Efficient Querying of XML Data Through Arbitrary Security Views

Houari Mahfoud[1](✉) and Abdessamad Imine[2]

[1] Abou-Bekr Belkaïd University, Tlemcen, Algeria
houari.mahfoud@gmail.com
[2] University of Lorraine and INRIA-LORIA, Nancy, France

Abstract. We study the problem of querying virtual security views of XML data that has received a great attention during the past years. A major concern here is that user XPath queries posed on recursive views cannot be rewritten to be evaluated on the underlying XML data. Existing rewriting solutions are based on the non-standard language, "Regular XPath", which makes rewriting possible under recursion. However, query rewriting under Regular XPath can be of exponential size. We show that query rewriting is always possible for arbitrary security views (recursive or not) by using only the expressive power of the standard XPath. We propose a more expressive language to specify XML access control policies as well as an efficient algorithm to enforce such policies. Finally, we present our system, called SVMAX, that implements our solutions and we show that it scales well through an extensive experimental study based on real-life DTD.

Keywords: XML access control · Security views · Materialization · Query rewriting · XPath · Regular XPath · XML databases · Confidentiality and integrity

1 Introduction

In parallel with the rapid growth of the World Wide Web, an increasing amount of data have become available electronically to humans and programs. Such data may be combined from heterogeneous systems based on different data formats, and need to be maintained in a self-describing format to accommodate a variety of ever-evolving business needs. This has led a need for a neutral and flexible way for exchanging data among different devices, systems, and applications. The solution to this problem came with the advent of XML [1,2].

The *eXtensible Markup Language* (XML) is a W3C recommendation that encodes data in a format which can be processed easily and exchanged across multiple platforms. XML has been universally received as the de facto standard for representing and exchanging data. An XML document represents not only base information, but also information about the relationship of data items to each other in the form of the hierarchy (*hierarchical structure*). Moreover, it

DOI: 10.1007/978-3-662-48567-5_3

can be searched or updated without requiring a static definition of the schema (*schema-less property*). XML brings a number of powerful capabilities to information modeling: (a) *Heterogeneity* (each record can contain different data fields); (b) *Extensibility* (new types of data can be added at will and do not need to be determined in advance); and (c) *Flexibility* (data fields can vary in size and configuration from instance to instance). These features and capabilities have made of XML the most used format for several needs and within various situations:

- *XML-based technologies*: XML has emerged as a critical enabler to various technology initiatives. Service-oriented architectures (*SOA*), enterprise application integration (*EAI*), web services, and standardization efforts in many industries all rely on or make use of XML as an underlying technology.
- *XML-based languages*: XML contributes on the creation of many markup languages for various domains such as *MathML* for mathematic, *CML* for chemistry, *SBML* and *BIOPAX* for biology, *SCORM* for e-learning.
- *Desktop applications*: *Open-office* files, *Ant's Build* files, and *Mac plist* configuration files are all written in XML format.

Specifically, we focus on situations where XML does not serve just as a technology or a configuration model, as explained above, but as a primordial format for data representation and exchange. Such situations are often encountered when the managed data has a *volatile schema* and is *inherently hierarchical* in nature. The properties of XML make it an *unavoidable* and more suitable format for this kind of data. We take first the case of medical data which is often presented with XML. A simple scenario of that is the "*Electronic Health Record* (*ERH*)[1]", an ongoing national project started in France at 2004, which has as goal to allow each one to access electronically to his own medical data (e.g. personal information, appointments, analysis results, medical and surgical history). Data of two different patients may not have the same rigid structure, e.g. one patient may have some surgical information whereas the other does not have any surgical item. Each department within the hospital may maintain his own volatile and local schema of data, and all schema may be combined to form the global data schema of the hospital. Moreover, the hospital data may be exchanged with other hospitals or laboratories that are not supposed to use the same schema. According to this context, it is much more natural to use XML for local data representation and to ensure efficiently the mapping between the different schema [3,4]. Many XML-based solutions are proposed for managing medical data: the *hData* [5] and *MEDOX* [6] frameworks, the *HL7* standard[2], solutions for interoperability of health-care applications [7], security [8,9] and integration [10,11] of health-care data. The other case occurs with the e-business where XML is an unavoidable standard not only for data representation, but essentially for ensuring interoperability of different systems. For this purpose, many XML-based

[1] The original name is the *DMP*, that refers in French to "*Dossier Médical Personnel*".
[2] Available at: http://www.hl7standards.com/.

solutions have been proposed: the *IBM jStart team* [12], the *DITA OASIS Standard* [13], the *ebXML consortium* [14], and the *Oracle* solutions [15] rely all on XML to address needs of managing and publishing business information.

Day to day operations that use XML data need to be easy to use, quick to carry out, and more importantly *safe* from *unauthorized* accesses. For instance, electronic commerce transactions require enforcement of some security constraints ensuring that crucial information will be accessible only to authorized entities. In addition, many organizations (mostly medical and commercial) manipulate *sensitive* information that should be *selectively exposed* to different classes of users based on their *access privileges*. A good example of such sensitive data is the "*EHR*" explained above. All patients' data are stored in a centralized database, and can be accessed totally/partially by different health personnels: nurses, doctors, pharmacists, insurance company staff, etc. Due to the sensitive nature of this data, a security policy is applied that controls access to different parts of the health-care data. For instance, grant to an insurance company a read access that concerns only medication information. The general scenario that can be found in practice is the following. For some XML data there may be multiple user groups which want to query the same data. For these user groups, different access privileges may be imposed, specifying what parts of the data are accessible to the users. The problem of secure XML access is to enforce these privileges. The well-established security specification and enforcement approaches of relational databases cannot be easily adapted for XML databases. This can be explained by the fact that XML has its own properties: an *hierarchical structure*, *schema-less* and *node relationship* properties. Consequently, the problem of secure access to XML data has its own particular flavor and requires specific solutions.

1.1 Motivation

It is increasingly common nowadays to find virtual views used to protect access to data as supported by many database systems (e.g. *Oracle 11g*, *IBM DB2*). Different models have been proposed that study such kind of protection [16–21]. Most of them deal only with read access rights. Given an XML document T that conforms to a schema D, a security view S is defined that heads some inaccessible information from D. According to S, a schema view D_v is derived first and provided to the user that describes the accessible data (s)he is able to see. Moreover, a virtual data view T_v is extracted that displays only accessible parts of T. XPath [22,23] is the most used language to query such virtual data view. For each XPath query Q posed on T_v, the *query rewriting* principle consists on rewriting Q into another one Q' such that: evaluating Q over T_v yields the same result as the evaluation of Q' over the original document T. Many rewriting algorithms have been proposed during the last decade [16,18,19,21,24,25].

Although a tremendous effort has been done on improving query rewriting over virtual XML views, most of existing algorithms are limited in the sens that they deal only with non recursive schema[3]. We investigate the use of DTD grammar as data schema. Recursive DTDs often arise in practice when specifying for instance (bio)medical and biological data. Examples of such DTDs are GedML and BIOML. The study done in [26] shown that most of the real-world DTDs are recursive. The rewriting process over virtual views becomes more challenging when manipulating recursive DTDs. Specifically, for two *accessible* nodes A and B, there may be some *inaccessible* nodes that connect A with B at the original data, these nodes are hidden in the view and thus B appears as immediate child of A in the virtual data view. Each query A/B must be rewritten to return only accessible B nodes that are either immediate children of some accessible nodes A or connected to them with only inaccessible nodes. Roughly, to rewrite a query A/B it remains to find all the inaccessible paths[4] that connect accessible nodes A with accessible B at the original data. Because of recursion, these paths may lead to an infinite set which cannot be explicitly expressed with the standard XPath. Thus, the query rewriting over recursive views is still an open problem.

For this reason, Fan et al. [17,27] proposed, as extension of their previous work [25], the first algorithm for coping with recursive security views. Their algorithm has been refined later by Groz et al. [18] by considering different types of DTDs and larger class of queries. The key idea behind these three works was to use the Regular XPath language [28] that is more expressive than the standard XPath and offers possibility to define recursive paths by means of the *Kleene star* operator "*". Although Regular XPath ensures query rewriting over arbitrary security views (recursive or non), this process may be costly since rewritten queries may be of exponential size. Regular XPath based investigations cannot be easily applied in practice: no tool exists to evaluate Regular XPath queries. Furthermore, more commercial database systems (e.g. *Oracle 11g*, *IBM DB2*, *eXist-db*, *Sedna*) provide support for the standard XPath to manipulate XML data. Therefore, there is a need for an XPath-based practical solution to secure XML data over arbitrary views.

Given the above, our first motivation at the outset was to develop some *practical* security solutions that can be easily and efficiently integrated within existing systems that provide support for managing XML data. We have focused principally on shortcomings of security-view-based approaches [17,18,24,25], and investigated some practical and efficient solutions to overcome them. This paper is thus a continuation to the important effort done during the two decades to design and implement XML access control models.

1.2 Contributions

An Efficient Approach for Coping with Arbitrary XML Security Views. While, in case of recursive security views, the query rewriting is not always possible over the downward fragment of XPath[5] [17] (class of queries

[3] A *recursive* schema has at least an element defined (in)directly in terms of itself.

[4] Paths composed by only inaccessible nodes.

[5] This fragment is more used both in practice and in theory, and several theoretical results have been found around this fragment [29,30].

with *child*-axis, *descendant*-axis, and complex predicates), we show that the expressive power of the standard XPath is sufficient to overcome this rewriting limitation. We extend the access specification language of Fan et al. [25] with new annotation types in order to define compact and more expressive XML access control policies. Then, we show that by extending the downward fragment of XPath with some axes and operators, the query rewriting becomes possible under arbitrary security views (recursive or non). As explained in Sect. 5, our rewriting approach can deal with a larger class of XPath queries that includes downward-axes (*child*, *descendant*), upward-axes (*parent*, *ancestor*). Moreover, it can be easily extended to rewrite horizontal-axes (*preceding*, *following*). We propose finally an efficient algorithm to rewrite XPath queries over arbitrary security views. Compared with the one presented in [17,27], our algorithm uses only the access specification (i.e. the read-access annotations) to rewrite any user query rather than using an auxiliary structure, like automatons, which can be costly or even impracticable in some cases. Moreover, our algorithm runs in linear time in the size of the query.

SVMAX system has been implemented to show the practicality and efficiency of our results. To our knowledge, SVMAX (*Secure and Valid MAnipulation of XML*) is the first system that provides secure handling of XML data over arbitrary views (recursive or non).

Further Contributions. We emphasize that SVMAX implements some other solutions, that are not explained here, but which are based on the results of this paper and then deserve a little discussion to complete the description of the system. We studied the XML access control by considering the operations of the XQuery Update Facility [31]. Our results in this context are based principally on the contribution of this paper. More precisely, we proposed in [32,33] a fine-grained language to specify XML update policies and which overcomes expressiveness limits of existing models [21,34]. Our update specification language is an extension of the read-access specification language that we describe in Sect. 4. SVMAX implements a linear time algorithm to enforce our XML update policies.

As we shall explain, SVMAX provides visual editor that helps the administrator to specify either read and update policies. These policies are enforced through the rewriting modules of the system: *XPath Rewriter* and *XQuery Update Rewriter* to rewrite safely, and w.r.t the corresponding policy, read-access queries and update queries respectively.

The wide use of W3C standards in practice makes of SVMAX a useful system that can be easily integrated, as an API, within commercial database systems. See [35] for more details of the system.

1.3 Outline of the Paper

The remainder of the paper is organized as follows. Section 2 provides essentially background about XML and XPath query language. We explain in Sect. 3 the main problem we tackle throughout this paper. Section 4 presents formal description of our access control model, and especially our access specification language. Policies based on such language are enforced through the rewriting approach explained in Sect. 5. Section 6 presents a brief overview of our system, followed by an extensive experimental study based on real-world DTDs. Related work is reviewed in Sect. 7. Finally, we conclude the paper in Sect. 8.

Additional parts of our contributions (algorithms, proofs,...) can be found on-line at https://tel.archives-ouvertes.fr/tel-01093661/.

2 Preliminaries

We present basic notions and definitions that are used throughout the paper.

2.1 Document Type Definitions

Definition 1 (DTD [1]). *A* Document Type Definition *(DTD) D is a triple (Σ, P, Root), where Σ is a finite set of* element types*; Root is a distinguished type in Σ called the* root type*; and P is a function defining element types such that for any A in Σ, P(A) is a regular expression α, called the* content model *of A, and 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 Σ, α','α denotes concatenation, and α'|'α denotes disjunction.* A → P*(A) refers to the* production rule *of* A*. For each element type* B *occurring in P(A), we refer to* B *as a* child type *of* A *and to* A *as a* parent type *of* B*. Moreover, P(A) can be defined 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)directly in terms of itself.*

Example 1. We consider the *department* DTD (*Σ*,*P*,*dept*) with *Σ*={*dept*, *course*, *project*, *cname*, *takenBy*, *givenBy*, *students*, *scholarship*, *student*, *sname*, *mark*, *professor*, *pname*, *grade*, *type*, *private*, *public*, *descp*, *results*, *result*, *members*, *member*, *name*, *qualif*, *theoretical*, *experimental*, *sub-project*}. The production rules of this DTD are defined as follows:

$$\begin{array}{ll}
dept & \rightarrow (course+,\ project*) \\
course & \rightarrow (cname,\ takenBy,\ givenBy) \\
takenBy & \rightarrow (students) \\
students & \rightarrow (scholarship?,\ student+) \\
scholarship & \rightarrow (student+) \\
student & \rightarrow (sname,\ mark) \\
givenBy & \rightarrow (professor+) \\
professor & \rightarrow (pname,\ grade) \\
project & \rightarrow (type,\ descp,\ results,\ members,\ sub\text{-}project) \\
type & \rightarrow (private \mid public) \\
results & \rightarrow (\texttt{str} \mid result)* \\
members & \rightarrow (member+) \\
member & \rightarrow (name,\ qualif,\ (theoretical \mid experimental)*) \\
sub\text{-}project & \rightarrow (project*)
\end{array}$$

The element types *private* and *public* are empty, while the remaining element types (e.g. *mark*, *result*) are text elements. A *dept* element has a list of *course* elements as well as zero or more *project* elements. A *course* consists of *cname* (course name), and lists of *students* and *professor* elements defined via the relations *takenBy* and *givenBy* respectively. A *student* who has registered for the *course* has a name (*sname*), a *mark* and may be part of a *scholarship* program. A *professor* is defined by his name (*pname*) and *grade*. A *project* is presented by its *type* (that can be either *private* or *public*), a description (*descp*), some *results*, and may be composed by zero or more than one projects (through the *sub-project* relation). A *member* of a given *project* is presented by his *name*, a qualification (denoted *qualif* that can be *professor*, *student*, *external researcher* etc.), and a list of his contributions (that can be either *theoretical* or *experimental*). Notice that *results* element type has mixed content (combination of text values that serve as comments, and *result* elements). Moreover, *member* element type has complex content, i.e. a sequence container that has the choice container (*theoretical* | *experimental*)*. □

2.2 XML Documents

We model an XML document with a finite node-labeled sibling-ordered unranked tree. Let Σ be a finite set of node labels (with a special label `str`) and C an infinite set of text values. We represent our XML documents with a structure, called *XML Tree*, defined as follows:

Definition 2 (XML Tree). *An XML tree T over Σ is a structure (N,root, $R_\downarrow$, $R_\rightarrow$, λ, ν), where N is a set of nodes, root $\in N$ is a distinguished root node, $R_\downarrow \subseteq N \times N$ is the parent-child relation, $R_\rightarrow \subseteq N \times N$ is a successor relation on (ordered) siblings, $\lambda : N \rightarrow \Sigma$ is a function assigning to every node its label, and $\nu : N \rightarrow C$ is a function assigning a text value to each node with label `str`.*

The relations $R_{\downarrow^*}$ and $R_{\rightarrow^*}$ represent the reflexive transitive closure of $R_\downarrow$ and $R_\rightarrow$ respectively. We use $R_\uparrow$ and $R_\leftarrow$ to denote the converse of $R_\downarrow$ and $R_\rightarrow$ respectively. In addition, $R_{\uparrow^*}$ and $R_{\leftarrow^*}$ denote respectively the converse of $R_{\downarrow^*}$ and $R_{\rightarrow^*}$. Contrary to the model defined in [28], we define the function ν to associate data values with nodes since data value comparison is supported by our XPath fragments defined subsequently.

Definition 3 (Validation of XML trees w.r.t DTD [28]). *An XML tree $T = (N, r, R_\downarrow, R_\rightarrow, \lambda, \nu)$, defined over the set Σ of node labels, conforms to a DTD D=($Ele, P, root$) if the following conditions hold:*

1. *The root of T is mapped to root (i.e. $\lambda(r)$=root);*
2. *Each node in T is labeled either with an element type A in* Ele*, called an* A element*, or with `str`, called a* text node*, therefore $\Sigma = Ele \cup \{$`str`$\}$;*
3. *For each A element with k ordered children $n_1, ..., n_k$, the word $\lambda(n_1), ..., \lambda(n_k)$ belongs to the regular language defined by* P*(A);*
4. *Each text node n (i.e. with $\lambda(n)$ = `str`) carries a string value $\nu(n)$ (i.e. `PCDATA`) and is the leaf of the tree.*

Note that elements of T are a set of nodes of N that are labeled with *Ele*, while nodes represent both elements and text nodes (i.e. nodes labeled with `str`). Subsequently, we use the terms of node and element interchangeably.

We call T an instance of a DTD D if T conforms to D. We denote by $\mathcal{T}(D)$ the set of all XML trees that conform to D. For instance, Fig. 1 depicts[6] an XML document that conforms to the *department* DTD of Example 1.

2.3 XPath Queries

We define here the different fragments of XPath [23] that are used throughout this paper.

Definition 4 (XPath Downward fragment). *We denote by $\mathcal{X}$ the* downward *fragment of XPath [36] that is defined as follows:*

$$\begin{aligned} p &:= \alpha::\eta \mid p[q]\cdots[q] \mid p/p \mid p \cup p \\ q &:= p \mid p{=}c \mid q \wedge q \mid q \vee q \mid \neg\ (q) \\ \alpha &:= \varepsilon \mid \downarrow \mid \downarrow^{+} \mid \downarrow^{*} \end{aligned}$$

where p denotes an XPath query and it is the start of the production, η 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, filter *or* predicate.

A qualifier q is said *valid* at a node n, denoted by $n \vDash q$, if and only if one of the following conditions holds: (i) q is an atomic predicate that, when evaluated over n, returns at least one node (i.e. there are some nodes reachable from n via q); (ii) q is given by $\alpha::text()=c$ and there is at least one node, reachable according to axis α from n, that has a text node with value c; (iii) q is a boolean expression and it is evaluated to true at n (e.g. $n \vDash \neg(q)$ if and only if the query q evaluates to empty set over n). We note by $\mathcal{S}[\![Q]\!](T)$ the set of nodes resulted from the evaluation of the query Q over the XML tree T.

We define in the following more expressive fragments of XPath that are used to overcome the query rewriting limitation discussed latter in Sect. 3.2.

Definition 5 (Extended fragment). *We consider an extended fragment of $\mathcal{X}$, denoted by $\mathcal{X}^{\Uparrow}_{[n,=]}$, and defined as follows:*

$$\begin{aligned} p &:= \alpha::\eta \mid p[q]\cdots[q] \mid p/p \mid p \cup p \mid p[1] \\ q &:= p \mid p{=}c \mid q \wedge q \mid q \vee q \mid \neg\ (q) \mid p = \varepsilon::* \\ \alpha &:= \varepsilon \mid \downarrow \mid \downarrow^{+} \mid \downarrow^{*} \mid \uparrow \mid \uparrow^{+} \mid \uparrow^{*} \end{aligned}$$

[6] We recall that indices in our examples of XML trees are used to distinguish between elements of the same type, e.g. *course*$_1$ and *course*$_2$. Moreover, because of space limitation we focus only on some nodes while $\triangle$ denotes the remaining ones.

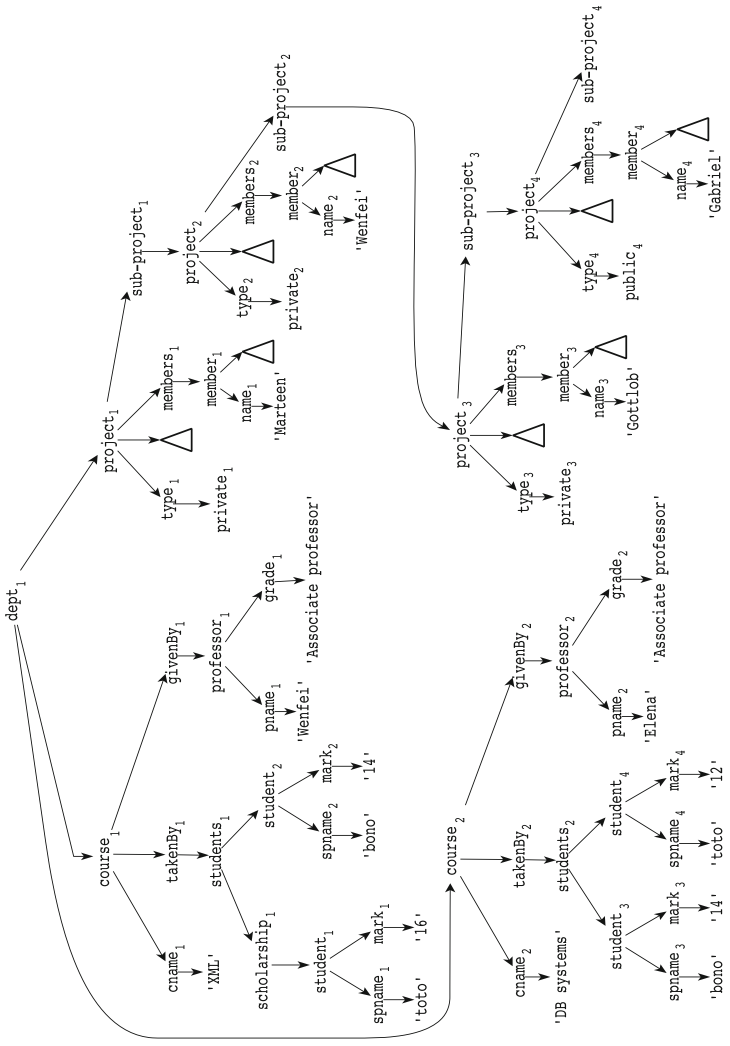

Fig. 1. Example of *department* XML document.

We enrich then $\mathcal{X}$ by the three upward-axes parent *($\uparrow$),* ancestor *($\uparrow^+$), and* ancestor-or-self *($\uparrow^*$), as well as the* position *and the* node comparison *predicates [23].*

In general [23], the position predicate, defined with $[k](k \in \mathbb{N})$, is used to return the k^{th} node from an ordered set of nodes. For instance, since we model XML documents as ordered trees, the query $\downarrow::*[2]$ at a node n returns its second child node. The *node comparison* is used to check the identity of two nodes. Specifically, the predicate $[p_1{=}p_2]$ is valid at a node n only if the evaluation of the right and left XPath queries at n results in exactly the same single node. Note that if p_1 and/or p_2 refer to more than one single node then a dynamic error is raised. The original XPath notation of the predicate $[p = \varepsilon::*]$ is given by $[p$ **is** $\varepsilon::*]$. However, we use "**=**" instead "**is**" for simplification. As an example, the predicate $[\uparrow^*::*[1]{=}\varepsilon::*]$ is valid at any node n since the queries $\uparrow^*::*[1]$ and $\varepsilon::*$ are equivalent and return the same single node over any context node. Contrary to the global definitions of position predicate (i.e. $[k]$ with $k \in \mathbb{N}$) and node comparison predicate (i.e. $[p_1{=}p_2]$) [23], for our purpose we need only the forms $[\mathit{1}]$ and $[p{=}\varepsilon::*]$ respectively. We define both restrictions since the resulting predicates are sufficient to overcome the limitation of XPath query rewriting as we shall show later. Furthermore, based on these restrictions our fragment of Definition 5 requires less evaluation time compared to the global fragment (defined with the global position and node comparison predicates).

We summarize our extensions of fragment $\mathcal{X}$ by the following subsets: $\mathcal{X}^{\Uparrow}$ ($\mathcal{X}$ with upward-axes), $\mathcal{X}^{\Uparrow}_{[n]}$ ($\mathcal{X}^{\Uparrow}$ with position predicate), and $\mathcal{X}^{\Uparrow}_{[n,=]}$ ($\mathcal{X}^{\Uparrow}_{[n]}$ with node comparison predicate). It should be noticed that we use fragment $\mathcal{X}$ to specify security policies as well as to formulate user requests (i.e. access queries and update operations). We will explain later how the augmented fragments of $\mathcal{X}$ defined above can be used to preserve confidentiality and integrity of XML data.

2.4 Regular XPath Queries

We talk about the extension of XPath queries with the *transitive closure operator* "*". For instance, the *reflexive transitive closure* of the XPath query $\downarrow::A$, denoted by $(\downarrow::A)^*$, is the infinite union (where ϵ denotes the empty query): $\epsilon \cup \downarrow::A \cup \downarrow::A/\downarrow::A \cup \downarrow::A/\downarrow::A/\downarrow::A \cup \ldots$. Transitive closure is a natural and useful operation that allows definition of recursive paths, and many languages for semistructured data support it (e.g. recursive SQL queries [37,38]). The major concern here is that XPath [22,23] does not support transitive closure, and thus arbitrary recursive paths are not expressible in this language [39].

In spite of its clear practical benefits, no XML engine supports the transitive closure operator. This has led researchers to define some extensions of the XPath language in order to enable definition of recursive path expressions. A useful study is given in [28] to know more about the theoretical properties of XPath 1.0 extended with regular path expressions. Based on their definitions, our class of Regular XPath queries, denoted by $\mathcal{X}_{reg}$, is defined as follows (p^* denotes an infinite repetition of the query p):

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

Based on the formal evaluation algorithm of $\mathcal{X}_{reg}$ queries described in [17], that relies on MFAs (*Mixed Finite state Automatas*), we get the following results:

Proposition 1. *Given an $\mathcal{X}_{reg}$ query Q defined over a DTD D, Q can be translated into an equivalent MFA M of size at most $O(|Q|.|D|)$ in at most $O(|Q|^2.|D|^2)$ time. Moreover, M can be evaluated over any instance T of D in at most $O(|Q|^2.|D|^2 + |Q|.|D|.|T|)$ time and space.*

3 Problem Statement

We present in this section the basic problem we tackle, namely answering XML queries over recursive security views.

3.1 XML Security Views

The notion of *security view*, introduced first by [40], consists on defining for each group of users a view of the underlying XML document that displays all and only parts of the document these users are allowed to access. Fan et al. [25] refined this notion by introducing first a language to specify fine-grained access control policies and a rewriting algorithm to enforce such policies. Security views are now the basic of most existing XML access control models [16–19,21,24,25,27,41,42].

Let T be an XML document that conforms to a DTD D. This document may be queried simultaneously by different users having different access privileges. An access control policy, as defined in [25], is an extension of the document DTD D associating *accessibility conditions* to element types of D. These conditions specify elements of T the users are granted access to. More specifically, an access specification is defined as follows:

Definition 6 (Access Specification *[25]***).** *An* access specification *S is a pair $(D$, $\texttt{ann})$ consisting of a DTD D and a partial mapping $\texttt{ann}$ such that, for each production rule $A \rightarrow P(A)$ and each element type B in $P(A)$, $\texttt{ann}(A, B)$, if explicitly defined, is an annotation of the form:*

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

where $[Q]$ is an XPath predicate. The root type of D is annotated Y by default.

Intuitively, the specification values Y, N, and $[Q]$ indicate that the B children of A elements in an instantiation of D are *accessible*, *inaccessible*, or *conditionally accessible* respectively. If $\texttt{ann}(A, B)$ is not explicitly defined, then B inherits the accessibility of A. On the other hand, if $\texttt{ann}(A, B)$ is explicitly defined then B may *override* the accessibility inherited from A.

Example 2. We consider the *department* DTD of Example 1 and we define some access privileges for professors. Assume that a professor, identified by his name $PNAME, can access to all his courses information except the information denoting whether or not a given student holds a scholarship. The access specification, $S{=}(dept,\ \boldsymbol{ann})$, corresponding to these privileges can be specified as follows:

$$\boldsymbol{ann}(dept,\ course) = \underbrace{[\downarrow::givenBy/\downarrow::professor/\downarrow::pname = \$\text{PNAME}]}_{Q_1}$$

$$\boldsymbol{ann}(students,\ scholarship) = N$$

$$\boldsymbol{ann}(scholarship,\ student) = [\uparrow^{+}::course[Q_1]]$$

Here $PNAME is treated as a constant parameter, i.e. when a concrete value, e.g., *Eichten*, is substituted for $PNAME, the specification defines the access control policy for the professor *Eichten*. Observe that $\boldsymbol{ann}(course,\ takenBy)$ is not explicitly defined, which means that in an instantiation of the *department* DTD, an *takenBy* element inherits its accessibility from its parent element *course*, this accessibility is either Y or N according to the evaluation of the predicate $[Q_1]$ at this *course* element. Similarly for *cname*, *students*, *givenBy* and his descendant types. The annotation $\boldsymbol{ann}(students,\ scholarship){=}N$ over a *scholarship* element overrides the accessibility inherited from its ancestor *course* to make this *scholarship* element inaccessible. Moreover, the annotation $\boldsymbol{ann}(scholarship,\ student){=}[\uparrow^{+}::course[Q_1]]$ overrides the accessibility N, inherited from *scholarship* element, and indicates that *student* children of *scholarship* elements are conditionally accessible (i.e. they are accessible if the professor is granted to access to their ancestor element *course*). □

Access control policies based on the specification of Definition 6 are enforced through the derivation of a *security view* [25]. A security view is an extension of the original XML document and the DTD that: (*1*) may be automatically derived from an access specification, (*2*) displays to the user all and only accessible parts of the XML document, (*3*) provides the user with a schema of all his accessible data so he can formulate and optimize his queries, and (*4*) allows a safe translation of user queries to prevent access to sensitive data[7]. More formally, an XML security view is defined as follows:

Definition 7 (Security View [25]). *Given an access specification $S{=}(D,\ \boldsymbol{ann})$ defined over a non-recursive DTD D, a* security view *V is a pair $(D_v,\ \sigma)$ where D_v is the DTD view of D that presents the schema of all and only data the user is granted access to, and σ is a function defined as follows: for any element type A and its child type B in D_v, $\sigma(A,\ B)$ is a set of XPath expressions that when evaluated over an A element of an XML tree T of D, returns all its accessible children B. In other words, σ maps each instance of D to an instance of D_v that contains only accessible data.*

[7] This translation is necessary only if the views of the data are virtual, i.e. not materialized.

The DTD view D_v is given to the user for formulation and optimization of queries. However, the set of XPath expressions defined by σ are hidden from the user and used to extract for any XML tree $T \in \mathcal{T}(D)$, a view T_v of T that contains all and only accessible nodes of T.

Example 3. Consider the access specification S=($dept$, $\boldsymbol{ann}$) of Example 2. The DTD view $dept_v$=(Σ_v, $dept$, P_v) of the *department* DTD can be computed by eliminating the *scholarship* element type, i.e. $\Sigma_v := \Sigma \setminus \{scholarship\}$, and by changing the definition of *dept* and *students* element types as follows:

$$\begin{array}{ll} P_v(dept) & := (course*,\ project*) \\ P_v(students) & := (student+) \\ P_v(A) & := P(A), \text{ for all remaining element types } A \text{ in } \Sigma_v \end{array}$$

The function σ is defined over the production rules of $dept_v$ as follows: (refer to Example 2 for the definition of $[Q_1]$)

$$\begin{array}{lll} dept & \rightarrow & P_v(dept): \\ & & \sigma(dept, course) = \downarrow::course[Q_1] \\ & & \sigma(dept, project) = \downarrow::project \\ students & \rightarrow & P_v(students): \\ & & \sigma(students, student) = \\ & & \downarrow::student \cup \downarrow::scholarship/\downarrow::student[\uparrow^+::course[Q_1] \\ A & \rightarrow & P_v(A)\text{: (for each remaining element type } A \text{ in } \Sigma_v) \\ & & \sigma(A, B) = \downarrow::B \text{ (for each child type } B \text{ in } P_v(A)) \end{array}$$

Using the resulting security view V=($dept_v$, σ), the view of the XML document of Fig. 1 is derived and depicted in Fig. 2, this view shows all and only parts of the original XML document that are accessible w.r.t the specification S=($dept$, $\boldsymbol{ann}$). Note that all descendants of the element $project_1$ are still unchanged. □

Given a security view V=(D_v, σ) defined for an access specification S=(D, $\boldsymbol{ann}$), then, for each instance T of D and its view T_v computed using the σ function, one can either materialize T_v and evaluate user queries directly over it [24,43], or keep T_v virtual for some reasons [17–19,21]. In case of virtual views, the *query rewriting* principle is used to translate each user query Q defined in D_v over the virtual view T_v, into a safe one Q^t defined in D over the original document T such that: evaluating Q over T_v returns the same set of nodes as the evaluation of the rewritten query Q^t over T.

Example 4. Consider the query $\downarrow::dept/\downarrow::course$ of the professor *Wenfei* defined over the view of Fig. 2. This query can be rewritten, using the security view of Example 3, to $\downarrow::dept/\sigma(dept, course)$ that is equal to:

$$\downarrow::dept/\downarrow::course[\downarrow::givenBy/\downarrow::professor/\downarrow::pname=\text{“}Wenfei\text{”}]$$

The evaluation of this query over the original XML document of Fig. 1 returns only accessible *course* elements, i.e. $course_1$. □

Since most existing approaches for securing XML data are based on the security view model, we discuss thereafter the major limits of this model.

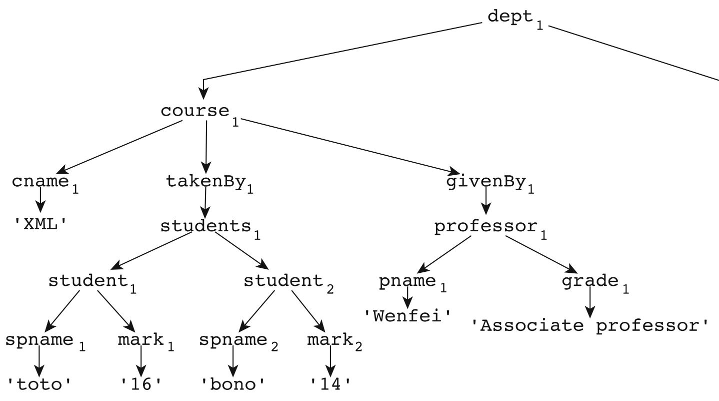

Fig. 2. The view of the *dept* XML document w.r.t the policy of Example 2.

3.2 Security View's Drawbacks

We study only the case of querying virtual XML data, then problems related to manipulation of materialized XML views [24,43] are outside the topic of interest of this work. More precisely, we discuss subsequently obstacles encountered when manipulating recursive views[8] and this at the stage of query rewriting. Even if the rewriting of XPath queries is quite straightforward for non-recursive XML security views, some obstacles may arise in the presence of recursive views that make this rewriting process impossible for some class of XPath queries. More precisely, the rewriting process is based on the definition of the function σ that, in case of recursive DTDs, cannot be defined in XPath as we show by the following example.

Example 5. We consider the *department* DTD of Example 1 and we assume that a personal of some department, identified by his name \$PNAME, can access to information of any project in which he is a member, as well as information of all public projects. The access specification, S=(*dept*, ***ann***), corresponding to these privileges is defined with:

$$\boldsymbol{ann}(dept, project) = \boldsymbol{ann}(sub\text{-}project, project) = \\ \underbrace{[\downarrow::type/\downarrow::public \vee \downarrow::members/\downarrow::member[\downarrow::name = \$\text{PNAME}]]}_{Q_2}$$

Note that if the predicate $[Q_2]$ is valid at a given *project* element then all its descendant elements inherit this accessibility except *sub-project* elements that may override it (that depends to the evaluation of $[Q_2]$). Consider the case of the professor "*Wenfei*", the view of the XML document of Fig. 1 is derived and depicted in Fig. 3. Given an accessible *dept* element, there is an infinite set of paths that connect this element to its accessible children of type *project*. More precisely, $\sigma(dept, project)$ can be defined using the transitive closure operator "$*$" with: $\sigma(dept, project) = (\downarrow::project[\neg(Q_2)]/\downarrow::sub\text{-}project)*/\downarrow::project[Q_2]$.

[8] A security view is *recursive* if it is defined over a recursive DTD.

The recursive path $(\downarrow::project[\neg(Q_2)]/\downarrow::sub\text{-}project)*$ is defined over only inaccessible elements. Thus, the expression $\sigma(dept, project)$ has to extract, over each accessible element of type *dept* in the original data, the accessible descendants of type *project* that appear in the view of the data as immediate children of this *dept* element. In other words, an element m of type *project* is shown in the view of Fig. 3 as an immediate child of some *dept* element n if and only: m and n are both accessible in the original tree of Fig. 1, and either m is an immediate child of n or separated from n with only inaccessible elements. Take the case of the elements $dept_1$ and $project_2$ of the tree of Fig. 1. After hiding the inaccessible element $project_1$, $project_2$ appears in the view of Fig. 3 as immediate child of $dept_1$. The same principle is applied for the elements $project_2$ and $project_4$. □

Authors of [28] showed that the kleen star operator "*" cannot be expressed in XPath. For this reason, the function σ of Example 5 cannot be defined in the standard XPath which makes the query rewriting process more challenging. We are principally motivated by studding the *closure* of a significant class of XPath queries (denoted by $\mathcal{X}$) under query rewriting, i.e. whether all queries of this class can be rewritten over arbitrary security views (recursive or not). We define formally the *closure property* as follows:

Definition 8. *An XML query language L is* closed *under query rewriting if there exists a function $\mathcal{R}$: $L \rightarrow L$ that, for any access specification $S=(D,\mathtt{ann})$ and any DTD view D_v of D, translates each query Q of L defined over D_v into another one $\mathcal{R}(Q)$ defined in L over D such that: for any $T \in \mathcal{T}(D)$ and its virtual view T_v, $\mathcal{S}[\![\mathcal{R}(Q)]\!](T)=\mathcal{S}[\![Q]\!](T_v)$.*

Note that Fan et al. [25] shown that the fragment $\mathcal{X}$ (Definition 4) is closed under query rewriting in case of non-recursive security views. However, in case of recursion, that is no longer the case as shows the following theorem:

Theorem 1 ([17,36]). *In case of recursive XML security views, the XPath fragment $\mathcal{X}$ is not closed under query rewriting.*

Finally, we should emphasize that no practical solution exists to respond to XML queries over recursive security views. Some theoretical results exist that are based on Regular XPath language which allows definition of recursive queries. According to [17,18], the fragment $\mathcal{X}_{reg}$ of Sect. 2.4 is closed under query rewriting. However, some major drawbacks are to be noted: no standard solution exists to evaluate regular queries, Regular XPath evaluation is more costly than standard XPath in general, and since contemporary database systems provide support for XPath only as XML query language, the results found around Regular XPath are still impractical.

4 Access Control with Arbitrary DTDs

Figure 4 presents our XML access control framework. It is designed particularly for native XML databases where XML data is stored in its native format. The

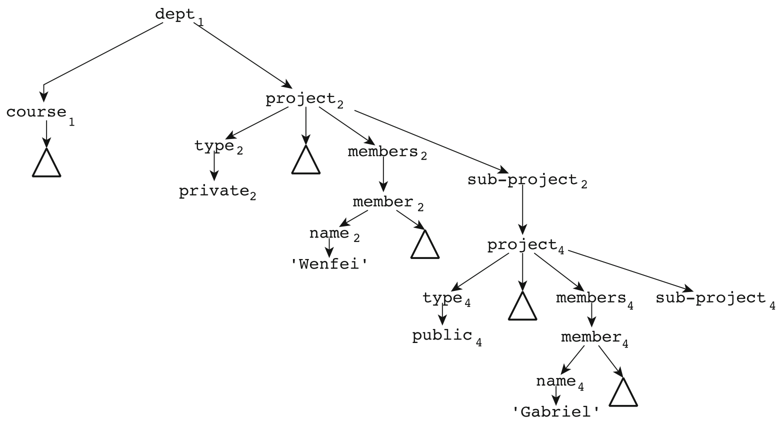

Fig. 3. The view of the *dept* XML document w.r.t the policy of Example 5.

module *Policy Specifier* allows the administrator to specify, for each group of users, the document they can query and an access control policy to handle this querying. According to this policy, the module *View Generator* computes a virtual view of their related document as well as a view (or an approximated view) of its corresponding DTD. This DTD view is used by the users to formulate their queries and query the virtual data view that is provided to them. Recall that the fragment $\mathcal{X}$ is used for user queries formulation. Each $\mathcal{X}$ query is rewritten into a safe one, defined in the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$, and evaluated over the original document. The results of this evaluation are given to the user as a set of sub-trees where each one presents an accessible node referred to by the input query.

We present in the following the *hospital DTD* that corresponds to a real-life patient medical data [44] and which is used throughout the rest of this paper.

Example 6. The *hospital DTD* (Σ,P,*hospital*) is defined with the following production rules (definitions of elements whose type is `str` are omitted):

$$\begin{array}{ll}
hospital & \rightarrow (department*) \\
department & \rightarrow (name,\ patient*) \\
patient & \rightarrow (pname,\ wardNo,\ parent?,\ sibling?, \\
 & \quad symptoms*,\ intervention) \\
parent & \rightarrow (patient*) \\
sibling & \rightarrow (patient*) \\
symptoms & \rightarrow (symptom*) \\
intervention & \rightarrow (doctor,\ treatment) \\
doctor & \rightarrow (dname,\ specialty)
\end{array}$$

$$\begin{array}{ll}
treatment & \rightarrow (type,\ Tresult*,\ diagnosis) \\
diagnosis & \rightarrow (Dresult*,\ implies?) \\
implies & \rightarrow (treatment \mid intervention)
\end{array}$$

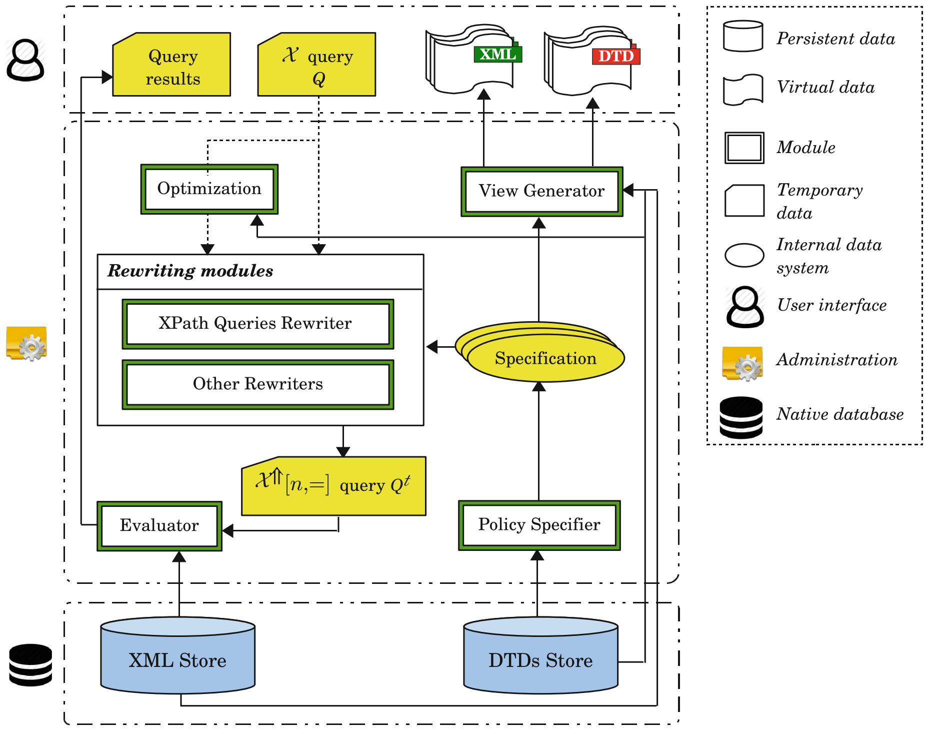

Fig. 4. Our XML access control framework.

A hospital DTD document consists of a list of departments, each *department* (defined by its *name*) has a list of patients currently residing in the hospital. For each *patient*, the hospital maintains her name (*pname*), a ward number (*wardNo*), a family medical history by means of the recursively defined *parent* and *sibling* relations, as well as a 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 responsible *doctor* (represented by its 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 diagnosis results (*Dresult*), either another treatment is planned or the intervention of another doctor/specialist/expert is solicited[9]. An instance of this hospital DTD is given in Fig. 5 (some text contents are abbreviated by '...'). □

[9] According to [44], this may happen when the required treatment is outside the area of expertise of the current responsible doctor.

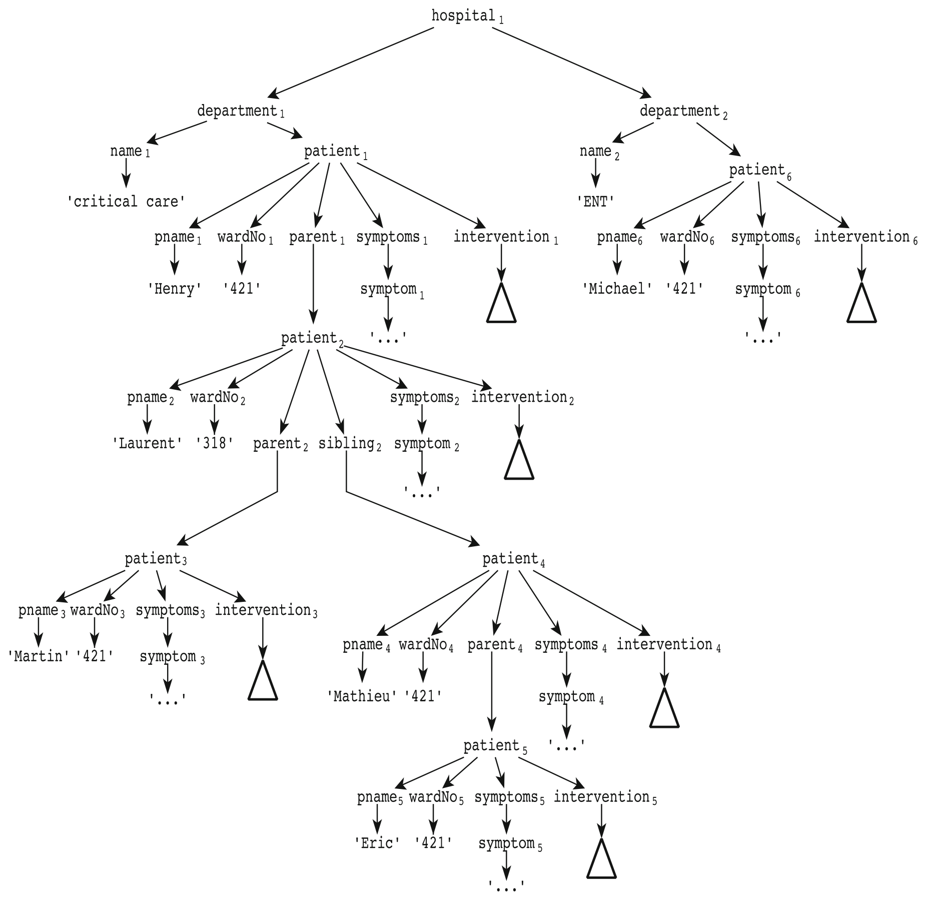

Fig. 5. Example of hospital data.

4.1 Access Specification

Fan et al. [25] proposed the first language for the specification of XML access control policies through annotation of DTD grammars. Moreover, authors of [24] studied the classification of such policies w.r.t the default annotation, the inheritance and the overriding of annotations. In this work we consider only the case of *top-down* access control policies where the root node of the XML tree is accessible by default and each intermediate node can either inherit the annotation of its parent node or override it (see Definition 6). Although both access specification languages defined in [24,25] are based on the same principle, i.e. annotating element types of DTDs with Y, N and $[Q]$, there is a significant difference in the use of conditional annotations (i.e. annotations of the form $[Q]$). We consider the following example for more details:

Example 7. We suppose that there are two annotations $\textit{ann}(A,\ B)$=[$\neg$ ($\downarrow$::D)] and $\textit{ann}(C,\ D)$=Y defined over a simple XML tree composed by only one path:

$$R \rightarrow A \rightarrow B \rightarrow C \rightarrow D$$

Note that the predicate [$\neg$ ($\downarrow$::D)] is invalid at the element node B. According to [25], all the subtree rooted at this B element is inaccessible and thus the second annotation that concerns the element node D does not take effect. According to [24] however, the element node D overrides the value N inherited from its ancestor element B and becomes accessible. □

In general, let n be an element node that is concerned by an annotation of the form [Q]. For the former work, if $n \nvDash Q$ then all the subtree rooted at n is inaccessible and no annotation defined over descendants of n can take effect. For the second work however, even if $n \nvDash Q$, descendants of n can override this annotation to become accessible.

We assume that the two definitions are useful and in practice applications may require the application of both kinds of annotations, even within the same scenario. For this reason, we present a refined and more expressive access specification language whose access specifications are defined as follows:

Definition 9 (Extended Access Specification). *We define an* access specification *S as a pair (D, $\textit{ann}$) consisting of a DTD D and a partial mapping $\textit{ann}$ such that, for each production rule $A \rightarrow P(A)$ and each element type B in $P(A)$, $\textit{ann}(A,\ B)$, if explicitly defined, is an annotation of the form:*

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

where [Q] is an XPath predicate. Annotations of the form N_h and $[Q]_h$ are called downward-closed annotations. *The root type of D is annotated Y by default.*

Recall from Definition 6 that annotations of the form Y, N, and [Q] indicate that an B element, child of an A element, is *accessible*, *inaccessible*, or *conditionally accessible* respectively. We allow overriding between annotations of the three previous forms. In other words, each element concerned by an annotation of the form Y, N, or [Q] overrides its inherited annotation if it is defined with one of these three forms. The special specification values N_h and $[Q]_h$ indicate that overriding is *denied* or *conditionally allowed* respectively. More specifically, let $n_1, \ldots, n_l$ ($l \geq 2$) be element nodes of types $A_1, \ldots, A_l$ respectively where each n_i ($1 \leq i < l$) is parent node of n_{i+1}. The annotation $\textit{ann}$ (A_1, A_2)=N_h indicates that all the subtree rooted at n_2 is inaccessible and no element under n_2 can override this annotation. Thus, if some annotation $\textit{ann}$ (A_i, A_{i+1})=Y|[Q] is explicitly defined then the element node n_{i+1} remains inaccessible even if $n_{i+1} \vDash Q$. However, the annotation $\textit{ann}$ (A_1, A_2)=$[Q_2]_h$ indicates that annotations defined over descendant types of A_2 take effect only if Q_2 is valid. In other words, given the annotation $\textit{ann}$ (A_i, A_{i+1})=Y (resp. [Q_{i+1}]), the element node n_{i+1} is accessible if and only if: $n_2 \vDash Q_2$ (resp. $n_2 \vDash Q_2 \wedge n_{i+1} \vDash Q_{i+1}$).

Example 8. Suppose that the hospital wants to impose some restrictions that allow some nurse to access only information of patients who are being treated in the *critical care* department and residing at the ward *421*. In addition, all sibling data should be inaccessible. This policy can be specified using our specification language with an access specification $S{=}(D, \textit{ann})$ where D is the hospital DTD and the function $\textit{ann}$ defines the three following annotations:

R_1: $\textit{ann}(hospital, department){=}[\underbrace{\downarrow ::name = \text{“}critical\ care''}_{Q_1}]_h$

R_2: $\textit{ann}(department, patient){=}\textit{ann}(parent, patient){=}[\underbrace{\downarrow ::wardNo = \text{“}421''}_{Q_2}]$

R_3: $\textit{ann}(patient, sibling){=}N_h$

According to this specification, the view of the data of Fig. 5 is extracted and depicted in Fig. 6. This view displays all and only the data the nurse is granted access to. All the data of the *ENT* department is hidden, i.e. the subtree rooted at the $departement_2$ element. Since R_1 is downward-closed and $departement_2 \nvDash Q_1$, then the annotation R_2 cannot be applied at $patient_6$ element which remains inaccessible even with $patient_6 \vDash Q_2$. Notice that $departement_1 \vDash Q_1$ which means that the $departement_1$ element is accessible and overriding of annotations is allowed for its descendants. Thus, the elements $patient_1$ and $patient_3$ are accessible along with their immediate children since Q_2 is valid at these elements, while the element $patient_2$ (with $patient_2 \nvDash Q_2$) overrides the annotation Y inherited from $patient_1$ and becomes inaccessible along with all its immediate children. In this way, $patient_3$ element appears at the view of Fig. 6 as immediate child of $parent_1$. Finally, since $sibling_2$ element is concerned by the downward-closed annotation R_3 with value N_h, then all the subtree rooted at $sibling_2$ is inaccessible and annotation R_2 cannot take effect over the elements $patient_4$ and $patient_5$. □

Our access specification language is more expressive than existing ones in the sens that the access policies of many current approaches can be specified in our language using only few annotation values as shown in Table 1. For instance, the policy of Example 8 cannot be specified in the fragment $\mathcal{X}$ using the specification languages presented in [24,25]. This can be done using a more expressive fragment, like $\mathcal{X}^{\Uparrow}$, but the annotations may be more verbose and difficult to manage.

The *completeness* and *consistency* of access control policies have been defined in [45] as follows. Let P be an access control policy and T be an XML tree. If a node n in T is not concerned by any access rule of P then P is *incomplete.* Moreover, if there are both a negative and a positive access rule for the same node n (i.e. n is both accessible and inaccessible) then P is *inconsistent.* Consider our access specifications of Definition 9, we define the notions of *completeness* and *consistency*, along the same lines as [24,25], as follows:

Definition 10. *Given an access specification $S{=}(D, \textit{ann})$ and an XML tree $T \in \mathcal{T}(D)$, then, we say that S is* complete *and* consistent *if and only if the*

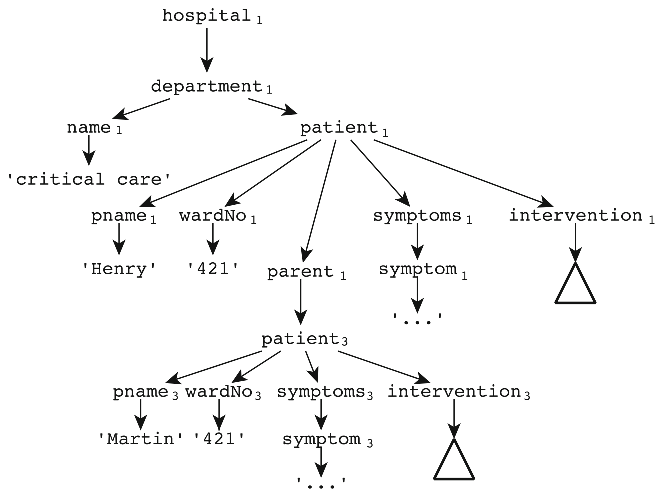

Fig. 6. View of the tree of Fig. 5 computed w.r.t the policy of Example 8.

accessibility *of each node in* T *is* uniquely *defined, i.e. it is either* accessible *or* inaccessible.

Proposition 2. *The access control policies based on Definition 9 are* complete *and* consistent.

Proof. Authors of [24] have proved that access policies defined with specification values of the form Y, N and $[Q]$ are complete and consistent. The case of downward-closed annotations is straightforward and the proof of the latter work can be easily extended to handle this kind of annotations. □

Table 1. Current approaches' policies specified with our language.

Access policies	Required specification values					Remark
	Y	N	N_h	$[Q]$	$[Q]_h$	
[17,25,27]	✓	✓			✓	
[24]	✓	✓		✓		case of top-down policies
[18]	✓	✓		✓		
[21]	✓		✓		✓	
[19]	✓	✓				
[46]	✓		✓		✓	*deny overwrites* as the conflict resolution policy
[47]	✓		✓		✓	with *denial downwards* consistency requirement

4.2 Accessibility

The enforcement of our access control policies relies principally on the definition of *node accessibility*. Inspired from [18,46], we define a single XPath filter, that can be constructed for any access specification, which checks whether a given XML node is *accessible* or not w.r.t this specification.

Definition 11. *Let n be an B element that is child of an A element. A given annotation* $\textit{ann}(A, B)$ *is* valid *at n if and only if* $\textit{ann}(A, B){=}Y|[Q]|[Q]_h$ *with* $n \vDash Q$. *Otherwise, it is* invalid, *i.e.* $\textit{ann}(A, B){=}N|N_h|[Q]|[Q]_h$ *with* $n \nvDash Q$.

If $\textit{ann}(A, B){=}[Q]_h$ with $n \vDash Q$ (resp. $\textit{ann}(A, B){=}N_h|[Q]_h$ with $n \nvDash Q$) then we talk about *valid* (resp. *invalid*) downward-closed annotation. Given the above, we define the node accessibility as follows:

Definition 12. *Let $S{=}(D, \textit{ann})$ be an access specification, T be an instance of D, and n be an element node in T of type B having parent node of type A. The element node n is* accessible *w.r.t S if and only if the following conditions hold:*

(i) Either there exists an explicitly defined annotation $\textit{ann}(A, B)$ that is valid at n; or the first annotation explicitly defined over ancestors of n is valid.
(ii) There is no invalid downward-closed annotation defined over ancestors of n.

More specifically, consider the element nodes $n_1, \ldots, n_k$ ($k \geq 2$) of element types $A_1, \ldots, A_k$ respectively where n_1 is the root node. Take the case of the element node n_k, the condition (i) of Definition 12 refers to one of the following three cases:

(a) Only the default annotation $\textit{ann}$ $(A_1){=}Y$ is defined over the types $A_1, \ldots, A_k$. Thus, n_k inherits its accessibility from the root node n_1.
(b) The annotation $\textit{ann}$ (A_{k-1}, A_k) is explicitly defined and valid at n_k.
(c) The annotation $\textit{ann}$ (A_{i-1}, A_i) is explicitly defined and valid at the element n_i $(1 < i < k)$, and no annotation is defined over the types $A_{i+1}, \ldots, A_k$. Thus, n_k inherits its accessibility from its ancestor node n_i.

The condition (ii) of Definition 12 implies that for any downward-closed annotation $\textit{ann}(A_{i-1}, A_i)$ defined over ancestor n_i of n_k (with $1 < i < k$), either $\textit{ann}(A_{i-1}, A_i) \neq N_h$ or $\textit{ann}(A_{i-1}, A_i){=}[Q]_h$ with $n_i \vDash Q$. Finally, note that a text node is accessible if and only if its parent element is accessible.

Definition 13. *Given an access specification $S{=}(D, \textit{ann})$, we define two $\mathcal{X}^{\Uparrow}_{[n]}$ predicates $\mathcal{A}^{acc}_1$ and $\mathcal{A}^{acc}_2$ as follows:*

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

$\mathcal{A}_2^{acc}$:= $\bigwedge_{(\textit{ann}(A',A)=[Q]_h)\in \textit{ann}}$ $\neg$ $(\uparrow^{+}::A[\neg\ (Q)]/\uparrow::A')$ $\bigwedge_{(\textit{ann}(A',A)=N_h)\in \textit{ann}}$ $\neg$ $(\uparrow^{+}::A/\uparrow::A')$

The predicates $\mathcal{A}_1^{acc}$ and $\mathcal{A}_2^{acc}$ satisfy the conditions (i) and (ii) of Definition 12 respectively.

The first predicate checks whether the node n is explicitly concerned by a valid annotation (case **b**) or inherits its accessibility from a valid annotation defined over its ancestors (cases **a** and **c**). The second predicate checks whether the node n is not in the scope of an invalid downward-closed annotation. The predicate $[allAnn]$ consists of a disjunction of all annotations, while $[validAnn]$ presents disjunction of only valid annotations. More precisely, the evaluation of the predicate $\uparrow^{*}::*[allAnn]$ at a node n returns an ordered set of nodes N that contains the node n and/or some of its ancestors such that each one is "explicitly" concerned by an annotation of S, i.e. $N \subseteq \{n\} \cup \textit{ancestors}(n)$[10], and $\forall m \in N$, m is of type B and has a parent node of type A where $\textit{ann}(A, B)$ is explicitly defined in S. The predicate $\uparrow^{*}::*[allAnn][1]$ (i.e. $N[1]$) returns the first node in N, i.e. either the node n (if it is explicitly concerned by an annotation), the first ancestor of n that is explicitly concerned by an annotation, or the root node (if only the default annotation is defined). The last predicate $[validAnn]$ checks whether the annotation defined over the node $N[1]$ is valid: this means that either the node n is explicitly concerned by a valid annotation or it inherits its accessibility from one of its ancestors that is concerned by a valid annotation (condition *(i)*). The use of the second predicate $\mathcal{A}_2^{acc}$ is obvious: if $n \vDash \mathcal{A}_2^{acc}$ then all the downward-closed annotations defined over $\textit{ancestors}(n)$ are valid (condition *(ii)*).

Lemma 1. *Given an access specification S=(D, **ann**), we define the* accessibility predicate *$\mathcal{A}^{acc}$:=$\mathcal{A}_1^{acc} \wedge \mathcal{A}_2^{acc}$ such that: for any XML tree $T \in \mathcal{T}(D)$, a node n of T is accessible if and only if $n \vDash \mathcal{A}^{acc}$.*

According to this lemma, for any access specification S=(D, **ann**) and any XML tree $T \in \mathcal{T}(D)$, the query $\downarrow^{*}::*[\mathcal{A}^{acc}]$ over T returns the set of all accessible nodes of T where $\mathcal{A}^{acc}$ is computed w.r.t S.

Example 9. Consider the access policy of nurses defined in Example 8 with the following annotations:

$$\textit{ann}(hospital, department) = [\underbrace{\downarrow ::name = \text{“}critical\ care''}_{Q_1}]_h$$

$$\textit{ann}(department, patient) = \textit{ann}(parent, patient) = [\underbrace{\downarrow ::wardNo = \text{“}421''}_{Q_2}]$$

$$\textit{ann}(patient, sibling) = N_h$$

According to these annotations, the predicates $\mathcal{A}_1^{acc}$ and $\mathcal{A}_2^{acc}$, that compose $\mathcal{A}^{acc}$, are defined as follows:

[10] We use $\textit{ancestors}(n)$ to refer to all ancestors of the node n.

$\mathcal{A}_1^{acc}$:= $\uparrow^*$::$*$[$allAnn$][1][$validAnn$], where:
$allAnn := \varepsilon::root \vee \varepsilon::department/\uparrow::hospital \vee \varepsilon::patient/\uparrow::department \vee \varepsilon::patient/\uparrow::parent \vee \varepsilon::sibling/\uparrow::patient$

$validAnn := \varepsilon::department[Q_1]/\uparrow::hospital \vee \varepsilon::patient[Q_2]/\uparrow::department \vee \varepsilon::patient[Q_2]/\uparrow::parent \vee \varepsilon::root$

$$\mathcal{A}_2^{acc} := \neg\ (\uparrow^{+}::departement[\neg\ (Q_1)]/\uparrow::hospital) \wedge \\ \neg\ (\uparrow^{+}::sibling/\uparrow::patient)$$

Consider the case of the element $patient_1$ of Fig. 5. The predicate $\uparrow^*::*[allAnn]$ at $patient_1$ returns the set N=$\{patient_1, departement_1, hospital_1\}$ (each element is concerned by an explicit annotation). We have N[1]= $\{patient_1\}$ and the predicate [$validAnn$] is valid at $patient_1$ (since $patient_1 \vDash Q_2$). Thus, the predicate $\mathcal{A}_1^{acc}$ is valid at $patient_1$. It is clear to see that $\mathcal{A}_2^{acc}$ is also valid at $patient_1$. We conclude that $patient_1 \vDash (\mathcal{A}_1^{acc} \wedge \mathcal{A}_2^{acc})$ which means that the element $patient_1$ is accessible. Consider now the element $patient_2$, $\uparrow^*::*[allAnn]$ at $patient_2$ returns the set N'=$\{patient_2, patient_1, departement_1, hospital_1\}$, N'[1]=$\{patient_2\}$, however, the predicate [$validAnn$] is not valid at $patient_2$ (since $patient_2 \nvDash Q_2$). Thus, $patient_2 \nvDash \mathcal{A}_1^{acc}$ and then the element $patient_2$ is not accessible. For the element $patient_4$, although $patient_4 \vDash \mathcal{A}_1^{acc}$, $patient_4$ is inaccessible since $patient_4 \nvDash \mathcal{A}_2^{acc}$ (i.e. $patient_4$ is descendant of $sibling_2$ element that is concerned by an invalid downward-closed annotation). Finally, the query $\downarrow^*::*[\mathcal{A}^{acc}]$ over the Fig. 5 returns all the accessible elements that compose the view of Fig. 6. □

5 Query Rewriting

We discuss in this section the basic principle of our XML access control approach. We recall that the fragment $\mathcal{X}$ (see Definition 4) is used in our approach for specification of access control policies as well as for formulation of user queries. However, we use more larger fragments of XPath to overcome the query answering problem presented in Sect. 3.2. More precisely, the access control policies based on Definition 9 are enforced through a rewriting technique. Let S=(D, ann) be an access specification, T be an instance of D, T_v be the virtual view of T computed w.r.t S, and Q be a query defined in $\mathcal{X}$. Our goal is to define a rewriting function *Rewrite* such that:

$$\mathcal{X} \longrightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$$
$$Q \longmapsto Rewrite(Q) \text{ such that } \mathcal{S}[\![Rewrite(Q)]\!](T)=\mathcal{S}[\![Q]\!](T_v)$$

5.1 Queries Without Predicates

Let us now consider queries without predicates, postponing rewriting of predicates to the next subsection. We consider the case of $\mathcal{X}$ queries of the form $\alpha_1::\eta_1/\cdots/\alpha_k::\eta_k$ ($k \geq 1$) where $\alpha_i \in \{\varepsilon, \downarrow, \downarrow^*, \downarrow^+\}$ and η_i can be any element type, $*$-label, or *text()* function. The union of queries is discussed later. We show first that the rewriting limitation for this kind of queries is encountered when manipulating the $\downarrow$ axis, however, the remaining axes can be rewritten in a simple manner using only the accessibility predicate.

Example 10. Consider the XML tree of Fig. 5 and its view depicted in Fig. 6 that is computed w.r.t the access policy of Example 8. We suppose the the nurse formulates the query $\downarrow^{+}{::}departement/\downarrow^{+}{::}patient$ over its data view which returns the nodes $patient_1$ and $patient_3$. It is easy to see that this query can be rewritten over the original data into $\downarrow^{+}{::}departement[\mathcal{A}^{acc}]/\downarrow^{+}{::}patient[\mathcal{A}^{acc}]$ where the predicate $\mathcal{A}^{acc}$ is given in Example 9. Obviously, this rewritten query selects first accessible *departement* elements of Fig. 5, i.e. $departement_1$ element, and then returns all its accessible descendants of type *patient*, i.e. $patient_1$ and $patient_3$. The accessibility of these nodes are checked using $\mathcal{A}^{acc}$. Consider now another query over the data view of nurses defined by $\downarrow^{*}{::}parent/\downarrow{::}*$ and which must return only the node $patient_3$. Since there is a cycle between the *patient* and *parent* elements of the hospital DTD, this latter query cannot be rewritten using only the accessibility predicate. More precisely, the query $\downarrow^{*}{::}parent[\mathcal{A}^{acc}]/\downarrow{::}*[\mathcal{A}^{acc}]$ over the original document returns no element since it selects first the accessible element $parent_1$, while its immediate child $patient_2$ is not accessible. Moreover, a cycle cannot be captured by replacing $\downarrow$ axes with $\downarrow^{*}$ axes. The query $\downarrow^{*}{::}parent[\mathcal{A}^{acc}]/\downarrow{::}*[\mathcal{A}^{acc}]$ over the original document returns both the node $patient_3$ as well as other additional elements: $pname_3$, $symptoms_3$, $symptom_3$, etc. □

We show in the following how that the upward axes and the position predicate of the XPath fragment $\mathcal{X}^{\Uparrow}_{[n]}$ can be used to overcome the rewriting limitation encountered when considering $\mathcal{X}$ queries without predicates.

Definition 14. *Given an access specification S=(D, **ann**) and an element type B, then we define two $\mathcal{X}^{\Uparrow}_{[n]}$ predicates $\mathcal{A}^{+}$ and $\mathcal{A}^{B}$ with: $\mathcal{A}^{+} := \uparrow^{+}{::}*[\mathcal{A}^{acc}]$, and $\mathcal{A}^{B} := \uparrow^{+}{::}*[\mathcal{A}^{acc}][1]/\varepsilon{::}B$. For any element node n, the evaluation $\mathcal{S}[\![\mathcal{A}^{+}]\!](\{n\})$ returns all the accessible ancestors of n, while $\mathcal{S}[\![\mathcal{A}^{B}]\!](\{n\})$ returns the first accessible ancestor of n whose type is B.*

Finally, we give the details of our rewriting function. Given an access specification S=(D, ***ann***), we define the function ***Rewrite***: $\mathcal{X} \longrightarrow \mathcal{X}^{\Uparrow}_{[n]}$ that rewrites any $\mathcal{X}$ query Q, of the form $\alpha_1{::}\eta_1/\cdots/\alpha_k{::}\eta_k$ ($k \geq 1$), into another one defined in the fragment $\mathcal{X}^{\Uparrow}_{[n]}$ as follows:

$$\textbf{\textit{Rewrite}}(Q) := \downarrow^{*}{::}\eta_n[\mathcal{A}^{acc}][prefix^{-1}(\alpha_1{::}\eta_1/\cdots/\alpha_k{::}\eta_k)]$$

The qualifier $prefix^{-1}(\alpha_1{::}\eta_1/\cdots/\alpha_k{::}\eta_k)$ presents a recursive rewriting in a descendant manner where each sub-query $\alpha_i{::}\eta_i$ is rewritten over all the sub-queries that precede it in the query Q. In other words, for each sub-query $\alpha_i{::}\eta_i$ ($1 \leq i \leq k$), $prefix^{-1}(\alpha_1{::}\eta_1/\cdots/\alpha_{i-1}{::}\eta_{i-1})$ is already computed and used to compute $prefix^{-1}(\alpha_1{::}\eta_1/\cdots/\alpha_i{::}\eta_i)$ as follows:[11]

[11] For $\alpha_i \in \{\downarrow^{+}, \downarrow^{*}\}$, α_i^{-1}=$\uparrow^{+}$ if α_i=$\downarrow^{+}$ and $\uparrow^{*}$ otherwise.

– $\alpha_i = \downarrow$:
  $prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_i::\eta_i) := \mathcal{A}^{\eta_{i-1}}[prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_{i-1}::\eta_{i-1})]$
– $\alpha_i \in \{\downarrow^+, \downarrow^*\}$:
  $prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_i::\eta_i) := \alpha_i^{-1}::\eta_{i-1}[\mathcal{A}^{acc}][prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_{i-1}::\eta_{i-1})]$
– $\alpha_i = \varepsilon$:
  $prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_i::\eta_i) := \varepsilon::\eta_{i-1}[prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_{i-1}::\eta_{i-1})]$

As a special case, the first sub-query is rewritten over the root type. Thus, we have $prefix^{-1}(\downarrow::\eta_1)$=$\mathcal{A}^{root}$, $prefix^{-1}(\downarrow^+::\eta_1)$=$\uparrow^+::root$, while for the remaining axes, $\alpha_1 \in \{\varepsilon, \downarrow^*\}$, $prefix^{-1}(\alpha_1::\eta_1)$ is empty.

Example 11. Let us consider the query Q=$\downarrow^*::parent/\downarrow::*$ of Example 10 posed over the data view of Fig. 6. By considering the access specification of Example 8, this query can be rewritten as follows: $\mathit{Rewrite}(Q)$=$\downarrow^*::*[\mathcal{A}^{acc}][\mathcal{A}^{parent}]$. By replacing $\mathcal{A}^{parent}$ with its value, we obtain: $\downarrow^*::*[\mathcal{A}^{acc}][\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::parent]$. Recall that the definition of the predicate $\mathcal{A}^{acc}$ w.r.t the access specification of Example 8 is given in Example 9. The evaluation of the query $\downarrow^*::*[\mathcal{A}^{acc}]$ over the original document of Fig. 5 returns a node set N composed by all the accessible nodes depicted in Fig. 6. The evaluation of $[\mathcal{A}^{parent}]$ over the set N returns only those elements having as the first accessible ancestor, an element of type $parent$, thus the query $\downarrow^*::*[\mathcal{A}^{acc}][\mathcal{A}^{parent}]$ over the original document returns the element $patient_3$ that is the only element that satisfies the predicate $[\mathcal{A}^{parent}]$: $\mathcal{S}[\![\mathcal{A}^{parent}]\!](\{patient_3\})$ returns the element $parent_1$, i.e. $patient_3 \vDash \mathcal{A}^{parent}$. Therefore, the query $\mathit{Rewrite}(Q)$ over the original document of Fig. 5 returns only the element $patient_3$ as does the query Q over the data view of Fig. 6. □

5.2 Rewriting Predicates

We discuss in this section the rewriting of predicates of the fragment $\mathcal{X}$ to complete the description of our rewriting approach. Given an access specification S=(D, ann), we define the function $\mathit{RW\_Pred}$: $\mathcal{X} \rightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$ that rewrites any $\mathcal{X}$ predicate P, of the form $\alpha_1::\eta_1/\cdots/\alpha_k::\eta_k$ ($k \geq 1$), into another one defined in the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$. In a descendant manner, $\mathit{RW\_Pred}(P)$ is recursively defined over sub-predicates of P as follows:

– α_i =$\downarrow$:
  $\mathtt{RW\_Pred}(\alpha_i::\eta_i/\cdots/\alpha_k::\eta_k)$:=
    $\downarrow^+::\eta_i[\mathcal{A}^{acc}][\mathtt{RW\_Pred}(\alpha_{i+1}::\eta_{i+1}/\cdots/\alpha_k::\eta_k)]/\mathcal{A}^+[1]$=$\varepsilon::*$
– $\alpha_i \in \{\downarrow^+, \downarrow^*\}$:
  $\mathtt{RW\_Pred}(\alpha_i::\eta_i/\cdots/\alpha_k::\eta_k)$:=
    $\alpha_i::\eta_i[\mathcal{A}^{acc}][\mathtt{RW\_Pred}(\alpha_{i+1}::\eta_{i+1}/\cdots/\alpha_k::\eta_k)]$
– $\alpha_i = \varepsilon$:
  $\mathtt{RW\_Pred}(\alpha_i::\eta_i/\cdots/\alpha_k::\eta_k) := \varepsilon::\eta_i[\mathtt{RW\_Pred}(\alpha_{i+1}::\eta_{i+1}/\cdots/\alpha_k::\eta_k)]$

As a special case, the predicate $\alpha::\eta/text()$=*'c'* (text-content comparison) is rewritten, according to the axis α, as follows:

- $\texttt{RW\_Pred}(\downarrow::\eta/text()=\text{'}c\text{'}) := \downarrow^{+}::\eta[\mathcal{A}^{acc}][self::*/text()=\text{'}c\text{'}]/\mathcal{A}^{+}[1] = \varepsilon::*$
- For $\alpha \in \{\downarrow^{+}, \downarrow^{*}\}$, $\texttt{RW\_Pred}(\alpha::\eta/text()=\text{'}c\text{'}) := \alpha::\eta[\mathcal{A}^{acc}]/text()=\text{'}c\text{'}$
- $\texttt{RW\_Pred}(\varepsilon::\eta/text()=\text{'}c\text{'}) := \varepsilon::\eta/text()=\text{'}c\text{'}$

Example 12. Consider the access specification of Example 8 and the data view of Fig. 6. It is clear that the predicate $[\underbrace{\downarrow::patient/\downarrow::wardNo = \text{"}421''}_{P}]$ is satisfied only over the element node $parent_1$. This predicate is rewritten into $[\mathit{RW\_Pred}(P)]$ as follows:

- $[\mathit{RW\_Pred}(P)] = [\downarrow^{+}::patient[\mathcal{A}^{acc}][\mathit{RW\_Pred}(\downarrow::wardNo=\text{"}421\text{"})]/\mathcal{A}^{+}[1]=\varepsilon::*]$
- $[\mathit{RW\_Pred}(\downarrow::wardNo=\text{"}421\text{"})] =$
  $[\downarrow^{+}::wardNo[\mathcal{A}^{acc}][\varepsilon::*/text()=\text{"}421\text{"}]/\mathcal{A}^{+}[1]=\varepsilon::*]$

Consider the XML document of Fig. 5, it is easy to check that the predicate $[\mathit{RW\_Pred}(P)]$ is satisfied only over the element node $parent_1$. □

Finally, we generalize the definition of the function *Rewrite* to take into account all queries of the fragment $\mathcal{X}$. Given an access specification $S=(D, \mathit{ann})$, the function $\mathit{Rewrite}: \mathcal{X} \longrightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$ is redefined to rewrite any $\mathcal{X}$ query Q, of the form $\alpha_1::\eta_1[p_1]/\cdots/\alpha_k::\eta_k[p_k]$ $(k \geq 1)$, into another one defined in the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ as follows (where $p_i^t=\mathit{RW\_Pred}(p_i)$ for $1 \leq i \leq k$):

$$\mathit{Rewrite}(Q) := \downarrow^{*}::\eta_k[\mathcal{A}^{acc}][p_k^t][prefix^{-1}(Q)]$$

The qualifier $prefix^{-1}(Q)$ is recursively defined as follows:

- $\alpha_i = \downarrow$:
  $prefix^{-1}(\alpha_1::\eta_1[p_1]/\cdots/\alpha_i::\eta_i[p_i]) :=$
  $\mathcal{A}^{\eta_{i-1}}[p_{i-1}^t][prefix^{-1}(\alpha_1::\eta_1[p_1]/\cdots/\alpha_{i-1}::\eta_{i-1}[p_{i-1}])]$
- $\alpha_i \in \{\downarrow^{+}, \downarrow^{*}\}$:
  $prefix^{-1}(\alpha_1::\eta_1[p_1]/\cdots/\alpha_i::\eta_i[p_i]) :=$
  $\alpha_i^{-1}::\eta_{i-1}[p_{i-1}^t][\mathcal{A}^{acc}][prefix^{-1}(\alpha_1::\eta_1[p_1]/\cdots/\alpha_{i-1}::\eta_{i-1}[p_{i-1}])]$
- $\alpha_i = \varepsilon$:
  $prefix^{-1}(\alpha_1::\eta_1[p_1]/\cdots/\alpha_i::\eta_i[p_i]) :=$
  $\varepsilon::\eta_{i-1}[p_{i-1}^t][prefix^{-1}(\alpha_1::\eta_1[p_1]/\cdots/\alpha_{i-1}::\eta_{i-1}[p_{i-1}])]$

As a special case, query of $\mathcal{X}$ of the form $Q_1 \cup \cdots \cup Q_k$ $(k \geq 1)$ is rewritten into $\mathit{Rewrite}(Q_1) \cup \cdots \cup \mathit{Rewrite}(Q_k)$.

Example 13. Consider the access specification defined in Example 8. The $\mathcal{X}$ query $Q{=}\downarrow^{+}::parent/\downarrow::patient[\underbrace{\downarrow::pname = \text{"}Martin''}_{P}]$ over the data view of Fig. 6 is rewritten over the original data of Fig. 5 as follows:

$$\mathit{Rewrite}(Q){=}\downarrow^{*}::patient[\mathcal{A}^{acc}][\mathit{RW\_Pred}(P)][\uparrow^{+}::*[\mathcal{A}^{acc}][1]/\varepsilon::parent]$$

$$\mathit{RW\_Pred}(P) = [\downarrow^{*}::pname[\mathcal{A}^{acc}][\varepsilon::*/text()=\text{"}Martin\text{"}]/\mathcal{A}^{+}[1]=\varepsilon::*]$$

The evaluation of the query $\mathtt{Rewrite}(Q)$ over the original data returns the element node $patient_3$ as does the query Q over the data view. □

We emphasize that the generalization of the function $\mathtt{RW\_Pred}$ to handle complex predicates is quite straightforward. For instance, $\mathtt{RW\_Pred}(P_1 \vee P_2)$ is given by $\mathtt{RW\_Pred}(P_1) \vee \mathtt{RW\_Pred}(P_2)$. Moreover, $\mathtt{RW\_Pred}(P_1[P_2])$ is given by $\mathtt{RW\_Pred}(P_1[\mathtt{RW\_Pred}(P_2)])$.

5.3 Coping with $\mathcal{X}^{\Uparrow}$ queries

We show how our rewriting function $\mathtt{Rewrite}$ can be extended to rewrite the upward axes $\{\uparrow, \uparrow^+, \uparrow^*\}$. Let $S{=}(D,\ ann)$ be an access specification. Firstly, the function $\mathtt{Rewrite}$: $\mathcal{X}^{\Uparrow} \longrightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$ is redefined to rewrite any $\mathcal{X}^{\Uparrow}$ query Q, of the form $\alpha_1{::}\eta_1[p_1]/\cdots/\alpha_k{::}\eta_k[p_k]$ $(k \geq 1)$, into another one defined in the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ as follows (we consider only the case where $\alpha_i \in \{\uparrow, \uparrow^+, \uparrow^*\}$ since the case of the remaining axes is already studied):

$$\mathtt{Rewrite}(Q) := \downarrow^*{::}\eta_k[\mathcal{A}^{acc}][p_k^t][prefix^{-1}(Q)]$$

The qualifier $prefix^{-1}(Q)$ is recursively defined as follows:

- $\alpha_i = \uparrow$:
  $prefix^{-1}(\alpha_1{::}\eta_1[p_1]/\cdots/\alpha_i{::}\eta_i[p_i]) :=$
  $\downarrow^+{::}\eta_{i-1}[\mathcal{A}^{acc}][p_{i-1}^t][prefix^{-1}(\alpha_1{::}\eta_1[p_1]/\cdots/\alpha_{i-1}{::}\eta_{i-1}[p_{i-1}])]/$
  $\mathcal{A}^+[1]{=}\varepsilon{::}\eta_i$
- $\alpha_i \in \{\uparrow^+, \uparrow^*\}$: ($\alpha_i^{-1}{=}\downarrow^+$ if $\alpha_i{=}\uparrow^+$ and $\downarrow^*$ otherwise)
  $prefix^{-1}(\alpha_1{::}\eta_1[p_1]/\cdots/\alpha_i{::}\eta_i[p_i]) :=$
  $\alpha_i^{-1}{::}\eta_{i-1}[\mathcal{A}^{acc}][p_{i-1}^t][prefix^{-1}(\alpha_1{::}\eta_1[p_1]/\cdots/\alpha_{i-1}{::}\eta_{i-1}[p_{i-1}])]$

The function $\mathtt{RW\_Pred}$: $\mathcal{X}^{\Uparrow} \longrightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$ is redefined to rewrite any $\mathcal{X}^{\Uparrow}$ predicate P, of the form $\alpha_1{::}\eta_1/\cdots/\alpha_k{::}\eta_k$ $(k \geq 1)$, into another one defined in the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ as follows (only the case of upward axes is considered):

- $\alpha_i = \uparrow$:
  $\mathtt{RW\_Pred}(\alpha_i{::}\eta_i/\cdots/\alpha_k{::}\eta_k) := \mathcal{A}^{\eta_i}[\mathtt{RW\_Pred}(\alpha_{i+1}{::}\eta_{i+1}/\cdots/\alpha_k{::}\eta_k)]$
- $\alpha_i \in \{\uparrow^+, \uparrow^*\}$:
  $\mathtt{RW\_Pred}(\alpha_i{::}\eta_i/\cdots/\alpha_k{::}\eta_k) := \alpha_i{::}\eta_i[\mathcal{A}^{acc}][\mathtt{RW\_Pred}(\alpha_{i+1}{::}\eta_{i+1}/\cdots/\alpha_k{::}\eta_k)]$

5.4 Theoretical Results

We present briefly some results that concern the evaluation of the overall answering time of our rewriting approach as well as its correctness.

Lemma 2. *Every $\mathcal{X}^{\Uparrow}_{[n,=]}$ query Q can be evaluated over an XML document T in time $O(|Q|.|T|)$.*

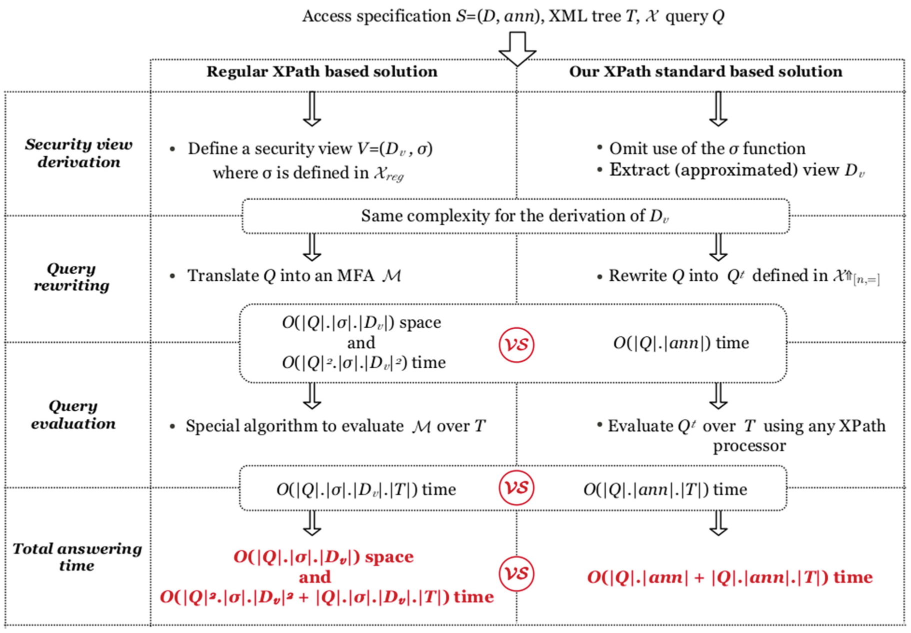
Access specification S=(D, ann), XML tree T, 𝒳 query Q
Regular XPath based solution
Our XPath standard based solution
Security view derivation
• Define a security view V=(D_v, σ) where σ is defined in 𝒳_reg
• Omit use of the σ function
• Extract (approximated) view D_v
Same complexity for the derivation of D_v
Query rewriting
• Translate Q into an MFA ℳ
• Rewrite Q into Q^t defined in 𝒳⇑[n,=]
O(|Q|.|σ|.|D_v|) space and O(|Q|².|σ|.|D_v|²) time
VS
O(|Q|.|ann|) time
Query evaluation
• Special algorithm to evaluate ℳ over T
• Evaluate Q^t over T using any XPath processor
O(|Q|.|σ|.|D_v|.|T|) time
VS
O(|Q|.|ann|.|T|) time
Total answering time
O(|Q|.|σ|.|D_v|) space and O(|Q|².|σ|.|D_v|² + |Q|.|σ|.|D_v|.|T|) time
VS
O(|Q|.|ann| + |Q|.|ann|.|T|) time

Fig. 7. Comparing our solution with that of [17].

The proof of this lemma is based on the results of the XPath query complexity analysis detailed in [48].

Theorem 2. *Given an access specification S=(D, $\texttt{ann}$), an XML tree $T \in \mathcal{T}(D)$ and its virtual view T_v computed w.r.t S. There exists an algorithm* `Rewrite` *that translates any $\mathcal{X}$ query Q over T_v into an $\mathcal{X}^{\Uparrow}_{[n,=]}$ query Q^t over T at most in time $O(|Q|)$. Moreover, Q^t can be evaluated over T at most in time $O(|Q|.|\texttt{ann}|.|T|)$.*

Theorem 3. *The query rewriting approach is correct for any query of the fragment $\mathcal{X}$.*

Theorem 3 shows the correctness of our query rewriting approach. More specifically, for any access specification S=(D,$\texttt{ann}$), any XML tree $T \in \mathcal{T}(D)$ and its virtual view T_v, our rewriting algorithm `Rewrite` translates any $\mathcal{X}$ query Q over T_v into a safe one Q^t defined over T such that: $\mathcal{S}[\![Q]\!](T_v)$=$\mathcal{S}[\![Q^t]\!](T)$.

Our algorithm `Rewrite` and the detailed proofs are given on-line at https://tel.archives-ouvertes.fr/tel-01093661/.

Finally, we make a brief comparison of our XPAth-based solution with that of [17] that is based on Regular XPath. We consider the same access specification, the same XML tree, and we show how an $\mathcal{X}$ query Q over this tree can be answered using both our solution and that of [17]. Figure 7 details the results of this comparison at each step of the XML access control processing.

6 Implementation and Experimental Study: The SVMAX Framework

We recall that our results on read-access control have been successfully extended to secure the update operations of the XQuery Update Facility [31] (see [32,33]). We have developed the SVMAX, a system that facilitates specification and enforcement of both read and update access rights for XML data. It provides general and expressive access control models that overcome limitations of existing approaches. Both of read and update rights of SVMAX are defined by annotating DTD grammars and enforced through the rewriting principle. SVMAX is well-suited to efficiently rewrite such queries and updates, and to be integrated within database systems that provide support for the W3C standards: XPath and XQuery Update Facility.

6.1 System Overview

SVMAX is composed by the following major modules: (1) a *Policy Specifier*, for the specification of read and update privileges; (2) a *View Generator*, for the generation of DTD and data views; (3) an *XPath Rewriter* [49] and (4) an *XQuery Update Rewriter* [33], for the rewriting of read and update queries respectively; (5) the *Validator* that applies an incremental validation after each update operation is performed[12]. These modules are implemented as an API allowing SVMAX to be integrated within existing native XML database systems that are aware of the XML data structure and support W3C standards.

On the other hand, SVMAX can run in standalone mode through its visual tool, SVMAX$^{\mathcal{V}}$. This latter is a GUI tool that monitors the previous modules. More precisely, SVMAX$^{\mathcal{V}}$ is used by the administrator to specify read and update policies, generate virtual views of the DTD and the XML data, and provide these views to the user. The user requests (XPath queries or XQuery update operations) are rewritten, using the adequate rewriter module, to be safely evaluated over the original XML data and then evaluation results are returned to the user. See [35] for more descriptions and screenshots of the system.

We should emphasize that in case of recursive DTDs, the DTD view generation is not always guaranteed [18] or can be of exponential size [50]. More specifically, hiding some information from the DTD may result in a context-free grammar that cannot be captured with a regular grammar[13]. In such situations, our *View generator* module generates an *approximated* DTD view. Our approximations are based on the well-known sufficient conditions for regularization of context-free grammars [51].

6.2 Performance Evaluation

In this section we present an evaluation of SVMAX. Our system is provided both as a Java API and a visual tool, the SVMAX$^{\mathcal{V}}$. Using this latter, one

[12] This is still an ongoing work: we deal only with simple kinds of DTDs and update operations, however, the global case is part of our perspective.

[13] It is undecidable in general to find a regular solution for a context-free grammar.

can choose a document DTD, specify access and update policies, and enforce these policies over underlying XML data. We focused in our experiments on the overall-time required for *rewriting* and *evaluation* of XPath queries. The study is conducted on the following aspects: (1) measure of scalability and degradation of our rewriting approaches, and (2) comparison of SVMAX with respect to naive approach in terms of overall answering time. Since our system can be integrated within existing NXDs, the other concern of experimentation is (3) a study of the integration efficiency.

(1) Scalability. We measure the time required by SVMAX to rewrite general XPath queries. We use the complex real-life recursive DTD **GedML**[14] and we generate randomly 10 access specifications by varying the number of annotations (from 20 into 200). After, we define different XPath queries of size[15] 400 that include most features of the XPath fragment $\mathcal{X}^{\Uparrow}$: with $\downarrow^*$-axis (Q_1); with $\downarrow^*$-axis and predicates (Q_2); with $\downarrow$-axis (Q_3); with $\downarrow$-axis and predicates (Q_4); with $\downarrow^*$-axis, predicates, and $*$-labels inside predicates (Q_5); with $\downarrow$-axis, predicates, and $*$-labels inside predicates (Q_6). Note that the used predicates contain different operators (e.g. $\wedge$, $\vee$, and text comparison).

Using SVMAX, we rewrite these queries according to each of the access specifications previously generated. Figure 8 shows the overall rewriting times. Notice that the rewriting time obtains a constant nature, i.e., it does not increase with the growth of the number of access annotations. This can be explained by the fact that, for an XPath query in input, our rewriter parses all its sub-queries (with the form *axis*::*label*) and rewrites them using the *accessibility predicate*. The computation time of this latter is negligible (less than 10 ms for large access specifications), and thus, our rewriting time depends basically on the parsing of the query, then on the size of the query. Since our queries have the same size, the overall rewriting time does not depend on the number of access annotations and still remains constant at some point. Moreover, we remark that in general, a query with $\downarrow^+$-axis requires more rewriting time than a query with $\downarrow$-axis (Q_1 w.r.t Q_3), also a query with predicates consumes some additional time (Q_2 w.r.t Q_1; and Q_4 w.r.t Q_3). The $*$-labels require less rewriting time (Q_2 w.r.t Q_5; and Q_4 w.r.t Q_6).

(2) Policy Enforcement. We measure the end-to-end processing time of our system for larger access specifications and general XPath queries. Since no tool exists in practice to secure querying of recursive XML views, we compare our system only w.r.t some naive approach as explained in the following.

We generate an XML document T of size 10MB that conforms to the GedML DTD, and different access specification S^i=($GedML$, ann) of size i (i=$|ann|$), where i is varying from 10 to 150. We define after a complex XPath query with

[14] Genealogy Markup Language: http://xml.coverpages.org/gedml-dtd9808.txt.

[15] The size of an XPath expression is the occurrence number of all its element types, $*$-labels, and text() functions.

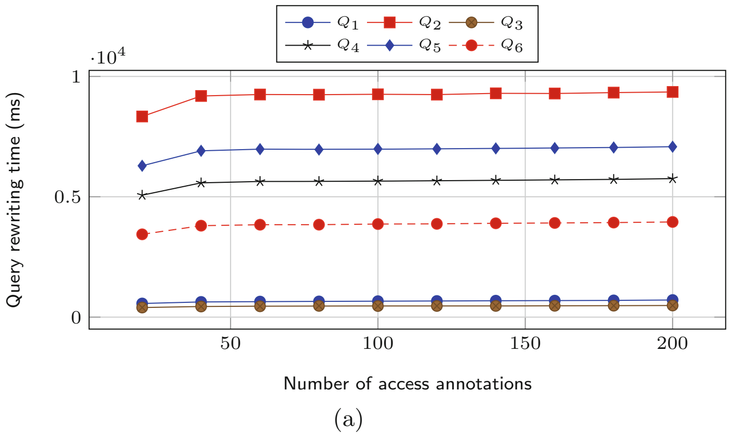

Fig. 8. SVMAX rewriting degradation for read update rights.

important size, different axes and complex predicates. This query is rewritten, w.r.t each specification S^i, both with our approach and using the on-the-fly materialization [24] as the naive approach. Figure 9 shows the answering times of each approach[16]. It is clearly shown that in case of large size of specifications and XML data, our system requires a small answering time and achieves an improvement of the naive approach by up to a factor of 10.

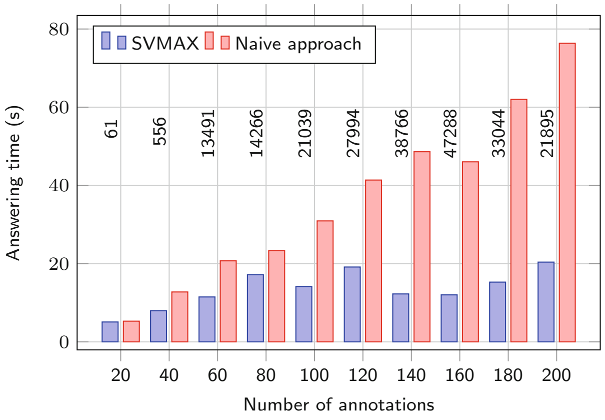

Fig. 9. Overall answering time: SVMAX versus naive approach.

(3) Integrating SVMAX Within NXDs. Finally, we use SVMAX as a simple Java API and we integrate it within different native XML databases: (1) *BaseX*, (2) *Sedna* and (3) *eXist*. The selection of these NXDs is done according to their growing use, as well as to their supports for querying and updating

[16] In the following figures, the numbers of queried nodes are depicted at the middle.

of XML data. The XPath language is supported by the three NXDs. However, only *BaseX* provides implementation for the XQuery update facility; each of the remaining systems provides a proprietary update language.

The communication between the SVMAX API and the underlying database system is ensured by using the APIs XQJ and XML:DB, present in most systems. The goal of this integration is to offer existing databases easy-to-use and efficient support to securely manipulate (recursive) XML views, as well as to leverage advantages of these systems (e.g. query optimization technologies).

We generate a simple XML document of 2 MB, a general query, and some policies P^1,...,P^{10} defined with the same principle explained in the previous subsection. Using the SVMAX rewriters, the query is safely rewritten w.r.t the different policies and sent to the underlying database for evaluation. The overall answering times (rewriting and evaluation) are depicted in Fig. 10. We remark that *eXist* database takes more time than the other (282 s for the simple policy P^1, i.e., with 20 annotations). The *BaseX* XQuery processor overcomes noticeably the *Sedna* processor in general by up to a factor of 2.

The first result of this study shows that our system has been successfully and easily integrated within such database systems. Since there are various implementation of the W3C standards, the other benefit of this study is to know with which XPath (resp. XQuery) processor the SVMAX rewritten queries may provide more efficiency.

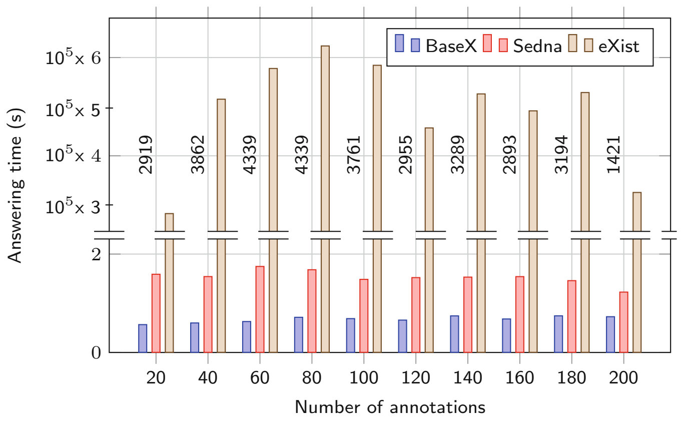

Fig. 10. Integration of SVMAX within NXDs.

7 Related Work

Figure 11 summaries the evolution of the XML access control models during the two decades. At the outset, used approaches [47,52] consisted on annotating naively the XML data with some security labels to specify which actions can be performed on which XML nodes, and thus restrict access to sensitive data through these labels. Although, some improvements [41,53] have been made in order to avoid the costly re-annotation of the data, these naive approaches are time consuming and generally difficult to apply for example in case of different users, multiple actions, and dynamic policies. Other models have been proposed [34,46] that define access policies without any labeling of data, and enforce these policies during the evaluation of users requests (read-access queries or update operations). An access policy is defined as a set of XPath expressions, each one refers to a set of XML nodes over which the user can execute some actions (read or update). The users requests are *rewritten* w.r.t the underlying access policies by adding some XPath predicates in order to execute the requested action only on authorized data (i.e. data that can be queried and/or updated). These XPath-based approaches outperform the instance-based approaches in most cases. However, the major limitation of these models is the lack of support for authorized users to access the data: the schema of accessible data is necessary for the users in order to formulate and optimize their queries; as well as for the security administrator for understanding how the authorized view of the XML data, for a group of users, will actually look like.

To overcome limitations of node-labeling protection and XPath-based protection, Stoica and Farkas [40] introduced the notion of *XML security view* that consists on defining, for each group of users, a view of the XML document that displays all and only accessible information. This notion has been refined later and used in different ways by providing each group of users with (1) a *materialized* view of accessible data; (2) a *virtual* view; or (3) a view that consists of a combination of materialized and virtual sub-views [42]. Fan et al. [25] proposed an expressive language which aims to define such security views and based on the notion of schema annotation. Roughly, the schema of the XML data is paired with a collection of XPath expressions that, when evaluated over the data, extract only accessible information. The server defines, for each group of users, such collections of XPath expressions representing users access policies. According to each access policy, the schema (e.g. a DTD) is then sanitized by eliminating information of inaccessible data, the resulted *schema view* is provided to the users who use it for formulation and optimization of their queries. While the users may query the views, they are not allowed to directly query the underlying XML data. An important issue is to answer queries posed on the views and to ensure the selective exposure of data to different classes of users.

One way to do this is to provide each group of users with a *materialized* view of all and only accessible data (as studied in [24]), which is used to evaluate users queries directly over it and offers faster access to the data. However, when the XML data and/or the access policies are changed, all users views should be (incrementally) maintained [55–58]. Note that in some cases, incremental maintenance of materialized views leads to the same performances as re-computation of the views from scratch. In addition to the maintenance cost, materialization of all users views within the server is time and memory consuming.

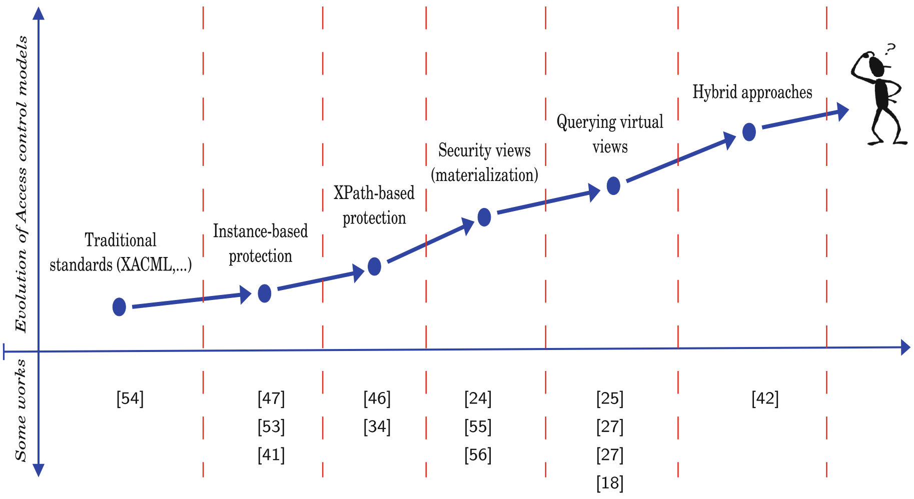

Fig. 11. Evolution of XML Access Control Models.

The *view virtualization* is the adequate and more scalable solution in case of huge data, an important number of users, and dynamic policies. Fan et al. [25] defined the notion of *query rewriting* that consists on translating queries posed over virtual views into equivalent ones to be evaluated over the original data. Since DTDs found in practice are often recursive [26], many authors have refined this work to use more expressive query language [17,18,27], namely Regular XPath. Regular XPath is more expressive than XPath and allows definition of recursive paths. The use of this language to secure XML data has been more studied in [17,18]. However, Regular XPath based solutions still a theoretical achievement and may be impractical since rewriting of Regular XPath expressions may be of exponential cost as we have shown in Sect. 5.4. In addition, Regular XPath is not commonly used in practice[17] and most of the commercial database systems (e.g. eXistdb) offer support for the W3C standards: XPath and XQuery. Thus, the securing of such queries remains a strong necessity.

8 Conclusions

We aimed to provide a practical solution for the open problem that consists on rewriting XPath queries under DTDs recursion. We have investigated the extension of the downward class of XPath with some axes and operators, and showed that the resulted XPath fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ can be used to rewrite efficiently any $\mathcal{X}$ query, over the data view, into a safe one that can be evaluated directly over the original data. Our proposal yields the first practical solution for the rewriting problem. The conducted experimentation shows the efficiency of our

[17] Note that no tool exists in practice to evaluate Regular XPath queries.

approach. Most importantly, the translation of queries from $\mathcal{X}$ to $\mathcal{X}^{\Uparrow}_{[n,=]}$ does not impact the performance of the queries answering.

Recall that a previous solution of the rewriting problem has been investigated in [27] that relies on the non-standard Regular XPath language. By the following comparison, we show that XPath-based rewriting is more efficient than the one based on Regular XPath since this later can lead to an exponential cost. Given an access specification S=(D, ann), an XML tree $T \in \mathcal{T}(D)$, let Q be an $\mathcal{X}$ query posed over the virtual view T_v of T. Whatever the type of D (recursive or not), we make possible the answering of Q over T in at most $O(|Q|.|\mathit{ann}|.|T|)$ time, while [17] do this in $O(|Q|.|\sigma|.|D_v|)$ space and $O(|Q|^2.|\sigma|.|D_v|^2 + |Q|.|\sigma|.|D_v|.|T|)$ time. We should emphasize that $|\mathit{ann}|$ is bounded by $O(|D|^2)$ (i.e. we can define at most $|D|^2$ annotations). However, the size of the function σ is, in general, larger than $O(|D|^2)$. In other words, the number of the paths presented by the function σ may be exponential on the size of the DTD as we show by the following example.

Example 14. Consider the DTD D=$(\{Root, A_1, \ldots, A_n\}, P, Root)$ where $n \in \mathbb{N}$ and the production rules are given as follows:

$$\begin{array}{ll} P(Root) & := (A_1|\cdots|A_n) \\ P(A_i) & := (A_1|\cdots|A_{i-1}|A_{i+1}|\cdots|A_n),\ i \leq n \end{array}$$

We define now the access specification S=(D, ann) where ann contains only the default annotation $\mathit{ann}(Root)$=Y, i.e. all element types of D are accessible. It is easy to prove that, for any element types A_i, A_j ($i \leq n$ and $j \leq n$), the number of paths presented by $\sigma(A_i, A_j)$ may be bounded by: $\Sigma_{1 \leq i \leq n-2} \frac{(n-2)!}{(n-2-i)!}$. □

Finally, we conclude that our rewriting approach is more efficient in practice than the one based on Regular XPath and requires an answering time that is linear on the size of the input query, the number of annotations, and the size of the XML data. This would lead for an efficient integration of our solution within some existing database systems. Moreover, by working with the XPath standard, we make possible the use of a bulk of interesting results found around the XPath language (e.g. XPath queries optimization [59,60] and efficient evaluation [61]).

References

1. Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J., Siméon, J.: Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation (2008). http://www.w3.org/TR/2008/REC-xml-20081126/
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., Cowan, J.: Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation (2006). http://www.w3.org/TR/2006/REC-xml11-20060816/
3. Amavi, J., Chabin, J., Halfeld-Ferrari, M., Réty, P.: A toolbox for conservative XML schema evolution and document adaptation. In: Decker, H., Lhotská, L., Link, S., Spies, M., Wagner, R.R. (eds.) DEXA 2014, Part I. LNCS, vol. 8644, pp. 299–307. Springer, Heidelberg (2014)

4. Chabin, J., Halfeld Ferrari, M., Musicante, M.A., Réty, P.: Conservative type extensions for XML data. In: Hameurlain, A., Küng, J., Wagner, R. (eds.) TLDKS IX. LNCS, vol. 7980, pp. 65–94. Springer, Heidelberg (2013)
5. Gerald, B., Sleeper, H., Gregorowicz, A., Dingwell, R.: hData - a simple XML framework for health data exchange. In: Proceedings of Balisage: The Markup Conference, Montral, Canada, August 11–14, 2009, vol. 3, pp. 299–307 (2009)
6. Fried, E., Geng, Y., Ullrich, S., Kneer, D., Grottke, O., Rossaint, R., Deserno, T.M., Kuhlen, T.: MEDOX: an XML-based approach of medical data organization for segmentation and simulation. In: Bildverarbeitung für die Medizin 2010 - Algorithmen - Systeme - Anwendungen, Aachen, Germany, March 14–16, 2010. CEUR Workshop Proceedings, vol. 574, 251–255. CEUR-WS.org (2010)
7. Cavalini, L.T., Cook, T.W.: Use of XML schema definition for the development of semantically interoperable healthcare applications. In: Gibbons, J., MacCaull, W. (eds.) FHIES 2013. LNCS, vol. 8315, pp. 125–145. Springer, Heidelberg (2014)
8. la Rosa Algarin, A.D., Demurjian, S.A., Berhe, S., Pavlich-Mariscal, J.A.: A security framework for XML schemas and documents for healthcare. In: 2012 IEEE International Conference on Bioinformatics and Biomedicine Workshops, BIBMW 2012, Philadelphia, USA, October 4–7, 2012, pp. 782–789. IEEE (2012)
9. Steele, R., Gardner, W., Chandra, D., Dillon, T.S.: Framework and prototype for a secure XML-based electronic health records system. IJEH **3**(2), 151–174 (2007)
10. Kumar, C.S., Govardhan, A., Rao, C.V.G.: Usage of XML technology in electronic health record for effective heterogeneous systems integration in healthcare. IJMEI **1**(4), 399–406 (2009)
11. Thuy, P.T.T., Lee, Y., Lee, S.: Semantic and structural similarities between XML schemas for integration of ubiquitous healthcare data. Pers. Ubiquit. Comput. **17**(7), 1331–1339 (2013)
12. IBM jStart team: IBM Emerging Technology's client engagement team. http://www-01.ibm.com/software/ebusiness/jstart/
13. DITA OASIS Standard: An XML architecture for designing, writing, managing, and publishing information. http://dita.xml.org/
14. ebXML consortium: Electronic Business using eXtensible Markup Language. http://www.ebxml.org/
15. Oracle White Paper: Sun Storage 7000 Unified Storage Systems and XML-Based Archiving for SAP Systems, April 2010. http://www.oracle.com/us/solutions/sap/database/ss7000-sap-implementation-guide-352637.pdf
16. 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)
17. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Rewriting regular xpath queries on XML views. In: ICDE, pp. 666–675. IEEE (2007)
18. 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)
19. Luo, B., Lee, D., Lee, W.C., Liu, P.: Qfilter: rewriting insecure XML queries to secure ones using non-deterministic finite automata. VLDB J. **20**(3), 397–415 (2011)
20. Cong, G.: Query and update through XML views. In: Bhalla, S. (ed.) DNIS 2007. LNCS, vol. 4777, pp. 81–95. Springer, Heidelberg (2007)
21. Damiani, E., Fansi, M., Gabillon, A., Marrara, S.: A general approach to securely querying XML. Comput. Stand. Interfaces **30**(6), 379–389 (2008)

22. Clark, J., DeRose, S.: XML path language (XPath) 1.0. W3C Recommendation, November 1999. http://www.w3.org/TR/xpath/
23. 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/
24. Kuper, G.M., Massacci, F., Rassadko, N.: Generalized XML security views. Int. J. Inf. Sec. **8**(3), 173–203 (2009)
25. Fan, W., Chan, C.Y., Garofalakis, M.N.: Secure XML querying with security views. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 587–598. ACM (2004)
26. Choi, B.: What are real dtds like? In: Fifth International Workshop on the Web and Databases (WebDB), pp. 43–48 (2002)
27. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: SMOQE: a system for providing secure access to XML. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 1227–1230. ACM (2006)
28. 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)
29. Wood, P.T.: Containment for XPath fragments under DTD constraints. In: Calvanese, D., Lenzerini, M., Motwani, R. (eds.) ICDT 2003. LNCS, vol. 2572, pp. 297–311. Springer, Heidelberg (2002)
30. Neven, F., Schwentick, T.: On the complexity of Xpath containment in the presence of disjunction, DTDs, and variables. Logical Methods in Computer Science 2(3) (2006)
31. Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J., Siméon, J.: Xquery update facility 1.0. W3C Recommendation, March 2011. http://www.w3.org/TR/xquery-update-10/
32. Mahfoud, H., Imine, A.: A general approach for securely updating XML data. In: Proceedings of the 15th International Workshop on the Web and Databases (WebDB 2012), pp. 55–60 (2012)
33. Mahfoud, H., Imine, A.: On securely manipulating XML data. In: Garcia-Alfaro, J., Cuppens, F., Cuppens-Boulahia, N., Miri, A., Tawbi, N. (eds.) FPS 2012. LNCS, vol. 7743, pp. 293–307. Springer, Heidelberg (2013)
34. Fundulaki, I., Maneth, S.: Formalizing XML access control for update operations. In: SACMAT, pp. 169–174. ACM (2007)
35. Mahfoud, H., Imine, A., Rusinowitch, M.: SVMAX: a system for secure and valid manipulation of XML data. In: Proceedings of the 17th International Database Engineering & Applications Symposium (IDEAS), pp. 154–161. ACM (2013)
36. Jia, X.: From Relations to XML: Cleaning, Integrating and Securing Data. Doctor of philosophy, Laboratory for Foundations of Computer Science. School of Informatics. University of Edinburgh (2007)
37. Fan, W., Yu, J.X., Li, J., Ding, B., Qin, L.: Query translation from XPath to SQL in the presence of recursive dtds. VLDB J. **18**(4), 857–883 (2009)
38. Krishnamurthy, R., Chakaravarthy, V.T., Kaushik, R., Naughton, J.F.: Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In: Proceedings of the 20th International Conference on Data Engineering (ICDE 2004), pp. 42–53. IEEE Computer Society (2004)
39. ten Cate, B.: The expressivity of XPath with transitive closure. In: Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2006), pp. 328–337. ACM (2006)

40. Stoica, A., Farkas, C.: Secure XML views. In: Research Directions in Data and Applications Security, IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security. IFIP Conference Proceedings, vol. 256, pp. 133–146. Kluwer (2002)
41. Duong, M., Zhang, Y.: An integrated access control for securely querying and updating XML data. In: Proceedings of the Nineteenth Australasian Database Conference (ADC). CRPIT, vol. 75, pp. 75–83. Australian Computer Society (2008)
42. Thimma, M., Tsui, T.K., Luo, B.: HyXAC: a hybrid approach for XML access control. In: 18th ACM Symposium on Access Control Models and Technologies (SACMAT), ACM (2013)
43. Fegaras, L.: Incremental maintenance of materialized XML views. In: Hameurlain, A., Liddle, S.W., Schewe, K.-D., Zhou, X. (eds.) DEXA 2011, Part II. LNCS, vol. 6861, pp. 17–32. Springer, Heidelberg (2011)
44. Shastry, P.D.N.M.: Integrated Healthcare IHE Pathway for the Patients: Patient Treatment Lifecycle Management (PTLM). Radiology Clinic, United Kingdom (2000). (October 2012) http://www.clinrad.nhs.uk/
45. Samarati, P., di Vimercati, S.C.: Access control: policies, models, and mechanisms. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 137–146. Springer, Heidelberg (2001)
46. Fundulaki, I., Marx, M.: Specifying access control policies for XML documents with XPath. In: SACMAT 2004, 9th ACM Symposium on Access Control Models and Technologies, pp. 61–69, ACM (2004)
47. Murata, M., Tozawa, A., Kudo, M., Hada, S.: XML access control using static analysis. ACM Trans. Inf. Syst. Secur. **9**(3), 292–324 (2006)
48. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Trans. Database Syst. **30**(2), 444–491 (2005)
49. Mahfoud, H., Imine, A.: Secure querying of recursive XML views: a standard XPath-based technique. In: WWW (Companion Volume), pp. 575–576. ACM (2012)
50. Kuper, G.M., Massacci, F., Rassadko, N.: Generalized XML security views. In: 10th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 77–84. ACM (2005)
51. Andrei, S., Chin, W.N., Cavadini, S.V.: Self-embedded context-free grammars with regular counterparts. Acta Inf. **40**(5), 349–365 (2004)
52. Murata, M., Tozawa, A., Kudo, M., Hada, S.: XML access control using static analysis. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), pp. 73–84. ACM (2003)
53. Duong, M., Zhang, Y.: Dynamic labelling scheme for XML data processing. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part II. LNCS, vol. 5332, pp. 1183–1199. Springer, Heidelberg (2008)
54. Oasis extensible access control markup language (XACML) TC, January 3013. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml
55. Bonifati, A., Goodfellow, M.H., Manolescu, I., Sileo, D.: Algebraic incremental maintenance of XML views. In: 14th International Conference on Extending Database Technology (EDBT), pp. 177–188. ACM (2011)
56. Nica, A.: Incremental maintenance of materialized views with outerjoins. Inf. Syst. **37**(5), 430–442 (2012)
57. Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Eng. Bull. **18**(2), 3–18 (1995)
58. Gupta, A., Mumick, I.S., Rao, J., Ross, K.A.: Adapting materialized views after redefinitions: techniques and a performance study. Inf. Syst. **26**(5), 323–362 (2001)

59. Maneth, S., Nguyen, K.: XPath whole query optimization. PVLDB **3**(1), 882–893 (2010)
60. Georgiadis, H., Charalambides, M., Vassalos, V.: A query optimization assistant for XPath. In: Proceedings of the 14th International Conference on Extending Database Technology (EDBT 2011), ACM (2011)
61. Hsu, W.C., Liao, I.E.: CIS-X: a compacted indexing scheme for efficient query evaluation of XML documents. Inf. Sci. **241**, 195–211 (2013)