Efficient Access Control to XML Data: Querying and Updating Problems

Houari Mahfoud

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Contrôle d'Accès Efficace pour des Données XML: problèmes d'interrogation et de mise-à-jour

∴ ∴ ∴

Efficient Access Control to XML Data: Querying and Updating Problems

THÈSE

présentée et soutenue publiquement le 18 Février 2014

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Houari Mahfoud

Composition du jury

Président :	Ye-Qiong Song	Université de Lorraine
Rapporteurs :	Mírian Halfeld Ferrari Alves	Université d'Orléans
	Nabil Layaida	INRIA Rhône-Alpes
Examinateur :	Nora Cuppens-Boulahia	Telecom Bretagne
Directeurs de thèse :	Abdessamad Imine	Université de Lorraine
	Michaël Rusinowitch	INRIA Nancy

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Acknowledgments

First of all, I owe a special and warm thank-you to Imine Abdessamad for giving me the opportunity to do this PhD with him. His high competence combined with his human qualities provided me a very pleasant and valuable supervision. I thank him for his great and constant availability, for his clear and precise answers to my questions, for his numerous advices and comments, for his constant human support during this adventure.

I thank Michaël Rusinowitch who warmly welcomed me in his team and taught me how to do research. His remarkable efforts, his indescribable vision of human analysis, encouraging statements in pessimistic situations, vital experience and moral support enabled me to look forward and face the challenges with confidence. He encouraged me to aim high and provided me with invaluable help. Without him, this thesis would not have been possible.

Then I want to thank Mírian Halfeld Ferrari and Nabil Layaida for accepting to review my thesis, for their detailed reviews and for their relevant remarks. I also thank Nora Cuppens-Boulahia and Ye-Qiong Song for the honor of having them in my jury.

Finally, I thank all the members of CASSIS team, special thank to Walid Belkhir, and a big thank-you to everyone who helped me with this thesis.

...to my family.

Table of contents

List of Figures

1

Introduction

Contents

1.1 General Context

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 are managed via a large number of data models and access techniques, and may be combined from several heterogeneous systems based on different data formats. Having proprietary data formats was sufficient for managing and communicating data across a small number of partners; but this model was not scalable. Enterprises and different companies need a neutral and flexible way for exchanging data among different devices, systems, and applications. Data need to be maintained in a self-describing format to accommodate a variety of ever-evolving business needs. This has led communities (mostly scientific, but not only) to search for a highly standardized common data format for data exchange between applications. The solution to this problem came with the advent of XML 1.0 [RCD+08] and XML 1.1 [BPSM+06].

The *eXtensible Markup Language* (XML) is a W3C recommendation that encodes text and data in a format which can be processed easily and exchanged across multiple platforms. An XML document consists of nested elements that collectively form a tree. The content of an XML element is a sequence of elements and text. Moreover, a list of attributes can be attached to an element to represent supplementary information. A simple XML document is shown in Figure. 1.1. This document represents a catalog of two books, each one is described by a list of authors, the title, the publication year, and its series. It is increasingly common to find the XML format used to define the concepts of theoretical computer science (e.g. automata [MLMK05], graph [FGMP13], logic [Bet08], programming language [FGK03]).

```
<catalog>
  <book id="VS01">
    <authors>
      <author firstname="Véronique" lastname="Cortier"/>
      <author firstname="Steve" lastname="Kremer"/>
    </authors>
    <title>Formal Models and Techniques for Analyzing Security Protocols</title>
    <series>Cryptology and Information Security Series</series>
    <publication_year>2011</publication_year>
  </book>
  <book id="WF01">
    <authors>
      <author firstname="Wenfei" lastname="Fan"/>
      <author firstname="Floris" lastname="Geerts"/>
    </authors>
    <title>Foundations of Data Quality Management</title>
    <series>Synthesis Lectures on Data Management</series>
    <publication_year>2012</publication_year>
  </book>
</catalog>
```

Figure 1.1: An XML document example.

1.1.1 Benefits and Use Cases of XML

XML has enjoyed considerable popularity and has been universally received as the de facto standard for representing and exchanging data. XML brings a number of powerful capabilities to information modeling:

1. *Heterogeneity*: where each record can contain different data fields. The real world is not neatly organized into tables, rows, and columns. There is great advantage in being able to express information, as it exists, without restrictions.
2. *Extensibility*: Where new types of data can be added at will and do not need to be determined in advance. This allows us to embrace, rather than avoid, change.
3. *Flexibility*: Where data fields can vary in size and configuration from instance to instance. XML imposes no restrictions on data; each data element can be as long or as short as necessary. XML is also self-describing; applications can use this feature to automatically build themselves with little programming required.

We refer in the following to different real-life cases where XML plays an important role:

- *Modeling systems components*: Companies such as *TIBCO*, *IBM*[1], *Oracle*[2] and *Microsoft* offer frameworks for building applications, with a minimum of effort, where XML is used as a universal information-structuring tool.

[1] `http://www-01.ibm.com/software/websphere/`.
[2] `http://www.oracle.com/us/solutions/sap/database/ss7000-sap-implementation-guide-352637.pdf`.

- *XML-based technologies*: XML has emerged as a critical enabler to various technology initiatives. Service-oriented architectures (*SOA*), enterprise application integration (*EAI*), enterprise information integration (*EII*), web services, and standardization efforts in many industries all rely on or make use of XML as an underlying technology.

- *XML-based business solutions*: Many solutions exist today that rely on XML to address business needs. For instance, the solutions proposed by the IBM's jStart team[3] and the DITA OASIS Standard[4] XML architecture for designing, writing, managing, and publishing information.

- *Desktop applications*: *OpenOffice* files, *Ant's Build* files, *Microsoft Visual Studio .Net* project files, and *Mac plist* configuration files are all written in XML format.

- *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, *GML* and *KML* for geography, *SVG* for graphics, *SCORM* for e-learning, and *NLM-DTD*, *ODT* and *OOXML* for documents.

As XML becomes more critical to the operations of an enterprise, it becomes an asset that needs to be shared, searched, secured, maintained, and integrated with other data.

1.1.2 Managing XML Data

Many business database systems have proposed supports that are aware of the XML data structure and offer efficient management. It is necessary to note that the area of XML databases is relatively new and rapidly evolving. As a consequence, the boundaries between different types of XML databases are not clear. This is due to the fact that it is seldom easy to classify XML applications as *data-centric* or *document-centric*. Thus, the distinction between these two broad application areas of XML technologies must be understood in order to identify the type of the XML database that is more suitable for a particular application.

Data-centric XML applications produce and use XML documents that mark up highly structured information. They are typically generated by machines, designed for machine consumption, and generally used to communicate between companies and applications. Examples of such *data-centric XML documents* are sales orders, flight schedules, scientific data, financial transaction information, programming language data structures. Data-centric documents are characterized by fairly regular structure, fine-grained data (that is, the smallest independent unit of data is at the level of a PCDATA-only, an element, or an attribute), and little or no mixed content [5]. The order in which sibling elements and PCDATA occurs is generally not significant, except when validating the document. Data-centric documents can originate both in the database (in which case we want to expose it as XML) and outside the database (in which case we want to store it in a database).

[3] `http://www-01.ibm.com/software/ebusiness/jstart/`
[4] `http://dita.xml.org/`.
[5] Mixed content refers to an XML element that has both element and text nodes as child nodes.

Example 1.1. Consider the following XML document:

```
<Meetings>
  <Meeting date="27/05/2013" time="10:30AM">
    <Member name="Michael Rusinowitch"/>
    <Member name="Houari Mahfoud"/>
    <Subject>
      Discuss the redaction plan
    </Subject>
    <Room>
      A209
    </Room>
  </Meeting>
</Meetings>
```

This document is clearly data-centric: it has rigid structure, fine-grained data and contains no mixed content. An illustrative example of data-centric documents are web pages on Amazon.com that display information about a book. Although the page is largely text, the structure of that text is highly regular, much of it is common to all pages describing books, and each piece of page-specific text is limited in size. Thus, the page could be constructed, from a database, as a data-centric XML document that contains the information about a single book. In general, any Web site that dynamically constructs HTML documents today by filling a template with database data can probably be replaced by a series of data-centric XML documents and one or more XSL stylesheets. □

Document-centric XML applications produce and use XML documents that are designed for human consumption. Examples of such *document-centric XML documents* are books, email, advertisements, technical manuals, legal documents, product catalogs, and almost any hand-written XHTML document. These documents are either without structure, with a variable structure (the schema evolves over the time), or with a fixed structure but the application does not take advantage of this structure. They are characterized also by larger grained data (that is, the smallest independent data unit might be at the level of an element or the entire document itself) and lot of mixed content (**semi-structured**). The order in which sibling elements and PCDATA occurs is almost always significant. Document-centric documents are usually written by hand in XML or some other format (e.g. RTF, PDF, or TEX) which is then converted to XML. Unlike data-centric documents, they usually do not originate in the database.

Example 1.2. The following XML document is document-centric:

```
<Meetings>
  <Meeting>
    Please can <Member name="Michael Rusinowitch"/> and <Member name="Houari Mahfoud"/>
    come to <Room>A209</Room> on <MeetingDate>27/05/2013</MeetingDate> at
    <MeetingTime>10:30AM</MeetingTime> to
    <Subject>discuss the redaction plan</Subject>
  </Meeting>
</Meetings>
```

□

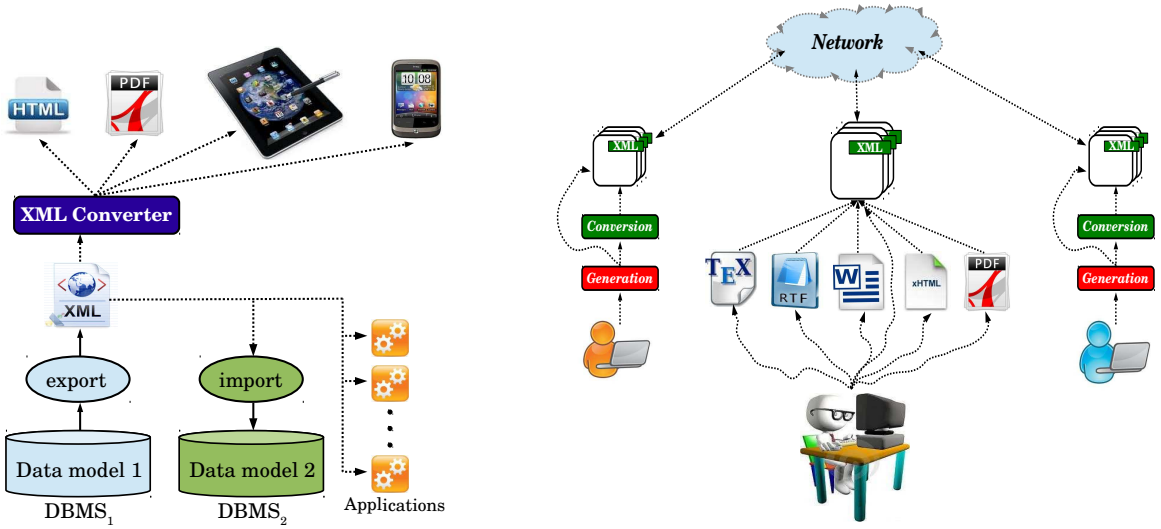


Figure 1.2: The architecture of (a) Data-centric and (b) Document-centric XML applications.

The general architecture of data-centric and document-centric XML applications is depicted in Figure 1.2. Note that characterizing the XML documents of an application as data-centric or document-centric will usually suffice to decide the type of the suitable database to use.

Originally, existing relational database systems were adapted to manipulate XML data, along with their traditional well-established technologies. XML was used only as a medium to transport data between partners. In this way, XML data was stored as relational tables. More precisely, a special *XML-enabling layer* was used to decompose XML data into tables (the *shredding* process) and store it within underlying database; and as well as to extract data from relational tables and exchange it in XML format (the *publishing process*). The most widely used language is SQL/XML[6] which extends SQL with some special functions for shredding and publishing of XML data. This kind of databases, called *XML-enabled databases* (XEDs), quickly became for a long time the de facto solution for deploying data-centric applications that managed large volumes of data and either wanted to be able to communicate with other businesses or to expose their data on the web. Examples of such systems are: *MySQL* and *PostgreSQL*.

The XEDs were somewhat limited: they were not capable for supporting huge XML documents, as shredding and publishing process require a high number of join operations. Therefore, they lead to unacceptable performance, both in retrieving documents and in querying them. This technology gap was covered by *native XML databases* (NXDs), that is database systems specifically developed for storing XML documents and commonly used by document-centric applications. They define an *XML data model* (e.g. XPath, DOM, and XML Infoset models), and store and retrieve XML data according to that model. An XML data model must preserve, in a much more efficient manner, the hierarchical form of the document by supporting at minimum elements, attributes, text, and document order. The most used NXDs are: *BaseX*, *Sedna*, and *eXist-db*. Most of the existing NXDs provide support for the W3C standards (e.g. XPath [BBC+10], XQuery [BCFF+10]) for querying and updating XML data.

[6] `http://sqlxml.org/`.

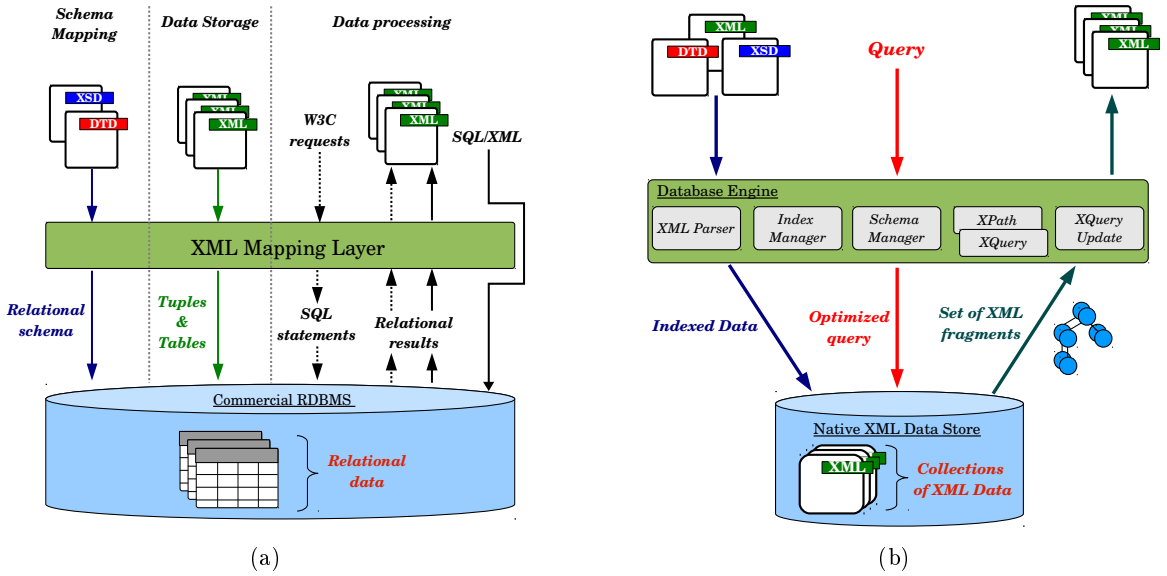


Figure 1.3: The architecture of (a) XEDs and (b) NXDs.

Figure 1.3 depicts the overall architecture of XEDs and NXDs. The need to combine the features of both native and XML-enabled databases brought on a new category of databases called *Hybrid XML Databases* (HXDs). A hybrid XML database provides XML data management in both native and XML-enabled fashion. Most of the nowadays' RDBM (e.g. *IBM DB2* and *Oracle*) are hybrid and allow, depending on the applications requirements, shredding and publishing of the XML document to and from relational tables as well as native storage. These systems present a significant advantage where interoperability between XML and other data is needed.

The W3C and some other organizations offered solutions for better manipulation of XML data: DTD, RELAX NG, XML Schema, XPath, XQuery, XUpdate. Some commercial database systems developed their own proprietary XML languages (e.g. those developed by *Sedna* and *eXist-db* for updating XML). In this dissertation, only the W3C standards and solutions are investigated which are widely used both in practice and in theory, and around them interesting results are found (e.g. W3C XML query optimization [MN10a], containment [tCL09], evaluation [BK08, GKPS05a], implementation and integration [LCS12, Wei11]).

1.1.3 Protecting XML Data

XML databases are widely used in various domains: business environments, social networks, biological and health-care systems. This has significantly increased the need for more efficient management of XML data. Consider day to day operations consist on querying and/or updating XML data. These tasks 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 "*Personal Medical Folder*" (DMP[7]), 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, treatment, analysis, medical and surgical history). 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. The patient can modify some parts of his DMP as personal information or adding some information that seems useful and pertinent. Due to the sensitive nature of this data, a security policy is applied that controls access to different parts of the DMPs. For instance, grant to an insurance company a read access that concerns only medication information; and an update access to the nurses that allows insertion and deletion of treatment results.

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 and/or update the same data. For these user groups, different read and update privileges may be imposed, specifying what parts of the data are accessible and/or updatable by the users. The problem of secure XML access is to enforce these privileges. More precisely, the goal is to ensure that the evaluation of any read query returns only data the users are allowed to access; as well as the execution of any update query makes changes of only data the users are allowed to update.

The well-established security specification and enforcement approaches of relational databases cannot be easily adapted for XML databases. This can be explained by the following facts:

- *Schemaless*: Unlike relational tables where the structure is known ahead of time, XML documents do not necessarily have a schema.

- *Node relationship*: While relational tables exist as standalone entities, an XML node depends to its ancestors, and its children are dependent on the node itself. This can be meaningful, for XML but not for relational context, where a node is required to be inaccessible if one of its ancestor is concerned by a deny access. Moreover, in case of ordered XML documents, each node depends to its right/left sibling.

- *Hierarchical structure*: It is useful in some cases to require a given node to be accessible only if some conditions (defined as predicates) are satisfied at this node. A generalization of this meaning consists to grant or deny access to some parts of the document.

Consequently, the problem of secure access to XML data has its own particular flavor and requires specific solutions. An XML security model should ideally provide expressiveness as well as efficiency by preserving, according to the application requirements, confidentiality and/or integrity of data. It must support:

1. An easy to write and powerful fine grained authorization mechanism that can control access to both content and structure (e.g. restricting access to entire subtrees or specific elements in the document based on their content or location).

2. An efficient mechanism for the enforcement of security policy (i.e. efficiently determine whether the access to an element or attribute is granted or denied).

[7] That refers in French to "*Dossier Médical Personnel*".

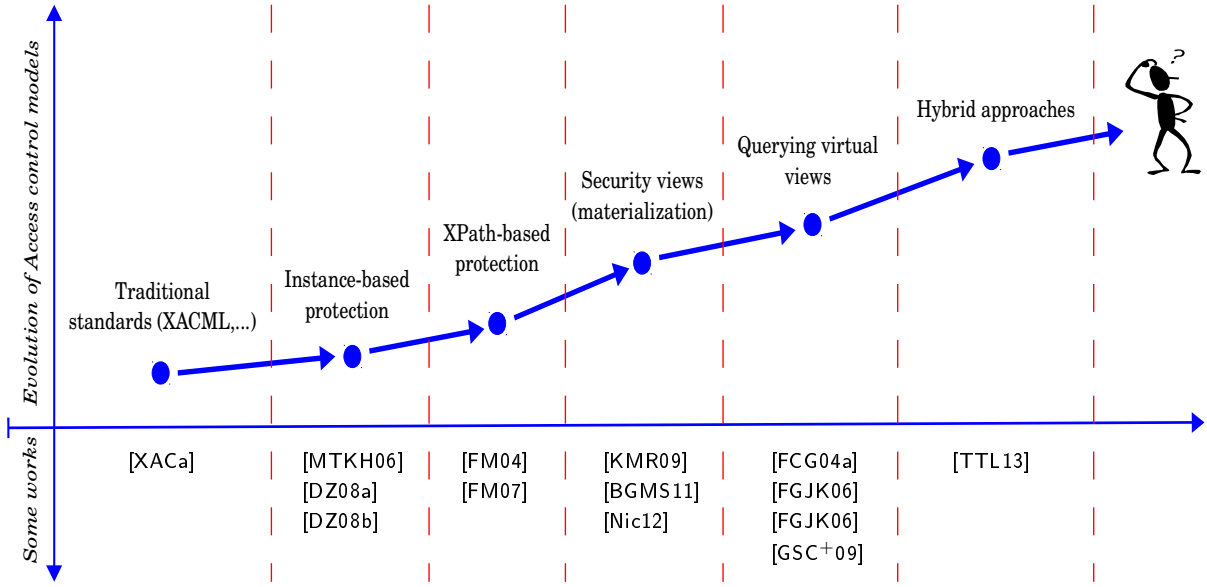


Figure 1.4: Evolution of XML Access Control Models.

3. Schema information that characterizes, for each user, all and only accessible data.

XML access control has emerged as one of the main challenges for computer security. This is not a trivial subject as can be seen with the large number of different approaches suggested in the literature [SF02a, MTKH03, FM04, FCG04a, KMR09, BCFS12, TTL13].

1.2 Motivation

Several access control mechanisms have been proposed to mitigate the risks of unauthorized access to resources, which could jeopardize the secrecy of sensitive data and cause loss of competitive advantages. In general, an access control model specifies and enforces policies that define which users, group of users, or role, can have access to which data, to perform which kinds of operations. Implementing such policies increases the exchange of data between different organizations and allows preserving confidentiality and integrity. Recently proposed approaches are by no means a panacea of all security issues; but the adequate approach is chosen according to the context and the users requirements. They involve varying degrees of complexity, use different languages (e.g. XPath, XQuery, XUpdate), and take several forms: annotation of the data with security labels, use of XPath expressions to protect access to data, extract materialized/virtual view for each group of users, etc.

Figure 1.4 summaries the evolution of the XML access control models during the two decades. A little more detail of this evolution is discussed in Chapter 2. Firstly, numerous methods, protocols and standards (e.g. ACL, SAML [SAM], OAuth [OAu], XACL [HK], XACML [XACa]) have been proposed (most of them are in active development) that provide efficient manners to specify, enforce, and (possibly) exchange access rights for different types of resources (e.g. system files, database tables, Web pages). The ACL and XACML are widely used in practice for access control purpose, however, they still remain slightly limited in protecting hierarchical data. Different works have been conducted in order to design access control models specific for

XML data. At the outset, used approaches [MTKH03,MTKH06] 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 [DZ08a, DZ08b] 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.

Some models have been proposed [FM04,FM07] 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[8] 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.

1.2.1 View-Based Security Models

To overcome the above limitations, Stoica and Farkas [SF02a] 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 subviews. Fan et al. [FCG04a] proposed an expressive language which aims to define such security views and based on the notion of schema annotation[9]. 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 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 [KMR05, KMR09]), 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 [GM95, GMRR01, BGMS11, Nic12]. 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. Consider the third most viewed website at the world, YouTube, that is accessed by a million of people per day. Since different users may have different interests (e.g. special kind of music, documentaries, favorite channels), the server may wish to provide a personalized view of the content to its users, and allow users to search such views. In such case, it is wasteful

[8] Schema of updatable data will be discussed in Chapter 5.

[9] When it is clear from the context, we refer by schema both to XML schemas [XML] and DTD grammars.

to materialize all user views because there are many users (more than 300 million of YouTube accounts are created) and their content is often overlapping, which could lead to data duplication and its associated space-overhead. For this reason, the server may keep the viewing history of the user, that reflects his area of interest, and suggest on-the-fly some links of videos that may interest this user [ZLC+12].

The *view virtualization* is the adequate and more scalable solution in case of huge data, an important number of users, and dynamic policies. Materialized and virtual views can differ in one very significant aspect, the *lifetime*: materialized views could live for a long time, while virtual views are necessarily *temporary* and could live only the time the user is connected to the server. A virtual view does not contain necessarily all the accessible data stored within the server, but it gives only some information that helps the users to query the underlying data. Consider the case of XML data, a materialized XML view contains all the accessible data and is often stored in a format that allows XML querying (e.g. DOM API). However, a virtual XML view provides to the user some information that reflects how the actual accessible data look like. Virtual XML views are often provided in text or HTML format that displays a bulk of nested accessible elements where relationships are not explicitly materialized. These formats require less storage cost but cannot be queried using XML technologies. With this comes the need to answer queries posed on virtual views. Fan et al. [FCG04a] 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. Many authors have refined this work by defining different types of policies [Ras06,KMR09], or by using more expressive query language [FGJK06, FGJK07, GSC+09].

Although virtualization of views gives in most case more advantages than the materialization, protecting data over virtual views is not always straightforward and becomes challenging in some case (e.g. over complex schemas and/or policies, rewritten requests may be complex, very large, and take more evaluation time). During the redaction of this dissertation, a new hybrid approach has been proposed [TTL13] that defines both materialized and virtual parts of the same XML view in order to benefit from the advantages of each approach.

1.2.2 Addressed Issues

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*). Due to their benefits, virtual XML views have received more attention from the research community. There has been a substantial amount of work on studying typical problems raised from the use of such views: querying [Ras06,FGJK07,GSC+09,LLLL11], querying and updating [Con07,DFGM08], consistency of update policies [BCFS12], view update translation [SBG10,BCG+11,LLHY13], integration with relational data [FYL+09]. This dissertation summarizes our results on developing approaches that provide safe and efficient manipulation of XML data, namely *querying*, *updating*, and *validation* of data. We have focused principally on shortcomings of security-view-based approaches, briefly presented in the following, and investigated some solutions to overcome them. This thesis is thus a continuation to the important effort done during the two decades to design and implement XML access control models.

Read Access Control Models Different approaches exist that use virtual views to protect access to XML data. 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 view T_v is extracted

that displays only accessible parts of T. For each user 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 that differ on:

- The query language used (e.g. the XPath standard or some variants of it [Mar04]);
- The class of queries supported (e.g. downward axes, upward axes);
- The type of the schema considered (XML schema, DTD grammar, recursive, non-recursive, normalized[10]);
- The type of the read-access policies that can be defined (e.g. top-down policies, bottom-up policies [KMR09]);
- The rewriting manner (schema-only-based rewriting [FCG04a,GSC+09,Ras06], automaton-based rewriting [DFGM08, LLLL11]).

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 schemas [11]. We investigate in the rest of this dissertation the use of only 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 [Ged] and BIOML [BIO]. The study done in [Cho02] shown that most of the real-world DTDs are recursive. We present in the following our Hospital DTD extracted from a real-life case of medical data [Sha12], and that is used throughout this dissertation.

Example 1.3. Figure 1.5 is a graph representation of our Hospital DTD. We use '$*$' on an edge to indicate a list, '?' to indicate an optional edge, while dashed edges represent disjunction. A hospital document conforming to this recursive DTD consists of a list of departments, each department (defined by its *name*) has a list of children representing patients currently residing in the hospital. For each *patient*, the hospital maintains her name (*pname*) and ward number (*wardNo*), a family medical history by means of the recursively defined *parent* and *sibling*, 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 intervention *date*, 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[12]. □

An instance of the Hospital DTD is given in Figure 1.6. We use the notation X_i to distinguish between different instances of element type X (e.g. $patient_1$, $department_2$). Due to space limitation, the content of some nodes is not depicted and is simply abbreviated by "..." (e.g. $symptom_1$, $intervention_1$). 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

[10] Definition is given in Chapter 3

[11] A schema is *recursive* if there is an element that is defined (in)directly in terms of itself.

[12] According to [Sha12], this may happen when the required treatment is outside the area of expertise of the current responsible doctor.

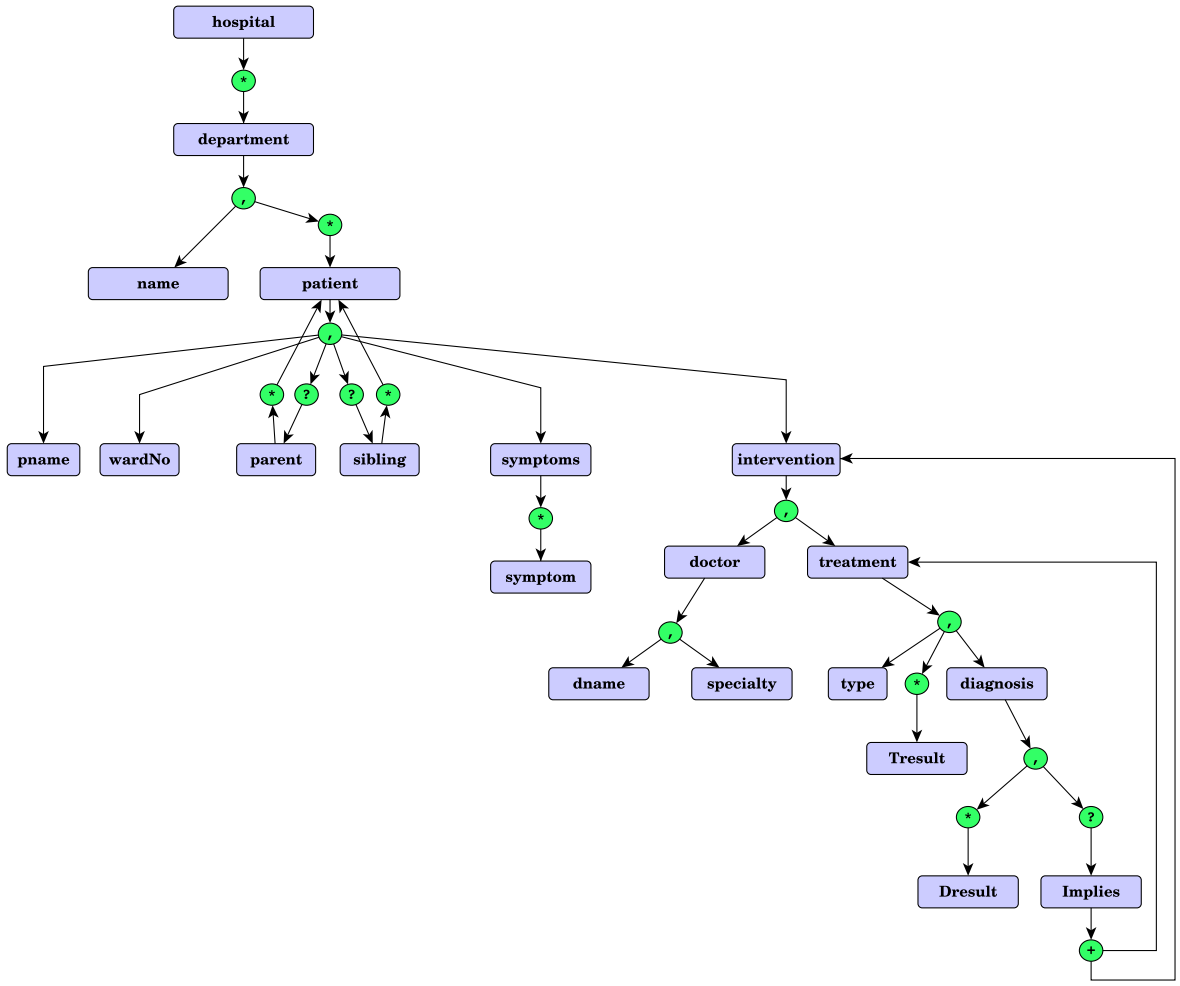


Figure 1.5: Example of Hospital DTD.

and residing at the ward *421*. According to this policy, a view is extracted and depicted in Figure 1.7, it displays all and only the data the nurse is granted access to. All the patients of the *ENT* department are hidden (i.e. $patient_6$). Moreover, all the data that belong to the *critical care* department are accessible except those that concern $patient_2$ (residing in the ward *318*). Intuitively, the node $patient_2$ and all its subordinate nodes ($pname_2$, $wardNo_2$, $parent_2$, $sibling_2$, $symptoms_2$, $intervention_4$) are hidden, and thus the nodes $patient_3$ and $patient_4$ appear as immediate children of the node $parent_1$ as can be seen in the nurse view of Figure 1.7. Any user query should be rewritten to return only accessible data. For instance, the XPath query $descendant::department[name="critical\ care"]/descendant::patient$ over the nurse view may be rewritten into $descendant::department[name="critical\ care"]/descendant::$patient$[wardNo="421"]$ [13] that, when evaluated over the original data of Figure 1.6, returns only the accessible nodes: $patient_1$, $patient_3$, $patient_4$, and $patient_5$.

In case of recursive DTDs however, the interaction between recursion in XPath queries (use of the *descendant* axis) and recursion in DTDs makes the rewriting process over virtual views more challenging. More specifically, for two *accessible* nodes A and B, there

[13] See Chapter 3 for more details of the class of XPath queries that we consider.

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. Each (sub)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. Consider the XPath query $descendant::patient[pname="Henry"]/parent/patient$ defined over the nurse view and that should return only the nodes $patient_3$ and $patient_4$. Directly evaluate this query over the original data returns the inaccessible node $patient_2$. To overcome this disclosure, we may rewrite each *patient* element in the query into $patient[wardNo="421"]$. The resulted query $descendant::patient[pname="Henry"]/parent/patient[wardNo="421"]$ returns no node over the original data. This is due to the fact that the nodes $patient_3$ and $patient_4$ are immediate children of the node $parent_1$ at the view, while some inaccessible nodes separate them from $parent_1$ at the original data. Rewriting the child relationship $parent/patient$ into $parent/descendant::patient$ is not the adequate solution: the resulted query $descendant::patient[pname="Henry"]/parent/descendant::patient[wardNo="421"]$ over the original data returns the nodes $patient_3$, $patient_4$, but also $patient_5$ that is not an immediate child of $parent_1$ at the data view. Roughly, to rewrite a (sub)query A/B it remains to find all the inaccessible paths [14] that connect accessible nodes A and B at the original data. In case of recursion, these paths may lead to an infinite set which cannot be expressed with the standard XPath.

For this reason, Fan et al. [FGJK06,FGJK07] proposed, as extension of their previous work [FCG04a], the first algorithm for coping with recursive security views. Their algorithm has been refined later by Groz et al. [GSC+09] 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 [Mar04] that is more expressive than the standard XPath and offers possibility to define recursive paths by means of the *Kleene star* operator "*". For instance, the previous query may be rewritten in Regular XPath into:

$$descendant::patient[pname="Henry"]/parent/(inaccessiblePaths)*/patient[wardNo="421"]$$

where:

$$inaccessiblePaths = patient[not\ (wardNo="421")]/(self::parent\ union\ self::sibling)$$

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 (as well as XQuery [BCFF+10]) to manipulate XML data. We emphasize that our first motivation at the outset was to develop some *practical* solutions that can be easily and efficiently integrated within existing systems that provide support for managing XML data.

Update Access Control Models Most of the native XML databases provide support for updating XML data either by using a standard language (e.g. the XQuery Update Facility [RCD+11]) or by introducing a proprietary language (e.g. the XQuery Update Extension provided by *eXist-db* [15], the Sedna update language[16]). The update primitives supported (e.g. *insert*, *delete*, *rename*, *replace*) have almost the same semantic but defined with different syntaxes. Within

[14] Paths composed by only inaccessible nodes.
[15] The *eXist-db* update language: `http://exist-db.org/exist/apps/doc/update_ext.xml`.
[16] The *Sedna* update language: `http://www.sedna.org/progguide/ProgGuidesu6.html`.

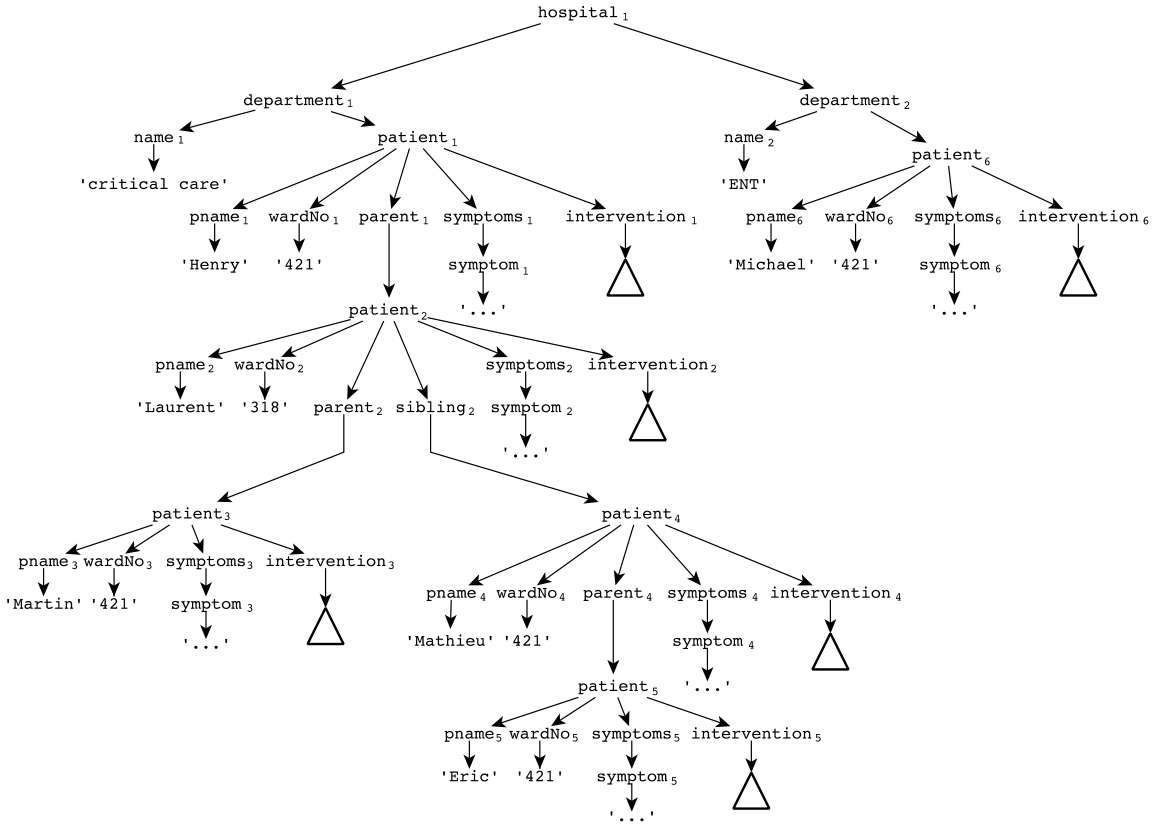


Figure 1.6: Example of Hospital data.

XML-enabled databases however, XML updates are supported either by using some SQL/XML functions (the updates are applied directly over the relational tables representing the XML data) or through the use of the XQuery Update Facility [LCS12] (the XML data is extracted from one or many relational tables, updated, and then shredded and stored back to the database). With this widespread use of XML update languages comes the need of a general access control model that would be on duty to specify which group of users, can perform which kind of updates, to modify which parts of an XML document. The requirement to impose such update rights is often encountered with collaborative editing. For instance, we consider the *Confluence* tool[17] that allows different teams to create, share and simultaneously edit different wiki pages. For each page, the administrator can make its edition granted only to some users. The update access control model supported is limited in the sens that the granularity is defined in terms of pages only and it is not possible to define update rights for section, subsection, paragraph, etc.

In this context, our motivation was to propose an expressive and fine-grained model to specify and enforce XML update policies. This problem has not received more attention since most of existing XML access control models deal only with read-access rights. Furthermore, only a few works [FM07, DFGM08, DZ08b] have tackled the control of XML update operations. In what follows, we show briefly the major limitations of these models.

[17] The *Confluence* tool: `http://www.atlassian.com/software/confluence`.

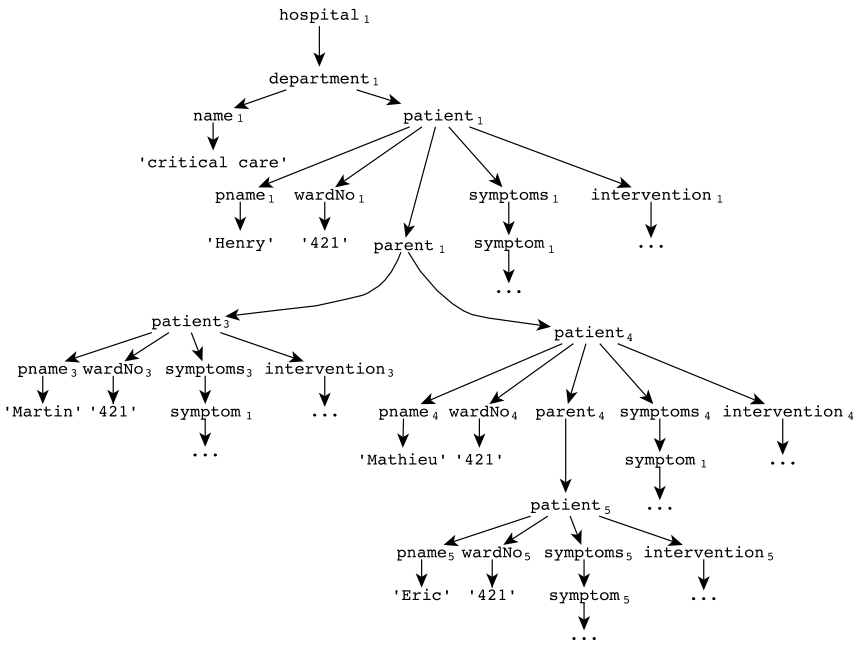


Figure 1.7: Example of data view.

Example 1.4. Due to space limitation, the different interventions done for $patient_1$ of Figure 1.6 are depicted in Figure 1.8. Suppose that the hospital wants to impose an update policy that authorizes each doctor to update only the details of the treatments she has done. For instance, the doctor *Imine* could update the data of $treatment_1$, $treatment_2$, and $treatment_4$ (e.g. add new diagnosis results into $diagnosis_4$, i.e. insert new *Dresult* sub-tree under $diagnosis_4$; change the result of $treatment_4$, i.e. update the text content of $Tresult_4$). However, the $treatment_3$ is done by another doctor (*Michael*) and thus cannot be updated by him. □

We show that the few proposed update access control models are not expressive enough to specify and enforce some update policies. Using the model proposed by Damiani et al. [DFGM08], an update policy is defined by annotating the XML schema with security attributes. For instance, assigning the attribute @insert=[*type*="*Blood Analysis*"] into element type *treatment* of the Hospital DTD of Figure 1.5 specifies that new elements can be inserted into *treatment* nodes having "*Blood Analysis*" as *type*. However, only *local annotations*[18] can be defined which makes the proposed model restricted to non-recursive schemas only. To show this limitation, we consider the update policy defined in Example 1.4. Using the discussed model, one cannot prevent the doctor *Imine* to update the third treatment (i.e. $treatment_3$). Specifically, assigning the attribute @insert=[*parent*::*intervention*[*doctor*/*dname*="*Imine*"]] into the *treatment* element type makes only the nodes $treatment_1$ and $treatment_4$ of Figure 1.8 updatable by *Imine* while discarding the authorized update of $treatment_2$. Moreover, the update annotation @insert=[*ancestor*::*intervention*[*doctor*/*dname*="*Imine*"]] over the *treatment* element type makes all treatment nodes of Figure 1.8 updatable by *Imine*, including $treatment_3$. The adequate update annotation remains to define the paths that refer to all the treatment nodes created under an intervention whose responsible doctor is *Imine*. As the Hospital DTD is recursive, these paths

[18]The update annotation concerns only the node and not its descendants.

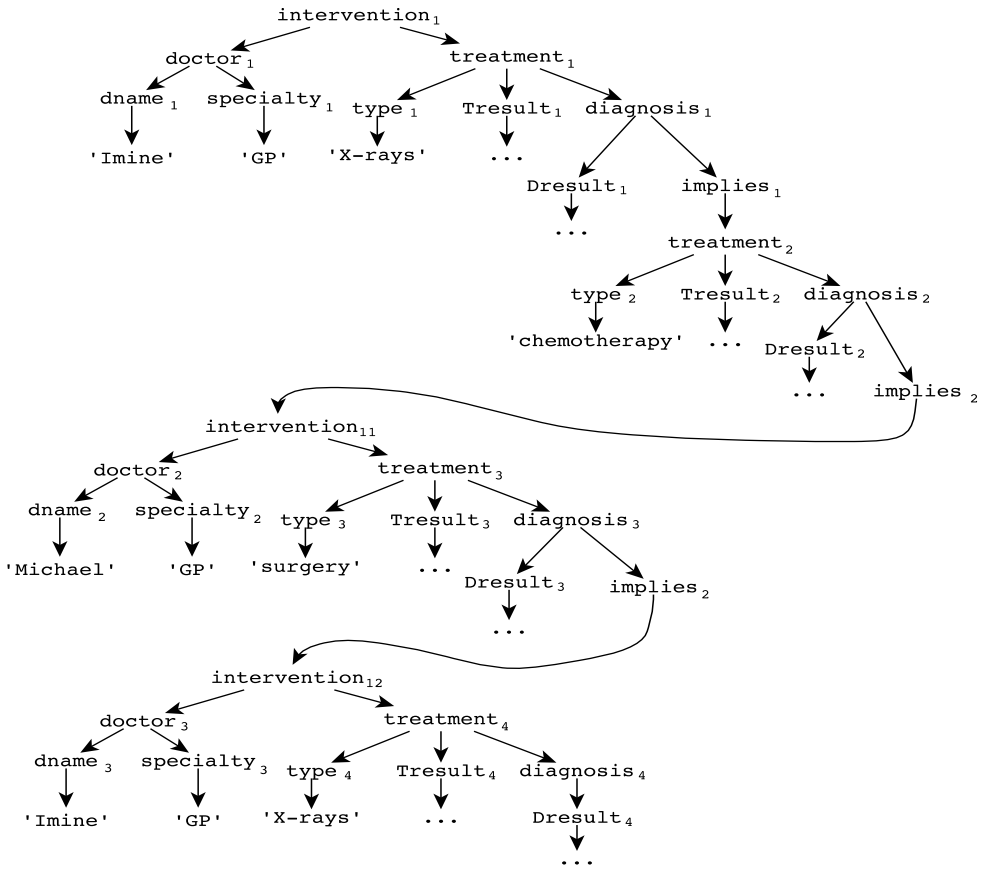


Figure 1.8: The interventions done for the patient "*Henry*" of Figure 1.6.

are infinite. Using the *transitive closure operator* "*", this update annotation would be defined by assigning the following attribute into element type *treatment*:

$$\begin{aligned}\text{@insert}=[(&parent::implies/parent::diagnosis/parent::treatment)*/\\ &parent::investigation[doctor/dname=\$\text{DNAME}]]\end{aligned}$$

Here $DNAME is treated as a constant parameter; e.g. if the concrete value *Imine* is substituted for $DNAME, then the previous annotation specifies the update right for doctor *Imine*. This update annotation ensures that a doctor with name $DNAME could update all and only the data of treatments she has done. However, the *transitive closure operator* cannot be expressed in the standard XPath as outlined in [tC06]. Thus, the update policy defined in Example 1.4 cannot be specified by using the model of [DFGM08].

Fundulaki et al. [FM07] propose an XPath-based access control model, called the `XACU`, for controlling the update primitives of the W3C XQuery Update Facility [RCD+11]. An update policy is defined by a set of `XACU` rules, each one refers to some parts of the XML document that the user is allowed/forbidden to update by using some update operation. Specifically, an `XACU` rule has the form ⟨*object, action, effect*⟩ where: 1) *object* is an XPath expression that returns the XML nodes concerned by the update; 2) *action* is an XQuery update operation; and 3) *effect* takes the value '+/-' to specify whether the rule grants or denies (case of *positive* and *negative*

rule resp.) the execution of *action* on the nodes referred to by *object*. The XACU policies are enforced by fixing: 1) a *default semantics* and 2) an *override policy*. The first one says that for a given update operation and a given node that is not explicitly concerned by an XACU rule, the user is either by default *allowed* or by default *forbidden* to perform this operation at this node. In the case when a given node is in the scope of both a positive and a negative XACU rules, the *override policy* specifies that either a positive rule overrides a conflicting negative one (*grant overrides* principle) or the other way around (*denies overrides* principle). We give the following example to show some limits of the XACU language.

Example 1.5. Consider the update policy of Example 1.4 and the XML tree of Figure 1.8. We define a positive and a negative XACU rules as follows:

(1) $\langle$*descendant*::*intervention*[*doctor*/*dname*=$DNAME]/*descendant*::*treatment*, *insert*, +$\rangle$,
(2) $\langle$*descendant*::*intervention*[*doctor*/*dname*$\neq$$DNAME]/descendant::*treatment*, *insert*, -$\rangle$.

By substituting $DNAME with "*Imine*", the first positive rule indicates that the doctor *Imine* can insert new data into the treatments that he has done (i.e. $treatment_1$, $treatment_2$, and $treatment_4$). However, since the DTD is recursive, the rule could be applied for the treatments data of another doctors, i.e. the $treatment_3$ of Figure 1.8 can be updated by *Imine*. We try to avoid this violation by defining the second negative rule that forbids each doctor to add new data into treatments done by another doctor. Observe that, $treatment_3$ and $treatment_4$ of Figure 1.8 are in the scope of both the first and the second rules. By defining the *grant overrides* as conflict resolution policy, the node $treatment_3$, done by *Michael* becomes updatable by *Imine*. In addition, applying the *deny overrides* principle makes the doctor *Imine* unable to update some of his data (i.e. the node $treatment_4$). Notice that in the two cases, the update policy of Example 1.4 is not enforced. □

Duong et al. [DZ08b] present a fine-grained access control model, named SecureX, which supports read and write privileges. Along the same lines as [DFGM08], the read and update rights are defined by annotating the XML schema with security attributes. More detail of SecureX is reported to Chapter 2. In conclusion, the model suffers from the same limit discussed above: only non recursive schemas are supported.

Given the above, the problem of securing XML updates deserves more attention. To the best of our knowledge, no model exists for specifying and enforcing XML update policies over recursive schemas.

1.3 Contributions

With the widespread use of XML, there has been a substantial amount of work that deal with the different problems related to the manipulation of XML data (e.g. confidentiality, integrity, consistency, containment). We have been mainly interested in the security aspect. We proposed formal models for specifying XML access control policies, and efficient algorithms to enforce such policies. Our solutions are based on the query rewriting approach over virtual XML views. We remark a gap between the real-life use of XML data and the results found around securing XML content. While administrators and users are familiarized with some well-known languages to manipulate XML data (e.g. XPath, XQuery, SQL/XML), some interesting results remain impractical, or costly to implement and to integrate within existing systems since they are

based on non-standard languages (e.g. Regular XPath, XUpdate). We believe that standard-based models provide practical solutions due to the fact that most of the commercial database systems feature supports for XML standards (especially XPath and XQuery). Our results are implemented and provided as a W3C standard-based XML access control system and therefore easy to integrate within some of widely used database systems.

An efficient approach for coping with recursive XML security views While, in case of recursive security views, the query rewriting is not always possible over the downward fragment of XPath [19] [FGJK07] (class of queries 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. [FCG04a] 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). Our rewriting approach can deal with a larger class of XPath queries that includes downward-axes (*child*, *descendant*), upward-axes (*parent*, *ancestor*), and 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 [FGJK06, FGJK07], 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.

An expressive XML update access control model We study the XML access control by considering the operations of the XQuery Update Facility [RCD+11]. Firstly, a fine-grained specification language is proposed to define XML update policies and which aims to overcome expressiveness limits of existing models. Our update specification language is an extension of the access specification language presented in [FCG04a]. In a nutshell, we annotate element types of the DTD grammar with update attributes to specify which update operations the user is (conditionally) allowed/forbidden to perform and over which parts of the XML document. To enforce our update policies, we propose a linear time algorithm that translates each user update operation op into another one op' such that the execution of op' over any instance of the DTD grammar makes changes only over data the user is authorized to update.

SVMAX prototype 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 supports specification and enforcement of both read and update access policies over arbitrary XML views (recursive or non). SVMAX features a visual tool that allows:

1. The administrators to specify read and update policies, and to extract views (both of data and schema) that are provided to the users.

2. The users to query and update the original data through the use of virtual view, significant class of XPath queries and a complete set of update operations.

Furthermore, SVMAX features an additional module enabling efficient validation of XML documents after execution of W3C XQuery primitives update. 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 as we should show.

[19] This fragment is more used both in practice and in theory, and several theoretical results have been found around this fragment [Woo03, NS06].

Publications Our results have been published in /submitted to the following conference papers:

- Our model of read-access control has been presented in [MI12e].
- We presented our update access control in [MI12b] and [MI12c].
- The interaction between read and update privileges (formal solution and algorithm) was first introduced in [MI12d]. A journal paper is under preparation.
- The paper presenting our system SVMAX has been published in [MIR13].

1.4 Outline of Dissertation

The remainder of this thesis is organized as follows. Chapter 3 provides essentially background about XML access control. We survey more recent models and algorithms for specification and enforcement of XML access policies, approaches for validation of XML data, as well as some other related problems (e.g. consistency of policies, view update translation). Chapter 2 reviews some basic definitions and notions that are tackled throughout this manuscript. In Chapter 4, we introduce our model of security view together with our algorithm for rewriting XPath queries over virtual XML views. Chapter 5 explains how the security view notion can be extended to handle update operations. It provides a formal model for specification of XML update policies, an efficient algorithm for enforcing such policies, as well as a discussion about the interaction of read and update policies. Chapter 6.3 presents an overview of the basic features of our system, followed by an extensive experimental study based on real-world DTDs. Chapter 6.4 concludes the thesis and draws research perspectives.

2

State of the Art

Contents

Although access control for traditional databases has been studied for a long time (e.g. relational databases [ST90, LDS+90, BJS99], object-oriented databases [BJS93, BCCGY93]), the nature of XML data (i.e. hierarchical structure, nodes dependency, schema recursion) make the proposed approaches difficult to adapt for XML databases. The problem of protecting XML content is relatively new compared with traditional databases problems that have been widely studied for more than two decades. In addition, some problems related to the manipulation of XML data are still in their early stages (or in active study) such as XML update policies specification [BCFS12], XML queries optimization [HL13a, WTWS13], updating compressed XML data [BHJ13], querying encrypted XML data [CRK+13].

A substantial part of this chapter is dedicated to the main XML access control models proposed during the last years. Firstly, we interested ourselves with the use of XML views, models and algorithms to derive and query such views. We put emphasis on the strategy established to query the original data through a request that has been expressed on the view, and in which case this strategy is challenging. We investigate after the issues that arise when controlling access to XML data by considering XML updates. Finally, we discuss solutions provided to preserve validity of XML documents after updates, a constraint that is required by many systems and applications.

2.1 Basic Models

Originally, numerous models have been proposed to specify and enforce security policies that protect access to different kinds of data. The ACL (*Access Control List*) and XACML (*eXtensible Access Control Markup Language*) belong to the first efforts done in this context. Since the ACL-based security model is used in different domains (e.g. networking, file systems, databases), we discuss such security concept as supported by the *Oracle 11g* database system. An ACL is a

list of access control entries (ACEs). Each ACE is an XML element that either grants or denies access to some data by a particular principle (user or role). The order in which the ACEs of an ACL are stored is relevant. Before a principal accesses to some data that is protected by one or more ACLs, the evaluation of these ACLs is done first according to their order. For each ACL, the ACEs in it that apply to the principal are examined, in order. If one ACE grants a certain privilege to the current user and another ACE denies that privilege to the user, then a conflict arises. Different conflicts resolution strategies can be applied: *1)* only the first matched ACE is considered; *2)* grant takes precedence over deny; or *3)* deny takes precedence over grant. Although the ACL model is currently used in many database systems such as *Oracle 11g*, its enforcement strategy is rather brute-force in the sens that it grants/denies access to the entire resource. Consider case of XML data, in practice one often wants the query to return the parts of the document that the user is authorized to access, instead of reject the access.

The XACML [XACa] is an OASIS standard, now at version 3.0, that defines a language for the definition of policies, access requests, and a work-flow to achieve policy enforcement. These tasks are accomplished through different components, principally: the *Policy Administration Point* (PAP), the *Policy Decision Point* (PDP), and the *Policy Enforcement Point* (PEP). XACML policies are defined as XML documents expressing the privileges a user needs to have for accessing the resource. Each resource can be controlled with one or more policies. A policy is a set of rules where each one defines a TARGET and an EFFECT. The EFFECT specifies whether the user request is "*Permitted*" or "*Denied*", while the TARGET is composed of the sub-elements *Subject* (i.e. the user that wishes to perform some action over the resource), *Action* (e.g. read, update), *Resource*, and *Environment* (e.g. time, date). Each sub-element is accompanied with a matching function: used to check whether the policy rule is applicable to a given request, specifically, it matches the value of the sub-element of the rule with the value of the same sub-element of the user request. A policy also specifies an algorithm that defines what is the final decision for a request when there are (permit/deny) conflicts in the rule decisions. Upon receiving a request, the policies are retrieved from the PAP by the PDP. The PDP checks the matching between all values of the request and those of the retrieved rules, in order to *permit* or *deny* access to the resource. In addition to *Permit* and *Deny*, the PDP decision can also be *not-applicable* if no policy rule applies to the request; or *indeterminate* if the PDP fails to evaluate the access request (e.g. missing of request attributes, network errors). The authorization decision is enforced by the PEP which can have many different forms, e.g. part of a remote-access gateway, a Web server.

XACML is now the de facto standard for enforcing access control policies in service-oriented architectures. Many database systems provide implementation of XACML such as *eXist-db*. However, designing XACML access control policies is a difficult and error-prone task. For this reason several tools have been proposed to make easy edition of XACML policies [XACb, UMU].

The above discussed standards are more suitable for some domains like file systems, networks, and (traditional) database systems. However, they still remain slightly limited when considering hierarchical data like XML. A few works exist that have to extend the XACML standard in order to take into account hierarchical data [KA08, Xia12]. More specific access control models have been proposed during the two decades that are aware of the XML data structure and efficiently secure access to such data. As we shall explain in the following, each model has its relevant use case where it may ensure good performances.

2.2 XML Access Control Models

2.2.1 Instance-based Models

These models transmit access permissions from *policy-level* into *node-level*. A policy consists of a set of access control rules, each rule is composed of different sub-elements defining: a *subject* (a user ID, a role, or a group name); an *object* defined with an XPath expression and that specifies the XML nodes to control; an *action* that can be read/update (we consider only read actions as supported by existing models); a *permission* (grant/deny) that specifies if the *subject* is (not) allowed to execute the *action* on the *object* nodes; and a *propagation* that specifies if the rule is *local* (applied only on *object* nodes) or *recursive* (applied on *object* nodes together with their descendants). A conflict can arise when two or more rules define different access permissions for the same XML node (e.g. a node concerned by a local and a recursive rules that have different permissions). Different conflict resolution strategies can be applied (e.g. denial permission overrides any grant permission, recursive rules take precedence over local rules). The enforcement of such policies is done by annotating the XML document with the different access permissions. More specifically, for each user u, each rule r, and each node n concerned by r (i.e. n is referred to by the r XPath expression), n is annotated by adding an attribute *@access* that has value '+' (if r permission is grant) or '-' (if r permission is deny). If n is already annotated then the conflict is resolved using underlying strategy. The annotation is propagated then along with n descendants if r is recursive. Finally, when all the rules that concerns the user u are parsed, the remaining unlabeled nodes are annotated with '-' par default. Each user XPath expression is rewritten in order to return only accessible nodes (those annotated by '+') and without any disclosure of sensitive information. In [MTKH06], authors impose the *denial downward consistency* requirement that denies access to a node if only one of its ancestors is inaccessible. In this case, rewriting of XPath expressions is quite simple and done by adding a predicate [*@access*='+'] to the end of each user XPath expression. In the other case however, where the aforementioned requirement is not applied, the enforcement mechanism is not discussed. In our view, this can be simply done as follows: for a user XPath expression, each sub-expression *descendant*::*label* is rewritten into *descendant*::*label*[*@access*='+'], while each sub-expression *child*::*label* is rewritten into *descendant*::*label*[*@access*='+'][position()=1].

Regretfully, the annotation process is repeated for every user, every action a user takes, and each time the policy and/or the data are changed. This is time consuming and requires a lot of resources for XML data parsing and labeling. Additionally, instance-based approaches lead to poor performances in case of huge XML documents since user queries are evaluated over the whole XML document while, in practice, accessible data is more smaller that the original data. Authors of [MTKH06] proposed a static analysis which is in duty to determine *statically* whether the user request does not select any accessible nodes, and then reject it without any evaluation. However, this static analysis fails in different cases and the query must be evaluated over the whole original data.

Authors of [DZ08a] proposed a dynamic labeling scheme that avoids re-calculation of labels when the XML data is updated. Their algorithm can generate unique codes for every new node without re-labeling existing nodes. The approach is used by the same authors to propose a fine-grained access control model, named *SecureX*, which supports read and write privileges, introduces various access types, and outperforms naive labeling approaches by eliminating repetitive labeling when updating data. In case of dynamic policies however, the whole XML document may be re-labeled and then *SecureX* may lead almost to the same performances as naive labeling approaches.

2.2.2 Security views

The scenario for this group of proposals is to extract an authorized XML view which includes data relevant to the user's clearance: access control rules applicable to the user are used to partially type the XML tree with Y (+) and N (-) labels. After that, partial annotation is extended to full one. Finally, fully annotated label is sanitized, i.e., N labeled elements are either deleted completely [BBC+00, WKD04] or modified [DdVPS02]. A different approach to XML security views was shown in [SF02b] where there was an attempt to define a security view by a single XPath expression. In a nutshell, XML elements matching non-asterisk location steps of XPath are added to a view, while asterisks represent forbidden parts of the XML.

2.2.3 Schema-based security views

Stoica and Farkas [MSW05] proposed to produce single-level views of XML when conforming DTD is annotated by labels of different confidentiality levels. The key idea lies in analyzing semantic correlation between element types, modification of initial structure of DTD and using cover stories. Altered DTD then undergoes "filtering" when only element types of the confidentiality level, that is no higher than the level of the requester, are extracted. However, the proposal requires expert's analysis of semantic meaning of production rules, and this can be unacceptable if database contains a large amount of schemas which are changed occasionally. Another view-based approach is proposed by Fan et al. [FCG04b]. We summarize this proposal as follows. The process of an access control policies enforcement can be described as follows: (1) define access specification for each class of users, (2) derive a sound and complete DTD security view for a particular access specification, (3) supply the user with a corresponding security DTD view, (4) user issues a query in terms of kept DTD view, (5) a query over the view schema is rewritten to a query over the initial schema and optimized, (6) the optimized query is evaluated over the XML and the result is returned to the requester. The latest approach to schema-based security views was presented in Mohan et al. [GB01]. The solution allows a complete restructuring of a DTD and relies on a command-like specification language. However, it was mentioned in Mohan et al. [GB01] that many operations are not commutative and have restrictions that means a possibility of errors while designing access control policies.

2.3 Securing XML Updates

Among most of existing update access control models, for each update operation, an XPath expression is defined to specify the XML data at which the update is applied. To enforce an update policy, the *query rewriting* principle can be applied where each update operation (i.e., its XPath expression) is rewritten according to the update constraints into a safe one in order to be performed only over parts of the XML data that can be updated by the user who submitted the operation. However, this rewriting step is already challenging for a small class of XPath. Consider the *downward* fragment of XPath which supports *child* and *descendant-or-self* axes, union and complex predicates. We will show that, in case of recursive DTDs, an update operation defined in this fragment cannot be rewritten safely. More specifically, a safe rewriting of the XPath expression of an update operation can stand for an infinite set of paths which cannot be expressed in the downward fragment of XPath (even by using the upward-axes: *parent*, *ancestor*, and *ancestor-or-self*).

To overcome this rewriting limitation, one can use the '*Regular XPath*' language [Mar04], which includes the *transitive closure operator* and allows to express recursive paths. Note that

this language has been used in [FGJK07, GSC+09] for the rewriting of read-access queries. However, it remains a theoretical achievement since no tool exists to evaluate Regular XPath queries. Thus, no practical solution exists for enforcing update policies in the presence of recursive DTDs.

In the same work [FM07], Fundulaki et al. present another version of their language `XACU`, called the $\texttt{XACU}^{annot}$, in order to specify update policies in the presence of DTD grammar. The $\texttt{XACU}^{annot}$ is based on the notion of schema annotation presented in [FCG04a]. An $\texttt{XACU}^{annot}$ annotation is of the form $\texttt{ann}(gpath, op)\texttt{=}Y|N|[Q]$ where: (1) *gpath* is a full XPath expression defined from the root of the DTD into the node concerned by the update, (2) *op* is an update operation, and (3) the values $Y|N|[Q]$ specify that the user can (Y) | cannot(N) | can if the XPath expression Q is true, perform operation *op* at nodes returned by *gpath*. This model can be applied only for non-recursive DTDs. For instance, the update policy of Example 5.4 cannot be defined by the $\texttt{XACU}^{annot}$ language. Specifically, the XPath expressions denoting treatments data done by a given doctor stand for an infinite set of paths. For any update operation *op*, the adequate annotation could be:

$\texttt{ann}(//intervention[doctor/dname =\$\text{DNAME}]/$
$(treatment/diagnosis/implies)*/treatment, op) = Y$

However, the *transitive closure operator* '$*$' is outside the expressive power of the standard XPath [tC06].

3

XML Background

Contents

We consider the problem of securing XML data protected using arbitrary security views. These views are defined by associating some security attributes to the schemas that correspond to the underlying XML documents (Stoica and Farkas [SF02a]). The user can query the data views through any XML query language. The XML views that we consider are defined using DTD grammars (thus, definition of XML schema [XML] is omitted here) while the XPath query language [BBC+10] is used to query such views. The current chapter presents some basic notions and definitions that are used throughout this manuscript. We define some key problems that have motivated our contributions.

3.1 Document Type Definitions

Definition 3.1 (**DTD** [RCD+08])**.** 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:

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

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. We refer to $A \rightarrow P(A)$ as 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 terms of itself directly or indirectly.

This DTD definition allows complex contents (i.e. contents with both "," and "|"), as well as mixed contents that are mixture of text and elements. Examples of these types are given in the following.

Example 3.1. We consider the *department* DTD (Σ, P, *dept*) where 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:

dept	→	(*course*+, *project*∗)
course	→	(*cname*, *takenBy*, *givenBy*)
takenBy	→	(*students*)
students	→	(*scholarship*?, *student*+)
scholarship	→	(*student*+)
student	→	(*sname*, *mark*)
givenBy	→	(*professor*+)
professor	→	(*pname*, *grade*)
project	→	(*type*, *descp*, *results*, *members*, *sub-project*)
type	→	(*private* | *public*)
results	→	(`str` | *result*)∗
members	→	(*member*+)
member	→	(*name*, *qualif*, (*theoretical* | *experimental*)∗)
sub-project	→	(*project*∗)

The element types *private* and *public* are empty, while the remaining element types (e.g. *mark*, *result*) are text elements. A *department* element has a list of *course* elements as well as one 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 hold a *scholarship*. 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 one or more *sub-project*. 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*)∗. □

Authors of [FCG04a] use DTDs in *normal form* that do not contain mixed containers (i.e. the content of any element type must be either conjunction or disjunction of subelement types). For instance, the element type *member* of the previous example is defined with a mixed container, thus the *department* DTD is not in normal form. A *normalization* of the *department* DTD can be done by changing only the content of *member* element type as follows:

member	→	(*name*, *qualif*, *contribution*)
contribution	→	(*theoretical* | *experimental*)*

with this new content type the *department* DTD becomes in normal form. Authors of [FCG04a] claim that any DTD may be normalized in the purpose to easily derive its view with respect to some access rights. To keep DTDs in normal form, some additional element types may be added as we have done with the content of element type *member*. Note that in case of policy enforcement, the additional element types may reflect the structure of some sensitive data that should be hidden with the DTD view.

3.1.1 DTD Graphs

It is well-known that non-recursive DTDs may be modeled as a DAG. However, arbitrary DTDs deserve the introduction of a special structure in order to model order between elements and their dependencies (defined with operators $*$, $+$, etc.), cycles and all DTDs information. To the best of our knowledge, no formal definition of such structure exists in the literature. Kuper et al. [KMR09] have informally introduced an expressive graph representation to model all DTDs. We refine here their structure, called *DTD graph*, and give a formal definition of it.

Based on the definitions of *multigraph* and *edge-ordered* graph [GY03, JB99, TKCW09], we define first our *graph* structure as follows:

Definition 3.2 (**Graph structure**). A rooted vertex-labeled edge-ordered directed multigraph is a structure $G{=}(\Sigma, V, E, \lambda_V, v_{rt}, Order)$, where Σ is a set of labels, V is a set of vertices, $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ is a multiset of edges, λ_V: $V \to \Sigma$ assigns a label to each vertex, $v_{rt} \in V$ is a distinguished root vertex, $Order$: $\mathbb{N} \times V \to V$ defines a *unique order* between children of any vertex v of V such that: $Order(i, v)$ denotes the i^{th} child of v. □

Given a graph $G = (\Sigma, V, E, \lambda_V, v_{rt}, Order)$; $arity(v) = |\{(v, v') \mid (v, v') \in E\}|$ is a function that, for a given vertex $v \in V$, computes the number of its children in G. Based on the graph structure defined above, we formalize our notion of *DTD graph* as follows:

Definition 3.3 (**DTD graph**). A *DTD graph* is a graph $G = (\Sigma, V, E, \lambda_V, v_{rt}, Order)$ which satisfies the following conditions:

1. $\Sigma = \Sigma_1 \uplus \{\cdot, |, *, +, ?, \texttt{str}\}$ (where Σ_1 represents the element types of D);
2. $\lambda_V(v_{rt}) \in \Sigma_1$;
3. For any A in Σ_1, there exists exactly one vertex v in V such that: $\lambda_V(v) = A$;

For each v in V we have:

4. $arity(v) = 0$ if $\lambda_V(v) = \texttt{str}$ or $\lambda_V(v)$ corresponds to an element type with empty content;
5. $arity(v) \geq 2$ if $\lambda_V(v) \in \{\cdot, |\}$;
6. $arity(v) = 1$ if $\lambda_V(v) \in \{*, +, ?\}$;
7. $arity(v) \leq 1$ if $\lambda_V(v) \in \Sigma_1$;
8. If $\lambda_V(v) \in \{\cdot, |\}$, then $Order(i, v)$ must be defined for any $1 \leq i \leq arity(v)$. □

Given a DTD D, its DTD graph is denoted by G_D where each vertex v represents either an element type of D, if $\lambda_V(v) \in \Sigma_1$, a text node if $\lambda_V(v) = \texttt{str}$, or an operator otherwise. For each element type A of D, there is exactly one node in G_D that is labeled with A. However, there is a node labeled with "$\cdot$", "$|$", "`str`", "$*$", "$+$", or "?" for each occurrence of sequence container, choice container, `str` definition, operators $*$, $+$, or ? in the production rules of D. A DTD graph is *cyclic* if its corresponding DTD is recursive. According to our DTD definition 3.1, a B subelement type of an A element type may appear more than once in $P(A)$ (e.g. $A \rightarrow B, C, B$). Thus, there may be in G_D more than one edge connecting the same ordered pair of vertices. These edges are called *multiple edges* (i.e. edges with the same *source* and *target* vertices) and are represented by the *multiset* E. A DTD graph is *simple* if it does not contain multiple edges, otherwise, it is called *multigraph* [GY03].

Since we consider ordered XML trees, for a given production rule $A \rightarrow \alpha_1, \ldots, \alpha_k$ in a DTD D (resp. $A \rightarrow \alpha_1 + \ldots + \alpha_k$), order defined between element types and operators of each α_i must be preserved in the DTD graph G_D of D. An expression $\alpha_1, \ldots, \alpha_k$ (resp. $\alpha_1 + \ldots + \alpha_k$) is presented in G_D by a subgraph rooted at vertex v with label "$\cdot$" (resp. "$|$"). This vertex has k children $v_{\alpha_1}, \ldots, v_{\alpha_k}$ where each vertex v_{α_i} represents the structure of the sub-expression α_i. In order to preserve the order defined between sub-expressions α_i in D, the edges $(v, v_{\alpha_1}), \ldots, (v, v_{\alpha_k})$ must be uniquely ordered. For this reason, the function *Order* of Definition 3.2 must be explicitly defined for each vertex of G_D with label "$\cdot$" or "$|$" (condition (8) of Definition 3.3). In a nutshell, consider the production rule $A \rightarrow \alpha$ and let v_A be the vertex representing element type A in a given DTD graph G_D. The vertex v_A has no child if $\alpha = \epsilon$; otherwise, it has a child v_α representing the structure α of A. If the vertex v_α has no children then either $\lambda_V(v_\alpha)$=`str` (i.e. $A \rightarrow$ `str`); or, $\lambda_V(v_\alpha)$=B (i.e. $A \rightarrow B$). If the vertex v_α has children $v_1, \ldots, v_k$ ($k \geq 1$), then a *unique order* is defined, using the function *Order*, between the edges $(v_\alpha, v_1), \ldots, (v_\alpha, v_k)$, and this according to the occurrence of element types and/or operators $\lambda_V(v_1), \ldots, \lambda_V(v_k)$ from left to right in the expression α. Due to this order, we define our DTD graphs as *edge-ordered* graphs.

Example 3.2. Figure 3.1 depicts the DTD graph $G_{department}$ of the *department* DTD of Example 3.1 (the nodes are depicted by their labels). We omit the complete definition $(\Sigma, V, E, \lambda_V, v_r, Order)$ of $G_{department}$ since it can be easily defined according to Definition 3.3. In the remainder of this example, names in parentheses are abbreviations of vertices. Consider the simple case of element type *givenBy*, $G_{department}$ contains a vertex labeled with *givenBy* ($v_{givenBy}$). Since $P(givenBy)$ is enclosed by $+$, vertex $v_{givenBy}$ has a child labeled with $\oplus$ ($v_\oplus$), $v_\oplus$ points to a vertex labeled *professor* ($v_{professor}$) that represents the element type *professor*. Moreover, since $P(professor)$ is a sequence container, vertex $v_{professor}$ has a child labeled with $\odot$ ($v_\odot$), $v_\odot$ has two vertices labeled with *pname* and *grade* (v_{pname} and v_{grade} resp.) representing element types *pname* and *grade* respectively, each of these latter vertices has text node. The order between vertices v_{pname} and v_{grade} is preserved by the function *Order* defined over vertex $v_\odot$ with: $Order(1, v_\odot)$=v_{pname} and $Order(2, v_\odot)$=v_{grade}. We remark first that $G_{department}$ is a simple graph since there are no multiple edges. Moreover, $G_{department}$ is cyclic due to the existence of the cycle defined over the element types *project* and *sub-project* (i.e. a cycle $v_{project}, v_\odot, v_{sub-project}, v_*, v_{project}$ composed by some vertices labeled with *project*, $\odot$, *sub-project*, $*$, and *project* resp.). □

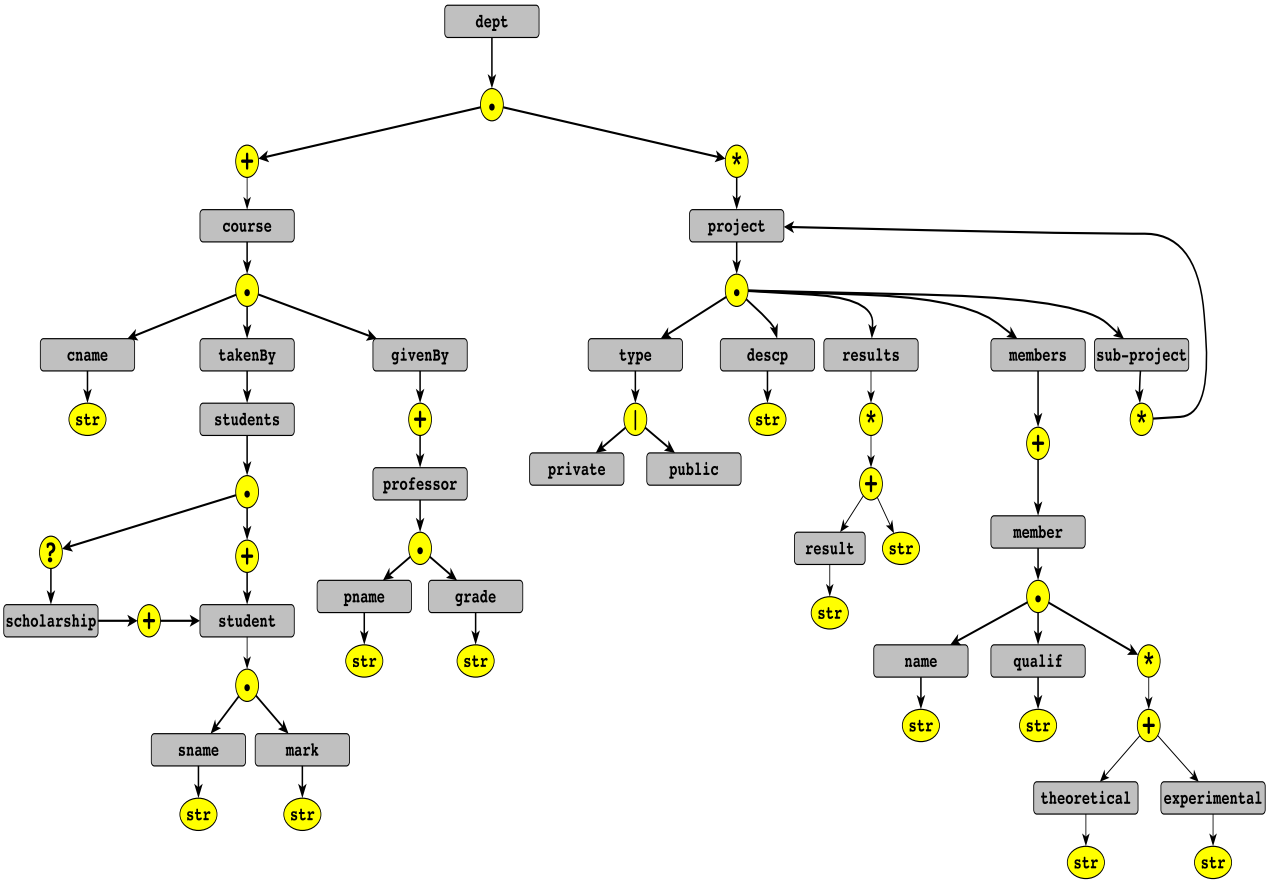


Figure 3.1: The *department* DTD graph.

Note.
Note that our definition of DTD graph differs from the ones defined in [BS03, LSAF05] which are sufficient to describe only some kind of DTDs. Moreover, the notion of *dependency graph* used in [FM07] represents only parent-child relationship between types of the DTD and not other constraints. We have proposed a formal definition of DTD graph since such structure is essential, as we will see later, for derivation of DTD views in case of read-access policies enforcement. The different steps of our DTD graphs construction, as well as the details and complexity of the corresponding algorithm are given in Annex B.

3.1.2 Extended DTDs

Papakonstantinou and Vianu [PV00] extended the expressive power of DTDs by adding types as in XML Schema [XML]. Given a DTD $D{=}(\Sigma, P, root)$, instead of having only one content model $P(A)$ for each element type A in Σ, they proposed to define on or more types (e.g. $A_1, \ldots, A_n$) to represent different content models of element type A (e.g. $P(A_1), \ldots, P(A_n)$). Types are from a finite set and each one is assigned to only one element type of the DTD. The root element type has only one possible type. The term *specialized DTDs* is used in [PV00] to refer to the resulted grammars (i.e. DTDs plus types). However, we prefer the term *extended DTDs*, as introduced

students	→	(*student**)
student	→	(*sname*, $degree^1$)
$degree^1$	→	(*year*, *title*, $degree^2$)
$degree^2$	→	(*year*, *title*, $degree^3$)
$degree^3$	→	(*year*, *title*)
sname	→	(`str`)
year	→	(`str`)
title	→	(`str`)

(a) EDTD E_1

students	→	(*student**)
student	→	(*sname*, $degree^1$)
$degree^1$	→	(*year*, *title*, $degree^2$)
$degree^2$	→	(*year*, *title*, $degree^3$)
$degree^3$	→	(*year*, *title*, ($degree^3$)?)
sname	→	(`str`)
year	→	(`str`)
title	→	(`str`)

(b) EDTD E_2

students	→	(*student**)
student	→	(*sname*, ($degree^1$)?)
$degree^1$	→	(*year*, *title*, ($degree^2$)?)
$degree^2$	→	(*year*, *title*, ($degree^3$)?)
$degree^3$	→	(*year*, *title*)
sname	→	(`str`)
year	→	(`str`)
title	→	(`str`)

(c) EDTD E_3

Figure 3.2: Production rules of three different EDTDs.

in [MNSB06], to express clearly that the power of the DTDs is amplified.

Definition 3.4 (EDTD). An *extended DTD* (EDTD) is a tuple $E=(\Sigma, \Delta, root, P, \mu)$ where Σ is a finite set of element types, Δ is a finite set of types, $(\Delta, root, P)$ is a DTD and μ is a mapping from Δ to Σ assigns an element type to each type. □

Standard DTDs (i.e. *nonextended* DTDs) correspond to those EDTDs $(\Sigma, \Delta, root, P, \mu)$ where $\Delta=\Sigma$ and μ is the identity mapping. The following simple example shows some content models that are not definable using standard DTDs.

Example 3.3. Consider the EDTD $E_1=(\Sigma, \Delta, students, P, \mu)$ where $\Sigma=\{$ *students*, *student*, *sname*, *degree*, *year*, *title* $\}$, $\Delta=\{$ *students*, *student*, *sname*, $degree^1$, $degree^2$, $degree^3$, *year*, *title* $\}$, $\mu(degree^i)=degree$ $(1 \leq i \leq 3)$ and $\mu(t) = t$ $(t \in \Delta \setminus \{degree^1, degree^2, degree^3\})$. The production rules of the DTD $D_1=(\Delta, students, P)$ are given in Figure 3.2 (a). Intuitively, EDTD E_1 consists of a list of students, each *student* is represented by his name (*sname*) and some of his degrees, each *degree* is defined with a *year* and a *title*. Notice that only three (latter) degrees of each student are represented. Thus, in any XML document satisfying the EDTD E_1, the nesting depth of *degree* elements is exactly 3. EDTDs E_2 and E_3 (whose production rules are depicted in Figures 3.2 (b) and (c) resp.) differ from E_1 in the sense that each *student* element is defined with at least 3 *degree* elements for the former and at most 3 for the latter. The three different content models of *student* element type, presented by EDTDs E_1, E_2 and E_3, can not be captured using standard DTDs. □

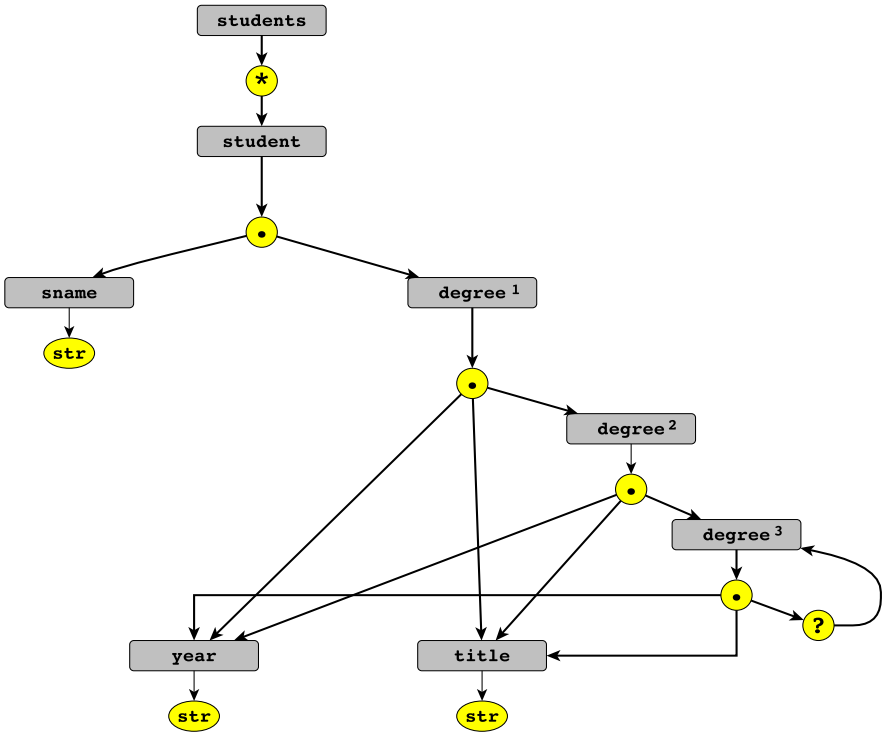


Figure 3.3: The EDTD graph of EDTD E_2 of Example 3.3.

Definition 3.5 (**EDTD graph**). Given an EDTD $E=(\Sigma, \Delta, root, P, \mu)$, the DTD graph corresponding to its DTD $(\Delta, root, P)$ is called *EDTD graph*. □

For instance, the EDTD E_2 of Example 3.3 is represented by the EDTD graph depicted in Figure 3.3. Finally, we note that there is a more generalized extension of DTDs defined by allowing use of context-free grammars in place of regular expressions.

Definition 3.6 (**GDTD** [GSC+09]). A *generalized DTD* (GDTD) is a tuple $H=(\Sigma, root, \Pi)$ where Σ and $root$ are the set of element types and the root type respectively; while Π is a function that maps Σ to context-free grammars over $\Sigma \cup \{\texttt{str}\}$. □

3.2 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 3.7 (**XML Tree**). An XML tree T over Σ is a structure defined as: $T=(N, root, R_{\downarrow}, R_{\rightarrow}, \lambda, \nu)$, 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 that assigns 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 [Mar04], we define the function ν to associate data values with nodes since data value comparison is supported by our XPath fragments defined subsequently.

Definition 3.8 (**Validation of XML trees w.r.t DTD/GDTD** [Mar04])**.** 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=($Ele, P, root$) (resp. a GDTD G=($Ele, root, \Pi$)) 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 $\Sigma = Ele \cup \{\texttt{str}\}$;
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 $P(A)$ (resp. the context-free language given by $\Pi(A)$);
4. Each text node n (i.e. with $\lambda(n) = \texttt{str}$) 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 give in the following the necessary conditions for an XML tree to be valid w.r.t an EDTD.

Definition 3.9 (**Validation of XML trees w.r.t EDTD** [Mar04])**.** An XML tree $T = (N, r, R_{\downarrow}, R_{\rightarrow}, \lambda, \nu)$ conforms to an EDTD E=($Ele, \Delta, root, P, \mu$) if there is a function L': $N \rightarrow \Delta$ such that:

1. For each node n of N with $\lambda(n) \neq \texttt{str}$, $L'(n) = t$ where $t \in \Delta$ and $\mu(t) \in Ele$. The tree T' resulting from the application of L' is called **witness** for T [MNSB06].
2. The witness tree T' conforms to the DTD $(\Delta, root, P)$ of E. □

We call T an instance of a DTD (resp. EDTD and/or GDTD) D if T conforms to D. We denote by $\mathcal{T}(D)$ the set of all XML trees that conform to D. For instance, Figure 3.4 depicts[20] an XML document that conforms to the *department* DTD of Figure 3.1.

3.3 XPath Queries

We define here the different fragments of XPath [BBC+10] that are used throughout this manuscript. Firstly, we consider a simple class of XPath queries that supports downward axes (*self*, *child*, *descendant*, *descendant-or-self*), union of queries and complex predicates. This class of XPath queries is commonly used in practice and many interesting results are found around this class. We introduce after some expressive classes of XPath used essentially to overcome the query rewriting limitation discussed in Section **??**.

[20]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 $\bigtriangleup$ denotes the remaining ones.

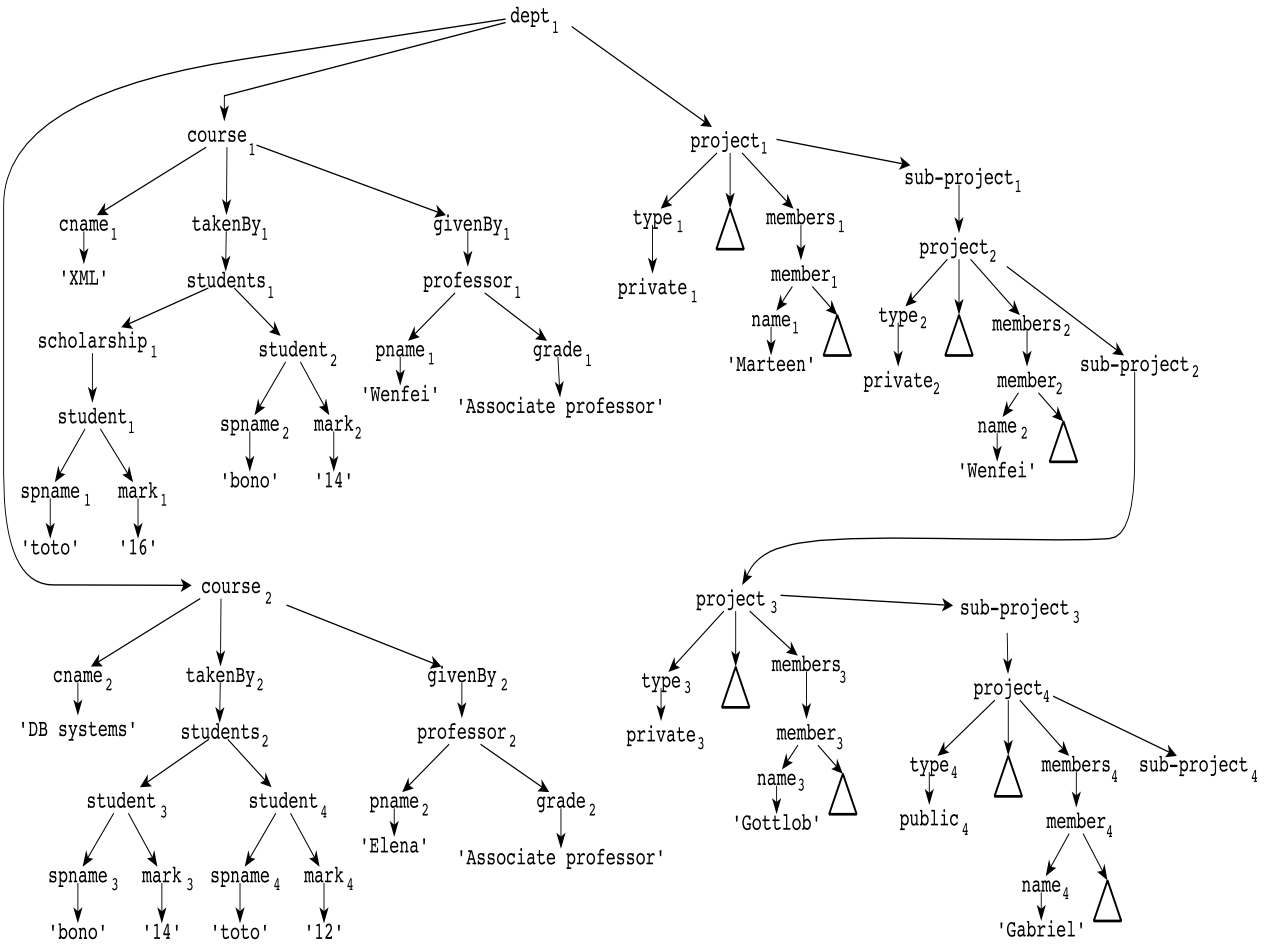


Figure 3.4: Example of *department* XML document.

Definition 3.10 (**XPath Downward fragment**). We denote by $\mathcal{X}$ the *downward* fragment of XPath [Jia07] 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 α::*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). See Section 3.3.1 for more semantics of our XPath queries.

Example 3.4. We consider the XML tree of Figure 3.4 and we define the following XPath query:

$$\downarrow^*::course[\downarrow::givenBy/\downarrow::professor/\downarrow::pname='Wenfei']$$

This query returns all *course* elements given by the professor *Wenfei*, i.e. the node $course_1$. In the official XPath notation [BBC+10], this query is written as $//course[givenBy/professor/pname='Wenfei']$. □

We define in the following more expressive fragments of XPath that are the core of the access control approaches proposed in this dissertation.

Definition 3.11 (Extended fragment). We consider an extended fragment of $\mathcal{X}$, denoted by $\mathcal{X}^{\Uparrow}_{[n,=]}$, and defined as follows:

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

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 [BBC+10]. □

In general [BBC+10], 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.

Note.

Contrary to the global definitions of position predicate (i.e. $[k]$ with $k \in \mathbb{N}$) and node comparison predicate (i.e. $[p_1{=}p_2]$) [BBC+10], for our purpose we need only the forms $[1]$ and $[p{=}\varepsilon::*]$ respectively. We define both restrictions since the resulting predicates are sufficient to overcome the limitation of XPath query (resp. XQuery update operations) rewriting as we shall show later. Furthermore, based on these restrictions our fragment of Definition 3.11 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 noted that we use fragment $\mathcal{X}$ to specify only 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.

$$\mathcal{S}[\![\alpha::\eta]\!](N) = \alpha(N) \cap T(\eta)$$
$$\mathcal{S}[\![p_1/p_2]\!](N) = \mathcal{S}[\![p_2]\!](\mathcal{S}[\![p_1]\!](N))$$
$$\mathcal{S}[\![p_1 \cup p_2]\!](N) = \mathcal{S}[\![p_1]\!](N) \cup \mathcal{S}[\![p_2]\!](N)$$
$$\mathcal{S}[\![(p_1 \cup p_2)/p]\!](N) = \mathcal{S}[\![p]\!](\mathcal{S}[\![p_1 \cup p_2]\!](N))$$
$$\mathcal{S}[\![p[q]]\!](N) = \mathcal{S}[\![p]\!](N) \cap \xi[\![q]\!]$$

$$\xi[\![p]\!] = \{n \in T \mid \mathcal{S}[\![p]\!](\{n\}) \neq \phi\}$$
$$\xi[\![q_1 \wedge q_2]\!] = \xi[\![q_1]\!] \cap \xi[\![q_2]\!]$$
$$\xi[\![q_1 \vee q_2]\!] = \xi[\![q_1]\!] \cup \xi[\![q_2]\!]$$
$$\xi[\![\neg\ (q)]\!] = \{n \in T\} \setminus \xi[\![q]\!]$$
$$\xi[\![p = c]\!] = \{n \in T \mid \varphi[\![c]\!](\mathcal{S}[\![p]\!](\{n\})) \neq \phi\}$$
$$\varphi[\![c]\!](N) = \{n \in N \mid \nu(n) = c\}$$
$$\xi[\![p[i]]\!] = (\xi[\![p]\!])[i]$$
$$\xi[\![p_1 = p_2]\!] = \{n \in T \mid \exists!\ m \in T, \mathcal{S}[\![p_1]\!](\{n\}) = \mathcal{S}[\![p_2]\!](\{n\}) = \{m\}\}$$

Table 3.1: Semantics of $\mathcal{X}^{\Uparrow}_{[n,=]}$ queries.

Example 3.5. Consider the *department* DTD of Example 3.1. We assume that the *student* subelement types of each *course* element are sorted according to the *mark* values, i.e. for a given *course* element, the first *student* gets the top *mark* and so on. We define the two following $\mathcal{X}^{\Uparrow}_{[n,=]}$ queries:

Q_1=$\downarrow^+$::*course*[$\downarrow^+$::*student*[1]/$\downarrow$::*sname*="*toto*"]
Q_2=$\downarrow^+$::*course*[$\downarrow^+$::*student*[1]=$\uparrow^+$::*dept*/$\downarrow^+$::*student*[$\downarrow$::*sname*="*toto*"]]

The first query returns all the *course* where the best student is named *toto*, i.e. when evaluated over the XML tree of Figure 3.4, Q_1 returns the element $course_1$. Let us take a look at the second query. Given first the element $course_2$. The predicate of Q_2 is valid at $course_2$ if the evaluation of the queries $\downarrow^+$::*student*[1] and $\uparrow^+$::*dept*/$\downarrow^+$::*student*[$\downarrow$::*sname*='*toto*'] return the same single node. The query $\downarrow^+$::*student*[1] over $course_2$ returns the element $student_2$ who is the best student of *DB systems* course. The query $\uparrow^+$::*dept*/$\downarrow^+$::*student*[$\downarrow$::*sname*='*toto*'] over $course_2$ returns the element $student_1$. It is clear that the predicate of query Q_2 is not valid at element $course_2$ since the two sub-queries refer to two different elements $student_2$ and $student_1$. However, it is easy to see that the predicate of query Q_2 is valid at element $course_1$ and thus, the query Q_2 over the XML tree of Figure 3.4 returns the element $course_1$. Notice finally that the two queries Q_1 and Q_2 are equivalent. □

3.3.1 Semantics and Equivalences of our XPath Queries

Since we use ordered XML trees, each query evaluated over a tree T returns a set of nodes ordered using document order [CD99]. Given an ordered set of nodes N, $N[i]$ returns the i^{th} node in N. In all what follows, we use for simplicity node set instead ordered set of nodes. Along the

same lines as [GKP02, KMR09], we define the semantic functions $\mathcal{S}$ and ξ for the evaluation of XPath queries and XPath predicates respectively. Intuitively, given a node set N of T, $\mathcal{S}[\![p]\!](N)$ gives all the nodes of T that are reachable from a node of N using the XPath query p. The $\xi[\![q]\!]$ function evaluates the qualifier q over T and returns all nodes that satisfy q. By $\varphi[\![c]\!](N)$ we denote the function that returns all nodes of N whose text value is equal to c.

Given an XPath axis α, we use $n\ \alpha\ m$ to say that the node m is reachable according to α from the node n. We refer by $\alpha(N)$ to all the nodes that are reachable according to α from a node in N. In other words, $\alpha(N)$=$\{m \in T \mid n\ \alpha\ m\ for\ n \in N\}$. By $T(\eta)$ we refer to all nodes of T that correspond to node test η. More specifically, $T(\eta)$ is defined as follows:

$$T(\eta) = \begin{cases} \{m \in T\} & if\ \eta\ =\ * \\ \{m \in T \mid \lambda(m) = A\} & if\ \eta\ is\ an\ element\ type\ A \\ \{m \mid \lambda(m) = \texttt{str}\ \} & if\ \eta\ is\ the\ text()\ function \end{cases}$$

The complete semantics of $\mathcal{X}^{\Uparrow}_{[n,=]}$ queries are given in Table 3.1 (inspired from [GKP05,KMR09]).

Definition 3.12 (XPath equivalence). Two $\mathcal{X}^{\uparrow}_{[n,=]}$ queries q_1 and q_2 are equivalent, denoted by $q_1 \equiv q_2$, if and only if: for any XML tree T: $\mathcal{S}[\![q_1]\!](T)$=$\mathcal{S}[\![q_2]\!](T)$. Moreover, if two $\mathcal{X}^{\uparrow}_{[n,=]}$ predicates f_1 and f_2 are equivalent then: for any node n of T, $n \vDash f_1$ if and only if $n \vDash f_2$. In other words, $\xi[\![f_1]\!](\{n\})$=$\xi[\![f_2]\!](\{n\})$. □

Let α be an axis of the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ that can be ε, $\downarrow$, $\downarrow^+$, $\downarrow^*$, $\uparrow$, $\uparrow^+$, $\uparrow^*$. We define the inverse axis of α, denoted α^{-1}, for the previous cases respectively by: ε, $\uparrow$, $\uparrow^+$, $\uparrow^*$, $\downarrow$, $\downarrow^+$, $\downarrow^*$. We define in the following some properties that we use in the rest of this manuscript (the proof is given in Annex A.1).

Property 3.1. Let T be an XML tree with *root* node. We define some equivalences between $\mathcal{X}^{\Uparrow}_{[n,=]}$ queries as follows:

1. If $q_1 \equiv q_2$ then $q_1[f] \equiv q_2[f]$.
2. $\alpha{::}\eta[f_1][\varepsilon{::}*[f_2]] \equiv \alpha{::}*[f_1][\varepsilon{::}\eta[f_2]] \equiv \alpha{::}\eta[f_1 \wedge f_2]$.
3. For any $\mathcal{X}^{\Uparrow}$ predicates f_1 and f_2: $\downarrow^*{::}\eta_1[f_1][\alpha{::}\eta_2[f_2]] \equiv \downarrow^*{::}\eta_2[f_2]/\alpha^{-1}{::}\eta_1[f_1]$.
4. $\alpha_1{::}\eta_1/\ldots/\alpha_k{::}\eta_k \equiv \downarrow^*{::}\eta_k[\alpha_k^{-1}{::}\eta_{k-1}/\ldots/\alpha_2^{-1}{::}\eta_1/\alpha_1^{-1}{::}root]$.
5. $m \in \mathcal{S}[\![\downarrow^*{::}\eta[f]]\!](T)$ if and only if $\xi[\![\varepsilon{::}\eta[f]]\!](\{m\})$=$\{m\}$. □

The proof of this property is given in Section A.1 of Appendix A.

3.4 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):

$$\epsilon \cup \downarrow{::}A \cup \downarrow{::}A/\downarrow{::}A \cup \downarrow{::}A/\downarrow{::}A/\downarrow{::}A \cup \ldots$$

Figure 3.5: Sample XML document.

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 [FYL+09, KCKN04]). The major concern here is that XPath 1.0 [CD99] and XPath 2.0 [BBC+10] do not support transitive closure, and thus arbitrary recursive paths are not expressible in these languages [tC06]. An attempt was done with the SAXON processor (version 6.5.5, released on 2005) to implement the transitive closure operator, however, all next versions[21] do not support this extension.

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 [Mar04] to know more about the theoretical properties of XPath 1.0 extended with transitive closure and conditional axis. Conditional axis for instance can have the form *do step while test* is valid at the resulted node. The conducted study is based on the Core XPath fragment [GKPS05b], four different extensions of this fragment are defined by restricting use of axes relations and filters, as well as by introducing conditional axes and regular path expressions. Based on the definitions done in [Mar04], our class of Regular XPath queries, denoted by $\mathcal{X}_{reg}$, is defined as follows:

[21] The latest version of Saxon is version 9.5, available at `http://saxon.sourceforge.net/`.

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

where $p*$ denotes an infinite repetition of the query p as explained above with example of the query $(\downarrow::A)*$.

Authors of [FGJK06,FGJK07] were the first to propose an efficient tool for the evaluation of Regular XPath queries. They have shown first that $\mathcal{X}_{reg}$ queries can be captured by a special class of MFAs (*Mixed Finite state Automatas*), namely, MFAs with the *split property*. Two algorithms `rewrite` and `HyPE` are proposed respectively for translating any $\mathcal{X}_{reg}$ query Q into its equivalent MFA M, and for the evaluation of the resulting automaton M over a given XML tree T. The translation and evaluation steps are done in the presence of a security view $V{=}(D_v, \sigma)$, therefore the founded theoretical results [FGJK07] are based on the size of σ and D_v. We try in the following to compute the complexity of evaluating $\mathcal{X}_{reg}$ queries independently to security views. For any access specification $S{=}(D, \mathtt{ann})$, we consider the security view $V{=}(D_v, \sigma)$ where $D_v{=}D$, and for any production rule $A \rightarrow P(A)$ and any B in $P(A)$, $\sigma(A, B){=}B$ (i.e. instances of D are entirely accessible). Given the above assumption and based on the complexities in [FGJK07], we get the following results:

Proposition 3.1. 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. □

Note.
Unfortunately, the system `SMOQE`, proposed in [FGJK06] for translation and evaluation of $\mathcal{X}_{reg}$ queries, is under improvements for additional research and no working version exists in the web. To the best of our knowledge, no practical tool exists today for evaluation of Regular XPath queries over XML data.

3.5 XML Update Operations

We review here some update operations of the W3C XQuery Update Facility recommendation [BCFF$^+$10]. We consider the following operations: *insert*, *delete*, *replace*, and *rename*. For each update operation, an XPath *target* expression is used to specify the set of XML nodes on which the update is applied. For *delete* operations, *target* specifies the XML nodes (denoted *target-nodes*) to be deleted. For the remaining operations however, *target* must specify a single XML node (denoted *target-node*), otherwise a dynamic error is raised. An additional argument, called *source* is required for *insert*, *replace* and *rename* operations which specifies, depending on the type of the operation, either a text value or a sequence of XML nodes.

In the following, names in brackets are abbreviations of operations. The update operations that we consider throughout this manuscript are detailed as follows[22]:

Insert For insert operations, the order defined between nodes of *source* must be preserved during the insertion. We distinguish different types of insert operations depending on the position of the insertion:

[22] We omit description that concerns attribute nodes (e.g. insertion of attribute nodes) since we do not consider attributes in our approach.

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

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

• **insert** *source* **into** *target* [*insertInto*]: Inserts the nodes in *source* as children of the single element node *target-node* (otherwise a dynamic error is raised). Note that the positions of the inserted nodes among the children of *target-node* are implementation-dependent [23]. Thus, the effect of executing an *insertInto* operation on *target-node* can be that of *insertAsFirst* / *insertAsLast* executed on *target-node*, or that of *insertBefore* / *insertAfter* executed at children of *target-node*.

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

Rename The operation "*rename target as source*" [*rename*] replaces the label of the single element node *target-node* (otherwise a dynamic error is raised) with the string value represented by *string-value*.

Replace We distinguish three types of replace operations:

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

• **replace value of** *target* **with** *string-value* [*replaceValue*]: *target-node* must evaluate to a single text node. This operation replaces the string value of *target-node* with *string-value*.

• **replace element-content of** *target* **with** *source* [*replaceElementContent*]: This operation replaces all children of the single element node *target-node* (otherwise a dynamic error is raised), together with its descendants, by the optional text node *source*. This operation will not be used in this work.

Example 3.6. Consider the XML tree of Figure 3.4 and assume that the missing children of nodes $project_1$, $project_2$, $project_3$ and $project_4$ (abbreviated by $\bigwedge$) are all the same. We define the following update operations:

1. **delete** $\downarrow^+::course[\neg\ (\downarrow^+::professor[\downarrow::pname='Wenfei'])]$
2. **delete** $\downarrow^+::scholarship$
3. **insert** <*student*><*sname*>toto</*sname*><*mark*>16</*mark*></*student*> **as first into** $\downarrow^+::students$
4. **delete** $\downarrow::dept/\downarrow::project/\downarrow::sub\text{-}project/\downarrow::project/\downarrow::sub\text{-}project/\downarrow^+::*$
5. **replace value of** $\downarrow^+::member[1]/\downarrow::name/text()$ **with** '*Wenfei*'
6. **replace** $\downarrow^+::sub\text{-}project/\downarrow^+::private$ **with** <*public*/>

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

7. **replace** $\downarrow^+::$*sub-project*/$\downarrow^+::$*name* **with** <*name*>Gabriel</*name*>

The first delete operation deletes information of any *course* that is not partially given by professor *Wenfei*, i.e. the subtree rooted at $course_2$ is deleted. The second one deletes all *scholarship* information, i.e. the subtree rooted at $scholarship_1$. After these two updates, $dept_1$ has only $course_1$ as child node which has only one *student* element. Given this element, i.e. $student_2$, the third operation inserts a new *student* element, with name '*toto*' and mark '*16*', in the preceding sibling of $student_2$. Consequently, $students_1$ is composed now by two children nodes $student_1$ and $student_2$ with names *toto* and *bono* respectively. The fourth operation deletes all descendants of the element $sub\text{-}project_2$. The resulting $dept_1$ element has only two *project* elements (elements $project_1$ and $project_2$ resp.). The fifth operation replaces the text value of the node $name_1$ (i.e. '*Marteen*') with '*Wenfei*'. The two last operations replace respectively the node $private_2$ with the new one *public*, and the node $name_2$ with another *name* element by changing its text value with '*Gabriel*'. □

We denote by $op(T)$ the XML tree resulted by performing an update operation *op* over an XML tree T. Our set of update operations is more general than the one used in [BCF07] while only atomic updates are used [24]. To simplify our proposals, however, we refer with *source* throughout this manuscript to a sequence of XML nodes with only the same type.

[24] I.e. the argument *source* of *insert* and *replace* operations must represent only a single XML node.

4

Secure XML Data with Security Views

Contents

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

time in the size of the query. An experimental evaluation demonstrates that our algorithm is efficient and scales well.

4.1 Problem Statement

We present in this section the basic problem we tackle, namely answering XML queries over recursive security views. Firstly, we give some definitions of security views and access specifications, as well as a formulation of the problem. A sketch of our solution is given at the end of this section.

4.1.1 XML Security Views

The notion of *security view*, introduced first by [SF02a], 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. [FCG04a] 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 [FCG04a,FGJK06, FGJK07,Ras06,DFGM08,DZ08b,KMR09,GSC+09,LLLL11,TTL13].

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 [FCG04a], 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 4.1 (**Access Specification** *[FCG04a]*)**.** An *access specification* S is a pair (D, $\boldsymbol{ann}$) consisting of a DTD D and a partial mapping $\boldsymbol{ann}$ such that, for each production rule $A \rightarrow P(A)$ and each element type B in $P(A)$, $\boldsymbol{ann}(A, B)$, if explicitly defined, is an annotation of the form:

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

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 $\boldsymbol{ann}(A, B)$ is not explicitly defined, then B inherits the accessibility of A. On the other hand, if $\boldsymbol{ann}(A, B)$ is explicitly defined then B may *override* the accessibility inherited from A.

Example 4.1. We consider the *department* DTD of Example 3.1 and we define some access privileges for professors. Assume that a professor, identified by his name $\$\text{PNAME}$, 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 ***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 ***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 ***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 4.1 are enforced through the derivation of a *security view* [FCG04a]. 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[25]. More formally, an XML security view is defined as follows:

Definition 4.2. Given an access specification S=(D, ***ann***) defined over a non-recursive DTD D, a *security view* V is a pair (D_v, σ) 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. □

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 4.2. Consider the access specification S=(*dept*, ***ann***) of Example 4.1. Firstly, the DTD view $dept_v$=(Σ_v, *dept*, P_v) of the *department* DTD can be computed easily 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:

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

The function σ is defined over the production rules of $dept_v$ as follows: (refer to Example 4.1 for the definition of $[Q_1]$)

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

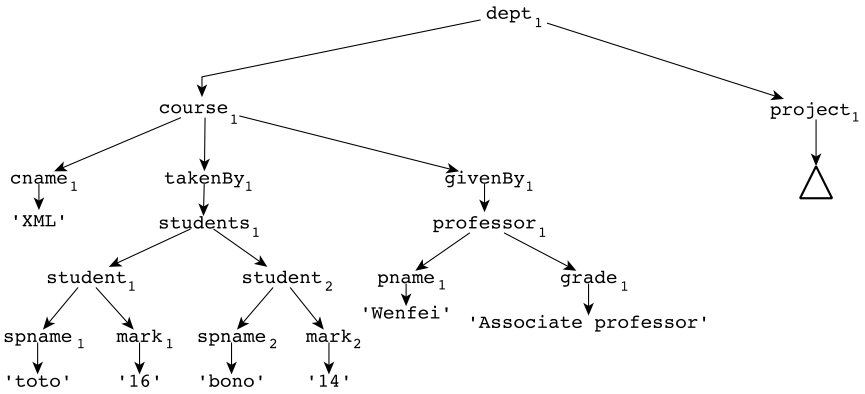


Figure 4.1: The view of the *dept* XML document w.r.t the policy of Example 4.1.

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

Using the resulting security view V=($dept_v$, σ), the view of the XML document of Figure 3.4 is derived and depicted in Figure 4.1, 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 [KMR09,Feg11], or keep T_v virtual for some reasons [FGJK07, DFGM08, GSC+09, LLLL11]. 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 4.3. Consider the query $\downarrow::dept/\downarrow::course$ of the professor $Wenfei$ defined over the view of Figure 4.1. This query can be rewritten, using the security view of Example 4.2, as follows:

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

The evaluation of this query over the original XML document of Figure 3.4 returns only accessible *course* elements, i.e. $course_1$. □

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

4.1.2 Security View's Drawbacks

Recall that security views should provide each group of users with a schema of all accessible data, extract virtual and/or materialized views [26] of the underlying XML data, and rewrite user queries over virtual views (if exist) to be evaluated over the original data. For non-recursive DTDs, several efficient solutions have been proposed in response to these needs. However, only few work has studied the use of security views in case of recursion. A security view is *recursive* if it is defined over a recursive DTD. This case deserves more attention since the emergence of the XML standard has spawned more complex data that, in different real-life scenarios, conform to recursive schemas [Cho02].

We study only the case of querying virtual XML data, then problems related to manipulation of materialized XML views [KMR09, Feg11] are outside the topic of interest of this work. More precisely, we discuss subsequently obstacles encountered when manipulating recursive DTDs and this at the stage of query rewriting. When 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. In case of recursive DTDs however, this function can not be defined in XPath as we show by the following example.

Example 4.4. We consider the *department* DTD of Example 3.1 and we assume that a personal of some department, identified by his name $\$\text{PNAME}$, 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:

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

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 Figure 3.4 is derived and depicted in Figure 4.2. 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, $\sigma(\textit{dept}, \textit{project})$ can be defined using the transitive closure operator "$*$" as follows:

$$\sigma(\textit{dept}, \textit{project}) = (\downarrow::\textit{project}[\neg(Q_2)]/\downarrow::\textit{sub-project})*/\downarrow::\textit{project}[Q_2]$$

The recursive path $(\downarrow::\textit{project}[\neg(Q_2)]/\downarrow::\textit{sub-project})*$ is defined over only inaccessible elements. Thus, the expression $\sigma(\textit{dept}, \textit{project})$ 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 Figure 4.2 as an immediate child of some *dept* element n if and only: m and n are both accessible in the original tree of Figure 3.4, and either m is an immediate child of n or separated from n with only inaccessible elements. Take the case of the elements $\textit{dept}_1$ and $\textit{project}_2$ of the tree of Figure 3.4. After hiding the inaccessible element $\textit{project}_1$, $\textit{project}_2$ appears in the view of Figure 4.2 as immediate child of $\textit{dept}_1$. The same principle is applied for the elements $\textit{project}_2$ and $\textit{project}_4$. □

[26] [TTL13] proposed hybrid approach that defines both virtual and materialized views of the same XML document.

[Mar04] showed that the kleen star operator "$*$" can not be expressed in XPath. For this reason, the function σ of Example 4.4 can not 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). Contrary to [FGJK07], we define formally the *closure property* as follows:

Definition 4.3. 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,$\boldsymbol{ann}$) 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, $\mathcal{S}[\![\mathcal{R}(Q)]\!](T)$=$\mathcal{S}[\![Q]\!](T_v)$. □

The definition given by [FGJK07] is not precise where the closure property of XPath is checked over security views. As we have shown however, security views can not be always derived, i.e. problem for defining DTD views and the function σ. Moreover, given an access specification S=(D, $\boldsymbol{ann}$) and its security view V=(D_v, σ). Since they impose no restriction for the definition of the σ function, this latter may be defined in a language that is more expressive than XPath (e.g. Regular XPath). Thus, it is clear that by using this view V some complex rewritten queries may not be presented in XPath. For this reason we refine the definition of closure property [FGJK07] to be checked over access specifications.

Note that [FCG04a] shown that the fragment $\mathcal{X}$ (defined in Section 3.10) 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 4.1 ([FGJK07,Jia07])**.** In case of recursive XML security views, the XPath fragment $\mathcal{X}$ is not closed under query rewriting. □

Proof 4.1. Assume by contradiction that the fragment $\mathcal{X}$ is closed under query rewriting, i.e. for any access specification, any query of the fragment $\mathcal{X}$ can be safely rewritten in $\mathcal{X}$. We consider the access specification of Example 4.4 and we define the XPath query Q=$\downarrow$::*dept*/$\downarrow$::*project*. According to the specification of this example, the query Q over the data view of Figure 4.2 must return only the element $project_2$ that is accessible to the professor *Wenfei* and appears as immediate child of element $dept_1$. Using the function σ as defined in Example 4.4, Q is translated into $\downarrow$::*dept*/σ(*dept*, *project*) where σ(*dept*, *project*) must represent all the paths that connect accessible elements *project* with accessible element $dept_1$. Each of these paths must have the form $n_1, \ldots, n_k$ ($k \geq 2$) where n_1 is $dept_1$, n_k is an accessible element of type *project*, and the remaining elements are all inaccessible, in this way n_k (i.e. $project_2$) appears in the data view as immediate child of $dept_1$ after hiding inaccessible elements that are between n_1 and n_k. Due to the cycle *project* $\rightarrow$ *sub-project* $\rightarrow$ *project* presented in the *department* DTD of Figure 3.1, we have seen that σ(*dept*, *project*) must be defined in Regular XPath in order to capture the infinite set of paths that connect accessible *project* elements into $dept_1$ element. The query Q is translated into:

$$\downarrow\text{::}dept/(\downarrow\text{::}project[\neg(Q_2)]/\downarrow\text{::}sub\text{-}project)*/\downarrow\text{::}project[Q_2]$$

The resulting query Q' over the data view of Figure 4.2 returns only the element $project_2$. According to [[Tho84, Mar04], a query of the form $(q_1/q_2)*$ is not expressible in XPath. Thus, Q' cannot be expressed in XPath which contradicts our assumption. We conclude that the XPath fragment $\mathcal{X}$ is not closed under query rewriting. □

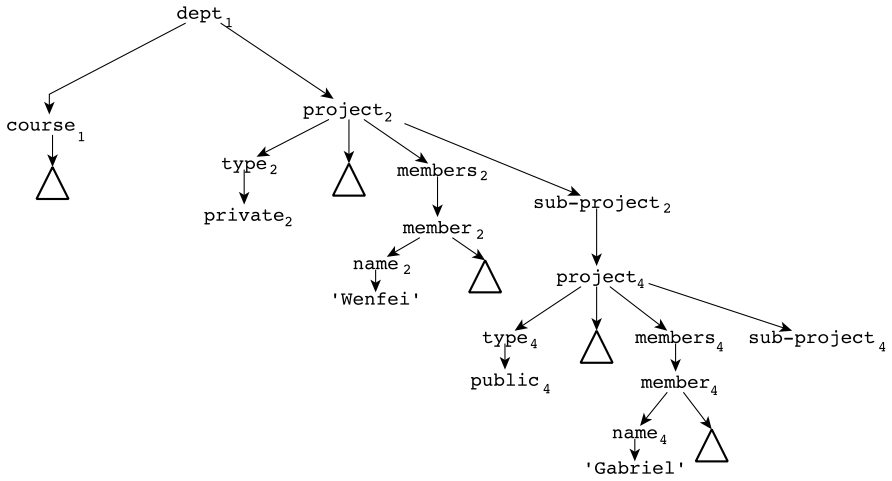


Figure 4.2: The view of the *dept* XML document w.r.t the policy of Example 4.4.

Note.
The proof of Theorem 4.1 given in [Jia07] is based on the definition of some security view for which the fragment $\mathcal{X}$ is not closed. However, the proof remains incomplete since no access specification can be assigned to the security view considered. In other words, as we have done, the proof must show that there exists an access specification (i.e. an access policy defined via the principle of schema annotation, Definition 4.1) for which query rewriting is not always possible in $\mathcal{X}$. While in [Jia07], the policy given as counter example cannot be specified using DTD annotation principle.

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 [FGJK07, GSC+09], the fragment $\mathcal{X}_{reg}$ of Section 3.4 is closed under query rewriting. However, some major drawbacks are to be noted: no standard solution exist 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.

4.1.3 Sketch of our Proposal

The major contribution of this work is a solution that makes possible querying of XML data using arbitrary security views (recursive or not). We make a brief comparison of our solution with some Regular XPath based solution that is still inefficient and impractical in general.

Several commercial database systems (e.g. Oracle 11g, Microsoft SQL Server, eXist-db, Sedna) are aware of the XML data structure and offer supports to efficiently manage business application data in XML format. They provide use of most W3C standards and particularly the XPath language [BBC+10]. This has motivated us to find an XPath-based solution for the query rewriting problem and that could be easily and efficiently integrated within such systems. In a nutshell, our solution consists in defining a rewriting function $\mathit{Rewrite}\colon \mathcal{X} \to \mathcal{X}^{\Uparrow}_{[n,=]}$ that, for any access specification $S{=}(D, \mathit{ann})$ and any XML tree $T \in \mathcal{T}(D)$, translates any $\mathcal{X}$ query

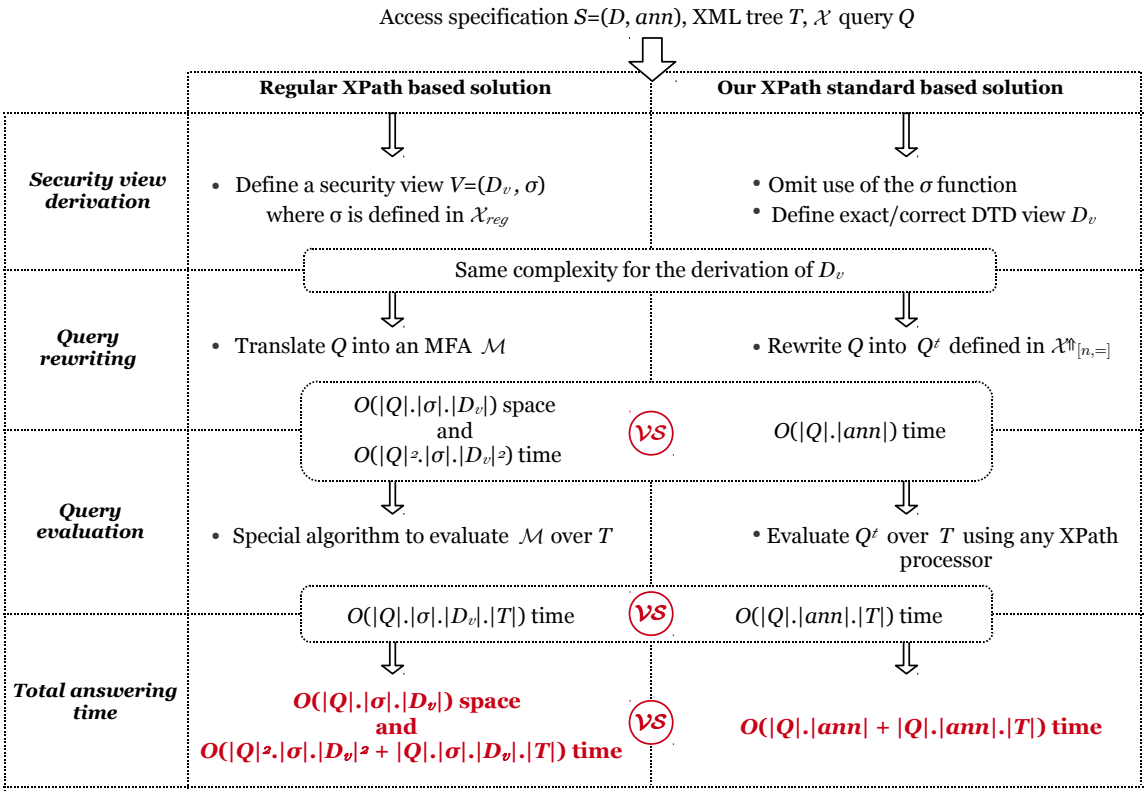


Figure 4.3: Comparing our solution with that of [FGJK07].

Q defined over the virtual view T_v of T into another one $\mathtt{Rewrite}(Q)$ defined in the extended fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ such that: $\mathcal{S}[\![\mathtt{Rewrite}(Q)]\!](T)=\mathcal{S}[\![Q]\!](T_v)$.

Note that one can use Regular XPath [Mar04] to overcome the query rewriting limitation as has been shown first by [FGJK07] and refined later by [GSC+09]. However, these approaches are costly to implement as we show subsequently. Figure 4.3 compares our solution, based on the XPath standard, with that of [FGJK07] 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 [FGJK07]. The comparison is done according to each step an XML access control model must follow:

1. ***Access policies specification.*** Along the same lines as [FCG04a], we present a general and expressive access specification language (see Section XX) that combines advantages of the languages presented in [FGJK07,KMR09] and overcomes their limits.

2. ***Security view derivation.*** This step consists in defining a security view $V=(D_v, \sigma)$ for some access specification $S=(D, \mathtt{ann})$. As we have shown, the σ function can not be defined in XPath if D is recursive. For this reason, we omit the definition and the use of such function in our approach. [FGJK07] claimed that the σ function they use is defined in Regular XPath and this is the basic idea of their solution. Moreover, all their theoretical results [FGJK07, Jia07] are based on the size of such function (i.e. $|\sigma|$). However, no algorithm is proposed for its construction which makes hard the comparison of their final results with those of recent solutions. We note finally that the derivation of the view D_v is problematical, thus we discuss only the use of some classical solutions for this problem.

3. *Query rewriting.* [FGJK07] translate an $\mathcal{X}$ query Q in input into a *mixed finite state automata* (MFA) $\mathcal{M}$. This translation requires $O(|Q|.|\sigma|.|D_v|)$ space and $O(|Q|^2.|\sigma|.|D_v|^2)$ time. On the other hand, we leverage the expressive power of the standard XPath to translate Q into another one Q^t, defined in the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$, and this in $O(|Q|.|\boldsymbol{ann}|)$ time. Note that the size of Q^t is bounded by $O(|Q|.|\boldsymbol{ann}|)$.

4. *Query evaluation.* According to the Theorem 4.4 of Section 3, any $\mathcal{X}^{\Uparrow}_{[n,=]}$ query can be evaluated in a linear time w.r.t the size of the XML data and the XPath query. Thus, our rewritten query Q^t can be evaluated over the XML tree T in $O(|Q^t|.|T|)$ time. On the other hand, since no algorithm has been proposed to evaluated Regular XPath queries, [FGJK07] proposed an algorithm, called `HyPE`, that evaluates an MFA $\mathcal{M}$ over an XML tree T in at most $O(|\mathcal{M}|.|T|)$ time.

Summing up. Given an access specification S=(D, $\boldsymbol{ann}$), an XML tree $T \in \mathcal{T}(D)$, and consider an $\mathcal{X}$ query Q 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 $O(|Q|.|\boldsymbol{ann}|.|T|)$ time, while [FGJK07] 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 $|\boldsymbol{ann}|$ is bounded by $O(|D|^2)$ (i.e. we can define at most $|D|^2$ annotations). However, the size of the function σ is more larger in general 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 4.5. 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:

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

We define now the access specification S=(D, $\boldsymbol{ann}$) where $\boldsymbol{ann}$ contains only the default annotation $\boldsymbol{ann}(Root)$=Y, i.e. all element types of D are accessible. It is clear 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 that of [FGJK07] 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 as we shall verify through an empirical study based on real-life DTDs. 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 [MN10b, GCV11] and efficient evaluation [HL13b]).

4.2 Access Control with Arbitrary DTDs

Figure 4.4 presents our XML access control framework. It is designed particularly for native XML databases [Bou10] where XML data is stored in its native format. The data our system is supposed to protect consists of a collection of XML documents and their corresponding DTDs. The 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

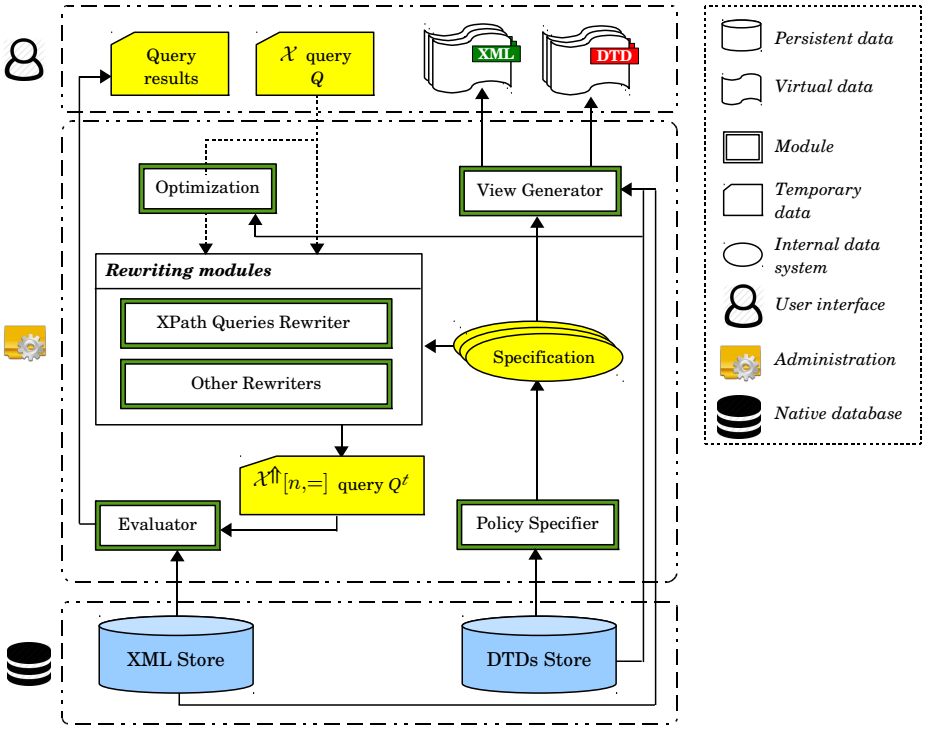


Figure 4.4: XML Access Control Framework.

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 XML node referred to by the query in input. The role of the *optimization* module is discussed later.

We present in the following the *hospital DTD* that corresponds to a real-life patient medical data [Sha12] and which is used throughout the rest of this manuscript.

Example 4.6. To facilitate comprehension, we recall here the hospital DTD and the hospital data presented at the beginning of this manuscript. 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 intervention *date*, 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

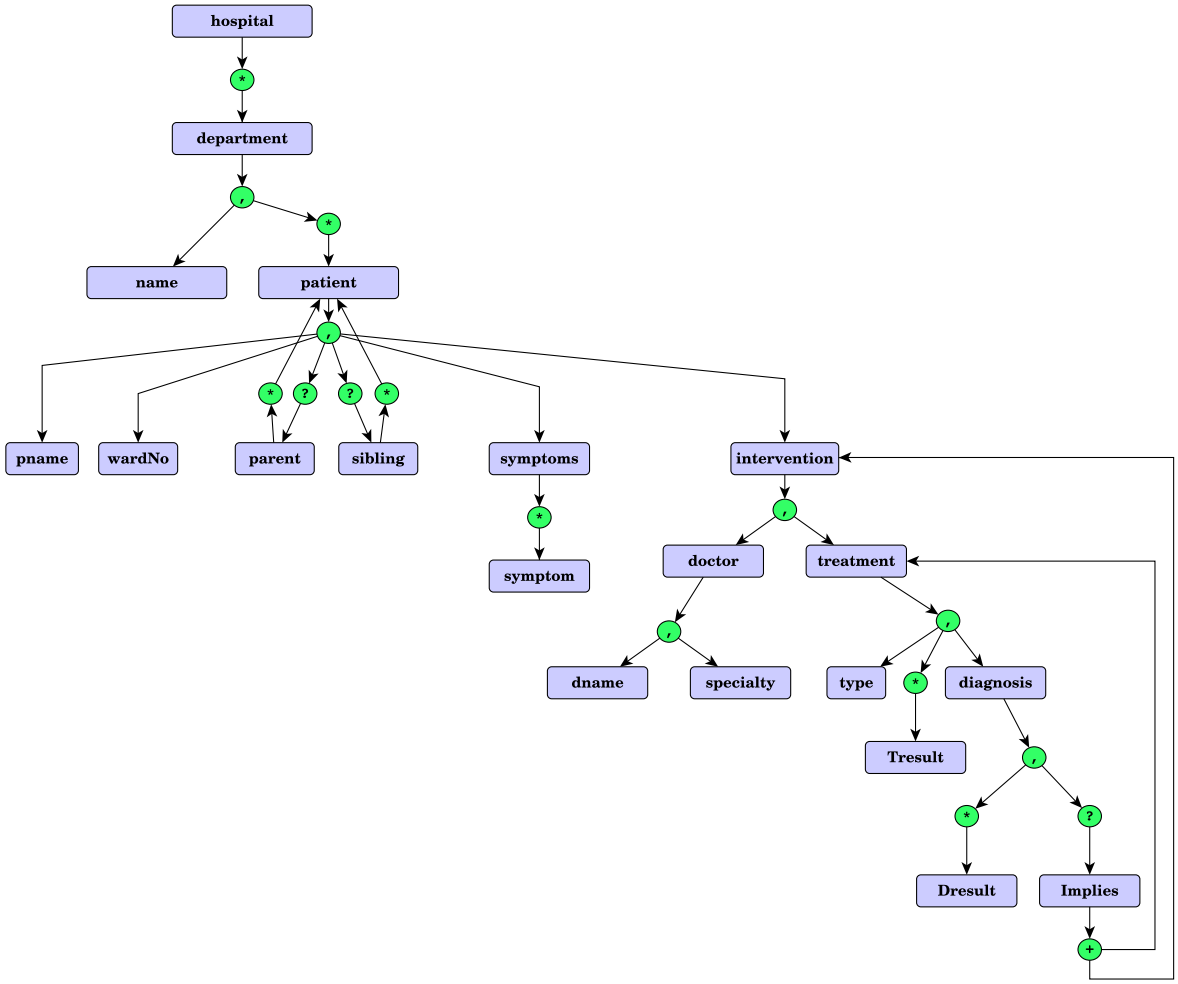


Figure 4.5: The hospital DTD.

solicited[27]. The DTD graph of our hospital DTD is depicted in Figure 4.5. An instance of this hospital DTD is given in Figure 4.6 (some text contents are abbreviated by '...'). □

4.2.1 Access Specification

[FCG04a] proposed the first language for the specification of XML access control policies through annotation of DTD grammars. Moreover, [KMR09] 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 4.1). Although both access specification languages defined in [FCG04a, KMR09] 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:

[27] According to [Sha12], this may happen when the required treatment is outside the area of expertise of the current responsible doctor.

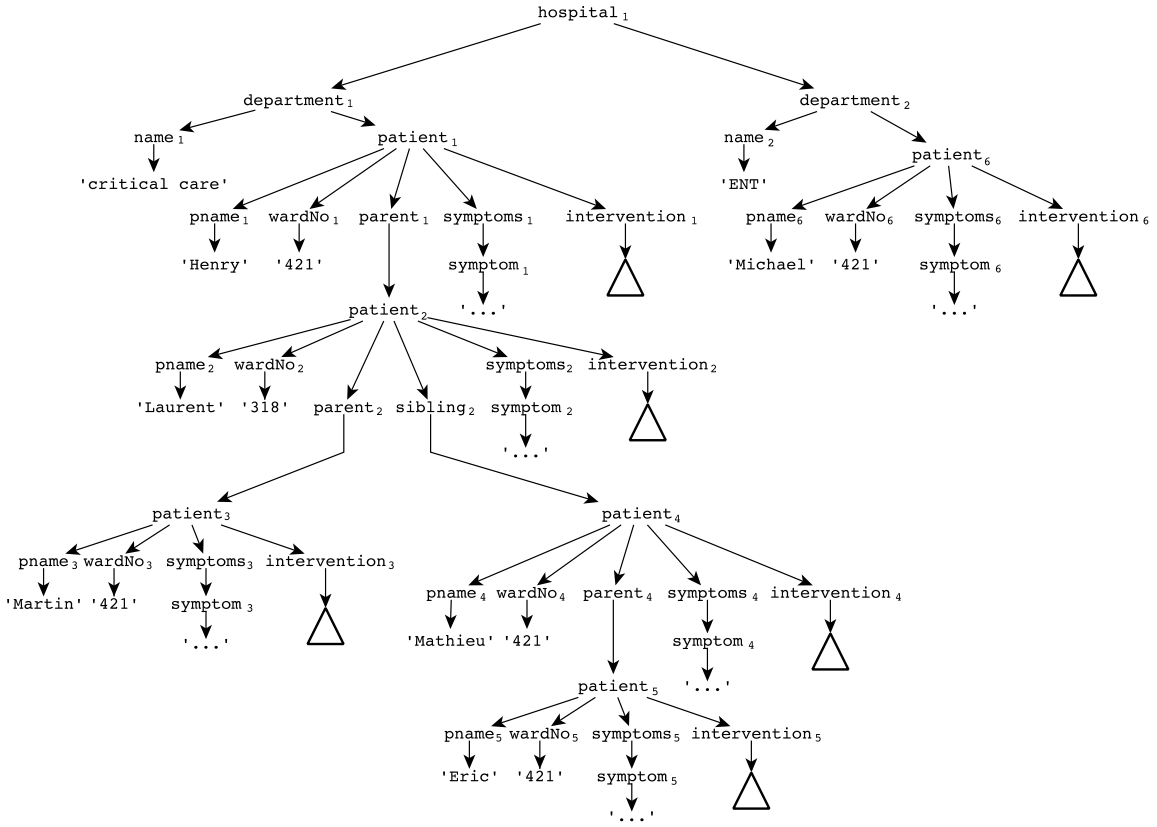


Figure 4.6: Example of Hospital data.

Example 4.7. We suppose that there are two annotations $\boldsymbol{ann}(A, B)$=[¬ (↓::D)] and $\boldsymbol{ann}(C, D)$=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 [¬ (↓::D)] is invalid at the element node B. According to [FCG04a], 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 [KMR09] 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 $n \not\models Q$ 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 $n \not\models Q$, 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 as we show later. For this reason, we present a refined and more expressive access specification language whose access specifications are defined as follows:

Definition 4.4 (**Extended Access Specification**). We define an *access specification* S as a pair (D, $\boldsymbol{ann}$) consisting of a DTD D and a partial mapping $\boldsymbol{ann}$ such that, for each production

Access policies	Required specification values					Remark
	Y	N	N_h	$[Q]$	$[Q]_h$	
[FCG04a,FGJK06,FGJK07]	✓	✓			✓	
[KMR09]	✓	✓		✓		case of top-down policies
[GSC+09]	✓	✓		✓		
[DFGM08]	✓		✓		✓	
[LLLL11]	✓	✓				
[FM04]	✓		✓		✓	*deny overwrites* as the conflict resolution policy
[MTKH06]	✓		✓		✓	with *denial downwards* consistency requirement

Table 4.1: Current approaches' policies specified with our language.

rule $A \rightarrow P(A)$ and each element type B in $P(A)$, $\mathit{ann}(A, B)$, if explicitly defined, is an annotation of the form:

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

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 4.1 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 $\mathit{ann}(A_1, A_2){=}N_h$ 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 $\mathit{ann}(A_i, A_{i+1}){=}Y|[Q]$ is explicitly defined then the element node n_{i+1} remains inaccessible even if $n_{i+1} \vDash Q$. However, the annotation $\mathit{ann}(A_1, A_2){=}[Q_2]_h$ indicates that annotations defined over descendant types of A_2 take effect only if Q_2 is valid. In other words, given the annotation $\mathit{ann}(A_i, A_{i+1}){=}Y$ (resp. $[Q_{i+1}]$), the element node n_{i+1} is accessible if and only if: $n_2 \vDash Q_2$ (resp. $n_2 \vDash Q_2 \wedge n_{i+1} \vDash Q_{i+1}$).

Note.
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 4.1.

Example 4.8. 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

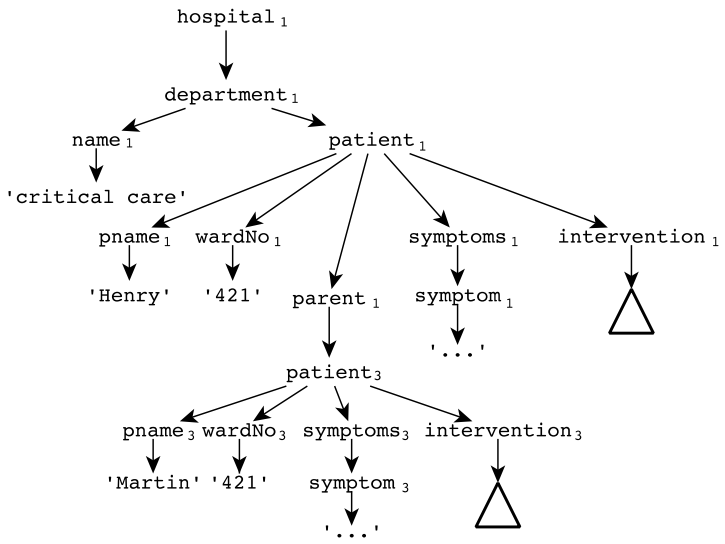


Figure 4.7: View of the tree of Figure 4.6 computed w.r.t the specification of Example 4.8.

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 $S{=}(D,\ \boldsymbol{ann})$ where D is the hospital DTD and the function $\boldsymbol{ann}$ defines the three following annotations:

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

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

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

According to this specification, the view of the data of Figure 4.6 is extracted and depicted in Figure 4.7. 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 $departement_2 \not\models Q_1$, then the annotation R_2 can not be applied at $patient_6$ element which remains inaccessible even with $patient_6 \models Q_2$. Notice that $departement_1 \models Q_1$ 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 $patient_2 \not\models Q_2$) 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 Figure 4.7 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 can not take effect over the elements $patient_4$ and $patient_5$. □

We emphasize that the policy of Example 4.8 can not be specified in the fragment $\mathcal{X}$ using the specification languages presented in [FCG04a, KMR09]. 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 [SdV00] 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 4.4, we define the notions of *completeness* and *consistency*, along the same lines as [FCG04a, KMR09], as follows:

Definition 4.5. Given an access specification $S{=}(D,\ \boldsymbol{ann})$ and an XML tree $T \in \mathcal{T}(D)$, then, we say that S is *complete* and *consistent* if and only if the *accessibility* of each node in T is *uniquely* defined, i.e. it is either *accessible* or *inaccessible*. □

Proposition 4.1. The access control policies based on Definition 4.4 are *complete* and *consistent*. □

Proof 4.2. Authors of [KMR09] 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. □

4.2.2 Accessibility

The enforcement of our access control policies relies principally on the definition of *node accessibility*. Inspired from [FM04, GSC+09], 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 4.6. Let n be an B element that is child of an A element. A given annotation $\boldsymbol{ann}(A,\ B)$ is *valid* at n if and only if $\boldsymbol{ann}(A,\ B){=}Y|[Q]|[Q]_h$ with $n \vDash Q$. Otherwise, it is *invalid*, i.e. $\boldsymbol{ann}(A,\ B){=}N|N_h|[Q]|[Q]_h$ with $n \nvDash Q$. □

If $\boldsymbol{ann}(A,\ B){=}[Q]_h$ with $n \vDash Q$ (resp. $\boldsymbol{ann}(A,\ B){=}N_h|[Q]_h$ 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 4.7. Let $S{=}(D,\ \boldsymbol{ann})$ 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 $\boldsymbol{ann}(A,\ B)$ 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 4.7 refers to one of the following three cases:

a) Only the default annotation $\boldsymbol{ann}(A_1){=}Y$ 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 $\boldsymbol{ann}(A_{k-1}, A_k)$ is explicitly defined and valid at n_k.

c) The annotation $\boldsymbol{ann}(A_{i-1}, A_i)$ 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 4.7 implies that for any downward-closed annotation $\boldsymbol{ann}(A_{i-1}, A_i)$ defined over ancestor n_i of n_k (with $1 < i < k$), either $\boldsymbol{ann}(A_{i-1}, A_i) \neq N_h$ or $\boldsymbol{ann}(A_{i-1}, A_i) = [Q]_h$ with $n_i \vDash Q$. Finally, note that a text node is accessible if and only if its parent element is accessible.

Definition 4.8. Given an access specification $S=(D, \boldsymbol{ann})$, we define two $\mathcal{X}^{\Uparrow}_{[n]}$ predicates $\mathcal{A}_1^{acc}$ and $\mathcal{A}_2^{acc}$ as follows:

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

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

The predicates $\mathcal{A}_1^{acc}$ and $\mathcal{A}_2^{acc}$ satisfy the conditions (i) and (ii) of Definition 4.7 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**). While 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. $N \subseteq \{n\} \cup \boldsymbol{ancestors}(n)$[28], and $\forall m \in N$, m is of type B and has a parent node of type A where $\boldsymbol{ann}(A, B)$ 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 $\boldsymbol{ancestors}(n)$ are valid (condition *(ii)*).

Lemma 4.1. Given an access specification $S=(D, \boldsymbol{ann})$, 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 $n \vDash \mathcal{A}^{acc}$. □

The proof of Lemma 4.1 is given in Annex. According to this lemma, for any access specification $S=(D, \boldsymbol{ann})$ 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.

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

Example 4.9. Consider the access policy of nurses defined in Example 4.8 with the following annotations:

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

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

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

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

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

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

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

Consider the case of the element $patient_1$ of Figure 4.6. 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 Figure 4.6 returns all the accessible elements that compose the view of Figure 4.7. □

4.3 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 3.10) 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 Section 4.1.2. More precisely, the access control policies based on Definition 4.4 are enforced through a rewriting technique. Let $S=(D, \mathit{ann})$ 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 $\mathit{Rewrite}$ such that:

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

Fundulaki et al. [FM04] proposed an XML access control approach where access policies are specified through a set of XPath expressions. The rewriting technique they proposed ensures that only accessible data can be returned by user queries. This is not sufficient since there approach allows disclosure of sensitive information as we show by the following example.

Example 4.10. We consider the access control policy of nurses presented in Example 4.8 and we define the following XPath query:

$$\downarrow^*::patient[\downarrow::pname="Henry"][\downarrow::parent/\downarrow::patient[(\downarrow::pname="Laurent") \wedge (\downarrow::wardNo="318")]]$$

This query is accepted by the approach proposed in [FM04] since when evaluated over the XML tree of Figure 4.6 returns only the accessible element $patient_1$. However, this discloses some sensitive information that is not shown at the view of Figure 4.7: the nurse deducts that the patient *Laurent*, whose information is not accessible for her, is residing at the ward *318*. □

For this reason, we make sure first that our rewriting function `Rewrite` ensures that only accessible data is returned by any user query. Moreover, contrary to Fundulaki et al. [FM04], we safely examine all intermediate parts of the query to overcome disclosure of sensitive data. For instance, the query of the previous example should be rejected.

4.3.1 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 4.11. Consider the XML document of Figure 4.6 and its view depicted in Figure 4.7 that is computed w.r.t the access policy of Example 4.8. 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 4.9. Obviously, this rewritten query selects first accessible *departement* elements of Figure 4.6, 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* element types of the hospital DTD, this latter query cannot be rewritten using only the accessibility predicate. More precisely, $\downarrow^*::parent[\mathcal{A}^{acc}]/\downarrow::*[\mathcal{A}^{acc}]$ 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 4.9. Given an access specification S=(D, $\mathtt{ann}$) and an element type B, then we define two $\mathcal{X}^{\Uparrow}_{[n]}$ predicates $\mathcal{A}^+$ and $\mathcal{A}^B$ as follows:

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

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, $\mathtt{ann}$), we define the function $\mathtt{Rewrite}$: $\mathcal{X} \longrightarrow \mathcal{X}^{\Uparrow}_{[n]}$ 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:

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

The qualifier $prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_k::\eta_k)$ presents a recursive rewriting in a descendant manner where each subquery $\alpha_i::\eta_i$ is rewritten over all the subqueries that precede it in the query Q. In other words, for each subquery $\alpha_i::\eta_i$ ($1 \leq i \leq k$), $prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_{i-1}::\eta_{i-1})$ is already computed and used to compute $prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_i::\eta_i)$ as follows: [29]

- $\alpha_i = \downarrow$:
  $prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_i::\eta_i) := \mathcal{A}^{\eta_{i-1}}[prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_{i-1}::\eta_{i-1})]$

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

- $\alpha_i = \varepsilon$:
  $prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_i::\eta_i) := \varepsilon::\eta_{i-1}[prefix^{-1}(\alpha_1::\eta_1/\cdots/\alpha_{i-1}::\eta_{i-1})]$

As a special case, the first subquery 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 4.12. Let us consider the query Q=$\downarrow^*::parent/\downarrow::*$ of Example 4.11 posed over the data view of Figure 4.7. By considering the access specification of Example 4.8, this query can be rewritten as follows: $\mathtt{Rewrite}(Q)$=$\downarrow^*::*[\mathcal{A}^{acc}][\mathcal{A}^{parent}]$. By replacing $\mathcal{A}^{parent}$ with its value, $\mathtt{Rewrite}(Q)$ is given by: $\downarrow^*::*[\mathcal{A}^{acc}][\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::parent]$. Recall that the definition of the predicate $\mathcal{A}^{acc}$ w.r.t the access specification of Example 4.8 is given in Example 4.9. The evaluation of the query $\downarrow^*::*[\mathcal{A}^{acc}]$ over the original document of Figure 4.6 returns a node set N composed by all the accessible nodes depicted in Figure 4.7. 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 $\mathtt{Rewrite}(Q)$ over the original document of Figure 4.6 returns only the element $patient_3$ as does the query Q over the data view of Figure 4.7. □

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

4.3.2 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:

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

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

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

Example 4.13. Consider the access specification of Example 4.8 and the data view of Figure 4.7. It is clear that the predicate $\underbrace{[\downarrow::\mathit{patient}/\downarrow::\mathit{wardNo} = \text{“}421''}_{P}]$ is satisfied only over the element node parent_1. This predicate is rewritten into $[\mathit{RW\_Pred}(P)]$ as follows:

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

Consider the XML document of Figure 4.6, 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 $\mathit{Rewrite}$ to take into account all queries of the fragment $\mathcal{X}$. Given an access specification S=(D, 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 $\mathtt{Rewrite}(Q_1) \cup \cdots \cup \mathtt{Rewrite}(Q_k)$.

Example 4.14. Consider the access specification of Example 4.8, the query $Q{=}\downarrow^+{::}parent/\downarrow{::}patient[\underbrace{\downarrow{::}\, pname = \text{“}Martin\text{”}}_{P}]$ over the data view of Figure 4.7 is rewritten over the original data of Figure 4.6 as follows:

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

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

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 $\mathtt{RW\_Pred}(P_1[\mathtt{RW\_Pred}(P_2)])$.

4.3.3 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 $S{=}(D,\ \mathtt{ann})$ 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 *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):

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

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

4.4 Rewriting Algorithm

Figure 4.8 presents our algorithm "*Rewrite*" for $\mathcal{X}$ queries rewriting. Given an access specification S=(D, *ann*) defined over arbitrary DTD (recursive or not), for any instance T of D, we extract the virtual view T_v of T that contains all the accessible nodes of T. For any query Q defined in $\mathcal{X}$ over T_v, our algorithm "*Rewrite*" translates it into an equivalent one Q_t defined in $\mathcal{X}^{\Uparrow}_{[n,=]}$ over the original document T such that, the evaluation of Q on T_v yields the same result as the evaluation of Q_t on T. We compute first the predicate $\mathcal{A}^{acc}$ w.r.t S as well as $\mathcal{A}^{+}$ whose definition is based on $\mathcal{A}^{acc}$. These predicates are constructed only one time after the definition of the access specification, then their construction time is negligible. Given an $\mathcal{X}$ query Q=$\alpha_1{::}\eta_1[P_1]/.../\alpha_k{::}\eta_k[P_k]$ $(k \geq 1)$, we compute the descending list of subqueries of Q. After, we parse these subqueries to define $prefix^{-1}(Q)$ where each subquery Q_i=$\alpha_i{::}\eta_i[P_i]$ is rewritten over Q_{i-1}=$\alpha_{i-1}{::}\eta_{i-1}[P_{i-1}]$ to ensure that only accessible nodes can be referred to by Q_i as well as to preserve the relationship defined between η_i and η_{i-1} (the first subquery Q_1 is rewritten over the *root* type of the DTD). It is clear to see in Figure 4.8 that each subquery Q_i is rewritten is a constant by manipulating the predicates $\mathcal{A}^{acc}$ and $\mathcal{A}^{+}$ already computed. For instance, the query Q=$\underbrace{\downarrow{::}*}_{Q_1}/\underbrace{\downarrow^+{::}\,B}_{Q_2}$ is rewritten in three steps as follows:

$$\underbrace{\downarrow^*{::}\,B[\mathcal{A}^{acc}][\underbrace{\uparrow^+{::}*[\mathcal{A}^{acc}]}_{Q_{2,1}}][\underbrace{\mathcal{A}^{+}[1]/\varepsilon :: root}_{Q_{1,root}}]}_{rewriting\ of\ Q}$$

We conclude that the rewriting of a query Q without predicates depends only on the parsing of all subqueries of Q. Note that the predicates P_i of the query Q in input are optional. The predicate P_i of each subquery Q_i, if exists, is rewritten using our predicate rewriting algorithm "*RW_Pred*" presented in Figure 4.9. In general, given the predicate f of the subquery $\alpha{::}\eta[f]$, we rewrite f to make tests over only accessible elements as well as to preserve the relationship defined between element types of f and the element type η. More precisely, let the predicate P_i be $p_1/.../p_m$ (where p_i=$\alpha_i{::}\eta_i$), we compute first the descending list of sub-predicates of P_i, after we parse this list and rewrite each sub-predicate in order to construct *RW_Pred*(P_i). Note that the first sub-predicate p_1 is rewritten according to the axis α_1 of p_1 and w.r.t the element type η_i of the subquery Q_i. However, for an intermediate sub-predicate $p_i/.../p_m$, p_i is rewritten according to the axis α_i of p_i and w.r.t the element type of p_{i-1}. The predicate *RW_Pred*$(p_i/.../p_m)$ is given by $\downarrow^+{::}\eta_i[\mathcal{A}^{acc}][$*RW_Pred*$(p_{i+1}/.../p_m)]/\mathcal{A}^{+}[1]$=$\varepsilon{::}*$, if α_i=$\downarrow$, or

Algorithm: *Rewrite*

input : An access specification S=(D, *ann*) and a query Q defined in $\mathcal{X}$.
output: Rewriting of Q w.r.t S.

```
1  compute the accessibility predicate A^acc w.r.t S;
2  if Q = Q1∪...∪Qn then
3  |  return ∪_{≤i≤n} Rewrite(Qi);
4  compute the descending list L of the subqueries Q1, ··· , Qk of Q (k ≥ 1);
5  each subquery Qi = αi::ηi[Pi];
6  prefix^-1 := ϵ; // presents prefix^-1(Q)

7  foreach Qi in the order of L do
8  |  case (αi =↓) :
9  |  |  if (prefix^-1 = ϵ) then // case of Q1
10 |  |  |  prefix^-1 := A^root;
11 |  |  else
12 |  |  |  prefix^-1 := A^{ηi-1}[prefix^-1];
13 |  case (αi ∈ {↓+,↓*}) :
14 |  |  if (prefix^-1 = ϵ) then // case of Q1
15 |  |  |  prefix^-1 := αi^-1::root;
16 |  |  else
17 |  |  |  prefix^-1 := αi^-1::ηi-1[A^acc][prefix^-1];
18 |  case (αi = ε) :
19 |  |  if (prefix^-1 = ϵ) then // case of Q1
20 |  |  |  prefix^-1 := αi::root;
21 |  |  else
22 |  |  |  prefix^-1 := αi::ηi-1[prefix^-1];
23 |  if (Pi-1 exists) then
24 |  |  prefix^-1 := prefix^-1[P^t_{i-1}];
   |  /* rewriting of predicate Pi                                    */
25 |  P^t_i := RW_Pred(Pi);
   /* rewritten query Qt of Q                                        */
26 if (Pk exists) then
27 |  Qt := ↓*::ηk[A^acc][P^t_k][prefix^-1];
28 else
29 |  Qt := ↓*::ηk[A^acc][prefix^-1]
30 return Qt;
```

Figure 4.8: $\mathcal{X}$ Queries Rewriting Algorithm.

by $\alpha_i{::}\eta_i[\mathcal{A}^{acc}][\mathit{RW\_Pred}(p_{i+1}/.../p_m)]$, in case of $\alpha_i \in \{\varepsilon, \downarrow^+, \downarrow^*\}$. For instance, the predicate $[\downarrow{::}A/\downarrow{::}B]$ is rewritten in $\mathcal{X}^{\Uparrow}_{[n,=]}$ into $[\downarrow^+{::}A[\mathcal{A}^{acc}][\downarrow^+{::}B[\mathcal{A}^{acc}]/\mathcal{A}^+[1]{=}\varepsilon{::}*]/\mathcal{A}^+[1]{=}\varepsilon{::}*]$. Recall that the predicates $\mathcal{A}^{acc}$ and $\mathcal{A}^+$ are added in a constant time. Thus, the rewriting of each sub-predicate P_i of a subquery Q_i depends only on the parsing of all sub-predicates of P_i.

As a conclusion, if we denote by $|Q|$ the size of all subqueries and sub-predicates of the query Q (e.g. $|\downarrow{::}*[\downarrow^+{::}*/\varepsilon{::}*]|$=3), then we deduct the following result:

Lemma 4.2. For any access specification S=(D, *ann*) and any $\mathcal{X}$ query Q defined over some virtual data view, our algorithm *Rewrite* translates Q into an equivalent one, defined over the original data, at most in time $O(|Q|)$. □

Algorithm: *RW_Pred*

input : An access specification S=(D, ***ann***) and a predicate P defined in $\mathcal{X}$.
output: A rewritten predicate P_t of P.

/* *text-comparaison is optional, if it does not exists then* [ε::*/*text*()='c'] *below is omitted* */

1 $P_t := false$;
2 **if** *(P is a single predicate α::η[F]/*text*()='c')* **then**
3 $F^t :=$ *RW_Pred*(F);
4 **if** *($\alpha = \downarrow$)* **then**
5 $P_t := \downarrow^+$::$\eta[\mathcal{A}^{acc}][F^t][\varepsilon$::*/*text*()='c']/$\mathcal{A}^+$[1]=$\varepsilon$::*;
6 **else if** *($\alpha \in \{\downarrow^+, \downarrow^*\}$)* **then**
7 $P_t := \alpha$::$\eta[\mathcal{A}^{acc}][F^t][\varepsilon$::*/*text*()='c'];
8 **else** // $\alpha = \varepsilon$
9 $P_t := \alpha$::$\eta[F^t][\varepsilon$::*/*text*()='c'];
10 **else if** *(P is P_1/P_r where $P_1 = \alpha_1$::$\eta_1[F_1]$ and P_r is the remaining part)* **then**
11 $F_1^t :=$ *RW_Pred*(F_1);
12 **if** *($\alpha_1 = \downarrow$)* **then**
13 $P_t := \downarrow^+$::$\eta_1[\mathcal{A}^{acc}][F_1^t]$[*RW_Pred*($P_r$)]/$\mathcal{A}^+$[1]=$\varepsilon$::*;
14 **else if** *($\alpha_1 \in \{\downarrow^+, \downarrow^*\}$)* **then**
15 $P_t := \alpha_1$::$\eta_1[\mathcal{A}^{acc}][F_1^t]$[*RW_Pred*($P_r$)];
16 **else** // $\alpha = \varepsilon$
17 $P_t := \alpha_1$::$\eta_1[F_1^t]$[*RW_Pred*(P_r)];
18 **else if** *(P is $P_1 \wedge ... \wedge P_n$)* **then**
19 $P_t := \bigwedge_i$*RW_Pred*(P_i);
20 **else if** *(P is $P_1 \vee ... \vee P_n$ or $P_1 \cup ... \cup P_n$)* **then**
21 $P_t := \bigvee_i$*RW_Pred*(P_i);
22 **else if** *case of not (P)* **then**
23 $P_t :=$ not (*RW_Pred*(P));
24 **return** P_t;

Figure 4.9: $\mathcal{X}$ Predicates Rewriting Algorithm.

4.5 Theoretical Results

We present in this section some results that concern the evaluation of the overall answering time of our rewriting algorithm as well as the correctness of our approach.

An interesting study [GKP02] has shown that contemporary XPath processors are inefficient and can have running times that are exponential in the size of the query and the XML data. Based on dynamic programming, authors proposed efficient algorithms for processing XPath queries that have polynomial time and space complexity. Their results have been improved noticeably in [GKP05] to evaluate arbitrary XPath queries in time $O(|T|^4.|Q|^2)$ and space $O(|T|^2.|Q|^2)$, where $|T|$ and $|Q|$ are the size of the XML document and the size of the XPath query respectively. Another idea for evaluating XPath queries, as used in e.g. [BP08] and improved in [BP11], is to compile the query into finite-state tree automaton that can be evaluated on the XML tree in linear time. Using the algorithms proposed in [BP08] XPath queries can be evaluated in $O(|Q|^3|T|)$ time, moreover, queries including Kleene star can be evaluated in $O(2^{O(|Q|)}.|T|)$ time or in $O(|Q|^3.|T|log|T|)$ time.

A practical useful fragment of XPath, *Core XPath*, has been introduced first in [GKP02] that can be evaluated in $O(|D|.|Q|)$ time with respect the size of the data ($|D|$) and that of the query ($|Q|$). This fragment is defined as follows:

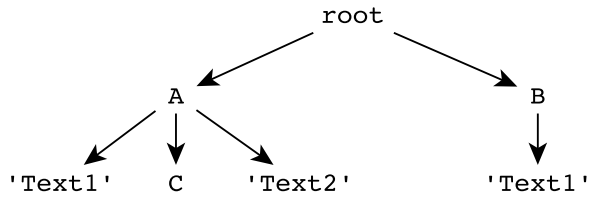


Figure 4.10: Sample XML document.

Definition 4.10. [**Core XPath** [GKP03,GKPS05c]] We denote by $\mathcal{CX}$ the *Core XPath* fragment defined as follows:

$$
\begin{aligned}
&cxpath \;:=\; path \mid /path \mid path \cup path\\
&path \;:=\; step \mid step/step\\
&path \;:=\; \alpha::ntst \mid \alpha::ntst[pred]\cdots[pred]\\
&pred \;:=\; cxpath \mid pred \wedge pred \mid pred \vee pred \mid \neg\ (pred)\\
&\alpha \;:=\; \varepsilon \mid \downarrow \mid \downarrow^{+} \mid \downarrow^{*} \mid \leftarrow \mid \rightarrow \mid \leftarrow^{*} \mid \rightarrow^{*}
\end{aligned}
$$

where *cxpath* denotes an Core XPath query and it is the start of the production, *path* is a relative path, */path* is an absolute path (evaluates starting from the root node), *pred* represents Core XPath predicates, and $\leftarrow$, $\rightarrow$, $\leftarrow^{*}$, $\rightarrow^{*}$ represent *horizontal axes* and stand for *preceding-sibling*, *following-sibling*, *preceding* and *following* axis respectively. □

The semantics of relative $\mathcal{CX}$ queries are the same as those of $\mathcal{X}^{\Uparrow}_{[n,=]}$ queries presented in Table 3.1 (horizontal axes have the same semantics as $\mathcal{X}^{\Uparrow}_{[n,=]}$ axes). However, the semantics of absolute $\mathcal{CX}$ queries are defined as follows:

$$\mathcal{S}[\![/\alpha :: \eta]\!](N) = \begin{cases} \alpha(\{root\}) \cap T(\eta) & if\ \{root\} \in N \\ \varnothing & otherwise \end{cases}$$

Note.
Note that the $\mathcal{CX}$ fragment, as defined in [GKP02,GKP05], does not allow union of queries. We consider the union operator in Definition 4.10 instead since the complexity time is still not changed [GKP03, GKPS05c, KMR09] (i.e. the query $Q_1 \cup Q_2$ is evaluated over a tree T in $O(|Q_1 + Q_2|.|T|)$ time).

We conduct thereafter a complexity analysis of the XPath fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ defined in Section
.

Compared to the fragment $\mathcal{X}^{\Uparrow}$ of Definition 3.10, $\mathcal{CX}$ does not allow text comparison inside predicates. We denote by $\mathcal{X}^{\Uparrow}_{noTest}$ queries of $\mathcal{X}^{\Uparrow}$ whose predicates do not contain text comparison, i.e. predicates of the form $p = c$ are not allowed. Since XPath axes have all the same evaluation time [GKP05], any sub-fragment resulted by eliminating some axes from $\mathcal{CX}$ has the same complexity of this latter. It is clear that $\mathcal{X}^{\Uparrow}_{noTest}$ is a subset of $\mathcal{CX}$, i.e. $\mathcal{CX}$ extends $\mathcal{X}^{\Uparrow}_{noTest}$ with the axes $\leftarrow$, $\rightarrow$, $\leftarrow^{*}$, and $\rightarrow^{*}$. Thus, any $\mathcal{X}^{\Uparrow}_{noTest}$ query Q can be evaluated over an XML tree T in $O(|Q|.|T|)$ time.

Lemma 4.3. Every $\mathcal{X}^{\Uparrow}$ query Q can be evaluated over an XML document T in time $O(|Q|.|T|)$. □

We emphasize a significant difference between the predicates α::η=c and α::η/*text()*=c. The former indicates that each node selected must have a single child node that is a text node having value c. However, the latter indicates that each node selected must have at least one text node with value c.

Example 4.15. Consider the XML tree of Figure 4.10 that contains three text nodes with the respective values '*Text1*', '*Text2*', and '*Text1*'. The XPath query $\downarrow^{+}$::*[ϵ::*='*Text1*'] over this tree returns the node labeled B, the query $\downarrow^{+}$::*[ϵ::*/*text()*='*Text1*'] returns the nodes labeled A and B, while the query $\downarrow^{+}$::*[ϵ::*='*Text2*'] returns only the node A. □

Proof 4.3. We should show that $\mathcal{X}^{\Uparrow}_{noTest}$ queries augmented with predicates of the form $p = c$ can be evaluated in linear time with respect to the size of the data and the size of the query. Consider an XML tree T and an $\mathcal{X}^{\Uparrow}_{noTest}$ query Q. By $|T|$ we denote the number of all element nodes and text nodes in T. Let τ be the largest text value in T. Let $|N|$ and $|S|$ be the size of element nodes and text nodes in T respectively. Each subquery q of Q, with the form α::η, can be evaluated in $O(|T|)$ time [GKP02] and returns a set of XML nodes bounded with $O(|N|)$. Let $n_1, \ldots, n_l$ $(1 \leq l \leq N)$ be the node set returned, each n_i may have $m_1, \ldots, m_{s_i}$ text nodes where $\Sigma_{1 \leq i \leq l} s_i \leq |S|$. By adding a text-comparison into the subquery q, i.e. q becomes q/*text()*=c, the value of each text node m_j of n_i is compared w.r.t the value c in $O(|\tau|)$ time. Thus, for $|S|$ text nodes we get at most $|S|.|\tau|$ comparison. Therefore, each subquery of Q augmented with text-comparison predicate can be evaluated in $O(|T| + |S|.|\tau|)$ time, i.e. $O(|T|.|\tau|)$ time (since $|S|$ is bounded by $|T|$). We conclude that a naive evaluation of a query Q, with text-comparison predicates, over an XML tree T can be done in $O(|Q|.|T|.|\tau|)$. This result can be significantly improved by classifying nodes with string values as done in [BP11]. We simplify the structure introduced in this latter work since we do not consider inequalities between text values. For each element node we parse its text nodes and we create a list that contains all string values of these text nodes. Each list is indexed w.r.t the text values, so it can be checked in constant time whether an element node n has a text node with value c. It is obvious to show that the resulted structure can be constructed in $O(|T|)$ time and space. Given an element node n, using the auxiliary structure the following tests can be done in a constant time:

1. n is selected by a query of the form α::η/*text()*=c: this can be done by checking whether the value c appears in the list of n.

2. n is selected by a query of the form α::η=c: this can be done by checking whether the value c appears in the list of n, as well as whether n has only one child node.

We conclude that predicate of the form $p = c$ can be evaluated in linear time w.r.t the size of the data and the size of the predicate. Therefore, any $\mathcal{X}^{\Uparrow}$ query Q over an XML tree T can be evaluated in $O(|Q|.|T|)$ time. □

We show in the following that extending the fragment $\mathcal{X}^{\Uparrow}$ with position predicates of the form [*1*] and node comparison predicate of the form p=ε::* does not lead to additional costs.

Lemma 4.4. Every $\mathcal{X}^{\Uparrow}_{[n,=]}$ query Q can be evaluated over an XML document T in time $O(|Q|.|T|)$. □

Proof 4.4. We use the data model of [GKP05] to present our XML trees. The *document order* is used to present order between all the nodes of a given XML tree, as well as the nodes returned by evaluating a given XPath expression. According to the W3C XPath specification [BBC+10], the *document order* is the order in which nodes appear in the XML serialization of a document. We use $<$ to denote the document order relation where $n < m$, for two nodes n and m of an XML tree T, if and only if the opening tag of n precedes the opening tag of m in the serialization of T. Within an XML tree $T{=}(N, root, R_{\downarrow}, R_{\rightarrow}, \lambda, \nu)$, the relation $<$ must satisfy the following conditions:

- $root < n$ for any node n of N [the root node occurs first].
- If $\{(n, n_1), \ldots, (n, n_k)\} \in R_{\downarrow}$ then $n < n_i$ for each node n_i of n $(1 \leq i \leq k)$ [each node occurs before all its children].
- If $\{(n_1, n_2), \ldots, (n_{k-1}, n_k)\} \in R_{\rightarrow}$ then $n_1 < \ldots < n_k$ [sibling order is maintained].
- If $\{(n_1, n_2), \ldots, (n_{k-1}, n_k)\} \in R_{\downarrow}$ then $n_1 < \ldots < n_k$ [each node occurs before all its descendants].
- If $\{(n, m_1), (m_1, m_2), \ldots, (m_{k-1}, m_k), (n, p_1), (p_1, p_2), \ldots, (p_{l-1}, p_l)\} \in R_{\downarrow}$ and $\{(m_1, p_1)\} \in R_{\rightarrow}$, then $m_i < \ldots < p_j$ $(1 \leq i \leq k$ and $1 \leq j \leq l)$ [descendants occurs before sibling nodes].

Note that using the forward axes $\downarrow, \downarrow^+, \downarrow^*$, the returned nodes are ordered using the document order, while for the upward axes $\uparrow$, $\uparrow^+$, $\uparrow^*$, the returned nodes are ordered using the reverse document order $<^{-1}$ ($n <^{-1} m$ if and only if $m < n$). For instance, the query $\downarrow^*::*$ over the root node R of Figure 4.10 returns the node set $\{R, A, 'Text1', C, 'Text2', B, 'Text1'\}$, notice that $R < A <' Text1' < C <' Text2' < B <' Text1'$. However, the query $\uparrow^*::*$ over the text node $'Text1'$ of the A element returns the node set $\{A, R\}$, notice that $A <^{-1} R$.

It is shown in [GKP05] that $\mathcal{CX}$ queries augmented with position predicates (e.g. $\downarrow::A/\downarrow::B[position() < k]$) lead to quadratic time. However, our position predicates (of the form $[1]$) are simple and can be evaluated in a constant time as we show below. We have shown that each $\mathcal{X}^{\Uparrow}$ query Q over an XML tree T returns a set of nodes, ordered w.r.t document order, in $O(|Q|.|T|)$. Consequently, for an $\mathcal{X}^{\Uparrow}$ predicate p of Q, $\xi[\![p]\!]$ is bounded by $O(|Q|.|T|)$. Let $N{=}\xi[\![p]\!]$, then the first node in N (i.e. $N[1]$) can be returned in a constant time. Since $\xi[\![p[1]]\!]$ $= (\xi[\![p]\!])[1]$, then the evaluation of $p[1]$ is bounded by $O(|Q|.|T|)$. We conclude that each $\mathcal{X}^{\Uparrow}_{[n]}$ query Q can be evaluated over an XML tree T in $O(|Q|.|T|)$ time. Finally, the predicate $p{=}\varepsilon::*$ checks whether the right and left expressions refer to the same single node. This can be done in a constant time using the relation $<$: *a*) if $\xi[\![p]\!]$ returns more than one node then the predicate $p{=}\varepsilon::*$ is not satisfied; *b*) if $\xi[\![p]\!]$ returns only one node, then this node must have the same document order as the node returned by $\varepsilon::*$, which can be checked in a constant time. Note that two node n and m are identical, i.e. have the same document order, if and only if $n \not< m$ and $m \not< n$. Since the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ extends the fragment $\mathcal{X}^{\Uparrow}$ with position predicates (of the form $p[1]$) and node comparison predicates (of the form $p{=}\varepsilon::*$), then we conclude that $\mathcal{X}^{\Uparrow}_{[n,=]}$ and $\mathcal{X}^{\Uparrow}$ have the same time complexity evaluation. □

Theorem 4.2. Given an access specification $S{=}(D,$ $\mathtt{ann}$$)$, an XML tree $T \in \mathcal{T}(D)$ and its virtual view T_v computed w.r.t S, then, any $\mathcal{X}$ query Q over T_v can be rewritten using the algorithm `Rewrite` and evaluated over T at most in time $O(|Q|.|\mathtt{ann}|.|T|)$. □

Proof 4.5. From the Definition 4.8, it is clear to see that for any access specification S=(D, ann), the size of the accessibility predicate $\mathcal{A}^{acc}$ is bounded by $O(|\mathit{ann}|)$. Moreover, the predicate $\mathcal{A}^{+}$ of Definition 4.9 is based on $\mathcal{A}^{acc}$ and then also bounded by $O(|\mathit{ann}|)$. We have seen in Section 4.4 that each subquery of the query Q is rewritten in a constant time by adding either the predicate $\mathcal{A}^{acc}$ and/or $\mathcal{A}^{+}$. Thus, the size of the rewritten query $\mathit{Rewrite}(Q)$ is bounded by $O(|Q|.|\mathit{ann}|)$. Summing up, according to Lemma 4.2 the query Q is rewritten in time $O(|Q|)$, and the rewritten query $\mathit{Rewrite}(Q)$, of size $|Q|.|\mathit{ann}|$, can be evaluated over the document T in time $O(|\mathit{Rewrite}(Q)|.|T|)$ as shows Lemma 4.4. Since the rewriting time is negligible w.r.t the evaluation time, then we conclude that the query Q can be answered over the document T at most in time $O(|Q|.|\mathit{ann}|.|T|)$. □

Theorem 4.3. The query rewriting algorithm $\mathit{Rewrite}$ is correct for any query of the fragment $\mathcal{X}$. □

Theorem 4.3 shows the correctness of our query rewriting approach. More specifically, for any access specification S=(D,ann), any XML tree $T \in \mathcal{T}(D)$ and its virtual view T_v, our rewriting algorithm $\mathit{Rewrite}$ translates any $\mathcal{X}$ query Q over T_v into a safe one Q^t defined over T such that: $\mathcal{S}[\![Q]\!](T_v)$=$\mathcal{S}[\![Q^t]\!](T)$. The correctness of Theorem 4.3 is given in Annex.

4.6 Conclusions

The proposed approach yields the first practical solution to rewrite XPath queries over recursive XML views using only the expressive power of the standard XPath. The extension of the downward class of XPath queries with some axes and operators has been investigated in order to make queries rewriting possible under recursion. The conducted experimentation shows the efficiency of our approach by comparison with the materialization approach. Most importantly, the translation of queries from $\mathcal{X}$ to $\mathcal{X}^{\Uparrow}_{[n,=]}$ does not impact the performance of the queries answering. We have discussed how our approach can be extended to deal with the upward-axes without additional cost. Lastly, a revision of the access specification language is presented to go beyond some limitations in the definition of some access privileges.

5

Toward a Safe Approach for Updating XML Data

Contents

The XQuery Update Facility language [BCFF+10] is a recommendation of W3C that provides a facility to modify some parts of an XML document and leave the rest unchanged, and this through different update operations. This includes rename, insert, replace and delete operations at the node level. The security requirement is the main problem when manipulating XML documents. An XML document may be queried and/or updated simultaneously by different users. For each class of users some rules can be defined to specify parts of the document which are accessible to the users and/or updatable by them. A bulk of work has been published in the last decade to secure the XML content, but only read-access rights has been considered over non-recursive DTDs [FCG04a,KMR09,DFGM08,Ras06]. Moreover, a few works have considered update rights [DFGM08,FM07,DZ08b].

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

For more comprehension, we recall in Figures 5.1 and 5.2 our examples of hospital DTD and hospital document that we consider in this manuscript.

5.1 Update Specifications

We follow the idea of *security annotations* presented in [FCG04a] and the *update access types* notion introduced in [BCF07] to define a language for specifying expressive and fine-grained XML

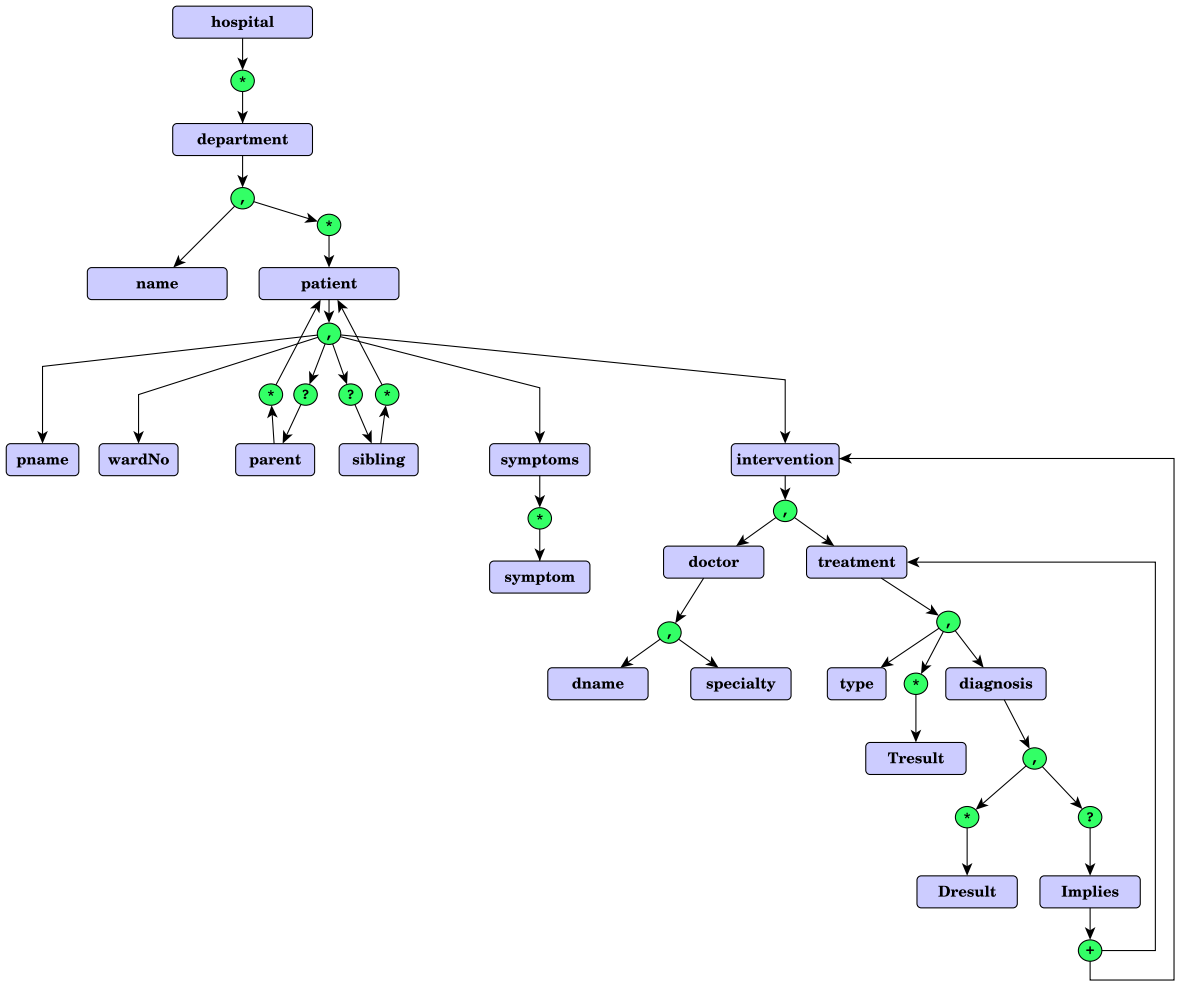


Figure 5.1: Example of Hospital DTD.

update policies in the presence of DTDs. An *update specification* S_{up} expressed in the language is a simple extension of the document DTD D associating element types with *update annotations* (XPath qualifiers), which specify for any XML tree T that conforms to D, the parts of T that can be updated by the user through a specific update operation.

Definition 5.1. Given a document DTD D, an *update type (ut)* defined over D is of the form `insertInto`$[B_i]$, `insertAsFirst`$[B_i]$, `insertAsLast`$[B_i]$, `insertBefore`$[B_i,B_j]$, `insertAfter`$[B_i,B_j]$, `delete`$[B_i]$, `replaceNode`$[B_i,B_j]$, `replaceValue`$[B_i]$, and `rename`$[B_i,B_j]$, where B_i and B_j are element types of D. □

Intuitively, each update type *ut* represents an update operation that is restricted to be applied only for specific element types. More specifically, each update type represents a set of update operations as follows:

Annotation	Semantic
$ann_{up}(A, \texttt{insertInto}[B_i]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, insert nodes of type B_i in an arbitrary position children of n.
$ann_{up}(A, \texttt{insertAsFirst}[B_i]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, insert nodes of type B_i as first children of n.
$ann_{up}(A, \texttt{insertBefore}[B_i, B_j]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, insert nodes of type B_j as preceding sibling nodes of any child node of n whose type is B_i.
$ann_{up}(A, \texttt{delete}[B_i]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, delete children of n whose type is B_i.
$ann_{up}(A, \texttt{replaceNode}[B_i, B_j]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, replace children of n of type B_i by a sequence of nodes of type B_j.
$ann_{up}(A, \texttt{replaceValue}[B_i]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, change text-content of children of n whose type is B_i.
$ann_{up}(A, \texttt{rename}[B_i, B_j]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, rename children of n whose type is B_i with the new label "B_j".

Table 5.1: Semantics of the update annotations Y, N, and $[Q]$.

$$
\begin{array}{lcl}
\texttt{insertInto}[B_i] & = & \{\textbf{insert } \textit{source} \textbf{ into } \textit{target} \setminus \textit{source} \subset \mathcal{N}(B_i)\} \\
\texttt{insertBefore}[B_i,B_j] & = & \{\textbf{insert } \textit{source} \textbf{ before } \textit{target} \setminus \textit{target-node} \in \mathcal{N}(B_i) \\
 & & \qquad \text{and } \textit{source} \subset \mathcal{N}(B_j)\} \\
\texttt{delete}[B_i] & = & \{\textbf{delete } \textit{target} \setminus \textit{target-nodes} \subset \mathcal{N}(B_i)\} \\
\texttt{replaceNode}[B_i,B_j] & = & \{\textbf{replace } \textit{target} \textbf{ with } \textit{source} \setminus \textit{target-node} \in \mathcal{N}(B_i) \\
 & & \qquad \text{and } \textit{source} \subset \mathcal{N}(B_j)\} \\
\texttt{replaceValue}[B_i] & = & \{\textbf{replace value of } \textit{target} \textbf{ with } \textit{string-value} \setminus \textit{target-node} \in \mathcal{N}(B_i)\} \\
\texttt{rename}[B_i,B_j] & = & \{\textbf{rename } \textit{target} \textbf{ with } \text{"}B_j\text{"} \setminus \textit{target-node} \in \mathcal{N}(B_i)\}
\end{array}
$$

The semantics of the update types $\texttt{insertAsFirst}[B_i]$ (resp. $\texttt{insertAsLast}[B_i]$) and $\texttt{insertAfter}[B_i]$ are defined in a similar way as $\texttt{insertInto}[B_i]$ and $\texttt{insertBefore}[B_i,B_j]$ respectively. Consider for instance the hospital DTD of Figure 5.1, the update type $\texttt{replaceValue}[Tresult]$ represents the set of update operations that consist in changing the text-content of elements $Tresult$.

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

Definition 5.2. An *update specification* S_{up} is a pair (D, ann_{up}) where D is a DTD and ann_{up} is a partial mapping such that, for each element type A in D and each update type ut defined over element types of D, $ann_{up}(A, ut)$, if explicitly defined, is an annotation of the form:

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

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

An update specification S_{up} is an extension of a DTD D associating update annotations with element types of D. In a nutshell, a value of Y, N, or $[Q]$ for $ann_{up}(A, ut)$ indicates that, for A elements in an instantiation of D, the user is *authorized*, *unauthorized*, or *conditionally authorized* respectively, to perform update operations of type ut at A (case of *insertInto*, *insertAsFirst*, or

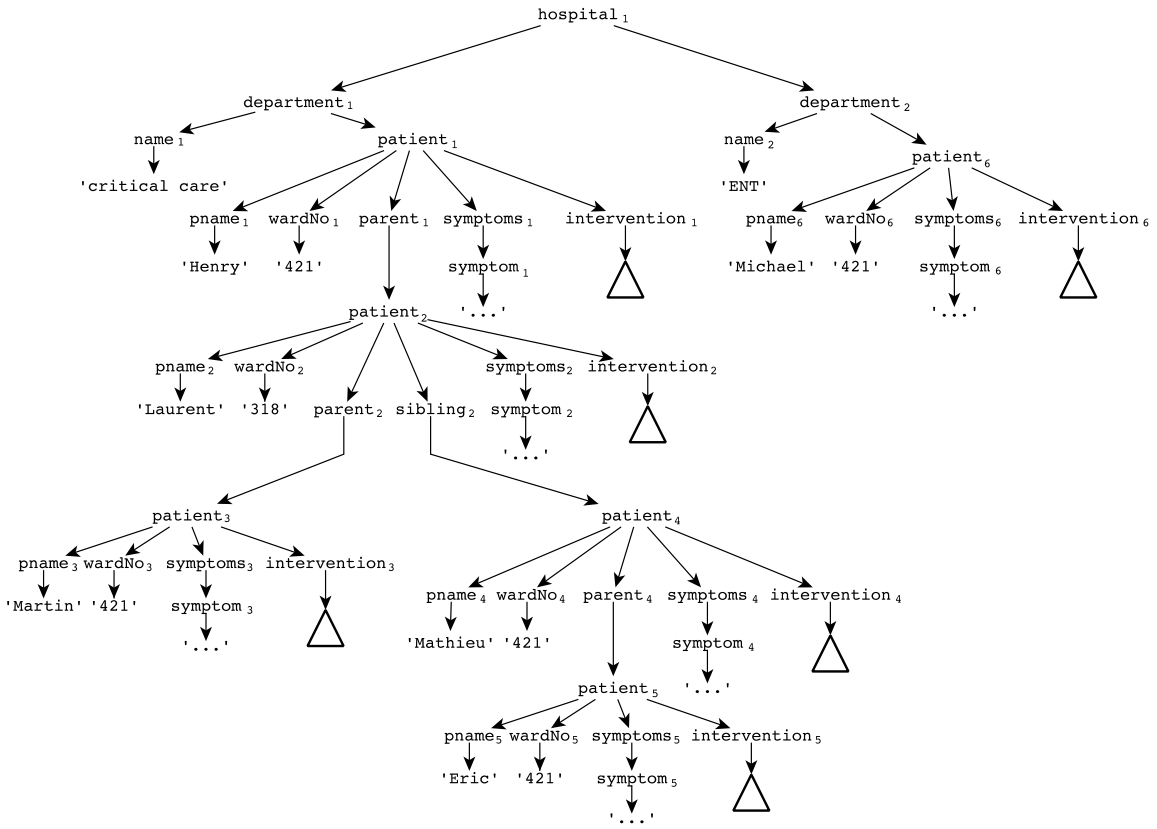


Figure 5.2: Example of Hospital data.

insertAsLast operations) or at children of A (case of the remaining operations). Table 5.1 presents more specifically the semantics of the update annotations Y, N, and $[Q]$ [30].

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

Example 5.1. We consider the hospital DTD of Figure 5.1, the XML tree of Figure 5.2, and we define the following update annotations for nurses:

R_1: $ann_{up}(department, \texttt{insertInto}\,[patient]) = Y$
R_2: $ann_{up}(sibling, \texttt{insertInto}\,[patient]) = N$

These annotations allow nurses to insert some data that concerns patients and their parents, while they deny insertion of sibling data. Specifically, the annotation R_1 allows insertion of new patients into departments of the hospital; i.e. new *patient* nodes can be inserted into the nodes $department_1$ and $department_2$. By inheritance, a nurse can perform updates of type

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

insertInto[*patient*] at the nodes $parent_1$, $parent_2$, and $parent_3$. The annotation R_2 denies insertion of new patient data under *sibling* subtrees. Accordingly, the node $sibling_2$ overrides the insertion authorization (Y) inherited from $department_1$ and makes insertion forbidden under it, such as insertion of new *patient* nodes into the node $parent_4$. □

We should emphasize that different semantics are defined for security qualifiers. In [KMR09], the evaluation of a qualifier $[Q]$ at a node n is mapped to either Y or N; and thus this security value can be overridden by another annotation at descendants of n (as we have shown with the two annotations of Example 5.1). We follow this semantic by using qualifiers of the form $[Q]$. In [FCG04a], however, a *false* evaluation of a qualifier $[Q]$ at a node n makes all the subtree rooted at n inaccessible (resp. not updatable) even if some annotations with value Y (resp. a valid qualifier $[Q']$) are defined under n. This principle is called *Denial Downward Consistency Requirement*, it defined first in [MTKH03] for read-access rights, and used in [DFGM08, FM04, FM07, LLLL11] [31]. We redefine this requirement for update rights as follows:
"For an XML tree/fragment T rooted at a node n, whenever an update policy denies the execution of an update operation at n, it must also deny the excecution of this operation at descendants of n. In other words, whenever an update operation is authorized at a node n' of T, execution of this update must be authorized as well at all ancestors of n' in T".

The advantage of the denial downward consistency is that it makes security policy enforcement easier even over recursive DTDs. Moreover, the security view derivation (see [GSC+09] for this problem) becomes always possible. For this reason, we define the new specification values N_h and $[Q]_h$ to make possible the application of this requirement for the security administrator.

The semantics of the specification values N_h and $[Q]_h$ are given in Table 5.2. The annotation $ann_{up}(A,ut)=N_h$ indicates that, for a node n of type A, update operations of type ut cannot be performed at any node of the subtree rooted at n, and no overriding of this authorization value is permitted for descendants of n. For instance, if n has a descendant n' whose type is A', then an update operation with the same type ut cannot be performed at n' even though the annotation $ann_{up}(A',ut)=Y$ is explicitly defined (resp. $ann_{up}(A',ut)=[Q']$ with $n' \vDash Q'$). As for the annotation $ann_{up}(A,ut)=[Q]_h$, qualifier Q must be valid at A elements, otherwise no annotation with update type ut can override the *false* evaluation of Q. For instance, let n and n' be two nodes of type A and A' respectively, and let n' be a descendant of n. The annotation $ann_{up}(A',ut)=[Q']$ indicates that an update operation of type ut can be performed at (children of) n' iff: $n' \vDash Q'$. Moreover, if the annotation $ann_{up}(A,ut)=[Q]_h$ is explicitly defined then the annotation $ann_{up}(A',ut)=[Q']$ takes effect at descendant n' of n only if $n \vDash Q$. This means that an update operation of type ut can be performed at (children of) n' iff: $(n \vDash Q \wedge n' \vDash Q')$. We call annotation with value N_h or $[Q]_h$ as *downward-closed* annotation.

Example 5.2. Suppose that each nurse is attached to only one department and only one ward within this department (denoted $nurseDept and $nurseWardNo resp.). Now, the hospital wants to impose an update policy that allows a nurse to update data of only patients having the same ward number as her (*Rule1*) and which are being treated at her department (*Rule2*). Moreover, all sibling data cannot be updated (*Rule3*). This policy can be specified by the following update annotations (ut denotes a general update type):

R_1: $ann_{up}(department,ut)=[\downarrow::name=\$nurseDept]_h$
R_2: $ann_{up}(patient,ut)=[\downarrow::wardNo=\$nurseWardNo]$

[31] In [FM04, FM07], the denial downward consistency requirement is applied by defining the *deny overrides* as conflict resolution policy. While in [LLLL11], it is applied by using the DEEP-EXCEPT operator.

Downward-closed Annotation	Semantic
$ann_{up}(A,ut) = N_h$ where ut can be any update type	Same principle as $ann_{up}(A,ut) = N$ of Table 5.1. Moreover, for a node n of type A, all annotations of type ut defined over descendant types of A are discarded regardless their truth values.
$ann_{up}(A,ut) = [Q]_h$ where ut can be any update type	Same principle as $ann_{up}(A,ut) = [Q]$ of Table 5.1. Moreover, for a node n of type A, if $n \not\models Q$ then all annotations of type ut defined over descendant types of A are discarded regardless their truth values.

Table 5.2: Semantics of the update annotations N_h and $[Q]_h$.

R_3: $ann_{up}(sibling,ut)=N_h$

Consider the case of the nurse having the ward number *421* and working at *Critical care* department, and let ut be `replaceValue`[*symptom*]. This nurse can update the values of all symptoms of Figure 5.2 except those of nodes: $symptom_2$ (since $patient_2$ has ward number *318*), $symptom_4$ and $symptom_5$ (representing sibling data), and $symptom_6$ (although $patient_6$ has ward number *421*, he is attached to *ENT* department). Notice that the annotations R_1 and R_3 must be defined as *downward-closed* to enforce the imposed policy, otherwise annotation R_2 may override a negative authorization inherited from R_1 or R_3 which may allow some updates that are forbidden by the imposed policy. Consider the following update annotations:

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

Since R_1' and R_3' are not downward-closed, the node $patient_4$ overrides the negative authorization inherited from $sibling_1$, using the annotation R_2', and makes the update of some sibling data possible (the node $symptom_4$). With the same principle, the node $patient_5$ overrides the negative authorization inherited from $department_2$, and allows the nurse to update some data of *ENT* department (the node $symptom_5$). Thus, using the XPath fragment $\mathcal{X}$, our update policy can be enforced only if the denial downward consistency requirement is enforced for *Rule1* and *Rule3* (which may be done using the downward-closed annotations R_1 and R_3). □

Example 5.3. Suppose that the hospital wants to impose a security policy that authorizes each doctor to update only the information of treatments that she has done. For instance, the doctor *Imine* could update the data of $treatment_1$, $treatment_2$, and $treatment_4$ (like insert new *Dresult* sub-tree into $diagnosis_4$ node) but not $treatment_3$. We show in the following that this update policy cannot be enforced by using the existing update specification languages. □

5.1.1 Compactness and Expressiveness

We show in this section that in some cases, our model can be more expressive than the existing ones and generates compact update policies.

The principle of denial downward consistency can be defined using only the annotation values $\{Y, N, [Q]\}$ but through a fragment of XPath larger than $\mathcal{X}$. For instance, the update policy of Example 5.2 (i.e. *Rule*1, *Rule*2, and *Rule*3) can be defined in the fragment $\mathcal{X}^{\Uparrow}$ of Section 4.5 as follows:

R_1'': $ann_{up}(department,ut)=[\downarrow::name=\$\text{NURSEDEPT}]$

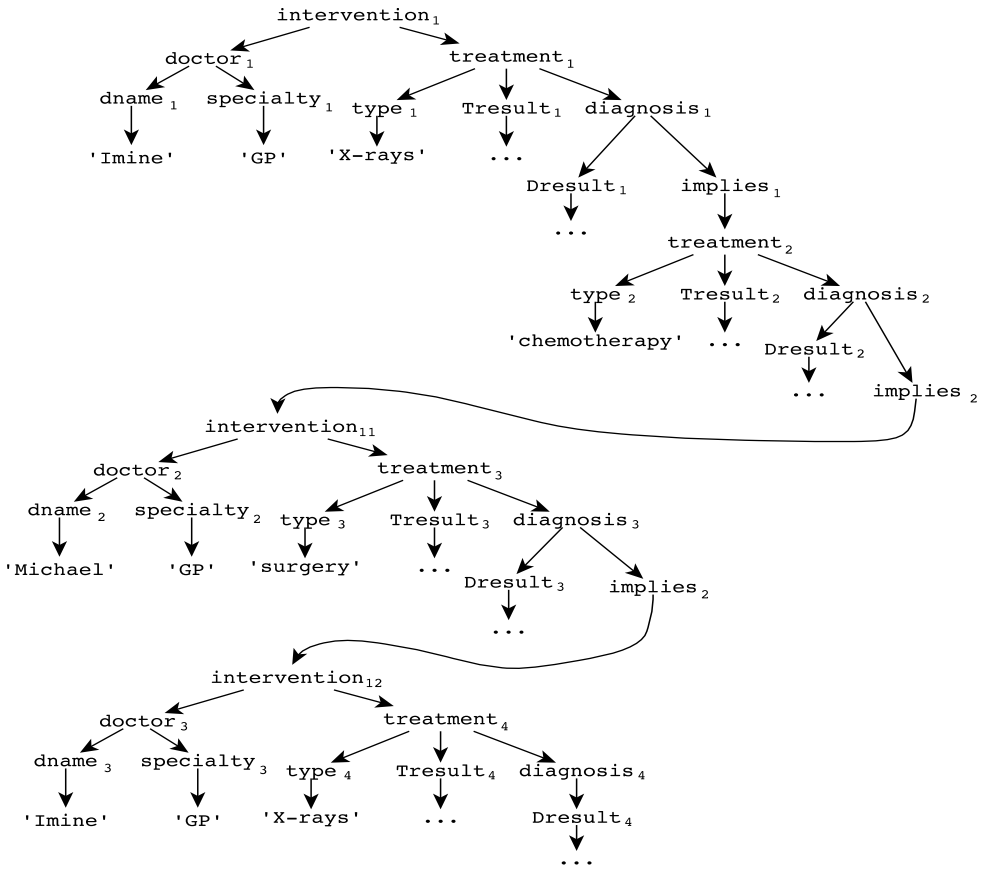


Figure 5.3: The interventions done for the patient "*Henry*" of Figure 5.2.

R_2'': $ann_{up}(patient,ut)=[(\downarrow::wardNo=\$\textsc{nurseWardNo})$
$\wedge\ (\uparrow^+::department[\downarrow::name=\$\textsc{nurseDept}])$
$\wedge\ \neg(\uparrow^+::sibling)]$

R_3'': $ann_{up}(sibling,ut)=N$

Here the denial downward consistency requirement must be applied for annotations R_1'' and R_3'' to enforce the update policy. For this reason, the annotation R_2 of Example 5.2 is redefined into R_2'' by including the values of R_1'' and R_3''. The redefined annotation R_2'' depends on R_1'' and R_3'' and must be redefined each time a modification is made on R_1'' and/or R_3''. Observe that, without the use of the annotations values $\{N_h, [Q]_h\}$, defining an annotation $ann_{up}(A, ut) = val$ as downward-closed amounts to redefine all the annotations of type ut defined over descendant types of A in order to take into account the value val. This can lead to verbose annotations, like annotation R_2''. Moreover, changing one annotation may require the modification of some annotations defined under it[32] which can make the policy administration complicated and time consuming in case of large DTDs.

The principle of inheritance and overriding of update annotations (defined first in [FCG04a] for read-access rights) allows our update access control model to be more expressive than the

[32]This is not recommended for some systems like collaborative editing where the update policies are dynamic and each change is propagated to all the users across the network [CIR11].

existing ones. We show by the following example how that our model can overcome the limitations of existing models [DFGM08, FM07].

Example 5.4. Suppose that the hospital imposes a policy that allows each doctor to update only his intervention data. In our case, this policy can be specified by defining only the following update annotation:

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

Where $DOCTORNAME is a constant parameter representing doctor's name, and *ut* can be any update type relevant to the update rules of Example 5.4. Consider the case of the update type $insertAfter[Tresult,Tresult]$ which allows a doctor to add new treatment results. The predicate $[\downarrow::doctor/\downarrow::dname='Imine']$ is valid at node $intervention_1$ of Figure 5.3; thus, the nodes $Tresult_1$ and $Tresult_2$ inherit this positive authorization which allows doctor *Imine* to insert new *Tresult* nodes as following siblings of these nodes. At node $intervention_{11}$, the predicate becomes invalid which forbids doctor *Imine* to insert new *Tresult* nodes as following sibling of node $Tresult_3$. Finally, the node $intervention_{12}$ overrides the negative authorization inherited from $intervention_{11}$, and makes possible for doctor *Imine* the insertion of *Tresult* nodes after the node $Tresult_4$. □

5.1.2 Rewriting Problem

As will be seen shortly, in the case of recursive DTDs, update operations rewriting is already challenging for the small fragment $\mathcal{X}$ of XPath.

Consider for instance the update policy of Example 5.4, with $DOCTORNAME=*Imine* and $ut=delete[Tresult]$. Due to the hospital DTD recursion, the update operation $delete\ \downarrow^+::treatment[\downarrow::type='Chemotherapy']/\downarrow::Tresult$ cannot be rewritten in $\mathcal{X}$ to be safe. Indeed, the *Tresult* nodes that doctor *Imine* is authorized to delete can be represented by an infinite set of paths. This set of paths can be captured in Regular XPath by rewriting the previous update into the following one:

$$delete\ \downarrow^+::intervention[\downarrow::dname=\$\text{DOCTORNAME}]/(\downarrow::treatment/\downarrow::diagnosis/\downarrow::implies)^*/\downarrow::treatment[\downarrow::type='Chemotherapy']/\downarrow::Tresult$$

which, when evaluated on the XML tree of Figure 5.3, has to delete only the node $Tresult_2$. However, the Kleene star cannot be expressed in XPath [Mar04].

Along the same lines as [FGJK07], we say that an XPath query language L is *closed* under update operations rewriting if there is a computable function $\mathcal{F} : L \to L$ that, given any update specification $S_{up} = (D, ann_{up})$ and any update operation of type *ut* with *target* expression defined in L, computes another expression $\mathcal{F}(target)$ in L such that for any XML tree $T \in \mathcal{T}(D)$, the evaluation of $\mathcal{F}(target)$ in T returns all and only the XML nodes that can be updated w.r.t. S_{up} using *ut* operations. The following theorem states the central problem studied in the rest of this chapter:

Theorem 5.1. For recursive DTDs, the fragment $\mathcal{X}$ is not closed under update operation rewriting. □

We study the *closure* property of XPath since it is preferable to rewrite update operations on XPath rather the use of a richer language such as XQuery or Regular XPath. In other words, it is more efficient to evaluate XPath queries than queries in the aforementioned languages (Authors

of [FGJK07] have shown that rewriting of XPath queries through a translation into Regular XPath may lead queries of exponential size). We explain in the next section how the extended fragment $\mathcal{X}^{\Uparrow}_{[n]}$, defined in Chapter 3, can be used to overcome this rewriting limitation of update operations.

5.2 Securely Updating XML

In this section we focus only on update rights and we assume that every node is read-accessible by all users. Given an update specification S_{up}=(D, ann_{up}), we discuss the enforcement of such update constraints where each update operation posed over an instance T of D must be performed only at the nodes of T that can be updated by the user w.r.t. S_{up}. We assume that the XML tree T remains valid after the update operation is performed, otherwise the update is rejected (this can be assumed using algorithms of [BPV04,BLS06]). In the following, we denote by S_{ut} the set of annotations defined in S_{up} with the update type ut and by $|S_{ut}|$ the size of this set. Moreover, for an annotation function ann (such as ann_{up} of an update specification S_{up}=(D, ann_{up})), we denote by $\{ann\}$ the set of all annotations defined with ann, and by $|ann|$ the size of this set.

5.2.1 Updatability

Here we want to specify when an update operation of type ut can be performed at a given node. We consider the privileges of *insert*, *delete*, *replace* and *rename*. The semantics of these privileges can be stated as follows:

- If the user holds the *insertInto* privilege on node n then she has the right to add new sub-tree to node n (in an arbitrary children position).
- If the user holds the *insertAsFirst* / *insertAsLast* privilege on node n then she has the right to add new sub-tree as first/last children of node n respectively.
- If the user holds the *insertBefore* / *insertAfter* privilege on node n then she has the right to add new sub-tree as preceding/following sibling nodes of some children of node n respectively.
- If the user holds the *delete* privilege on node n then she has the right to delete some children of node n.
- If the user holds the *replaceNode* privilege on node n then she has the right to replace some children of node n with a sequence of new sub-trees.
- If the user holds the *replaceValue* privilege on node n then she has the right to change the text-content of some children of node n.
- If the user holds the *rename* privilege on node n then she has the right to rename some children of node n.

According to these semantics and using our notion of update type (Definiton 5.1), we say that a node n is *updatable* w.r.t. update type ut if the user has the ut privilege on node n. For instance, if the user has the *insertInto*$[B]$ privilege on node n (i.e. node n is updatable w.r.t. *insertInto*$[B]$), then some nodes of type B can be inserted as children of n. Additionally, if a

node n is updatable w.r.t. $\texttt{replaceNode}[B_i,B_j]$, then children of n with type B_i can be replaced with some nodes of type B_j.

More formally, we define the *node updatability* property as follows:

Definition 5.3. Let $S_{up}=(D, ann_{up})$ be an update specification and ut be an update type. A node n in an instantiation of D is *updatable* w.r.t. ut if the following conditions hold:

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

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

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

Example 5.5. We consider the following update annotation:

$ann_{up}(intervention, \texttt{delete}[Tresult])$ = [↓::*doctor*/↓::*dname*=$DOCTORNAME]

Accordingly, the update **delete** ↓$^+$::*treatment*[↓::*type*='*surgery*']/↓::*Tresult* of doctor *Imine* has no effect over the XML tree of Figure 5.3 since the node concerned by this update is $Tresult_3$, so his parent node $treatment_3$ must be updatable w.r.t. $\texttt{delete}[Tresult]$ which is no longer the case: According to Definition 5.3, no annotation of type $\texttt{delete}[Tresult]$ is defined over element type *treatment*; thus node $treatment_3$ inherits his authorization from the first ancestor node that is concerned by an annotation of type $\texttt{delete}[Tresult]$; i.e. the node $intervention_{11}$. However, the previous annotation is not valid at node $intervention_{11}$. □

Given an update specification $S_{up}=(D, ann_{up})$, we define two predicates $\mathcal{U}^1_{ut}$ and $\mathcal{U}^2_{ut}$ (expressed in fragment $\mathcal{X}^{\Uparrow}_{[n]}$) to satisfy the conditions (*i*) and (*ii*) of Definition 5.3 with respect to an update type ut:

$$\mathcal{U}^1_{ut} := \uparrow^*::*[\vee_{(ann_{up}(A,ut)=Y|N|[Q]|N_h|[Q]_h)\in S_{ut}}\ \varepsilon::A][1]$$
$$[\vee_{(ann_{up}(A,ut)=Y)\in S_{ut}}\ \varepsilon::A\ \vee_{(ann_{up}(A,ut)=[Q]|[Q]_h)\in S_{ut}}\ \varepsilon::A[Q]]$$

$$\mathcal{U}^2_{ut} := \wedge_{(ann_{up}(A,ut)=N_h)\in S_{ut}}\ \text{not}\ (\uparrow^+::A)$$
$$\wedge_{(ann_{up}(A,ut)=[Q]_h)\in S_{ut}}\ \text{not}\ (\uparrow^+::A[not(Q)])$$

The predicate $\mathcal{U}^1_{ut}$ has the form $\uparrow^*::*[qual_1][1][qual_2]$. Applying $\uparrow^*::*[qual_1]$ on a node n returns an ordered set $\mathcal{S}$ of nodes (node n and/or some of its ancestor nodes) such that for each one an annotation of type ut is defined over its element type. The predicate $\mathcal{S}[1]$ returns either node n, if an annotation of type ut is defined over its element type; or the first ancestor node of n concerned by an annotation of type ut. Thus, to satisfy condition (*i*) of Definition 5.3, it amounts to check that the node returned by $\mathcal{S}[1]$ is concerned by a valid annotation of type ut; checked by the predicate $\mathcal{S}[1][qual_2]$ (i.e., $n \vDash \mathcal{U}^1_{ut}$). The second predicate is used to check that all downward-closed annotations of type ut defined over ancestor nodes of n are valid (i.e., $n \vDash \mathcal{U}^2_{ut}$).

Definition 5.4. Let $S_{up}=(D, ann_{up})$, ut, and T be an update specification, an update type and an instance of DTD D respectively. We define the *updatability predicate* $\mathcal{U}_{ut}$ which refers to an $\mathcal{X}^{\Uparrow}_{[n]}$ qualifier such that, a node n on T is *updatable* w.r.t. ut iff $n \vDash \mathcal{U}_{ut}$, where $\mathcal{U}_{ut} := \mathcal{U}^1_{ut} \wedge \mathcal{U}^2_{ut}$. □

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

Property 5.1. For an update specification S_{up}=(D, ann_{up}) and an update type ut, the updatability predicate $\mathcal{U}_{ut}$ can be constructed in at most $O(|ann_{up}|)$ time. Moreover, $|\mathcal{U}_{ut}|$=$O(|ann_{up}|)$. □

Proof 5.1. Given an update specification S_{up}=(D, ann_{up}) and an update type ut, the updatability predicate $\mathcal{U}_{ut}$ of Definition 5.4 is defined over the set S_{ut}. Intuitively, the definition of the set S_{ut} depends on the parsing of all annotations of S_{up} (i.e., the set $\{ann_{up}\}$) in $O(|ann_{up}|)$ time. The construction of each predicate $\mathcal{U}^1_{ut}$ and $\mathcal{U}^2_{ut}$ over annotations of S_{ut} takes $O(|S_{ut}|)$ time. Thus, the updatability predicate $\mathcal{U}_{ut}$ can be constructed in at most $O(|S_{ut}|+|ann_{up}|)$=$O(|ann_{up}|)$ time (since $|S_{ut}| \leq |ann_{up}|$). It is clear that we can have in at most $|\mathcal{U}^1_{ut}| = 2 * |ann_{up}|$ and $\mathcal{U}^2_{ut} = |ann_{up}|$. Thus, $|\mathcal{U}_{ut}|$=$O(|ann_{up}|)$. □

Example 5.6. We define the following update annotations according to the update rights of nurses defined in Example 5.2:

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

According to this update policy, the predicate $\mathcal{U}_{ut} := \mathcal{U}^1_{ut} \wedge \mathcal{U}^2_{ut}$ is defined with:

$$\mathcal{U}^1_{\textbf{replaceValue}[symptom]} := \uparrow^*::*[\varepsilon::department \vee \varepsilon::patients \vee \varepsilon::sibling][1] \\ [\varepsilon::department[\downarrow::name=\$\text{NURSEDEPT}] \\ \vee \varepsilon::patient[\downarrow::wardNo=\$\text{NURSEWARDNO}]]$$

$$\mathcal{U}^2_{\textbf{replaceValue}[symptom]} := \text{not } (\uparrow^+::department[\text{not } (\downarrow::name=\$\text{NURSEDEPT})]) \wedge \\ \text{not } (\uparrow^+::sibling)$$

Consider the case of the nurse having the ward number *421* and working at *Critical care* department. The predicate $\uparrow^*::*[\varepsilon::department \vee \varepsilon::patients \vee \varepsilon::sibling]$ over the node $patient_3$ of Figure 5.2 returns the ordered set $\mathcal{S}$=$\{patient_3, patient_2, patient_1, department_1\}$ of nodes (each one is concerned by an annotation of type **replaceValue**[symptom]); $\mathcal{S}[1]$ returns $patient_3$ and the predicate $[\varepsilon::department[\downarrow::name='Critical\ care'] \vee \varepsilon::patient[\downarrow::wardNo='421']]$ is valid at node $patient_3$ (i.e. $patient_3 \vDash \mathcal{U}^1_{\textbf{replaceValue}[symptom]}$). Also, we can see that $patient_3 \vDash \mathcal{U}^2_{\textbf{replaceValue}[symptom]}$. Consequently, the node $patient_3$ is updatable w.r.t. **replaceValue**$[symptom]$ (i.e., $patient_3 \vDash \mathcal{U}_{\textbf{replaceValue}[symptom]}$). This means that the user is granted to update text-content of *symptom* elements of $patient_3$ (e.g. node $symptom_3$). However, for node $patient_5$ we can check that the predicate $\mathcal{U}^1_{\textbf{replaceValue}[symptom]}$ is valid, while it is no longer the case for the predicate $\mathcal{U}^2_{\textbf{replaceValue}[symptom]}$ ($patient_5$ has an ancestor node $department_2$ with $name \neq$ *'Critical care'*). Thus, no update of type **replaceValue**$[symptom]$ can be applied at node $symptom_5$. □

Proposition 5.1. For any update specification $S_{up} = (D, ann_{up})$, any update operation ut, and any XML $T \in \mathcal{T}(D)$, the updatability of each node in T w.r.t. ut is uniquely defined, i.e., it is either updatable or not updatable.

5.2.2 Update Operations Rewriting

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

Delete/Replace Operations. According to our model of update, if the user holds the $\mathtt{delete}[A]$ right on a node n then he can delete children nodes of n of type A. Thus, given the update operation "**delete** *target*", for each node n of type A_i referred to by *target*, parent node n' of n must be updatable w.r.t $\mathtt{delete}[A_i]$ (i.e., $n' \vDash \mathcal{U}_{delete[A_i]}$). To this end, the *target* expression of $\mathtt{delete}$ operations can be rewritten into: $target[\bigvee_{A_i \in Ele} \varepsilon{::}A_i[\uparrow{::}*[\mathcal{U}_{delete[A_i]}]]]$. Consider now the update operation "**replace** *target* **with** *source*". A node n of type A_i referred to by *target* can be replaced with nodes in *source* if its parent node n' is updatable w.r.t $\mathtt{replace}[A_i,B]$ (i.e., $n' \vDash \mathcal{U}_{replace[A_i,B]}$). Therefore, the *target* expression of the replace operation can be rewritten into: $target[\bigvee_{A_i \in Ele} \varepsilon{::}A_i[\uparrow{::}*[\mathcal{U}_{replace[A_i,B]}]]]$.

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

Insert into Operation. In the following we assume that: if a node n is concerned by an annotation of type $\mathtt{insertInto}[B]$, then this annotation implies $\mathtt{insertAsFirst}[B]$ (resp. $\mathtt{insertAsLast}[B]$) rights for n, and $\mathtt{insertBefore}[B]$ (resp. $\mathtt{insertAfter}[B]$) rights for children nodes of n (inspired from [FM07]). In other words, if one can(not) insert children nodes of types B at any child position of some node n as specified by some annotations of type $\mathtt{insertInto}[B]$, then one can(not) insert nodes of type B in the first and last child position of n and in preceding and following sibling of children nodes of n (unless if there is some annotations of type $\mathtt{insertAsFirst}[B]$, $\mathtt{insertAsLast}[B]$, $\mathtt{insertBefore}[B]$, or $\mathtt{insertAfter}[B]$ respectively that specify otherwise). Thus, one can execute the update operation "**insert** *source* **into** *target*" over an XML tree T iff: (i) one has the right to execute update operations of type $\mathtt{insertInto}[B]$ on the node n ($n \in \mathcal{TS}[\![target]\!]$); and (ii) no annotation *explicitly prohibits* update operations of type $\mathtt{insertAsFirst}[B]/\mathtt{insertAsLast}[B]$ on node n (resp. $\mathtt{insertBefore}[B]/\mathtt{insertAfter}[B]$ on children nodes of n). When condition (ii) does not hold (e.g. update operations of type $\mathtt{insertAsFirst}$ is explicitly denied), this leads to situation where there is a *conflict* between $\mathtt{insertInto}$ and other insert operations.

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

an ancestor node n' of n such that: n' is the first ancestor node of n concerned by an annotation of type ut, and this annotation is invalid at n'; c) there is an ancestor node of n concerned by an invalid downward-closed annotation of type ut.

More formally, for an update specification S_{up}=(D, ann_{up}), we define the predicate $\mathcal{U}^{-1}_{ut} := Cnd_{a\vee b} \bigvee Cnd_c$ over an update type ut with:[33]

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

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

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

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

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

Theorem 5.2. For any update specification S_{up}=(D, ann_{up}) and any update operation *op* (defined in $\mathcal{X}$), there exists an algorithm "*Rewrite Updates*" that translates *op* into a safe one *op*′ (defined in $\mathcal{X}^{\Uparrow}_{[n]}$) in at most $O(|ann_{up}|)$ time. □

Proof. Our algorithm "*Rewrite Updates*" for XML update operations rewriting is given in Figure 5.4. As explained in Section 5.2.2, for any update specification S_{up}=(D, ann_{up}) with DTD D=$(Ele, Rg, root)$, the securing of an update operation *op* consists in the rewriting of its *target* expression (defined in $\mathcal{X}$) into a safe one *target*′ (defined in $\mathcal{X}^{\Uparrow}_{[n]}$) in order to refer only to XML nodes that can be updated by the user w.r.t S_{up}. Proving that *target*′ can be defined in $O(|ann_{up}|)$ time is intuitive and based on the proof of Property 5.1:

• A *delete* operation can be rewritten by adding the following predicate $[\bigvee_{A_i\in Ele} \varepsilon::A_i[\uparrow::*[\mathcal{U}_{delete[A_i]}]]]$ to its *target* expression. For each element type A_i in DTD D, $S_{delete[A_i]}$ is a subset of $\{ann_{up}\}$, i.e., $\bigcup_{A_i\in Ele} S_{delete[A_i]} \subseteq \{ann_{up}\}$. All these subsets can be computed by parsing only one time the set $\{ann_{up}\}$, i.e., in $O(|ann_{up}|)$ time. Next, each sub-predicate $\mathcal{U}_{delete[A_i]}$ is defined over the subset $S_{delete[A_i]}$ in $O(|S_{delete[A_i]}|)$ time, and all sub-predicates used in line 4 of Figure 5.4 can be defined in $O(\sum_i |S_{delete[A_i]}|)$=$O(|ann_{up}|)$ time. Therefore, the predicate $[\bigvee_{A_i\in Ele} \varepsilon::A_i[\uparrow::*[\mathcal{U}_{delete[A_i]}]]]$ can be defined in at most $O(|ann_{up}|)$ time, which is the rewriting time of *delete* operations. The same principle is applied for *replace* operations.

• For an *insertAsFirst* operation (resp. *insertAsLast*, *insertBefore*, and *insertAfter*) defined with *source* of nodes conform to type B, only one predicate is used to rewrite this operation; the predicate $[\mathcal{U}_{insertAsFirst[B]}]$ is constructed in at most $O(|ann_{up}|)$ time.

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

Algorithm: *Rewrite Updates*

input : An update specification S_{up}=(D, ann_{up}) and an update operation *op*.
output: a rewritten of *op* w.r.t S_{up}.

1 let D=$(Ele, Rg, root)$;
2 let *op* be defined with *target* and optional sequence *source* of nodes which conform to type B;
3 **case** (*delete operation*) :
4 | $target' := target[\bigvee_{A_i \in Ele} \varepsilon::A_i[\uparrow::*[\mathcal{U}_{delete[A_i]}]]]$;
5 **case** (*replace operation*) :
6 | $target' := target[\bigvee_{A_i \in Ele} \varepsilon::A_i[\uparrow::*[\mathcal{U}_{replace[A_i,B]}]]]$;
7 **case** (*insertAsFirst operation*) :
8 | $target' := target[\mathcal{U}_{insertAsFirst[B]}]$;
| *//same principle for* ***insertAsLast***, ***insertBefore***, *and* ***insertAfter*** *operations*;
9 **case** (*insertInto operation*) :
10 | $CRP_B := \mathcal{U}^{-1}_{insertAsFirst[B]} \bigvee \mathcal{U}^{-1}_{insertAsLast[B]}$
| $\bigvee \downarrow::*[\mathcal{U}^{-1}_{insertBefore[B]}] \bigvee \downarrow::*[\mathcal{U}^{-1}_{insertAfter[B]}]$;
11 | $target' := target[\mathcal{U}_{insertInto[B]} \bigwedge not(CRP_B)]$;
12 replace *target* of *op* with $target'$;
13 **return** *op*;

Figure 5.4: XML Update Operations Rewriting Algorithm.

• An *insertInto* operation defined with *source* of nodes conform to type B is rewritten by adding the predicate $[\mathcal{U}_{insertInto[B]} \bigwedge not(CRP_B)]$ to its *target* expression (line 11 of Figure 5.4). The predicate $\mathcal{U}_{insertInto[B]}$ is constructed in at most $O(|ann_{up}|)$ time, while the predicate CRP_B is based on the definition of some other predicates $\mathcal{U}^{-1}_{ut}$ for each update type *ut* in {*insertAsFirst*[B], *insertAsLast*[B], *insertBefore*[B], *insertAfter*[B]}. Similarly to the updatability predicate, the construction of each predicate $\mathcal{U}^{-1}_{ut}$ takes at most $O(|ann_{up}|)$ time (the same proof as Property 5.1). Thus, the overall complexity time of the rewriting of *insertInto* operations is $O(5 * |ann_{up}|)$=$O(|ann_{up}|)$ time. □

5.3 Conclusion

We have proposed a general model for specifying XML update policies based on the primitives of XQuery Update Facility. To enforce such policies, we have introduced a rewriting approach to securely updating XML over arbitrary DTDs and for a significant fragment of XPath. In the future, we will investigate the secure of XML data in the presence of security views. We plan to study the inference problem that can be caused by combining read and update privileges, and we aim propose solution to deal with such a problem.

6

Implementation and Experimental Study: The SVMAX framework

Contents

We study first the problem of rewriting XPath queries [BBC+10] over virtual views that can be stated as follows: Given a DTD D, a (recursive) security view S=(D_v,ann), and an XPath query Q over D_v. The rewriting problem consists in defining a rewriting function $\mathcal{R}$ that computes another XPath query $\mathcal{R}(Q)$ over the original document D such that: for any instance T of D and its view T_v computed w.r.t S, the evaluation of Q on T_v yields the same result as the evaluation of $\mathcal{R}(Q)$ on T. Furthermore, we attempt to secure the update primitives of the XQuery Update Facility [BCFF+10].

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 (see [MI12e, MI12c] for more details). Both of read and update rights of SVMAX (denoted in the following by ann_{read} and ann_{up} resp.) are defined by annotating DTD grammars. In case of security view S=(D_v, ann_{read}), the update rights ann_{up} (defined over the original DTD D) must be enforced by taking

into account the read-access rights ann_{read} and this in order to preserve confidentiality of sensitive information that can be disclosed by performing update operations [MI12a]. The combination problem of read and update rights, can be stated as follows: Given a DTD D, a (recursive) security view S=(D_v,ann_{read}), some update privileges ann_{up}, and an update operation *op* defined over D_v. A safe rewriting of the operation *op* consists in defining a rewriting function $\mathcal{R}$ that translates *op* into another one $\mathcal{R}(op)$ defined over the original document D such that: for any instance T of D and its view T_v computed w.r.t S, updating T_v with *op* yields the same result as the update of T with $\mathcal{R}(op)$.

SVMAX provides a safe solution to thoroughly combine read and update privileges in order to preserve confidentiality and integrity of XML data. Our goal at the outset was to secure XPath queries and the update operations of XQuery Update Facility. SVMAX is well-suited to efficiently rewrite such queries and updates, and to be integrated within database systems that provide support for the previous W3C standards.

Contributions. To the best of our knowledge, SVMAX is the first system that provides efficient support for securing both access and update queries over arbitrary XML views. The main features of SVMAX are the following:

• Along the same lines as [FCG04a], SVMAX supports the definition of read and update policies by annotating a DTD grammar with some privileges defined as XPath expressions.

• SVMAX provides use of a significant class of XPath queries (including descendant and upward axes).

• A large set of W3C XQuery Update operations is supported by SVMAX.

• SVMAX is able to rewrite XPath queries (resp. update operations) posed on possibly recursive XML views to be evaluated over original XML documents.

• Policies are defined through a visual tool (see Section 6.2) that helps users to annotate DTD grammars and define read and update privileges.

• A simple and efficient validation approach is presented by SVMAX that has to eliminate any auxiliary structure and then avoid maintenance and storage costs.

• Users can either perform static validation[34] or choose to automatically execute our validation checking each time an update is being performed.

• SVMAX implements a general algorithm for the definition of (approximated) DTD view.

• SVMAX can run either as a singleton or integrated as an API within some database systems.

We reproduce our Hospital DTD in Figure 6.1 and we define .

6.1 SVMAX Policies

An access control policy is a set of rules that determine whether a user is allowed to perform some action on the data. SVMAX proposes general and expressive models to specify both read and update access control policies. Our policies specification is based on the notion of DTD annotation where security labels are associated to production rules of the DTD specifying read as well as update privileges. We use the XPath fragment $\mathcal{X}^{\Uparrow}$ (defined in Section 4.5) to specify our policies as explained in the following.

[34] Validation from scratch of the whole XML document.

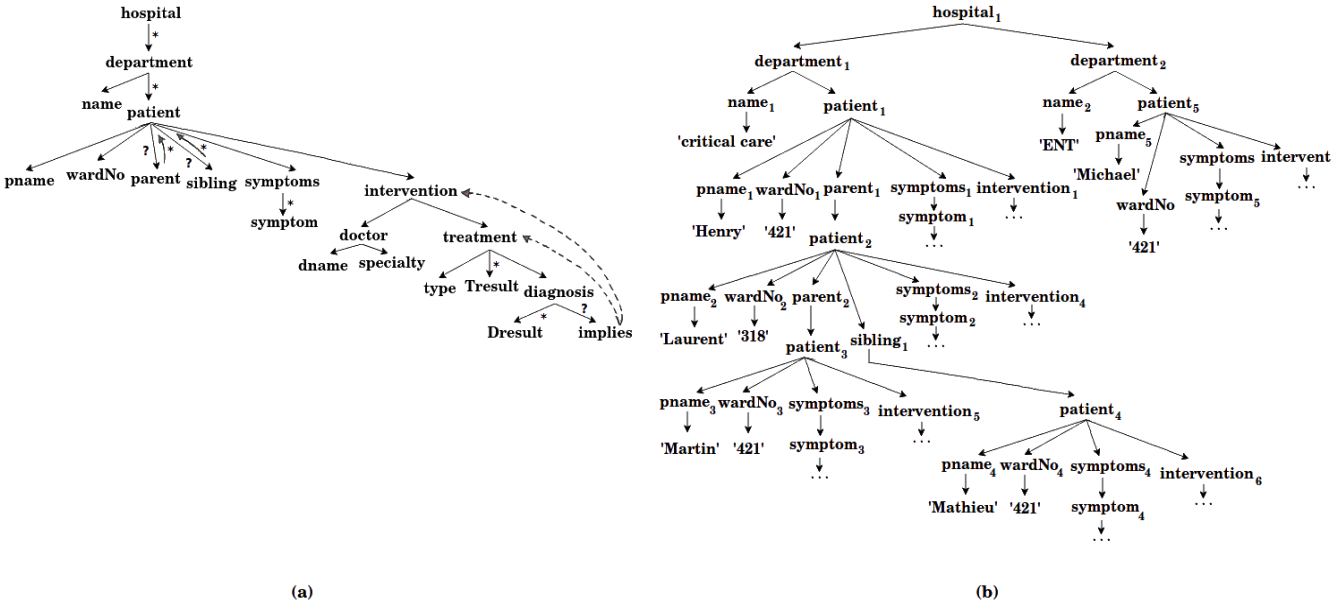


Figure 6.1: Example of a) Hospital DTD, and b) part of a valid XML document.

6.1.1 Read-Access Policies

An *access specification* S_{read} is defined as a pair (D, ann_{read}) consisting of a document DTD D and a partial mapping ann_{read} such that, for each production rule $A \rightarrow Rg(A)$ and each element type B occurring in $Rg(A)$, $ann_{read}(A, B)$, if explicitly defined, is an annotation of the form: $ann_{read}(A, B) := value$, where $value$ defines the user *access right* on B children of A elements, and can be Y, N, or $[Q]$ (Q is an $\mathcal{X}^{\Uparrow}$ predicate). The specification values Y , N , and $[Q]$ indicate that the B children of A elements in an instance of D are *accessible*, *inaccessible*, or *conditionally accessible* respectively. If $ann_{read}(A, B)$ is not explicitly defined, then B *inherits* the accessibility of A. On the other hand, if $ann_{read}(A, B)$ is explicitly defined it may *override* the accessibility inherited from A. The root is accessible by default.

Other specification values are proposed by SVMAX, denoted by N_h and $[Q]_h$, and called *downward-closed annotations*. In a nutshell, for a child B of an element A, if the annotation $ann_{read}(A, B)$=$N_h|[Q]_h$ (where $B \not\models Q$), then all the subtree rooted at B is inaccessible and no annotation defined over children of B can override this forbidden access.

Example 6.1. We consider the hospital DTD of Figure 6.1 (a) and define the access rights of nurses as follows:

R_1: $ann_{read}(hospital, department)$=$[\downarrow::name$=$\$\text{NURSEDEPT}]_h$

R_2: $ann_{read}(department, patient)$=$[\downarrow::wardNo$=$\$\text{NURSEWARD}]$

R_3: $ann_{read}(parent, patient)$=$[\downarrow::wardNo$=$\$\text{NURSEWARD}]$

R_4: $ann_{read}(patient, sibling)$=$N_h$

These annotations specify that each nurse can only access the data of patients in a department having a certain name (denoted by $NURSEDEPT) and which are being treated in a ward with a certain number (denoted $NURSEWARD). Moreover, nurses are not authorized to access to sibling data. Consider now the XML document of Figure 6.1 (b) and the case of nurse working

at *critical care* department and having the ward number *421*. This nurse can access to only data of $patient_1$ and $patient_3$. Note that the annotation R_1 must be defined as downward-closed, otherwise the negative authorization inherited from $department_2$ will be overridden by the annotation R_2 allowing access to $patient_5$ which does not belong to *critical care* department. Moreover, the node $sibling_1$ is inaccessible and the annotation R_4 is defined as downward-closed in order to avoid overriding of this forbidden access along the subtree rooted at $sibling_1$. □

6.1.2 Update-Access Policies

We define an *update type* (*ut*) as a restriction of an W3C update operation to be applied only for a specific element type. Given a DTD D, we define the following update types: *insertInto*[B_i], *insertAsFirst*[B_i], *insertAsLast*[B_i], *insertBefore*[B_i,B_j], *insertAfter*[B_i,B_j], *delete*[B_i], *replaceNode*[B_i,B_j], *replaceValue*[B_i], and *rename*[B_i,B_j], where B_i and B_j are element types of D. For instance, the update type *delete*[B_i] refers to all operations that delete element nodes of type B_i. Moreover, the update type *replaceNode*[B_i,B_j] refers to all operations that replace element nodes of type B_i with a sequence of nodes of type B_j.

An *update specification* S_{up} is defined as as a pair (D, ann_{up}) where D is a DTD and ann_{up} is a partial mapping such that, for each element type A in D and each update type ut defined over D element types, $ann_{up}(A, ut)$, if explicitly defined, is an annotation of the form: $ann_{up}(A, ut) ::= Y \mid N \mid [Q] \mid N_h \mid [Q]_h$. Table 5.1 presents the semantics of our update annotations [35]. For instance, the values Y, N, and $[Q]$ indicates that the user is *authorized*, *unauthorized*, or *conditionally authorized* respectively, to perform update operations of type ut over subtrees rooted at A elements.

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

6.1.3 Policies Enforcement

As the same lines as [FCG04a], we use the query rewriting principle to enforce our policies. We denote by $\mathcal{S}^{read}$ and $\mathcal{S}^{up}$ the sets of all access and update specifications respectively that can be defined with SVMAX. We define two rewriting functions $\mathcal{R}^{read}$ and $\mathcal{R}^{up}$ as follows:

$\mathcal{R}^{read}$: $\mathcal{S}^{read} \times \mathcal{X}^{\Uparrow} \rightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$

$\mathcal{R}^{up}$: $\mathcal{S}^{read} \times \mathcal{S}^{up} \times \mathcal{X}^{\Uparrow} \rightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$

For any access specification S_{read}=(D, ann_{read}), an XML tree T that conforms to D, and its virtual view T_v computed w.r.t S_{read}. We use the function $\mathcal{R}^{read}$ to translate any $\mathcal{X}^{\Uparrow}$ query Q posed over T_v into another one, defined in $\mathcal{X}^{\Uparrow}_{[n,=]}$, such that: $Q(T_v) = \mathcal{R}^{read}(S_{read}, Q)(T)$. Recall that when a security view is defined, update specifications are defined over the view of the DTD since one cannot define update rights over inaccessible data. Given an update specification S_{up}=(D_v, ann_{up}), where D_v is the DTD view of D computed w.r.t S_{read}, the user is provided by the virtual view T_v over which he formulates his update operations. These operations must be thoroughly rewritten to be safely performed over the original data. To preserve confidentiality and integrity of XML data, SVMAX translates update operations posed over the view w.r.t both read and update rights. Thus, for an update operation *op* posed over T_v with target expression

[35] The semantics of the other annotations are deduced with the same principle.

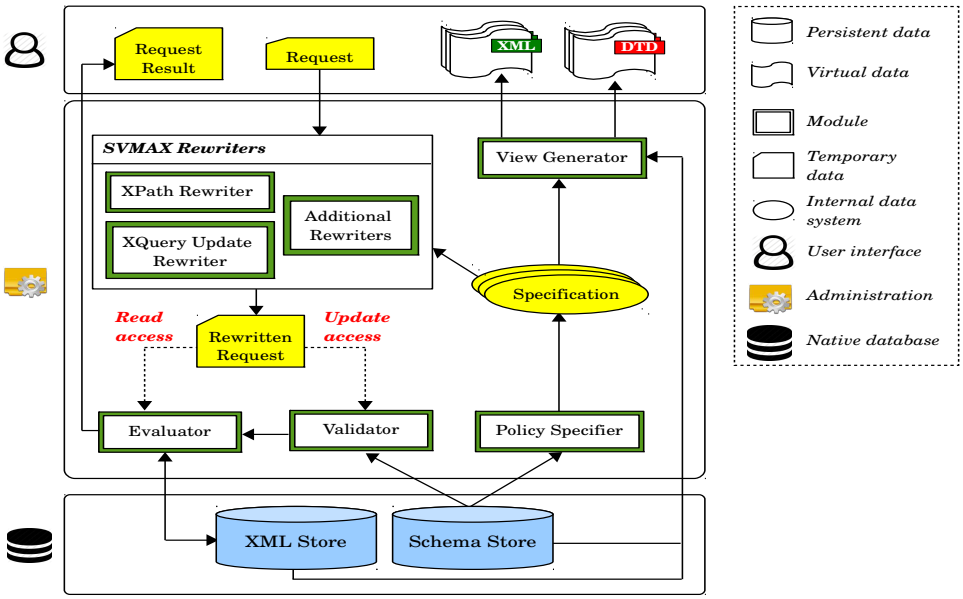


Figure 6.2: SVMAX Overall Architecture

defined in $\mathcal{X}^{\Uparrow}$, we use the function $\mathcal{R}^{up}$ to rewrite *op* into another one $\mathcal{R}^{up}(S^{read}, S^{up}, op)$ that, when evaluated over T, makes changes of only nodes that are accessible w.r.t S_{read} and can be updated w.r.t S_{up}. Let op' be the update operation *op* where its *target* expression is replaced by $\mathcal{R}^{read}(S_{read}, target)$ (i.e. rewritten w.r.t S_{read}). We should emphasize that: $op'(T_v) = \mathcal{R}^{up}(S^{read}, S^{up}, op)$.

Finally, consider the case where no security view is defined ($S_{read} = \phi$), then the update specification S_{up}=(D, ann_{up}) is defined over the original DTD D, and user update operations are posed directly over the original XML tree T. In this case, for any update operation *op* with target expression defined in $\mathcal{X}^{\Uparrow}$ over T, the evaluation of $\mathcal{R}^{up}(\phi, S^{up}, op)$ over T takes effect only at nodes that are updatable w.r.t S_{up}.

6.2 System Overview

The overall architecture of our system is depicted in Figure 6.2. SVMAX is composed by the following major modules: 1) a *Policy Specifier*, 2) a *View Generator*, 3) an *XPath Rewriter*, 4) an *XQuery Update Rewriter*, 5) and the *Validator*. 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 [Bou10].

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 safely evaluated over the original XML data and evaluation results are returned to the user. The SVMAX$^{\mathcal{V}}$ provides some additional features like view *materialization* and *static validation*. Although these additional modules are implemented for purpose of comparison, we keep them in our system as they may be useful in some cases: 1) the data view materialization is useful in the case where a small XML data are accessed only by a

few users, which means that materializing all views from the server may not be fastidious; 2) the static validation is provided *temporarily* by our system since validation of *composite updates* [36] is still remains an ongoing work.

6.2.1 Policy Specifier

As shown in Figure 6.3, SVMAX$^{\mathcal{V}}$ displays DTDs in both text format and graph format which are useful in case of simple and complex DTDs respectively. The graph depicted in this figure is the representation of our hospital DTD. For each group of users and a given DTD, the administrator can specify one and only one access and/or update specification(s). Specifications are directly written in text as done in Example 6.1. For instance, the bottom part of the interface of Figure 6.3 concerns the specification of access policies. After, the administrator chooses the XML documents (that conform to the used DTD) for which the defined specifications should be enforced.

6.2.2 View Generator

As shown in Figure 6.2, the *View Generator* module is responsible of the generation of *virtual* views of the DTD and the XML data. Given a document DTD D, an access specification S_{read}=(D, ann_{read}), and an XML document T that conforms to D. For each group of users, SVMAX$^{\mathcal{V}}$ generates a virtual view T_v w.r.t S_{read} by hiding all inaccessible parts of T. For instance, the tree at the right part of Figure 6.4 represents the virtual view of the XML data of Figure 6.1 (b), computed w.r.t the nurses rights of Example 6.1.

In order to overcome costs of views materialization and maintenance, the users data views are necessarily virtual, i.e., they are not materialized and are automatically eliminated once the users disconnect from the server.

The user needs a view of the DTD to formulate and optimize his queries and updates. This view must display only information of accessible data. Once S_{read} is generated as explained above, SVMAX$^{\mathcal{V}}$ automatically generates a DTD view D_v of D which is provided as a graph to the user. Consider for instance the access rights of nurses defined in Example 6.1. The DTD view that corresponds to these rights is computed by simply hiding the *sibling* element type as can be seen in the DTD graph of Figure 6.4.

We should emphasize that in case of recursive DTDs, the view generation is not always guaranteed [GSC$^+$09]. More specifically, hiding some information from the DTD may result in a context-free grammar that cannot be captured with a regular grammar [37]. In such situations, our *View generator* module generates an *approximated* DTD view. Our approximations are based on the the well-known sufficient conditions for regularization of context-free grammars [ACC04]. We should note that no algorithm exists in the literature for the derivation of recursive views. Our main contribution in this context is to propose a general algorithm that efficiently computes either the exact view of the DTD (if this is possible) or an approximated version of it.

6.2.3 Rewriters

The SVMAX leverages the expressiveness of XPath to overcome the query rewriting limitation discussed in Chapter 4, and this by using the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ (defined in Section 4.5). The basic rewriting modules of our system are the *XPath Rewriter* and the *XQuery Update Rewriter* that

[36] Set of different atomic updates of Section 3.5.

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

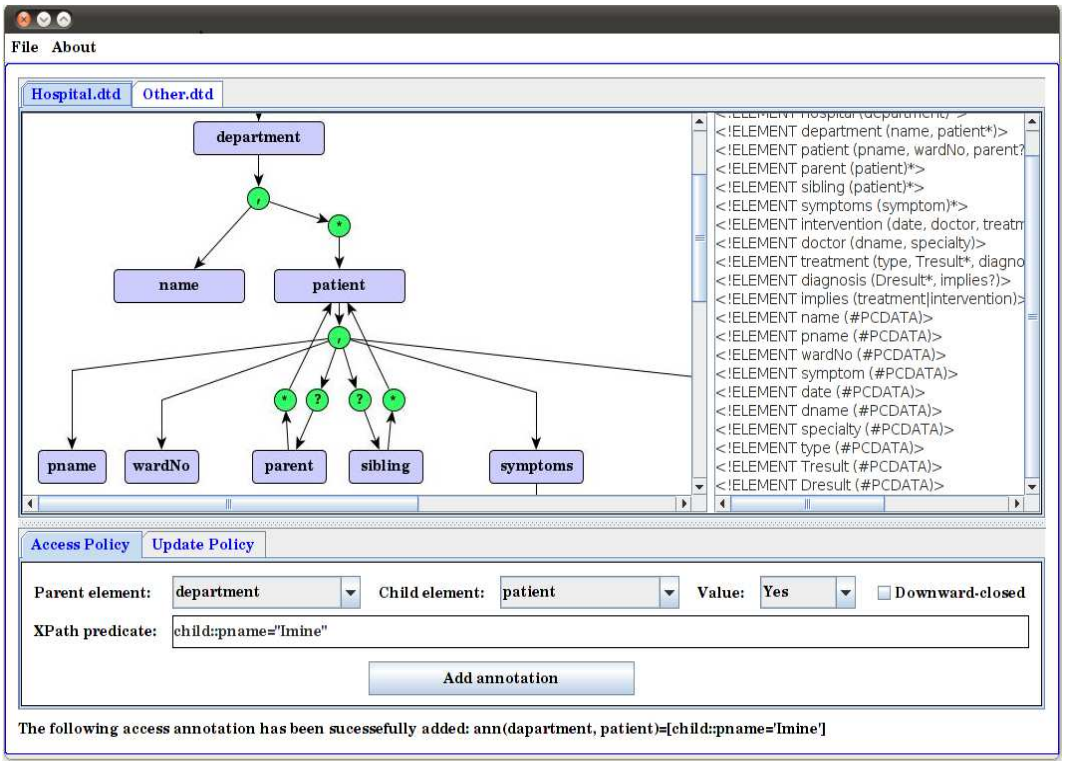


Figure 6.3: Specification of policies using SVMAX$^{\mathcal{V}}$

implement respectively the rewriting functions $\mathcal{R}^{read}$ and $\mathcal{R}^{up}$ of Section 6.1.3. In the following, we describe briefly the main principle of these functions.

Consider first the case of access privileges. For an access specification S_{read}=(D, ann_{read}), we define an $\mathcal{X}^{\Uparrow}$ predicate $\mathcal{A}$, called the *accessibility predicate*, such that: for any XML tree $T \in \mathcal{T}(D)$, a node n of T is *accessible* w.r.t S_{read} (i.e. n appears in the view T_v of T) iff: $n \vDash \mathcal{A}$. Each XPath query Q is rewritten, using the accessibility predicate, to return all and only accessible nodes referred to by Q. We show through the following example the rewriting of different axes of XPath.

Example 6.2. Consider the nurses access rights defined in Example 6.1, the query $\downarrow^*::parent/\downarrow::patient$ over the XML document of Figure 6.1 (b) must be rewritten in order to return only accessible *patient* elements that are immediate children of an accessible element *parent*. We define also the queries $\downarrow^*::parent[\downarrow::patient/\downarrow::pname='Laurent']$; and $\downarrow^*::symptoms/\uparrow::patient[\downarrow::pname='Henry']$. We safely rewrite these queries respectively into Q_1, Q_2, and Q_3 defined as follows:

Q_1: $\downarrow^*::patient[\mathcal{A}][\uparrow^+::*[\mathcal{A}][1]/\varepsilon::parent]$

Q_2: $\downarrow^*::parent[\mathcal{A}][\downarrow^+::patient[\mathcal{A}][\downarrow^+::pname[\mathcal{A}]$
$[\varepsilon::* = 'Laurent']/\uparrow^+::*[\mathcal{A}][1] = \varepsilon::*]/$
$\uparrow^+::*[\mathcal{A}][1] = \varepsilon::*]$

Q_3: $\downarrow^*::symptoms[\mathcal{A}][\uparrow^+::*[\mathcal{A}][1]/\varepsilon::patient[\downarrow^+::pname[\mathcal{A}]$
$[\varepsilon::* = 'Henry']/\uparrow^+::*[\mathcal{A}][1] = \varepsilon::*]]$

where the accessibility predicate $\mathcal{A}$ is given by:

$\uparrow^*::*[\varepsilon::department/\uparrow::hospital \vee \varepsilon::sibling/\uparrow::patient$
$\vee \varepsilon::patient/\uparrow::department \vee \varepsilon::patient/\uparrow::parent][1]$
$[\varepsilon::department[\downarrow::name=\$nurseDept] \vee$
$\varepsilon::patient[\downarrow::wardNo=\$nurseWard]][\neg (\uparrow^+::sibling) \wedge$
$\neg (\uparrow^+::department[\downarrow::name \neq \$nurseDept])]$

The evaluation of Q_1 over the XML document of Figure 6.1 (b) returns the node $patient_3$; Q_2 returns an empty set of nodes; while Q_3 returns the node $symptoms_1$. □

For an update specification S_{up}=(D, ann_{up}) and an update type ut, we define an $\mathcal{X}^{\Uparrow}_{[n,=]}$ predicate $\mathcal{U}_{ut}$, called the *updatability predicate*, such that: for any XML tree $T \in \mathcal{T}(D)$, a node n of T can be updated with an operation of type ut iff: $n \vDash \mathcal{U}_{ut}$. More specifically, given an update operation op of type ut and with *target* expression, we rewrite op into another one op' and this by simply replacing its *target* expression with $target[\mathcal{U}_{ut}]$. The rewritten operation op' concerns only nodes that are *updatable* w.r.t S_{up} through updates of type ut. When an access specification S_{read} is defined in addition to S_{up}, the *target* expression of the operation op is rewritten first w.r.t S_{read} into $target'$ (using the *accessibility predicate* as explained above), $target'$ is rewritten then w.r.t S_{up} into $target'[\mathcal{U}_{ut}]$. Let op' be the rewriting of op by replacing its *target* expression with $target'[\mathcal{U}_{ut}]$. We ensure that the execution of op' over any XML tree that conforms to D updates only *accessible* and *updatable* nodes. More details of our rewriting functions can be found in [MI12a].

For integration purpose, SVMAX provides additional modules to rewrite update operations of each proprietary update language of the database systems *Sedna* and *eXist*.

6.2.4 Validator

In most of the incremental validation approaches [BPV04,BLS06], the XML document is preprocessed and all its nodes and their relationships (e.g. parent-child, first/last child, sibling nodes) are presented with some indexed files. The resulted structure requires an additional storage cost[38], must be maintained each time the document is updated, and its size grows logarithmically with the size of the document.

To avoid these costs, we propose an efficient validation approach that translates each update operation op, defined in $\mathcal{X}^{\Uparrow}$ over a DTD D, into a *valid* one op', defined in $\mathcal{X}^{\Uparrow,\Leftrightarrow}_{[n,=]}$, such that: $op'(T) \in \mathcal{T}(D)$ for any XML tree $T \in \mathcal{T}(D)$. Our approach is based on the rewriting principle and consists in adding an $\mathcal{X}^{\Uparrow,\Leftrightarrow}_{[n,=]}$ predicate to the target of the update operation in order to ensure that the content of the elements concerned by the update still valid after the update is done. When the rewritten update is launched to be evaluated by any XQuery Update processor, the rewritten target expression is evaluated first to determine the XML nodes to be updated. In case of an *invalid* update operation op (i.e. $op(T) \notin \mathcal{T}(D)$), the target of the rewritten update op' must return an empty set of nodes, i.e., $op'(T) = T$.

Based on this rewriting principle, SVMAX transforms all update operations into valid ones, without using any auxiliary structure, and in a time that depends only on the size of the document DTD. Our validation approach is applied only for *atomic updates* (i.e. the primitive update of Section 3.5). As the validation of combined operations still remains an ongoing work, SVMAX temporarily provides a static validation to allow validation after sequence of update operations. Through SVMAX$^{\mathcal{V}}$ tool, the users can choose to execute our validation automatically after each atomic update, or to manually perform a static validation.

[38] This is prohibitively expensive in case of native XML databases where collections of large XML documents are stored.

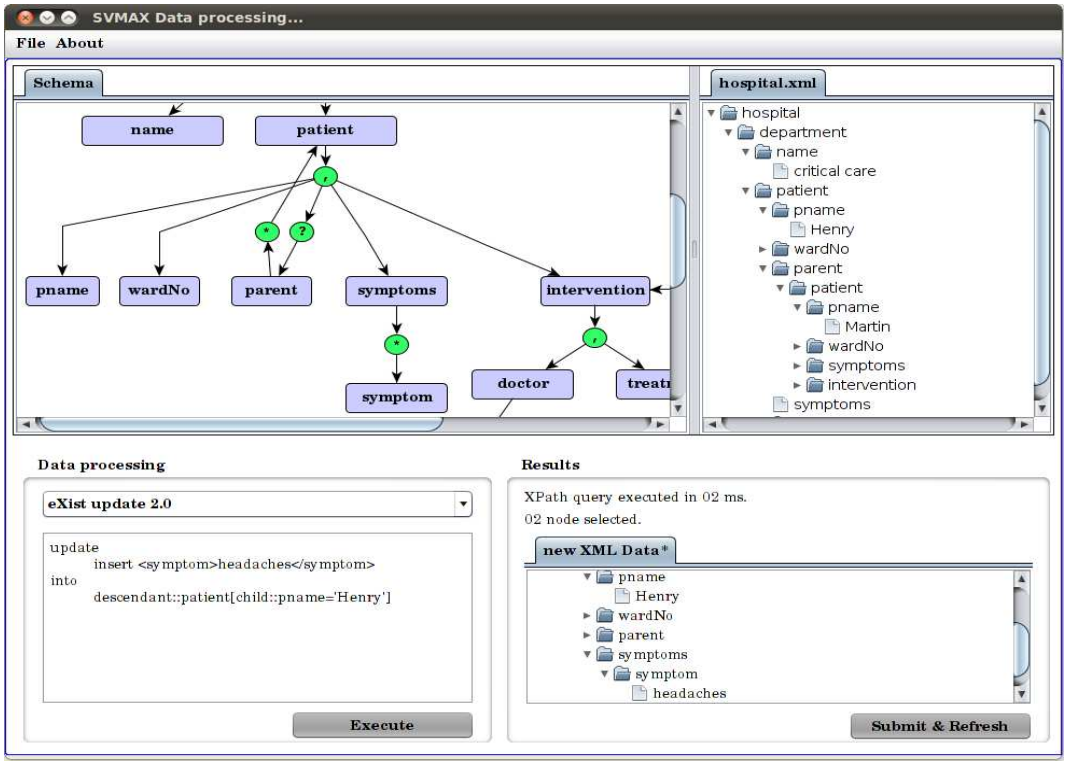


Figure 6.4: Processing XML data using SVMAX$^{\mathcal{V}}$

6.2.5 Evaluation

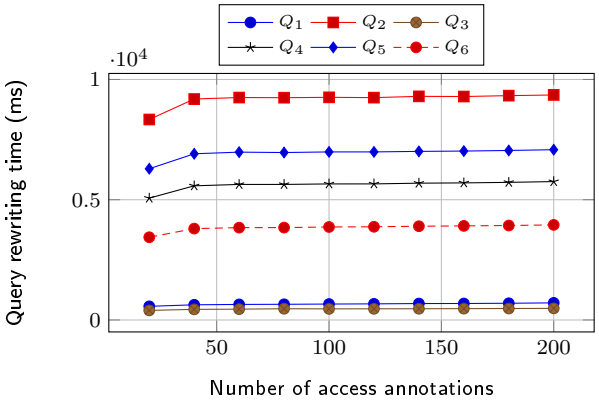
Consider the access rights of Example 6.1. Figure 6.4 represents the system interface that is provided to the nurse. The left part of the figure depicts the hospital DTD view; while the tree at the right part is the view of the original XML data shown in Figure 6.1 (b). Based on these views, the user can formulate her queries and update operations which are rewritten and evaluated over the original data. By default, SVMAX$^{\mathcal{V}}$ uses the *Saxon* processor[39] to evaluate our XPath queries and XQuery update operations. Moreover, other processors can be used to manipulate native XML databases: *BaseX*, *Sedna*, and *eXist* processors.

The evaluation results are the number of returned/updated nodes and consumed time. Each node referred to by an XPath query is returned along with their attributes and descendant nodes. In case of updates however, a virtual version of the updated XML document is shown first, the user can choose to apply the updates over the original data. The user is warned by messages for example in the case of insufficient privileges, or where the updates may result an invalid document.

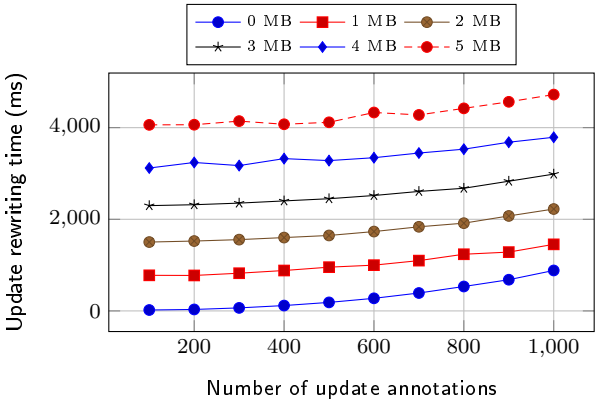
6.3 Performance Study

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 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 (resp.

[39] Available at: `http://www.saxonica.com/welcome/welcome.xml`.



(a)



(b)

Figure 6.5: SVMAX rewriting degradation for (a) read and (b) update rights.

XQuery updates). 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. Since our system can be integrated within existing NXDs, the other concern of experimentation is integration efficiency.

6.3.1 Scalability

We measure the time required by SVMAX for rewriting of general XPath queries and XQuery update operations. We use the complex real-life recursive DTD **GedML**[40] and we generate randomly 10 access and update specifications by varying the number of annotations (from 20 into 200 and from 100 into 1000 annotations for access and update rights resp.). After, we define different XPath queries of size[41] 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). The remaining $\mathcal{X}^{\Uparrow}$ axis are discarded since the rewriting performance of the

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

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

$\downarrow^*$-axis is almost the same as those of $\downarrow^+$, $\uparrow^+$ and $\uparrow^*$. Furthermore, $\uparrow$-axis requires almost the same time as $\downarrow$-axis.

Using SVMAX, we rewrite these queries according to each of the access specifications previously generated. Figure 6.5 (a) 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 subqueries (with the form $axis$::$label$) and rewrites them using a special predicate, *the accessibility predicate* (as explained in Section 6.2.3). 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. 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 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).

Consider now the update rights. We experiment only *insert* operations which have the form "**insert** *source* **before** *target*"; where *source* is a sequence of XML nodes and *target* can be any $\mathcal{X}^{\Uparrow}$ expression. Each update operation of type *ut* must be rewritten by adding an predicate $\mathcal{U}_{ut}$ as explained in Section 6.2.3. The definition of this *updatability predicate* depends only on the number of update annotations and takes a negligible time. In case of *insert* and *replace* operations, SVMAX performs an additional task to check whether all XML nodes defined with parameter *source* are valid w.r.t the DTD. For this reason, the number of update annotations and the size of *source* are the main parameters of our update experimentation.

Thus, we define different *insert* operations by varying the size of the XML nodes to be inserted from 0MB, 1MB,..., 5MB. The update operations defined are rewritten w.r.t the update specifications previously generated (with update annotations varying from 100 into 1000). The rewriting times are depicted in Figure 6.5 (b). We should emphasize that our rewriting times of *insert* and *replace* operations increase with the growth of the size of XML nodes to be inserted (due to the validation check). However, the rewriting time of the remaining update operations takes a constant nature (still independent to the number of update annotations).

6.3.2 Policy Enforcement Performance

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

We define different policies composed by both access and update rights as follows: P^i=$\{S^i_{read}$, $S^i_{up}\}$, where S^i_{read}=$(GedML, ann_{read})$ and S^i_{up}=$(GedML^v, ann_{up})$. As explained in Section 6.1.3, each update specification S^i_{up} is generated w.r.t the access specification S^i_{read}. Moreover, $GedML^v$ is the (approximated) view of the GedML DTD computed w.r.t S^i_{read}. The experiment consists in comparing SVMAX w.r.t the naive approach based on *materialization* [KMR09] and *node-level security annotation* [MTKH06].

Naive process. Consider a policy P^i=$\{S^i_{read}, S^i_{up}\}$. We compute first a materialized view T'_v of T by eliminating all nodes that are inaccessible w.r.t S^i_{read}. After, we annotate the nodes of T'_v, according to ann_{up}, with security attributes with the form @ut="+"/"-", specifying whether updates of type *ut* can be applied on these nodes. For instance, @$insertBefore[B]$="+" at a

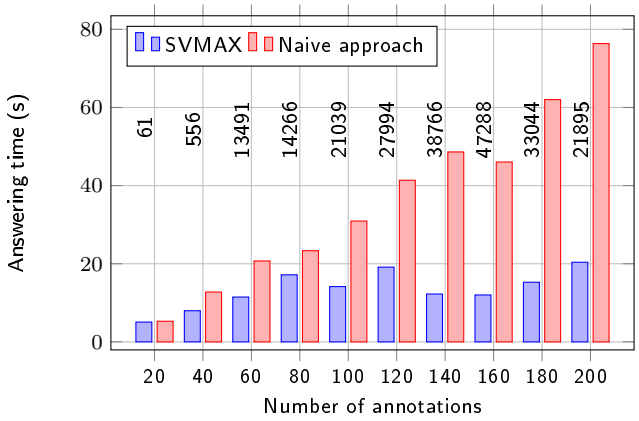


Figure 6.6: Overall answering time: SVMAX versus naive approach.

node n specifies that insertion of B children in sibling position of n is allowed. Let T_v'' be the resulted view which is provided to user to formulate his queries/updates (security labels are hidden from the user). An XPath query is evaluated directly over T_v''; while an update operation of type ut is rewritten first by adding the predicate @ut="+" into its *target* expression, and then evaluated over T_v''.

SVMAX process. Each user update operation is rewritten w.r.t S^i_{read} to ensure confidentiality and S^i_{up} to preserve integrity of data; and then evaluated over the original document T.

Setup. We generate an XML document T of size 10MB that conforms to the GedML DTD, a general update operation, and different policies P^i of size i (i=$|ann_{read}|$=$|ann_{up}|$) varying from 10 to 150. For each policy P^i, the update is rewritten over $2i$ access and update annotations and this using both our approach and the naive approach. Figure 6.6 shows the answering times of each approach[42]. 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.

6.3.3 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 of XML. 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 2MB, a general update operation, and some policies P^1,...,P^{10} defined with the same principle explained in the previous subsection. Using the SVMAX rewriters (the XQuery Update rewriter for *BaseX*, and a special rewriter for both *Sedna* and *eXist* as explained in Section 6.2.3), the update operation is safely rewritten w.r.t the different policies P^i and sent to the underlying database for evaluation. The overall answering times

[42] In the following figures, the numbers of updated nodes are depicted at the middle.

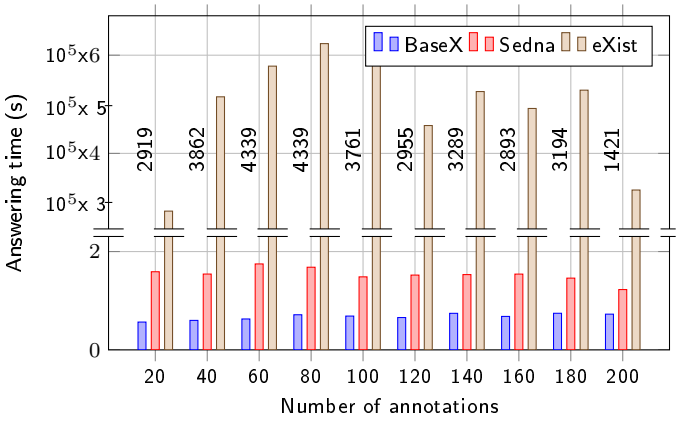


Figure 6.7: Integration of SVMAX within NXDs.

(rewriting and evaluation) are depicted in Figure 6.7. We remark first that *eXist* database takes more time than the other (282 seconds for the simple policy P^1, i.e., with 20 annotations). The *BaseX* XQuery processor overcomes noticeably the *Sedna* update processor in general by up to a factor of 2.

6.4 Conclusion

We plan to extend our system to take into account validation of composite updates. We also intend to study the integration of our system within XML-enabled databases [Bou10], include a consistency repair algorithm, and handle multiple updates in context of collaborative editing [CIR11].

Conclusion and Future Directions

XML has become a standard for representation and exchange of data across the internet. Collaborative work knows a growing and important use following a high availability of efficient hardware. Replication of data within different sites (e.g. XML documents) is used to increase availability of data by minimizing the time of user access to shared data. However, the safety of shared data remains an important issue in such systems. The aim of the thesis is to highlight a formal description of a model of access control of XML documents taking into account the type of access (read or update) and document structure (e.g. DTD grammar DTD or XML schema). On the other hand, the second part of this thesis is to put in practice our models of access control within a collaborative context, taking into account the dynamic aspect of user/group collaboration (a user can join or leave the session together at any time).

Most of existing works around the access control of XML documents are based on the definition of a view showing, for each class of users, the parts of data they are allowed to read and/or modify. This view is the result of the annotation of the DTD grammar, associated to the XML document, by different access conditions expressed as XPath expressions. To prevent access to confidential data, hidden by the view, each XPath query posed by the user in the view must be rewritten in order to be evaluated safely on the original document. However, the rewriting XPath queries in the case of recursive XML views remains an open problem. To overcome this problem, some studies have suggested working with a non-standard language which is more expressive than XPath, called Regular XPath, giving the possibility to define recursive queries through the Kleene star operator (i.e. the operator *). Nevertheless, these studies remain theoretical because no tool is available in practice for the evaluation of Regular XPath queries against XML data. We showed that the rewriting of XPath queries is always possible for recursive XML views without using transformations to other languages (such as Regular XPath), and this only by using the XPath standard. Our solution is based on the extension of the more used fragment of XPath, called *downward closed*, with some XPath operators such as position and the comparison operator. Based on this extension, we have proposed a linear algorithm for rewriting XPath queries against arbitrary XML views (recursive or not). To show the effectiveness of our solution, an experimental study was conducted using a recursive DTD and several forms of XPath queries. The experiment results clearly demonstrate the efficiency of our algorithm.

The majority of existing work around the XML access control only discuss access read. The XML access control considering update operations has not received enough attention. Our second contribution is an expressive model for specifying XML update policies by means of primitive updates of the W3C XQuery Update Facility. Given a DTD grammar D, we annotate the elements of D by different rights of update to specify the parts of the document the user is allowed to change. We have proposed defined a notion of inheritance and overriding of update annotations to overcome the limitations of the existing update access control models. Our approach to enforce these specifications is based on the notion of query rewriting. To do this, we have proposed a linear algorithm based only on the XPath standard, which allows to rewrite

each update operation, defined over an arbitrary DTD (recursive or not), into a safe operation that makes changes of only XML data the user is authorized to update. We studied then the interaction between the read and update rights, and we described a solution that preserves the confidentiality and integrity of XML data.

Our approaches are implemented as a working system that represents the first practical system for specification and enforcement of both read and update rights over arbitrary security views.

Finally, we plan to propose a tool for collaborative editing of documents XML that takes into account our results. Unlike existing work, the tool will be based only on standards (e.g. the XPath standard and the W3C update primitives).

A

Proofs

A.1 Proofs of Chapter 3

Property 3.1. Given an XML tree T with *root* node, we define some equivalences between $\mathcal{X}^{\Uparrow}_{[n,=]}$ queries as follows:

1. If $q_1 \equiv q_2$ then $q_1[f] \equiv q_2[f]$.
2. For any node test $\eta \neq$ *text()*: $\alpha{::}\eta[f_1][\varepsilon{::}*[f_2]] \equiv \alpha{::}*[f_1][\varepsilon{::}\eta[f_2]] \equiv \alpha{::}\eta[f_1 \wedge f_2]$.
3. For any $\mathcal{X}^{\Uparrow}$ predicates f_1 and f_2: $\downarrow^*{::}\eta_1[f_1][\alpha{::}\eta_2[f_2]] \equiv \downarrow^*{::}\eta_2[f_2]/\alpha^{-1}{::}\eta_1[f_1]$.
4. $\alpha_1{::}\eta_1/\ldots/\alpha_k{::}\eta_k \equiv \downarrow^*{::}\eta_k[\alpha_k^{-1}{::}\eta_{k-1}/\ldots/\alpha_2^{-1}{::}\eta_1/\alpha_1^{-1}{::}root]$.
5. $m \in \mathcal{S}[\![\downarrow^*{::}\eta[f]]\!](T)$ if and only if $m \in \xi[\![\varepsilon{::}\eta[f]]\!]$. □

Proof A.1. We consider in the following the XML tree $T = (N, r, R_{\downarrow}, R_{\rightarrow}, \lambda, \nu)$. Note that an XPath query Q over T is evaluated over the root node r of T, i.e. $\mathcal{S}[\![Q]\!](T)=\mathcal{S}[\![Q]\!](\{r\})$. Moreover, it is clear that $n\ \alpha\ m$ is equivalent to $m\ \alpha^{-1}\ n$. We prove now equivalences given in Property 3.1.

1. If $q_1 \equiv q_2$ then $q_1[f] \equiv q_2[f]$. Since $q_1 \equiv q_2$ then $\mathcal{S}[\![q_1]\!](T)=\mathcal{S}[\![q_2]\!](T)$. We have:

$$\mathcal{S}[\![q_1[f]]\!](T) = \mathcal{S}[\![q_1]\!](T) \cap \xi[\![f]\!] = \mathcal{S}[\![q_2]\!](T) \cap \xi[\![f]\!] = \mathcal{S}[\![q_2[f]]\!](T).$$

Therefore, $q_1[f] \equiv q_2[f]$.

2. For any node test $\eta \neq$ *text()*: $\alpha{::}\eta[f_1][\varepsilon{::}*[f_2]] \equiv \alpha{::}*[f_1][\varepsilon{::}\eta[f_2]] \equiv \alpha{::}\eta[f_1 \wedge f_2]$. According to the XPath standard [BBC+10], $\varepsilon{::}*$ over a node m checks whether m is an element node and not a text node, i.e. $m \in \xi[\![\varepsilon{::}*]\!]$ if and only if $\lambda(m) \neq \texttt{str}$. Intuitively, for an element node m, we have:

- $m \in \xi[\![\varepsilon{::}*]\!]$,
- $m \in \xi[\![\varepsilon{::}l]\!]$ iff $\lambda(m)=l$,
- $m \in \xi[\![\varepsilon{::}*[f]]\!]$ iff $m \in \xi[\![f]\!]$,
- $m \in \xi[\![\varepsilon{::}l[f]]\!]$ iff $\lambda(m)=l$ and $m \in \xi[\![f]\!]$.

Firstly, let η be an element type l (the case where $\eta = *$ is briefly discussed later). We consider the following deductions:

(a) Evaluation of $\alpha{::}\eta[f_1][\varepsilon{::}*[f_2]]$ *over* T*:*

- $\underbrace{\mathcal{S}[\![\alpha :: \eta[f_1][\varepsilon :: *[f_2]]]\!](\{r\})}_{A} = \mathcal{S}[\![\alpha{::}\eta[f_1 \wedge \varepsilon{::}*[f_2]]]\!](\{r\}) =$

$\mathcal{S}[\![\alpha{::}\eta]\!](\{r\}) \cap \xi[\![f_1 \wedge \varepsilon{::}*[f_2]]\!] = \underbrace{\mathcal{S}[\![\alpha :: \eta]\!](\{r\})}_{A_1} \cap \underbrace{\xi[\![f_1]\!] \cap \xi[\![\varepsilon :: *[f_2]]\!]}_{A_2}$

- $A_1 = \alpha(\{r\}) \cap T(\eta) = \{m \in T \mid r\ \alpha\ m, \lambda(m) = l\}$

- $\xi[\![\varepsilon{::}*[f_2]]\!] = \{n \in T \mid \mathcal{S}[\![\varepsilon{::}*[f_2]]\!](\{n\}) \neq \phi\} = \{n \in T \mid \lambda(n) \neq \texttt{str}, n \in \xi[\![f_2]\!]\}$[43]

- $A_2 = \xi[\![f_1]\!] \cap \xi[\![\varepsilon{::}*[f_2]]\!] = \{m \in T \mid m \in \xi[\![f_1]\!]\} \cap \{n \in T \mid \lambda(n) \neq \texttt{str}, n \in \xi[\![f_2]\!]\} = \{n \in T \mid \lambda(n) \neq \texttt{str}, n \in \xi[\![f_1]\!], n \in \xi[\![f_2]\!]\}$

- $A = A_1 \cap A_2 = \{m \in T \mid r\ \alpha\ m, \lambda(m) = l\} \cap \{n \in T \mid \lambda(n) \neq \texttt{str}, n \in \xi[\![f_1]\!], n \in \xi[\![f_2]\!]\} = \{m \in T \mid r\ \alpha\ m, \lambda(m) = l, \lambda(m) \neq \texttt{str}, m \in \xi[\![f_1]\!], m \in \xi[\![f_2]\!]\}$

Since $\lambda(m) = l$, i.e. m is an element node, then $\lambda(m) \neq \texttt{str}$ can be omitted. Therefore:

$$A = \{m \in T \mid r\ \alpha\ m, \lambda(m) = l, m \in \xi[\![f_1]\!], m \in \xi[\![f_2]\!]\}$$

(b) Evaluation of $\alpha{::}*[f_1][\varepsilon{::}\eta[f_2]]$ *over* T*:*

- $\underbrace{\mathcal{S}[\![\alpha :: *[f_1][\varepsilon :: \eta[f_2]]]\!](\{r\})}_{B} = \underbrace{\mathcal{S}[\![\alpha :: *]\!](\{r\})}_{B_1} \cap \underbrace{\xi[\![f_1]\!] \cap \xi[\![\varepsilon :: \eta[f_2]]\!]}_{B_2}$

- $B_1 = \alpha(\{r\}) \cap T(*) = \{s \in T \mid r\ \alpha\ s, \lambda(s) \neq \texttt{str}\}$

- $\xi[\![\varepsilon{::}\eta[f_2]]\!] = \{n \in T \mid \mathcal{S}[\![\varepsilon{::}\eta[f_2]]\!](\{n\}) \neq \phi\} = \{t \in T \mid \lambda(t) = l, t \in \xi[\![f_2]\!]\}$

- $B_2 = \xi[\![f_1]\!] \cap \xi[\![\varepsilon :: \eta[f_2]]\!] = \{t \in T \mid \lambda(t) = l, t \in \xi[\![f_1]\!], t \in \xi[\![f_2]\!]\}$

- $B = B_1 \cap B_2 = \{s \in T \mid r\ \alpha\ s, \lambda(s) \neq \texttt{str}\} \cap \{t \in T \mid \lambda(t) = l, t \in \xi[\![f_1]\!], t \in \xi[\![f_2]\!]\} = \{s \in T \mid r\ \alpha\ s, \lambda(s) = l, \lambda(s) \neq \texttt{str}, s \in \xi[\![f_1]\!], s \in \xi[\![f_2]\!]\}$

Since s is an element node, then the condition $\lambda(s) \neq \texttt{str}$ is useless and we conclude:

$$B = \{s \in T \mid r\ \alpha\ s, \lambda(s) = l, s \in \xi[\![f_1]\!], s \in \xi[\![f_2]\!]\}$$

(c) Evaluation of $\alpha{::}\eta[f_1 \wedge f_2]$ *over* T*:*

- $\underbrace{\mathcal{S}[\![\alpha :: \eta[f_1 \wedge f_2]]\!](\{r\})}_{C} = \underbrace{\mathcal{S}[\![\alpha :: \eta]\!](\{r\})}_{C_1} \cap \xi[\![f_1 \wedge f_2]\!]$

- $C_1 = \{u \in T \mid r\ \alpha\ u, \lambda(u) = l\}$

[43] $\lambda(n) \neq \texttt{str}$ indicates that n is an element node.

- $C = C_1 \cap \xi[\![f_1 \wedge f_2]\!]$ (i.e. all nodes of C_1 satisfying both f_1 and f_2 must be returned). Therefore:

$$C = \{u \in T \mid r\ \alpha\ u, \lambda(u) = l, u \in \xi[\![f_1]\!], u \in \xi[\![f_2]\!]\}$$

We remark that A=B=C, i.e. the three evaluations $\mathcal{S}[\![\alpha::l[f_1][\varepsilon::*[f_2]]]\!](T)$, $\mathcal{S}[\![\alpha::*[f_1][\varepsilon::l[f_2]]]\!](T)$, and $\mathcal{S}[\![\alpha::l[f_1 \wedge f_2]]\!](T)$ return the same set of nodes. For the case where $\eta = *$, $\mathcal{S}[\![\alpha::*]\!](\{r\})$ is given by $\{w \in T \mid r\ \alpha\ w\}$. It is obvious to show that the three evaluations return, in case of $\eta = *$, the following set of nodes:

$$\{w \in T \mid r\ \alpha\ w, w \in \xi[\![f_1]\!], w \in \xi[\![f_2]\!]\}$$

We conclude finally that, for any node test $\eta \neq$ *text()*, the queries $\alpha::\eta[f_1][\varepsilon::*[f_2]]$, $\alpha::*[f_1][\varepsilon::\eta[f_2]]$ and $\alpha::\eta[f_1 \wedge f_2]$ are equivalent.

3. $\downarrow^*\!\!::\eta_1[f_1][\alpha::\eta_2[f_2]] \equiv \downarrow^*\!\!::\eta_2[f_2]/\alpha^{-1}\!\!::\eta_1[f_1]$**.** We must show that for any predicates f_1 and f_2 of $\mathcal{X}^{\Uparrow}$, the queries $\downarrow^*::\eta_1[f_1][\alpha::\eta_2[f_2]] \equiv \downarrow^*::\eta_2[f_2]/\alpha^{-1}::\eta_1[f_1]$ are equivalent. For the sake of readability, we consider only simple $\mathcal{X}^{\Uparrow}$ predicates of size one, primarily predicates of the form $\alpha::\eta$ and $\alpha::\eta = c$ and composition of these two predicates (e.g. $(\alpha_1::\eta_1 \wedge \alpha_2::\eta_2)$, $\neg(\alpha::\eta = c)$). We should emphasize that the proof for complex $\mathcal{X}^{\Uparrow}$ predicates (i.e. for arbitrary size) can be done in a similar way.

• **Case of simple predicates** (f_1=$\alpha_{11}::\eta_{11}$, f_2=$\alpha_{22}::\eta_{22}$)**:** We must show that $\downarrow^*::\eta_1[\alpha_{11}::\eta_{11}][\alpha::l_2[\alpha_{22}::\eta_{22}]] \equiv \downarrow^*::\eta_2[\alpha_{22}::\eta_{22}]/\alpha^{-1}::\eta_1[\alpha_{11}::\eta_{11}]$.

(a) Evaluation of $\downarrow^*::\eta_1[\alpha_{11}::\eta_{11}][\alpha::\eta_2[\alpha_{22}::\eta_{22}]]$ *over* T*:*

- $\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_1[\alpha_{11} :: \eta_{11}][\alpha :: \eta_2[\alpha_{22} :: \eta_{22}]]]\!](\{r\})}_{A} =$

$$\mathcal{S}[\![\downarrow^*::\eta_1[\alpha_{11}::\eta_{11} \wedge \alpha::\eta_2[\alpha_{22}::\eta_{22}]]]\!](\{r\}) =$$
$$\mathcal{S}[\![\downarrow^*::\eta_1]\!](\{r\}) \cap \xi[\![\alpha_{11}::\eta_{11} \wedge \alpha::\eta_2[\alpha_{22}::\eta_{22}]]\!] =$$
$$\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_1]\!](\{r\})}_{A_1} \cap \underbrace{\xi[\![\alpha_{11} :: \eta_{11}]\!]}_{A_2} \cap \underbrace{\xi[\![\alpha :: \eta_2[\alpha_{22} :: \eta_{22}]]\!]}_{A_3}$$

- $A_1 = \{m_1 \in T \mid r\ \downarrow^*\ m_1, \lambda(m_1) \approx \eta_1\}$[44]

- $A_2 = \{m_{11}^{'} \in T \mid \mathcal{S}[\![\alpha_{11}::\eta_{11}]\!](\{m_{11}^{'}\}) \neq \varnothing\} =$
$\{m_{11}^{'} \in T \mid \exists m_{11} \in T, m_{11}^{'}\ \alpha_{11}\ m_{11}, \lambda(m_{11}) \approx \eta_{11}\}$

- $A_3 = \{m_2^{'} \in T \mid \mathcal{S}[\![\alpha :: \eta_2[\alpha_{22} :: \eta_{22}]](\{m_2^{'}\}) \neq \varnothing\} =$
$\{m_2^{'} \in T \mid \exists m_2 \in T, m_2^{'}\ \alpha\ m_2, \lambda(m_2) \approx \eta_2, m_2 \in \xi[\![\alpha_{22} :: \eta_{22}]\!]\} =$
$\{m_2^{'} \in T \mid \exists m_2, m_{22} \in T, m_2^{'}\ \alpha\ m_2, \lambda(m_2) \approx \eta_2, m_2\ \alpha_{22}\ m_{22}, \lambda(m_{22}) \approx \eta_{22}\}$

- $A = A_1 \cap A_2 \cap A_3$, this means that m_1 nodes of A_1 must coincide with $m_{11}^{'}$ and $m_2^{'}$ of A_2 and A_3 respectively. Therefore:

[44] $\lambda(n) \approx \eta$ denotes that label of n corresponds to the node test η, i.e. $\lambda(n) =$ `str` and $\eta =$ *text()*, or $\lambda(n) = \eta = l$ (for an element type l), or $\eta = *$ and $\lambda(n)$ is any element type.

$$A = \left\{ m_1 \in T \;\middle|\; \begin{array}{l} \exists m_{11}, m_2, m_{22} \in T, \\ \lambda(m_1) \approx \eta_1, \lambda(m_{11}) \approx \eta_{11}, \lambda(m_2) \approx \eta_2, \lambda(m_{22}) \approx \eta_{22}, \\ r \downarrow^* m_1, m_1 \ \alpha_{11} \ m_{11}, m \ \alpha \ m_2, m_2 \ \alpha_{22} \ m_{22} \end{array} \right\}$$

(b) Evaluation of $\downarrow^*::\eta_2[\alpha_{22}::\eta_{22}]/\alpha^{-1}::\eta_1[\alpha_{11}::\eta_{11}]$ *over* T*:*

- $\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_2[\alpha_{22} :: \eta_{22}]/\alpha^{-1} :: \eta_1[\alpha_{11} :: \eta_{11}]]\!](\{r\})}_{B} = \mathcal{S}[\![\alpha^{-1}::\eta_1[\alpha_{11}::\eta_{11}]]\!](\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_2[\alpha_{22} :: \eta_{22}]]\!](\{r\})}_{B_1})$

- $B_1 = \mathcal{S}[\![\downarrow^*:: \eta_2]\!](\{r\}) \cap \xi[\![\alpha_{22}::\eta_{22}]\!] =$
  $\{s_2 \in T \mid r \downarrow^* s_2, \lambda(s_2) \approx \eta_2, s_2 \in \xi[\![\alpha_{22} :: \eta_{22}]\!]\} =$
  $\{s_2 \in T \mid \exists s_{22} \in T, r \downarrow^* s_2, \lambda(s_2) \approx \eta_2, s_2 \ \alpha_{22} \ s_{22}, \lambda(s_{22}) \approx \eta_{22}\}$

- $B = \mathcal{S}[\![\alpha^{-1}::\eta_1[\alpha_{11}::\eta_{11}]]\!](B_1) =$
  $\{s_1 \in T \mid \exists s_2 \in B_1, s_2 \ \alpha^{-1} \ s_1, \lambda(s_1) \approx \eta_1, s_1 \in \xi[\![\alpha_{11} :: \eta_{11}]\!]\} =$
  $\{s_1 \in T \mid \exists s_2 \in B_1, s_{11} \in T, s_2 \ \alpha^{-1} \ s_1, \lambda(s_1) \approx \eta_1, s_1 \ \alpha_{11} \ s_{11}, \lambda(s_{11}) \approx \eta_{11}\}$

- From the definition of B_1, we conclude that:

$$B = \left\{ s_1 \in T \;\middle|\; \begin{array}{l} \exists s_{11}, s_2, s_{22} \in T, \\ \lambda(s_2) \approx \eta_2, \ \lambda(s_{22}) \approx \eta_{22}, \ \lambda(s_1) \approx \eta_1, \ \lambda(s_{11}) \approx \eta_{11}, \\ r \downarrow^* s_2, s_2 \ \alpha_{22} \ s_{22}, \ s_2 \ \alpha^{-1} \ s_1 \ , s_1 \ \alpha_{11} \ s_{11} \end{array} \right\}$$

Note that the condition $s_2 \ \alpha^{-1} \ s_1$ of B can be replaced with $s_1 \ \alpha \ s_2$. Finally, it remains to show that A=B, i.e. each node of A holds the conditions of B and vice versa. Let us compare between the two sets of nodes A and B representing the evaluations $\downarrow^*::\eta_1[\alpha_{11}::\eta_{11}][\alpha::\eta_2[\alpha_{22}::\eta_{22}]](T)$ and $\downarrow^*::\eta_2[\alpha_{22}::\eta_{22}]/\alpha^{-1}::\eta_1[\alpha_{11}::\eta_{11}](T)$ respectively:

$$A = \left\{ m_1 \in T \;\middle|\; \begin{array}{l} \exists m_{11}, m_2, m_{22} \in T, \\ \lambda(m_1) \approx \eta_1, \lambda(m_{11}) \approx \eta_{11}, \lambda(m_2) \approx \eta_2, \lambda(m_{22}) \approx \eta_{22}, \\ r \downarrow^* m_1 \ , m_1 \ \alpha_{11} \ m_{11}, m_1 \ \alpha \ m_2, m_2 \ \alpha_{22} \ m_{22} \end{array} \right\}$$

$$B = \left\{ s_1 \in T \;\middle|\; \begin{array}{l} \exists s_{11}, s_2, s_{22} \in T, \\ \lambda(s_1) \approx \eta_1, \ \lambda(s_{11}) \approx \eta_{11}, \ \lambda(s_2) \approx \eta_2, \ \lambda(s_{22}) \approx \eta_{22}, \\ r \downarrow^* s_2 \ , s_1 \ \alpha_{11} \ s_{11}, \ s_1 \ \alpha \ s_2 \ , s_2 \ \alpha_{22} \ s_{22} \end{array} \right\}$$

We define the mapping $m_1 \rightarrow s_1$, $m_{11} \rightarrow s_{11}$, $m_2 \rightarrow s_2$, and $m_{22} \rightarrow s_{22}$ from nodes of A to nodes of B. Thus, we remark that A and B have the same conditions except the additional conditions $r \downarrow^* m_1$ of A and $r \downarrow^* s_2$ of B. Thus, for each set (A or B), its additional condition must be satisfied by the nodes of the second set. Since r is the root node then for any node set and for any node t the condition $r \downarrow^* t$ holds. Finally, we conclude that for any simple predicates f_1 and f_2 of $\mathcal{X}^{\Uparrow}$ with the form $\alpha::\eta$, the queries $\downarrow^*::\eta_1[f_1][\alpha::\eta_2[f_2]]$ and $\downarrow^*::\eta_2[f_2]/\alpha^{-1}::\eta_1[f_1]$ are equivalent.

• **Case of predicates with text-comparison** (f_1 and f_2 are given by $\alpha_{11}::\eta_{11} = c_1$ and $\alpha_{22}::\eta_{22} = c_2$ resp.)**:** This case can be proven following the same principle as the previous case (i.e. predicates of the form $\alpha::\eta$). Thus, in the following deductions some intermediate steps are omitted. Recall that: $\xi[\![p = c]\!]$=$\{n \in T \mid \varphi[\![c]\!](\mathcal{S}[\![p]\!](\{n\})) \neq \phi\}$=$\{n \in T \mid \exists m \in T, m \in \mathcal{S}[\![p]\!](\{n\}), \nu(m) = c\}$.

(a) Evaluation of $\downarrow^*::\eta_1[\alpha_{11}::\eta_{11} = c_1][\alpha::\eta_2[\alpha_{22}::\eta_{22} = c_2]]$ *over* T*:*

- $\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_1[\alpha_{11} :: \eta_{11} = c_1][\alpha :: \eta_2[\alpha_{22} :: \eta_{22} = c_2]]]\!](\{r\})}_{A'} =$

$$\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_1]\!](\{r\})}_{A'_1} \cap \underbrace{\xi[\![\alpha_{11} :: \eta_{11} = c_1]\!]}_{A'_2} \cap \underbrace{\xi[\![\alpha :: \eta_2[\alpha_{22} :: \eta_{22} = c_2]]\!]}_{A'_3}$$

- $A'_1 = \{m_1 \in T \mid r \downarrow^* m_1, \lambda(m_1) \approx \eta_1\}$

- $A'_2 = \{m'_{11} \in T \mid \exists m_{11} \in T, m_{11} \in \mathcal{S}[\![\alpha_{11}::\eta_{11}]\!](\{m'_{11}\})), \nu(m_{11}) = c_1\} =$
$\{m'_{11} \in T \mid \exists m_{11} \in T, m'_{11}\ \alpha_{11}\ m_{11}, \lambda(m_{11}) \approx \eta_{11}, \nu(m_{11}) = c_1\}$

- $A'_3 = \{m'_2 \in T \mid \mathcal{S}[\![\alpha :: \eta_2[\alpha_{22} :: \eta_{22} = c_2](\{m'_2\}) \neq \varnothing\} =$
$\{m'_2 \in T \mid \exists m_2 \in T, m'_2\ \alpha\ m_2, \lambda(m_2) \approx \eta_2, m_2 \in \xi[\![\alpha_{22} :: \eta_{22} = c_2]\!]\} =$
$\{m'_2 \in T \mid \exists m_2, m_{22} \in T, m'_2\ \alpha\ m_2, \lambda(m_2) \approx \eta_2, m_2\ \alpha_{22}\ m_{22}, \lambda(m_{22}) \approx \eta_{22}, \nu(m_{22} = c_2\}$

- Finally, we conclude that:

$$A' = \left\{ m_1 \in T \;\middle|\; \begin{array}{l} \exists m_{11}, m_2, m_{22} \in T, \\ \lambda(m_1) \approx \eta_1, \lambda(m_{11}) \approx \eta_{11}, \lambda(m_2) \approx \eta_2, \lambda(m_{22}) \approx \eta_{22}, \\ r \downarrow^* m_1, m_1\ \alpha_{11}\ m_{11}, m\ \alpha\ m_2, m_2\ \alpha_{22}\ m_{22}, \nu(m_{11}) = c_1,\ \nu(m_{22}) = c_2 \end{array} \right\}$$

(b) Evaluation of $\downarrow^*::\eta_2[\alpha_{22}::\eta_{22} = c_2]/\alpha^{-1}::\eta_1[\alpha_{11}::\eta_{11} = c_1]$ *over* T*:*

- $\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_2[\alpha_{22} :: \eta_{22} = c_2]/\alpha^{-1} :: \eta_1[\alpha_{11} :: \eta_{11} = c_1]]\!](\{r\})}_{B'} =$

$\mathcal{S}[\![\alpha^{-1}::\eta_1[\alpha_{11}::\eta_{11} = c_1]]\!](\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_2[\alpha_{22} :: \eta_{22} = c_2]]\!](\{r\})}_{B'_1})$

- $B'_1 = \mathcal{S}[\![\downarrow^*:: \eta_2]\!](\{r\}) \cap \xi[\![\alpha_{22}::\eta_{22} = c_2]\!] =$
$\{s_2 \in T \mid \exists s_{22} \in T, r \downarrow^* s_2, \lambda(s_2) \approx \eta_2, s_2\ \alpha_{22}\ s_{22}, \lambda(s_{22}) \approx \eta_{22}, \nu(s_{22}) = c_2\}$

- $B' = \mathcal{S}[\![\alpha^{-1}::\eta_1[\alpha_{11}::\eta_{11} = c_1]]\!](B'_1) =$
$\{s_1 \in T \mid \exists s_2 \in B'_1, s_2\ \alpha^{-1}\ s_1, \lambda(s_1) \approx \eta_1, s_1 \in \xi[\![\alpha_{11} :: \eta_{11} = c_1]\!]\} =$
$\{s_1 \in T \mid \exists s_2 \in B'_1, s_{11} \in T, s_2\ \alpha^{-1}\ s_1, \lambda(s_1) \approx \eta_1, s_1\ \alpha_{11}\ s_{11}, \lambda(s_{11}) \approx \eta_{11}, \nu(s_{11}) = c_1\}$

- From the definition of B'_1, we conclude that: ($s_2\ \alpha^{-1}\ s_1$ is replaced by $s_1\ \alpha\ s_2$)

$$B' = \left\{ s_1 \in T \;\middle|\; \begin{array}{l} \exists s_{11}, s_2, s_{22} \in T, \\ \lambda(s_2) \approx \eta_2,\ \lambda(s_{22}) \approx \eta_{22},\ \lambda(s_1) \approx \eta_1,\ \lambda(s_{11}) \approx \eta_{11}, \\ r \downarrow^* s_2, s_2\ \alpha_{22}\ s_{22}, s_1\ \alpha\ s_2, s_1\ \alpha_{11}\ s_{11}, \nu(s_{11}) = c_1,\ \nu(s_{22}) = c_2 \end{array} \right\}$$

Remark that A' differs from A with only the additional conditions $\nu(m_{11}) = c_1$ and $\nu(m_{22}) = c_2$, as well as B' differs from B with the addition of the two conditions $\nu(s_{11}) = c_1$ and $\nu(s_{22}) = c_2$. By mapping m_i nodes of A into nodes s_i in B we have shown that A=B, and since equivalent conditions are added into nodes of A and B, then we conclude that A'=B'. Summing up, for simple $\mathcal{X}^{\Uparrow}$ predicates of the form $\alpha::\eta$ and $\alpha::\eta = c$, we have shown that the queries $\downarrow^*::\eta_1[f_1][\alpha::\eta_2[f_2]]$ and $\downarrow^*::\eta_2[f_2]/\alpha^{-1}::\eta_1[f_1]$ are equivalent.

• **Case of conjunction of predicates:** We must show that the queries $\downarrow^*::\eta_1[f_1 \wedge f'_1][\alpha::\eta_2[f_2 \wedge f'_2]]$ and $\downarrow^*::\eta_2[f_2 \wedge f'_2]/\alpha^{-1}::\eta_1[f_1 \wedge f'_1]$ are equivalent for conjunction of predicates of the forms $\alpha::\eta$ and/or $\alpha::\eta$=c. Based on the semantics given in Table 3.1, it is easy to prove that $\mathcal{S}[\![p[f_1 \wedge f_2]]\!](T)$

is equivalent to the query $\mathcal{S}[\![p[f_1]]\!](T) \cap \mathcal{S}[\![p[f_2]]\!](T)$. Thus, the queries $\downarrow^*::\eta_1[f_1 \wedge f_1^{'}][\alpha::\eta_2[f_2 \wedge f_2^{'}]]$ and $\downarrow^*::\eta_2[f_2 \wedge f_2^{'}]/\alpha^{-1}::\eta_1[f_1 \wedge f_1^{'}]$ can be easily translated into conjunction of some queries composed with only predicates of the form $\alpha::\eta$ and $\alpha::\eta = c$. More precisely, these queries can be translated as follows:

- $\downarrow^*::\eta_1[f_1 \wedge f_1^{'}][\alpha::\eta_2[f_2 \wedge f_2^{'}]] =$
$\downarrow^*::\eta_1[f_1][\alpha::\eta_2[f_2 \wedge f_2^{'}]] \cap \downarrow^*::\eta_1[f_1^{'}][\alpha::\eta_2[f_2 \wedge f_2^{'}]] =$
$\underbrace{\downarrow^*:: \eta_1[f_1][\alpha :: \eta_2[f_2]]}_{A_1} \cap \underbrace{\downarrow^*:: \eta_1[f_1][\alpha :: \eta_2[f_2^{'}]]}_{A_2} \cap \underbrace{\downarrow^*:: \eta_1[f_1^{'}][\alpha :: \eta_2[f_2]]}_{A_3} \cap \underbrace{\downarrow^*:: \eta_1[f_1^{'}][\alpha :: \eta_2[f_2^{'}]]}_{A_4}.$

- $\downarrow^*::\eta_2[f_2 \wedge f_2^{'}]/\alpha^{-1}::\eta_1[f_1 \wedge f_1^{'}] =$
$\downarrow^*::\eta_2[f_2]/\alpha^{-1}::\eta_1[f_1 \wedge f_1^{'}] \cap \downarrow^*::\eta_2[f_2^{'}]/\alpha^{-1}::\eta_1[f_1 \wedge f_1^{'}] =$
$\underbrace{\downarrow^*:: \eta_2[f_2]/\alpha^{-1} :: \eta_1[f_1]}_{B_1} \cap \underbrace{\downarrow^*:: \eta_2[f_2]/\alpha^{-1} :: \eta_1[f_1^{'}]}_{B_2} \cap \underbrace{\downarrow^*:: \eta_2[f_2^{'}]/\alpha^{-1} :: \eta_1[f_1]}_{B_3} \cap$
$\underbrace{\downarrow^*:: \eta_2[f_2^{'}]/\alpha^{-1} :: \eta_1[f_1^{'}]}_{B_4}.$

We shown that, for predicates f_1 and f_2 of the form $\alpha::\eta$ and $\alpha::\eta{=}c$, the queries $\downarrow^*::\eta_1[f_1][\alpha::\eta_2[f_2]]$ and $\downarrow^*::\eta_2[f_2]/\alpha^{-1}::\eta_1[f_1]$ are equivalent. Notice that the node sets A_1, A_2, A_3 and A_4 are equivalent to B_1, B_2, B_3 and B_4 respectively. Thus, we conclude that the queries $\downarrow^*::\eta_1[f_1 \wedge f_1^{'}][\alpha::\eta_2[f_2 \wedge f_2^{'}]]$ and $\downarrow^*::\eta_2[f_2 \wedge f_2^{'}]/\alpha^{-1}::\eta_1[f_1 \wedge f_1^{'}]$ are equivalent.

• **Case of disjunction of predicates:** We must show that the queries $\downarrow^*::\eta_1[f_1 \vee f_1^{'}][\alpha::\eta_2[f_2 \vee f_2^{'}]]$ and $\downarrow^*::\eta_2[f_2 \vee f_2^{'}]/\alpha^{-1}::\eta_1[f_1 \vee f_1^{'}]$ are equivalent for disjunction of predicates of the forms $\alpha::\eta$ and/or $\alpha::\eta{=}c$. This can be shown in a similar way as the previous case. In a nutshell, these queries can be translated as follows:

- $\downarrow^*::\eta_1[f_1 \vee f_1^{'}][\alpha::\eta_2[f_2 \vee f_2^{'}]] =$
$\downarrow^*::\eta_1[f_1][\alpha::\eta_2[f_2 \vee f_2^{'}]] \cup \downarrow^*::\eta_1[f_1^{'}][\alpha::\eta_2[f_2 \vee f_2^{'}]] = A_1 \cup A_2 \cup A_3 \cup A_4.$

- $\downarrow^*::\eta_2[f_2 \vee f_2^{'}]/\alpha^{-1}::\eta_1[f_1 \vee f_1^{'}] =$
$\downarrow^*::\eta_2[f_2]/\alpha^{-1}::\eta_1[f_1 \vee f_1^{'}] \cup \downarrow^*::\eta_2[f_2^{'}]/\alpha^{-1}::\eta_1[f_1 \vee f_1^{'}] = B_1 \cup B_2 \cup B_3 \cup B_4.$

Since the node sets A_1, A_2, A_3 and A_4 are equivalent to B_1, B_2, B_3 and B_4 respectively, then the queries $\downarrow^*::\eta_1[f_1 \vee f_1^{'}][\alpha::\eta_2[f_2 \vee f_2^{'}]]$ and $\downarrow^*::\eta_2[f_2 \vee f_2^{'}]/\alpha^{-1}::\eta_1[f_1 \vee f_1^{'}]$ are equivalent.

• **Case of negation of predicates:** We must show that the queries $\downarrow^*::\eta_1[\neg(f_1)][\alpha::\eta_2[\neg(f_2)]]$ and $\downarrow^*::\eta_2[\neg(f_2)]/\alpha^{-1}::\eta_1[\neg(f_1)]$ are equivalent for negation of predicates of the forms $\alpha::\eta$ and/or $\alpha::\eta{=}c$. It is clear that $\mathcal{S}[\![p[f]]\!](T)$ is equivalent to $\mathcal{S}[\![p]\!](T) \setminus \mathcal{S}[\![p[\neg(f)]]\!](T)$. The equivalence between the previous queries can be shown as follows:

- $\mathcal{S}[\![\downarrow^*::\eta_2[\neg(f_2)]/\alpha^{-1}::\eta_1[\neg(f_1)]]\!](T) = \mathcal{S}[\![\alpha^{-1}::\eta_1[\neg(f_1)]]\!](\underbrace{\mathcal{S}[\![\downarrow^*:: \eta_2[\neg(f_2)]]\!](T)}_{A}).$

- $\mathcal{S}[\![\alpha^{-1}::\eta_1[\neg(f_1)]]\!](A) = \underbrace{\mathcal{S}[\![\alpha^{-1} :: \eta_1]\!](A)}_{A_1} \setminus \underbrace{\mathcal{S}[\![\alpha^{-1} :: \eta_1[f_1]]\!](A)}_{A_2}.$

- $A = \underbrace{\mathcal{S}[\![\downarrow^*:: \eta_2]\!](T)}_{A_3} \setminus \underbrace{\mathcal{S}[\![\downarrow^*:: \eta_2[f_2]]\!](T)}_{A_4}.$

- $A_1 = \mathcal{S}[\![\alpha^{-1}::\eta_1]\!](A) = \mathcal{S}[\![\alpha^{-1}::\eta_1]\!](A_3 \setminus A_4) = \mathcal{S}[\![\alpha^{-1}::\eta_1]\!](A_3) \setminus \mathcal{S}[\![\alpha^{-1}::\eta_1]\!](A_4).$

- $A_2 = \mathcal{S}[\![\alpha^{-1}::\eta_1[f_1]]\!](A) = \mathcal{S}[\![\alpha^{-1}::\eta_1[f_1]]\!](A_3 \setminus A_4) = \mathcal{S}[\![\alpha^{-1}::\eta_1[f_1]]\!](A_3) \setminus \mathcal{S}[\![\alpha^{-1}::\eta_1[f_1]]\!](A_4).$

- $A_1 \setminus A_2 = (\mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](A_3) \setminus \mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](A_4)) \setminus (\mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](A_3) \setminus \mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](A_4)) =$
$\mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](A_3) \setminus (\mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](A_4) \cup \mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](A_3) \cup \mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](A_4))$.

- $\mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](A_3) = \mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](\mathcal{S}[\![\downarrow^*{::}\eta_2]\!](T)) = \mathcal{S}[\![\downarrow^*{::}\eta_2/\alpha^{-1}{::}\eta_1]\!](T) = \mathcal{S}[\![\downarrow^*{::}\eta_1[\alpha{::}\eta_2]]\!](T)$.

- $\mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](A_4) = \mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](\mathcal{S}[\![\downarrow^*{::}\eta_2[f_2]]\!](T)) = \mathcal{S}[\![\downarrow^*{::}\eta_2[f_2]/\alpha^{-1}{::}\eta_1]\!](T)$.

- $\mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](A_3) = \mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](\mathcal{S}[\![\downarrow^*{::}\eta_2]\!](T)) = \mathcal{S}[\![\downarrow^*{::}\eta_2/\alpha^{-1}{::}\eta_1[f_1]]\!](T)$.

- $\mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](A_4) = \mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](\mathcal{S}[\![\downarrow^*{::}\eta_2[f_2]]\!](T)) = \mathcal{S}[\![\downarrow^*{::}\eta_2[f_2]/\alpha^{-1}{::}\eta_1[f_1]]\!](T)$.

Notice that: $(\mathcal{S}[\![\alpha^{-1}{::}\eta_1]\!](A_4) \cup \mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](A_3) \cup \mathcal{S}[\![\alpha^{-1}{::}\eta_1[f_1]]\!](A_4)) =$
$(\underbrace{\mathcal{S}[\![\downarrow^*{::}\eta_1[\alpha :: \eta_2[f_2]]\!](T)}_{B_1} \cup \underbrace{\mathcal{S}[\![\downarrow^*{::}\eta_1[f_1][\alpha :: \eta_2]]\!](T)}_{B_2} \cup \underbrace{\mathcal{S}[\![\downarrow^*{::}\eta_1[f_1][\alpha :: \eta_2[f_2]]\!](T)}_{B_3})$.

- Since $B_3 \subset B_2$ then:
$A_1 \setminus A_2 = (\mathcal{S}[\![\downarrow^*{::}\eta_1[\alpha{::}\eta_2]]\!](T)) \setminus (B_1 \cup B_2) = (\mathcal{S}[\![\downarrow^*{::}\eta_1[\alpha{::}\eta_2]]\!](T) \setminus B_1) \setminus B_2 =$
$\mathcal{S}[\![\downarrow^*{::}\eta_1[\alpha{::}\eta_2[\neg(f_2)]]]\!](T) \setminus B_2 = \mathcal{S}[\![\downarrow^*{::}\eta_1[\neg(f_1)][\alpha{::}\eta_2[\neg(f_2)]]]\!](T)$.

We have shown that the queries $\downarrow^*{::}\eta_1[f_1][\alpha{::}\eta_2[f_2]]$ and $\downarrow^*{::}\eta_2[f_2]/\alpha^{-1}{::}\eta_1[f_1]$ are equivalent for simple predicates of the form $\alpha{::}\eta$ and $\alpha{::}\eta = c$, as well as for composition of them (i.e. conjunction, disjunction, and negation). It is still straightforward to generalize this proof in order to take into account general $\mathcal{X}^{\Uparrow}$ predicates of arbitrary size.

4. $\alpha_1{::}\eta_1/\ldots/\alpha_k{::}\eta_k \equiv \downarrow^*{::}\eta_k[\alpha_k^{-1}{::}\eta_{k-1}/\ldots/\alpha_2^{-1}{::}\eta_1/\alpha_1^{-1}{::}root]$**.** The following deductions provide this equivalence:

(a) Evaluation of $\alpha_1{::}\eta_1/\ldots/\alpha_k{::}\eta_k$ over T:

- $\mathcal{S}[\![\alpha_1{::}\eta_1]\!](\{r\}) = \{n_1 \in T \mid r\ \alpha_1\ n_1,\ \lambda(n_1) \approx \eta_1\}$

- $\mathcal{S}[\![\alpha_1{::}\eta_1/\alpha_2{::}\eta_2]\!](\{r\}) = \mathcal{S}[\![\alpha_2{::}\eta_2]\!](\mathcal{S}[\![\alpha_1{::}\eta_1]\!](\{r\})) =$
$\{n_2 \in T \mid \exists n_1 \in T,\ r\ \alpha_1\ n_1,\ n_1\ \alpha_2\ n_2,\ \lambda(n_1) \approx \eta_1,\ \lambda(n_2) \approx \eta_2\}$

Let A be the node set resulted from the evaluation $\mathcal{S}[\![\alpha_1{::}\eta_1/\ldots/\alpha_k{::}\eta_k]\!](\{r\})$. Intuitively, A is given by

$$A = \left\{ n_k \in T \;\middle|\; \begin{array}{l} \exists n_1, \ldots, n_{k-1} \in T,\ \forall_{1 \le i \le k}\ \lambda(n_i) \approx \eta_i, \\ r\ \alpha_1\ n_1,\ n_1\ \alpha_2\ n_2, \ldots,\ n_{k-2}\ \alpha_{k-1}\ n_{k-1},\ n_{k-1}\ \alpha_k\ n_k \end{array} \right\}$$

(b) Evaluation of $\downarrow^*{::}\eta_k[\alpha_k^{-1}{::}\eta_{k-1}/\ldots/\alpha_2^{-1}{::}\eta_1/\alpha_1^{-1}{::}r]$ over T:

- $\mathcal{S}[\![\downarrow^*{::}\eta_k]\!](\{r\}) = \underbrace{\{n_k \in T \mid r\ \downarrow^*\ n_k\ , \lambda(n_k) \approx \eta_k\ \}}_{B_1}$.

- $\mathcal{S}[\![\downarrow^*{::}\eta_k[\alpha_k^{-1}{::}\eta_{k-1}/\ldots/\alpha_2^{-1}{::}\eta_1/\alpha_1^{-1}{::}root]]\!](\{r\}) =$
$B_1 \cap \underbrace{\xi[\![\alpha_k^{-1} :: \eta_{k-1}/\ldots/\alpha_2^{-1} :: \eta_1/\alpha_1^{-1} :: root]\!]}_{B_2}$.

- $\xi[\![\alpha_k^{-1}{::}\eta_{k-1}]\!] = \{n_k \in T \mid \exists n_{k-1} \in T,\ n_k\ \alpha_k^{-1}\ n_{k-1},\ \lambda(n_{k-1}) \approx \eta_{k-1}\}$

- $\xi[\![\alpha_k^{-1}::\eta_{k-1}/\alpha_{k-1}^{-1}::\eta_{k-2}]\!] = \{n_k \in T \mid \exists n_{k-1}, n_{k-2} \in T,\ n_k\ \alpha_k^{-1}\ n_{k-1},\ n_{k-1}\ \alpha_{k-1}^{-1}\ n_{k-2},\ \lambda(n_{k-1}) \approx \eta_{k-1},\ \lambda(n_{k-2}) \approx \eta_{k-2}\}$

Intuitively B_2 is given as follows:

$$B_2 = \left\{ n_k \in T \;\middle|\; \begin{array}{l} \exists n_{k-1}, \ldots, n_1 \in T,\ \forall_{1\leq i\leq k-1}\ \lambda(n_i) \approx \eta_i, \\ n_k\ \alpha_k^{-1}\ n_{k-1}, \ldots,\ n_2\ \alpha_2^{-1}\ n_1,\ n_1\ \alpha_1^{-1}\ r \end{array} \right\}$$

Finally, B (i.e. $B_1 \cap B_2$) is given as follows:

$$B = \left\{ n_k \in T \;\middle|\; \begin{array}{l} \exists n_1, \ldots, n_{k-1} \in T,\ \forall_{1\leq i\leq k}\ \lambda(n_i) \approx \eta_i, \\ r\ \downarrow^*\ n_k,\ \boxed{n_k\ \alpha_k^{-1}\ n_{k-1}}, \ldots,\ \boxed{n_2\ \alpha_2^{-1}\ n_1},\ \boxed{n_1\ \alpha_1^{-1}\ r} \end{array} \right\}$$

Let us now compare between the two node sets A and B (each relation $n\ \alpha^{-1}\ m$ in B is replaced with $m\ \alpha\ n$):

$$A = \left\{ n_k \in T \;\middle|\; \begin{array}{l} \exists n_1, \ldots, n_{k-1} \in T,\ \forall_{1\leq i\leq k}\ \lambda(n_i) \approx \eta_i, \\ r\ \alpha_1\ n_1,\ n_1\ \alpha_2\ n_2, \ldots,\ n_{k-1}\ \alpha_k\ n_k \end{array} \right\}$$

$$B = \left\{ n_k \in T \;\middle|\; \begin{array}{l} \exists n_1, \ldots, n_{k-1} \in T,\ \forall_{1\leq i\leq k}\ \lambda(n_i) \approx \eta_i, \\ \boxed{r\ \alpha_1\ n_1},\ \boxed{n_1\ \alpha_2\ n_2}, \ldots,\ \boxed{n_{k-1}\ \alpha_k\ n_k},\ \boxed{r\ \downarrow^*\ n_k} \end{array} \right\}$$

Notice that the only difference between A and B is the additional condition $r \downarrow^* n_k$ in B. Since r is the root node, then for any node n_k we have $r \downarrow^* n_k$. Thus, the condition $r \downarrow^* n_k$ of B is useless and can be omitted, as it can be added to A without changing its semantic. We conclude then that A=B. Finally, we found that the queries $\alpha_1::\eta_1/\ldots/\alpha_k::\eta_k$ and $\downarrow^*::\eta_k[\alpha_k^{-1}::\eta_{k-1}/\ldots/\alpha_2^{-1}::\eta_1/\alpha_1^{-1}::r]$ are equivalent.

5. $m \in \mathcal{S}[\![\downarrow^*\!::\eta[f]]\!](T)$ if and only if $m \in \xi[\![\varepsilon::\eta[f]]\!]$. Given a node m of T, we must show first that if m is referred to by the query $\downarrow^*::\eta[f]$ over T, then $m \in \xi[\![\varepsilon::\eta[f]]\!]$. Secondly, if $m \in \xi[\![\varepsilon::\eta[f]]\!]$ then the node m must belong to the set of nodes returned by the evaluation $\mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$.

$\Longrightarrow$ Assume by contradiction that $m \in \mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$ and $m \notin \xi[\![\varepsilon::\eta[f]]\!]$. We have $\mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$=$\mathcal{S}[\![\downarrow^*::\eta[f]]\!](\{r\})$= $\{m \mid r\ \downarrow^*\ m,\ \lambda(m) \approx \eta,\ m \in \xi[\![f]\!]\ \}$. We conclude that m corresponds to the node test η and the predicate f is valid over m. On the other hand, assume that $m \notin \xi[\![\varepsilon::\eta[f]]\!]$ implies that either $\lambda(m) \not\approx \eta$ or $m \in \xi[\![f]\!]$, which contradicts our assumption (i.e. $m \in \mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$). Thus, if $m \in \mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$ then $m \in \xi[\![\varepsilon::\eta[f]]\!]$.

$\Longleftarrow$ Assume by contradiction that $m \in \xi[\![\varepsilon::\eta[f]]\!]$ and $m \notin \mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$. The first condition of the assumption implies that $\lambda(m) \approx \eta$ and $m \in \xi[\![f]\!]$. We have seen that $\mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$=$\{m \mid r\ \downarrow^*\ m,\ \lambda(m) \approx \eta,\ m \in \xi[\![f]\!]\ \}$. Thus, a node m does not belong to this node set iff: 1) m is not a descendant of r, 2) $\lambda(m) \not\approx \eta$, or 3) $m \notin \xi[\![f]\!]$. The first condition is not fulfilled since r is the root node, while the second and the third conditions contradict our assumption. Thus, if $m \in \xi[\![\varepsilon::\eta[f]]\!]$ then $m \in \mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$.

We conclude finally that for any node m of T, $m \in \mathcal{S}[\![\downarrow^*::\eta[f]]\!](T)$ if and only if $m \in \xi[\![\varepsilon::\eta[f]]\!]$. □

A.2 Proofs of Chapter 4

Given an XML tree T and its authorized view T_A (see Definition **??**), each accessible node n of T must appear in T_A and vice versa, thus we define a mapping between nodes of T and T_A. We denote by n_{T_A} a node of T_A and by n_T its corresponding node in T.

To simplify the proofs we redefine the semantic of the function ξ as follows. By $\xi[\![q]\!](N)$ we denote the nodes of the set N that satisfy the predicate q. Moreover, $\xi[\![q_1[q_2]]\!](N)$=$\xi[\![q_2]\!](\xi[\![q_1]\!](N))$, and $\mathcal{S}[\![p[q]]\!](N)$=$\xi[\![q]\!](\mathcal{S}[\![p]\!](N))$.

A.2.1 Correctness of Accessibility Predicate

Definition A.1. Given an access specification S=$(D, \boldsymbol{ann})$ and an XML tree $T \in \mathcal{T}(D)$, then, an element node n in T of type B with parent node of type A is *accessible* (i.e. shown in the authorized view T_A of T) if and only if the following conditions hold:

i) Either there exists an explicitly defined annotation $\boldsymbol{ann}(A, B)$ that is valid at n; or the first annotation explicitly defined over ancestors of n is valid.

ii) There is no downward-closed annotation defined over any ancestor node n' of n with value N_h or $[Q]_h$ where $n' \nvDash Q$. □

Specifically, consider the nodes $n_1, \ldots, n_k$ where n_1 is the root node, $k \geq 2$, and each n_i has element type A_i. According to the condition (i) of Definition A.1, an accessible node n_j $(j > 1)$ should have either (**a**) an explicitly defined annotation $\boldsymbol{ann}(A_{j-1}, A_j)$ that is valid at n_j (i.e. $\boldsymbol{ann}(A_{j-1}, A_j)$=$Y|[Q]|[Q]_h$ with $n_j \vDash Q$), or (**b**) an ancestor n_i concerned by a valid annotation (i.e. either the default annotation $\boldsymbol{ann}(A_i)$=Y if n_i is the root node, or an arbitrary annotation $\boldsymbol{ann}(A_{i-1}, A_i)$) such that this annotation is the first one defined over ancestors of n_j (i.e. no annotation $\boldsymbol{ann}(A_{l-1}, A_l)$ is explicitly defined for $i < l \leq j$). The condition (ii) of Definition A.1 implies that for any downward-closed annotation $\boldsymbol{ann}(A_{i-1}, A_i)$ defined over ancestor n_i of n_j (i.e. $i < j$), either $\boldsymbol{ann}(A_{i-1}, A_i) \neq N_h$ or $\boldsymbol{ann}(A_{i-1}, A_i)$=$[Q]_h$ with $n_j \vDash Q$.

Definition A.2. Given an access specification S=$(D, \boldsymbol{ann})$, 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:
$allAnn := \varepsilon::root \vee_{ann(A',A)\in \boldsymbol{ann}} \varepsilon::A/\uparrow::A'$
$validAnn := \varepsilon::root \vee_{(\boldsymbol{ann}(A',A)=Y)\in \boldsymbol{ann}} \varepsilon::A/\uparrow::A' \vee_{(\boldsymbol{ann}(A',A)=[Q]|[Q]_h)\in \boldsymbol{ann}} \varepsilon::A[Q]/\uparrow::A'$

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

$\mathcal{A}^{acc}_1$ and $\mathcal{A}^{acc}_2$ satisfy the conditions (i) and (ii) of Definition A.1 respectively. □

The first predicate checks whether the node n is explicitly concerned by a valid annotation or inherits its accessibility from a valid annotation defined over its ancestors. While the second predicate checks whether the node n is not in the scope of an invalid downward-closed annotation. More specifically, the evaluation of the predicate $\uparrow^*::*[allAnn]$ at a node n returns a node set 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. for any node $m \in \{n\} \cup \boldsymbol{ancestors}(n)$[45] of type B with a parent node of

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

type A, $m \in N$ if and only if $\boldsymbol{ann}(A, B)$ 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) or the first ancestor of n that is explicitly concerned by an annotation . 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 $\boldsymbol{ancestors}(n)$ are valid (condition *(ii)*).

Lemma A.1. Given an access specification S=(D, $\boldsymbol{ann}$), 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: $n \vDash \mathcal{A}^{acc}$. □

Proof A.2. Given an access specification S=(D, $\boldsymbol{ann}$), an XML tree $T \in \mathcal{T}(D)$, and its authorized view T_A, then, to prove that our accessibility predicate $\mathcal{A}^{acc}$ is "*correct*" (i.e. it can be used to select all and only accessible nodes, Lemma A.1), we have to show that: (i) for any node n_{T_A} of T_A, its corresponding node n_T must satisfy $\mathcal{A}^{acc}$; and (ii) for any node n_T of T that satisfies $\mathcal{A}^{acc}$, its corresponding node n_{T_A} must appear in T_A. In other words, one can extract all the nodes of T_A from T using our accessibility predicate $\mathcal{A}^{acc}$, moreover, all the accessible nodes of T (i.e. selected using $\mathcal{A}^{acc}$) represent all and only the nodes of T_A. Specifically, we should prove that: $\mathcal{S}[\![\downarrow^*::*]\!](T_A) = \mathcal{S}[\![\downarrow^*::*[\mathcal{A}^{acc}]]\!](T)$. We denote by R_1 and R_2 the nodes returned by $\mathcal{S}[\![\downarrow^*::*]\!](T_A)$ and $\mathcal{S}[\![\downarrow^*::*[\mathcal{A}^{acc}]]\!](T)$ respectively.

Consider an accessible node n_{T_A} of T_A and its corresponding node n_T in T. According to the Definition A.1, this means that there is no invalid downward-closed annotation defined over ancestors of n_{T_A}, and one of the following conditions hold:

i) n_{T_A} is the root node.

ii) n_{T_A} is of type B, its parent node is of type A, and $\boldsymbol{ann}(A, B)$ is explicitly defined and it is valid at n_{T_A}.

iii) No annotation is explicitly defined over the node n_{T_A} and this latter inherits its accessibility from its parent node m_{T_A}.

If n_{T_A} has at least one ancestor m'_{T_A} that is concerned by an invalid downward-closed annotation, then all the subtree rooted at m'_{T_A} is hidden (as we do using our materialization algorithm of Chapter 4), the node n_{T_A} becomes inaccessible and does not appear in T_A, which contradicts our assumption that n_{T_A} is accessible. By assuming that no invalid downward-closed annotation is defined over ancestors of n_{T_A}, the predicate $\mathcal{A}_2^{acc}$ (that is defined only over downward-closed annotations) is true at n_{T_A} and then our accessibility predicate $\mathcal{A}^{acc}$ can be reduced to $\mathcal{A}_1^{acc}$ (since $\mathcal{A}^{acc} = \mathcal{A}_1^{acc} \wedge \mathcal{A}_2^{acc}$). Now, it remains to prove that $R_1 \subseteq R_2$ and $R_2 \subseteq R_1$ for all the cases (i), (ii), and (iii).

Case I: (n_{T_A} *is the root node*)
Recall that for any access specification and any XML tree, the root node of this tree is accessible by default which is specified by the special annotation $\boldsymbol{ann}(root)$=Y. Assume that n_{T_A} is accessible, then since its corresponding node n_T is the root node of T then n_T must be accessible too. The predicate $\mathcal{A}^{acc}$ is defined over n_T as follows:

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

This predicate is valid at the node n_T as shown by the following deductions:

- $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\}) = \mathcal{S}[\![\uparrow^*::*[allAnn][1][validAnn]]\!](\{n_T\})$
$= \xi[\![[allAnn][1][validAnn]]\!](\underbrace{\mathcal{S}[\![\uparrow^*:: *]\!](\{n_T\})}_{\{n_T\}})$

- $\xi[\![[allAnn][1][validAnn]]\!](\{n_T\}) = \xi[\![[1][validAnn]]\!](\xi[\![[allAnn]]\!](\{n_T\}))$

- $\xi[\![[allAnn]]\!](\{n_T\}) = \xi[\![[\varepsilon::root \vee_{ann(A',A)\in \mathbf{ann}} \varepsilon::A/\uparrow::A']]\!](\{n_T\})$

- We have $\xi[\![[\varepsilon::root]]\!](\{n_T\})=\{n_T\}$. Moreover, since the root type is concerned only by $\mathbf{ann}(root)=Y$, then for each other annotation $\mathbf{ann}(A',A)$, $\xi[\![[\varepsilon::A/\uparrow::A']]\!](\{n_T\})=\varnothing$. Thus: $\xi[\![[allAnn]]\!](\{n_T\})=\xi[\![[\varepsilon::root \vee_{ann(A',A)\in \mathbf{ann}} \varepsilon::A/\uparrow::A']]\!](\{n_T\})=\{n_T\}$.

- $\xi[\![[1][validAnn]]\!](\xi[\![[allAnn]]\!](\{n_T\})) = \xi[\![[1][validAnn]]\!](\{n_T\})$
$= \xi[\![[validAnn]]\!](\xi[\![[1]\{n_T\}]\!]) = \xi[\![[validAnn]]\!](\{n_T\})$.

- Since n_T is the root of T, n_T is of type $root$ and then the predicate $[validAnn]$ is valid at the node n_T (i.e. the sub-predicate $\varepsilon::root$ is satisfied at n_T). Thus: $\xi[\![[validAnn]]\!](\{n_T\})=\{n_T\}$.

- We conclude that: $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\})=\{n_T\}$ which means that n_T is accessible.

We shown that for the root node n_{T_A} of T_A in R_2, its corresponding node n_T exists in R_2. We show now the reverse: whether for each node n_T of R_2, its corresponding node $n_{T_A} \in R_1$. We prove this by contradiction, assume that $n_T \vDash \mathcal{A}^{acc}$ but $n_{T_A} \notin R_1$ (i.e. n_{T_A} is not accessible). The first case of this proof consists in considering n_T and n_{T_A} as the root nodes of T and T_A respectively. Thus, the node n_{T_A} is inaccessible if and only if there is an explicitly defined annotation $\mathbf{ann}(root)=N$. Since the root node is not accessible, we eliminate $\varepsilon::root$ from $[validAnn]$ when computing $\mathcal{A}^{acc}$. In this case, the evaluation $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\})$ is given by:

- $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\}) = \xi[\![[allAnn][1][validAnn]]\!](\underbrace{\mathcal{S}[\![\uparrow^*:: *]\!](\{n_T\})}_{\{n_T\}})$
$= \xi[\![[allAnn][1][validAnn]]\!](\{n_T\})$
$= \xi[\![[1][validAnn]]\!](\underbrace{\xi[\![[allAnn]]\!](\{n_T\})}_{\{n_T\}}) = \xi[\![[1][validAnn]]\!](\{n_T\})$

- $\xi[\![[1][validAnn]]\!](\{n_T\}) = \xi[\![[validAnn]]\!](\xi[\![[1]\{n_T\}]\!]) = \xi[\![[validAnn]]\!](\{n_T\})$.

- $[validAnn]$ is given by $[\vee_{(\mathbf{ann}(A',A)=Y)\in \mathbf{ann}} \varepsilon::A/\uparrow::A' \vee_{(\mathbf{ann}(A',A)=[Q]|[Q]_h)\in \mathbf{ann}} \varepsilon::A[Q]/\uparrow::A']$ and this by assuming that $\mathbf{ann}(root)=N$.

- Since n_T is the root of T, for each other annotation $\mathbf{ann}(A',A)$: $\xi[\![[\varepsilon::A/\uparrow::A']]\!](\{n_T\})=\varnothing$. Thus, $\xi[\![[validAnn]]\!](\{n_T\})=\varnothing$.

- We conclude that: $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\})=\varnothing$ which means that $n_T \nvDash \mathcal{A}^{acc}$.

This contradicts our assumption that $n_T \vDash \mathcal{A}^{acc}$, then we conclude that n_{T_A} is accessible. In other words, for each n_T of R_2, its corresponding node $n_{T_A} \in R_1$. Finally, for the case (**I**) where n_T and n_{T_A} are the root nodes of T and T_A respectively, we shown that $R_1 = R_2$.

Case II: (n_{T_A} *is valid with an explicit annotation defined over its type*)
Assume that the node n_{T_A} is concerned by an explicit annotation $\boldsymbol{ann}(A, B)$, i.e. n_{T_A} if of type B, its parent node (which may be accessible or not) is of type A, and $\boldsymbol{ann}(A, B)$ is valid at n_{T_A} (this means that either $\boldsymbol{ann}(A, B)=Y$, or $\boldsymbol{ann}(A, B)=[Q]||[Q]_h$ with $n_{T_A} \vDash Q$). According to this assumption, we should show that $n_T \vDash \mathcal{A}^{acc}$. We denote by $B \circ \sigma(A, B)$ either B if $\boldsymbol{ann}(A, B)=Y$, or $B[Q]$ if $\boldsymbol{ann}(A, B)=[Q]||[Q]_h$. Note that the annotation $\boldsymbol{ann}(A, B)$ is valid at n_T iff: $n_T \vDash [\varepsilon :: B \circ \sigma(A, B)/\uparrow :: A]$. Let us take a look at the predicate $\mathcal{A}^{acc}$:

- $\mathcal{S}[\![\uparrow^* :: *]\!](\{n_T\}) = \{n_T, m_T^k, \ldots, m_T^0, root\}$, where $m_T^k, \ldots, m_T^0$ ($k \geq 0$) are the ancestors of n_T that connect it with $root$.

- $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\}) = \xi[\![[allAnn][1][validAnn]]\!](\mathcal{S}[\![\uparrow^* :: *]\!](\{n_T\}))$
$= \xi[\![[allAnn][1][validAnn]]\!](\{n_T, m_T^k, \ldots, m_T^0, root\})$
$= \xi[\![[1][validAnn]]\!](\xi[\![[allAnn]]\!](\{n_T, m_T^k, \ldots, m_T^0, root\}))$.

- $[allAnn]$ is given by $[\varepsilon :: root \vee \varepsilon :: B/\uparrow :: A \vee \cdots]$[46]

- $\xi[\![[allAnn]]\!](\{n_T, m_T^k, \ldots, m_T^0, root\}) = \xi[\![[\varepsilon :: root \vee \varepsilon :: B/\uparrow :: A \vee \cdots]]\!](\{n_T, m_T^k, \ldots, m_T^0, root\})$ $= \{n_T, \ldots, root\}$ (since we know that at least the nodes n_T and $root$ are concerned by some annotations, we denote by $\ldots$ the ancestors of n_T that are concerned by some annotations).

- $\xi[\![[1][validAnn]]\!](\xi[\![[allAnn]]\!](\{n_T, m_T^k, \ldots, m_T^0, root\})) = \xi[\![[1][validAnn]]\!](\{n_T, \ldots, root\})$
$= \xi[\![[validAnn]]\!](\xi[\![[1]]\!](\{n_T, \ldots, root\})) = \xi[\![[validAnn]]\!](\{n_T\})$.

- The predicate $[validAnn]$ is given by: $[\varepsilon :: root \vee \varepsilon :: B \circ \sigma(A, B)/\uparrow :: A \vee \cdots]$. It is clear that the predicate $[validAnn]$ is valid at the node n_T since there is at least the sub-predicate $\varepsilon :: B \circ \sigma(A, B)/\uparrow :: A$ which is satisfied at n_T according to our assumption.

- Thus:$\xi[\![[validAnn]]\!](\{n_T\})=\{n_T\}$. We conclude that: $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\})=\{n_T\}$ which means that n_T is accessible.

We have shown that for an n_{T_A} of T_A that is accessible w.r.t an explicit annotation $\boldsymbol{ann}(A, B)$, its corresponding node n_T satisfies $\mathcal{A}^{acc}$, i.e., for any n_{T_A} of R_1 that is concerned by an explicit valid annotation, $n_T \in R_2$. We show now the reverse by assuming that $n_T \vDash \mathcal{A}^{acc}$ but $n_{T_A} \notin R_1$. The node n_{T_A} is inaccessible, this means that either it inherits its accessibility from an invalid ancestor, or it is concerned by an invalid annotation (i.e. either $\boldsymbol{ann}(A, B)=N|N_h$, or $\boldsymbol{ann}(A, B)=[Q]||[Q]_h$ with $n \nvDash Q$). The former case contradicts the case (**II**) of this proof where we consider that n_{T_A} is concerned by an explicit annotation. While, the latter case contradicts our assumption that $n_T \vDash \mathcal{A}^{acc}$ as we show in the following. The predicate $[validAnn]$ is computed w.r.t to only valid annotations as follows:

- $\boldsymbol{ann}(A, B)=N|N_h$:
$[validAnn]=[\varepsilon :: root \vee_{(\boldsymbol{ann}(A'', A')=Y) \in \boldsymbol{ann}} \varepsilon :: A'/\uparrow :: A'' \vee_{(\boldsymbol{ann}(A'', A')=[Q]||[Q]_h) \in \boldsymbol{ann}} \varepsilon :: A'[Q]/\uparrow :: A'']$.
Where for each annotation $\boldsymbol{ann}(A'', A')$, $A'' \neq A$ and $A' \neq B$, i.e. the invalid annotation $\boldsymbol{ann}(A,B)$ is not considered when computing $[validAnn]$.

- $\boldsymbol{ann}(A, B)=[Q]||[Q]_h$:
$[validAnn]=[\varepsilon :: root\ \varepsilon :: B[Q]/\uparrow :: A \vee \cdots]$.

Thus, in case of $\boldsymbol{ann}(A, B)=N|N_h$, the sub-predicate $\varepsilon :: B/\uparrow :: A$ does not appear in $[validAnn]$, while if $\boldsymbol{ann}(A, B)=[Q]||[Q]_h$ then we include the sub-predicate $\varepsilon :: B[Q]/\uparrow :: A$ to check at run-time whether or not Q is valid at n_T. Consider the first case, we have $\xi[\![[\varepsilon :: root]]\!](\{n_T\})=\varnothing$

[46] We denote by $\cdots$ the disjunction of other annotations that are not defined over type of n_T.

since n_T is of type B. Moreover, $\xi[\![[\vee_{(\boldsymbol{ann}(A'',A')=Y)\in \boldsymbol{ann}}\ \varepsilon::A'/\uparrow::A'' \vee_{(\boldsymbol{ann}(A'',A')=[Q]|[Q]_h)\in \boldsymbol{ann}} \varepsilon::A'[Q]/\uparrow::A'']\!](\{n_T\}) = \varnothing$ since no sub-predicate $\varepsilon::B \circ \sigma(A,B)/\uparrow::A$ is defined in $[validAnn]$. Thus, $\xi[\![[validAnn]]\!](\{n_T\})=\varnothing$. For the second case, the sub-predicate $\varepsilon::B[Q]/\uparrow::A$ is evaluated to false at n_T (as we assume that n_T is concerned by an invalid annotation $\boldsymbol{ann}(A,B)=[Q]|[Q]_h$ with $n_T \not\models Q$), then $\xi[\![[validAnn]]\!](\{n_T\})=\varnothing$. In the two cases, we shown that $\xi[\![[validAnn]]\!](\{n_T\})=\varnothing$ which means that $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\})=\varnothing$. This contradicts our assumption that n_T is accessible and then we conclude that n_{T_A} is accessible.

Finally, we have shown that for any node n_{T_A} of R_1 that is explicitly concerned by a valid annotation, its corresponding node n_T satisfies $\mathcal{A}^{acc}$. Moreover, for any node n_T of R_2 its corresponding node n_{T_A} is accessible and appears in T_A. Therefore, for the case (**II**) the equality $R_1 = R_2$ is satisfied.

Case III. (n_{T_A} *inherits its accessibility from its parent node*)
Assume that the node n_{T_A} inherits its accessibility from its accessible parent node. This means that either *a*) n_{T_A} inherits its accessibility from the root node, or *b*) n_{T_A} has an accessible parent node m_{T_A} that is explicitly concerned by a valid annotation. We consider each of these cases separately:

a) The node n_{T_A} inherits its accessibility from the *root* node. Consider the path $root, m^0_{T_A}, \ldots, m^k_{T_A}, n_{T_A}$ ($k \geq 0$) that connects the node n_{T_A} with the *root* node. Since n_{T_A} inherits its accessibility from *root* then there is no annotation defined over the ancestors of n_{T_A} (i.e. the nodes $m^0_{T_A}, \ldots, m^k_{T_A}$). Let us take a look at the accessibility predicate $\mathcal{A}^{acc}$ defined over n_T:

- $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\}) = \xi[\![[allAnn][1][validAnn]]\!](\mathcal{S}[\![\uparrow^*::*]\!](\{n_T\}))$
  $= \xi[\![[allAnn][1][validAnn]]\!](\{n_T, m^k_T, \ldots, m^0_T, root\})$
  $= \xi[\![[1][validAnn]]\!](\xi[\![[allAnn]]\!](\{n_T, m^k_T, \ldots, m^0_T, root\})).$

  Since only the *root* node is concerned by an explicit annotation, then:
  $\xi[\![[allAnn]]\!](\{n_T, m^k_T, \ldots, m^0_T, root\})=\{root\}$.

- $[validAnn]$ is given by $[\varepsilon::root \vee \cdots]$ and it is clear that the predicate $[validAnn]$ is valid at *root*. We have: $\xi[\![[1][validAnn]]\!](\xi[\![[allAnn]]\!](\{n_T, m^k_T, \ldots, m^0_T, root\}))=$
  $\xi[\![[1][validAnn]]\!](\{root\})=\xi[\![[validAnn]]\!](\xi[\![[1]]\!](\{root\}))= \xi[\![[validAnn]]\!](\{root\})=\{root\}$.

- We conclude that: $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\})=\{root\}$ which means that $n_T \models \mathcal{A}^{acc}$.

We shown that for an n_{T_A} that inherits its accessibility from the root node, $n_T \in R_2$. We show now the reverse by assuming by contradiction that $n_T \models \mathcal{A}^{acc}$ but $n_{T_A} \notin R_1$. Considering that the node n_{T_A} is inaccessible means that either n_{T_A} is explicitly concerned by an invalid annotation or inherits its accessibility from an inaccessible ancestor. The former case contradicts the case (**III**) of this proof where we assume that the accessibility of n_T (resp. n_{T_A}) is inherited. Consider now the latter case where n_{T_A} is in the scope of an inaccessible ancestor. Since in case (**b**) we assume that n_{T_A} inherits its accessibility from the root node, then the latter case implies that the root node is inaccessible, i.e. the annotation $\boldsymbol{ann}(root)=N$ is explicitly defined. This contradicts our assumption that $n_T \models \mathcal{A}^{acc}$ as we show in the following. The node n_T inherits its accessibility from the root node, then no annotation is defined over n_T neither over its ancestors excepting the *root* node. Thus, $\xi[\![[allAnn]]\!](\mathcal{S}[\![\uparrow^*::*]\!](\{n_T\}))=\{root\}$ (since *root* is the only node that is concerned by an annotation), and $\xi[\![[allAnn][1]]\!](\{root\})=\{root\}$. We have $\xi[\![[validAnn]]\!](\{root\})=\varnothing$ since *root* is concerned by an invalid annotation. Therefore,

$\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\})=\varnothing$, which means that $n_T \not\models \mathcal{A}^{acc}$. This contradicts our assumption which considers that $n_T \models \mathcal{A}^{acc}$. We conclude then that for each node n_T of R_2, its corresponding node $n_{T_A} \in R_1$.

b) The node n_{T_A} inherits its accessibility from its parent node m_{T_A} that is explicitly concerned by a valid annotation. Consider the node m'_{T_A} to be the parent node of m_{T_A} where m'_{T_A} and m_{T_A} are of type A' and A respectively. The node n_{T_A} inherits its accessibility from m_{T_A} means that there is no annotation defined over the type of the node n_{T_A}. We can see easily that the predicate $\mathcal{A}^{acc}$ is valid at the node n_T of T. We have $\mathcal{S}[\![\uparrow^*::*]\!](\{n_T\})=\{n_T, m_T, m'_T, m^0_T, \ldots, m^k_T, root\}$, $\xi[\![[allAnn]]\!](\{n_T, m_T, m'_T, m^0_T, \ldots, m^k_T, root\})=\{m_T, \ldots, root\}$ (at least we know that there are m_T and $root$ which are concerned by explicit annotations, with $\ldots$ we denote the other nodes among $m'_T, m^0_T, \ldots, m^k_T$ that are concerned with some annotations). It is easy to verify that: $\xi[\![[validAnn][1]]\!](\xi[\![[allAnn]]\!](\mathcal{S}[\![\uparrow^*::*]\!](\{n_T\})))=\xi[\![[validAnn]]\!]\{m_T\}$. In addition, $[validAnn]$ is given by $\varepsilon::root \vee \varepsilon::A \circ \sigma(A', A)/\uparrow::A' \vee \cdots$. It is clear that $\xi[\![[validAnn]]\!](\{m_T\})=\{m_T\}$ since we assume that the node m_T is concerned by an explicit valid annotation $\boldsymbol{ann}(A',A)$, then we conclude that: $\mathcal{S}[\![\mathcal{A}^{acc}]\!]=\{m_T\}$, in other words, $n_T \models \mathcal{A}^{acc}$.

We show now the reverse: whether for each $n_T \in R_2$ its corresponding node $n_{T_A} \in R_1$. We assume by contradiction that $n_T \models \mathcal{A}^{acc}$ where n_{T_A} is inaccessible. Considering that the node n_{T_A} is inaccessible means that either n_{T_A} is explicitly concerned by an invalid annotation or in the scope of an inaccessible ancestor. The former case contradicts the case (**III**) of this proof where we assume that the accessibility of n_T (resp. n_{T_A}) is inherited. While, the latter case contradicts our assumption that $n_T \models \mathcal{A}^{acc}$ as we show in the following. Let $\boldsymbol{ann}(A, B)$ be the invalid annotation that is defined over some ancestor of n_T (denoted m^i_T in the following). We compute the predicate $\mathcal{A}^{acc}$ in this case as follows:

- $\begin{aligned}\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\}) &= \xi[\![[allAnn][1][validAnn]]\!](\mathcal{S}[\![\uparrow^*::*]\!](\{n_T\})) \\ &= \xi[\![[allAnn][1][validAnn]]\!](\{n_T, m^k_T, \ldots, m^0_T, root\})= \\ &= \xi[\![[1][validAnn]]\!](\xi[\![[allAnn]]\!](\{n_T, m^k_T, \ldots, m^0_T, root\})).\end{aligned}$

- Suppose that m^i_T is the ancestor of n_T that is concerned by an invalid annotation. Thus, $\xi[\![[allAnn]]\!](\{n_T, m^k_T, \ldots, m^0_T, root\})=\xi[\![[\varepsilon::root \vee \varepsilon::B/\uparrow::A \vee \cdots]]\!](\{n_T, m^k_T, \ldots, m^0_T, root\})=\{m^i_T, \ldots, root\}$ (since at least we know that m^i_T and $root$ are concerned by some annotations).

- $\xi[\![[1][validAnn]]\!](\{m^i_T, root\})=\xi[\![[validAnn]]\!](\xi[\![[1]]\!](\{m^i_T, root\}))=\xi[\![[validAnn]]\!](\{m^i_T\})$.

- $[validAnn]$ is given by $[\varepsilon::root \vee \varepsilon::B \circ \sigma(A, B)/\uparrow::A \vee \cdots]$ and it is clear that the predicate $[validAnn]$ is not valid at m^i_T since this node is concerned by only one annotation $\boldsymbol{ann}(A, B)$ which is invalid according to our assumption.

- Thus, $\xi[\![[validAnn]]\!](\{m^i_T\})=\varnothing$. We conclude that $\mathcal{S}[\![\mathcal{A}^{acc}]\!](\{n_T\})=\varnothing$ which means that $n_T \not\models \mathcal{A}^{acc}$. This contradicts our assumption that $n_T \models \mathcal{A}^{acc}$ and thus n_{T_A} is accessible.

In the case where the node n_T (resp. n_{T_A}) inherits its accessibility from its parent node, we have shown that for each node n_{T_A} of R_1, its corresponding node n_T satisfies $\mathcal{A}^{acc}$ in T, moreover, for each node n_T of T that satisfies $\mathcal{A}^{acc}$ (i.e. $n_T \in R_2$), its corresponding node $n_{T_A} \in R_1$. In other words, $R_1 = R_2$.

$\because \quad \therefore \quad \therefore \quad \therefore \quad \because$

Summing up, we have proven that for the different cases of nodes accessibility (case of root node, a node that is explicitly concerned by a valid annotation, or a node that inherits its accessibility from its accessible parent node), a node n_{T_A} of T_A can be selected from T using our accessibility predicate $\mathcal{A}^{acc}$, and each node n_T of T that satisfies $\mathcal{A}^{acc}$ appears in T_A. In other words, for each access specification S=($\mathtt{ann}$,D), an XML tree $T \in \mathcal{T}(D)$, and its authorized view T_A, we have: $\mathcal{S}[\![\downarrow^*::*]\!](T_A) = \mathcal{S}[\![\downarrow^*::*[\mathcal{A}^{acc}]]\!](T)$. □

A.2.2 Correctness of our Algorithm *Rewrite*

Recall that the fragment $\mathcal{X}$ of Definition 3.10 is used in our case only to define security policies as well as to formulate user requests (i.e. access queries and update operations). While, the extended fragment of Definition 3.11 is used to safely translate these requests, defined over virtual views, in order to be evaluated over the original data.

Definition A.3. Given an access specification S=(D,$\mathtt{ann}$), an XML tree $T \in \mathcal{T}(D)$ and its authorized view T_A. A rewriting function $\mathcal{R}$ is said to be *correct* w.r.t S for a class of XPath queries $\mathcal{C}$ if and only if: for any query Q of $\mathcal{C}$, the evaluation of Q over T_A yields the same set of nodes as the evaluation of the rewritten query $\mathcal{R}(Q)$ over T, i.e. $\mathcal{S}[\![\mathcal{R}(Q)]\!](T)$=$\mathcal{S}[\![Q]\!](T_A)$. □

To prove the correctness of our rewriting approach we have to show that our rewriting function $\mathtt{Rewrite}$ is correct for any query of the fragment $\mathcal{X}$. We denote by $\mathcal{X}_{noPred}$ all queries of fragment $\mathcal{X}$ defined without predicates. More formally, $\mathcal{X}_{noPred}$ is defined by:

$$p \;:=\; \alpha::ntst \;\mid\; p/p \;\mid\; p \cup p$$
$$\alpha \;:=\; \varepsilon \mid \downarrow \mid \downarrow^+ \mid \downarrow^*$$

The correctness of our rewriting function $\mathtt{Rewrite}$ is proven in two parts: first for queries without predicates, after we show that the rewriting of $\mathcal{X}$ predicates using our function $\mathtt{RW\_Pred}$ is correct.

Lemma A.2. The query rewriting algorithm $\mathtt{Rewrite}$ is correct for any query of the fragment $\mathcal{X}_{noPred}$. □

Proof A.3. Recall that in our case virtual views of the data,that are provided to the user, are never materialized. Moreover, authorized views mentioned here are used only to prove correctness of our approach. Let S=(D,$\mathtt{ann}$) be an access specification, $T \in \mathcal{T}(D)$ be an XML tree and T_A be the authorized version of T. Consider now the user query Q_n of fragment $\mathcal{X}_{noPred}$ defined over some virtual view of T with $q_1/\ldots/q_n$ where: q_i=$axis_i$::e_i, and e_i can be element type, $*$, or *text()* function. We should prove that: $\mathcal{S}[\![\mathtt{Rewrite}(Q_n)]\!](T)$=$\mathcal{S}[\![Q_n]\!](T_A)$. Let us prove it by induction.

Basis of induction: We show that $\mathcal{S}[\![\mathtt{Rewrite}(Q_1)]\!](T)$=$\mathcal{S}[\![Q_1]\!](T_A)$. We have Q_1=$axis_1$::e_1, this query is rewritten particularly over the *root* type of the DTD. Consider the different cases of $axis_1$ as follows:

•*Case of* $\downarrow::e_1$. We have to show that: $\mathcal{S}[\![\downarrow::e_1]\!](T_A)$=$\mathcal{S}[\![\mathtt{Rewrite}(\downarrow::e_1)]\!](T)$. The query Q_1 is evaluated directly over the *root* node of T_A. The evaluation $\mathcal{S}[\![\downarrow::e_1]\!](T_A)$ must return all

the accessible nodes of T_A that are of type e_1[47] and appear in T_A as immediate children of *root*. Let $\mathcal{S}[\![\downarrow::e_1]\!](T_A)=R_1$. Since nodes of R_1 are accessible, then for each node $n_{T_A} \in R_1$, its corresponding node n_T of T satisfies $\mathcal{A}^{acc}$ (see Lemma A.1). Moreover, n_T is either the immediate child of *root* in T, or separated from *root* with some nodes $n_1, \ldots, n_k$ where $n_i \nvDash \mathcal{A}^{acc}$ $(1 \leq i \leq k)$, for this reason n_{T_A} appears as immediate child of *root* in T_A after hiding the inaccessible nodes $n_1, \ldots, n_k$. The evaluation $\mathcal{S}[\![\downarrow^*::e_1[\mathcal{A}^{acc}]]\!](T)$ returns the *root* node as well as all accessible descendants of the *root* node in T. We define two sets of nodes $R_1^{'}$ and R_2 that represent respectively 1) the nodes which are connected to *root* with only inaccessible nodes, and 2) the nodes that are connected to *root* with at least one accessible node. It is clear, that each node n_T of $R_1^{'}$ appears in T_A as immediate child of *root* after hiding inaccessible nodes between *root* and n_T. This means that $R_1 = R_1^{'}$. Thus, $\mathcal{S}[\![\downarrow^*::e_1[\mathcal{A}^{acc}]]\!](T) = \{root\} \cup R_1 \cup R_2$.

We have $\mathtt{Rewrite}(\downarrow::e_1)=\downarrow^*::e_1[\mathcal{A}^{acc}][\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]$, and $\mathcal{S}[\![\mathtt{Rewrite}(\downarrow::e_1)]\!](T)$ is computed as follows:

1. $\mathcal{S}[\![\downarrow^*::e_1[\mathcal{A}^{acc}][\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]]\!](T)=\xi[\![[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]]\!](\mathcal{S}[\![\downarrow^*::e_1[\mathcal{A}^{acc}]]\!](T))=$
$\xi[\![[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]]\!](\{root\} \cup R_1 \cup R_2)$.

2. $\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]\!](\{root\})= \mathcal{S}[\![\varepsilon::root]\!](\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}][1]]\!](\{root\}))=$
$\mathcal{S}[\![\varepsilon::root]\!](\xi[\![[\mathcal{A}^{acc}][1]]\!](\mathcal{S}[\![\uparrow^+::*]\!](\{root\})))=\mathcal{S}[\![\varepsilon::root]\!](\xi[\![[\mathcal{A}^{acc}][1]]\!](\varnothing))=\varnothing$.

This means that the predicate $[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]$ is not satisfied at *root*. Thus: $\xi[\![[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]]\!](\{root\})=\varnothing$.

3. $\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]\!](R_1)= \mathcal{S}[\![\varepsilon::root]\!](\xi[\![[1]]\!](\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}]]\!](R_1)))$.
The query $\uparrow^+::*[\mathcal{A}^{acc}]$ over a node n returns all its accessible ancestors. Since nodes of R_1 are connected to *root* with only inaccessible nodes, then $\uparrow^+::*[\mathcal{A}^{acc}]$ over each node of R_1 returns $\{root\}$. Thus:

$\mathcal{S}[\![\varepsilon::root]\!](\xi[\![[1]]\!](\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}]]\!](R_1)))=\mathcal{S}[\![\varepsilon::root]\!](\xi[\![[1]]\!](\{root\}))=\mathcal{S}[\![\varepsilon::root]\!](\{root\})=\{root\}$.

In other words, all nodes of R_1 satisfy the predicate $[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]$, i.e. $\xi[\![[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]]\!](R_1)=R_1$.

4. Since R_2 is the set of nodes that are connected to the *root* with at least one accessible node, then for each node $n \in R_2$, the query $\uparrow^+::*[\mathcal{A}^{acc}]$ returns the set of all its accessible ancestors that are different to *root*. We have $\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}]]\!](\{n\})= \{m_1, \ldots, m_k, root\}$ where $m_1, \ldots, m_k$ $(k \geq 1)$ are the accessible nodes that exist between *root* and n (according to the definition of R_2).

$$\begin{aligned}\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]\!](\{n\}) &= \mathcal{S}[\![\varepsilon::root]\!](\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}][1]]\!](\{n\})) \\ &= \mathcal{S}[\![\varepsilon::root]\!](\xi[\![[1]]\!](\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}]]\!](\{n\}))) \\ &= \mathcal{S}[\![\varepsilon::root]\!](\xi[\![[1]]\!](\{m_1, \ldots, m_k, root\})) \\ &= \mathcal{S}[\![\varepsilon::root]\!](\{m_1\}) = \varnothing \text{ (since } m_1 \text{ is not of type } root).\end{aligned}$$

We shown that for each node $n \in R_2$: $\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]\!](\{n\})=\varnothing$. We conclude that $\xi[\![[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]]\!](\{n\})=\varnothing$. In other words, no node of R_2 satisfies the predicate $[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]$. Thus: $\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]\!](R_2)=\varnothing$.

[47] Node of type e_1 means: text node if e_1=*text()*, node with any type if e_1=$*$, or otherwise, node with an explicit type e_1.

$\mathcal{S}[\![\textit{Rewrite}(\downarrow::e_1)]\!](T)= \xi[\![[\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::root]]\!](\{root\} \cup R_1 \cup R_2)$. From (*2*), (*3*) and (*4*) we conclude that: $\textit{Rewrite}(\downarrow::e_1)=R_1$.

•*Case of* $\downarrow^*::e_1$. Lemma A.1 shows that $\mathcal{S}[\![\downarrow^*::*]\!](T_A)=\mathcal{S}[\![\downarrow^*::*[\mathcal{A}^{acc}]]\!](T)$. Let R be the set of nodes resulted from these two equivalent evaluations. Since R represents all the accessible nodes of T_A (resp. T), then $\xi[\![[\varepsilon::e_1]]\!](R)$ returns all the nodes of R that are of type e_1, i.e., all the accessible nodes of T_A (resp. T) that are of type e_1. Recall that $\textit{Rewrite}(\downarrow^*::e_1)=\downarrow^*::e_1[\mathcal{A}^{acc}]$. We show in the following that: $\mathcal{S}[\![\downarrow^*::e_1]\!](T_A)= \mathcal{S}[\![\downarrow^*::e_1[\mathcal{A}^{acc}]]\!](T)$.

1. $\xi[\![[\varepsilon::e_1]]\!](R)= \xi[\![[\varepsilon::e_1]]\!](\mathcal{S}[\![\downarrow^*::*]\!](T_A))= \mathcal{S}[\![\downarrow^*::*[\varepsilon::e_1]]\!](T_A)$.
Based on Property 3.1, $\downarrow^*::*[\varepsilon::e_1]$ is equivalent to $\downarrow^*::e_1$. We conclude that: $\xi[\![[\varepsilon::e_1]]\!](R)=\mathcal{S}[\![\downarrow^*::e_1]\!](T_A)$.

2. $\xi[\![[\varepsilon::e_1]]\!](R)= \xi[\![[\varepsilon::e_1]]\!](\mathcal{S}[\![\downarrow^*::*[\mathcal{A}^{acc}]]\!](T))= \mathcal{S}[\![\downarrow^*::*[\mathcal{A}^{acc}][\varepsilon::e_1]]\!](T)$.
Based on Property 3.1, $\downarrow^*::*[\mathcal{A}^{acc}][\varepsilon::e_1]$ is equivalent to $\downarrow^*::e_1[\mathcal{A}^{acc}]$. We conclude that: $\xi[\![[\varepsilon::e_1]]\!](R)=\mathcal{S}[\![\downarrow^*::*[\mathcal{A}^{acc}][\varepsilon::e_1]]\!](T)$.

Finally, form (*1*) and (*2*), we conclude that: $\mathcal{S}[\![\downarrow^*::e_1]\!](T_A)=\mathcal{S}[\![\textit{Rewrite}(\downarrow^*::e_1)]\!](T)$. Note that the case of $\downarrow^+::e_1$ can be done in the same way as $\downarrow^*::e_1$.

•*Case of* $\varepsilon::e_1$. Note that $\textit{Rewrite}(\varepsilon::e_1)= \varepsilon::e_1$. It is obvious then that: $\mathcal{S}[\![\varepsilon::e_1]\!](T_A)=\mathcal{S}[\![\textit{Rewrite}(\varepsilon::e_1)]\!](T)$.

Inductive assumption: Assume that for a query Q_n of fragment $\mathcal{X}_{noPred}$ with size n (i.e. $Q_n=q_1/\ldots/q_n$), $\mathcal{S}[\![Q_n]\!](T_A)=\mathcal{S}[\![\textit{Rewrite}(Q_n)]\!](T)$. Let R be the set of nodes resulted from these two equivalent evaluations.

Inductive step: Based on the assumption, we should show that the rewriting of Q_{n+1} (i.e. Q_n/q_{n+1}) using our rewriting function *Rewrite* is still correct. More formally: $\mathcal{S}[\![Q_{n+1}]\!](T_A)=\mathcal{S}[\![\textit{Rewrite}(Q_{n+1})]\!](T)$. Consider the different cases of q_{n+1} as follows:

•*Case of* $\downarrow::e_{n+1}$. Let R_1 and R_2 be the sets of nodes resulted from the evaluation of $\mathcal{S}[\![Q_{n+1}]\!](T_A)$ and $\mathcal{S}[\![\textit{Rewrite}(Q_{n+1})]\!](T)$ respectively. We have to prove that $R_1 = R_2$. In other words, for any node n_{T_A} that is selected from T_A by Q_{n+1}, its corresponding node n_T of T can be selected with $\textit{Rewrite}(Q_{n+1})$, and vice versa.

a) $R_1 \subseteq R_2$**.** We have $\mathcal{S}[\![Q_{n+1}]\!](T_A)= \mathcal{S}[\![q_{n+1}]\!](\mathcal{S}[\![Q_n]\!](T_A))= \mathcal{S}[\![q_{n+1}]\!](R)=R_1$. Thus, any node n_{T_A} of R_1 is of type e_{n+1} and has an accessible parent m_{T_A} of type e_n such that $m_{T_A} \in R$. Given a node $n_{T_A} \in R_1$ and assume (***i***) that its corresponding node in T is n_T, this means that n_T in T has the same properties as n_{T_A} in T_A (n_T is accessible, of type e_{n+1}, and has an accessible parent of type e_n that belongs to R). According to the inductive assumption, $R=\mathcal{S}[\![Q_n]\!](T_A)= \mathcal{S}[\![\textit{Rewrite}(Q_n)]\!](T)= \mathcal{S}[\![\downarrow^*::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]\!](T)$, Let m_T be the accessible parent of n_T that belongs to R, $m_T \in R$ implies that m_T satisfies the predicate $[prefix^{-1}(Q_{n-1})]$. Let us prove that $R_1 \subseteq R_2$ by contradiction. Assume that (***ii***) there exists a node $n_{T_A} \in R_1$ where its corresponding node n_T in T is not selected by $\textit{Rewrite}(Q_{n+1})$ (i.e. $n_T \notin R_2$). We have $\textit{Rewrite}(Q_{n+1})= \downarrow^*::e_{n+1}[\mathcal{A}^{acc}][\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]$. According to Property 3.1, $n_T \notin \mathcal{S}[\![\textit{Rewrite}(Q_{n+1})]\!](T)$ implies that: $\mathcal{S}[\![\varepsilon::e_{n+1}[\mathcal{A}^{acc}][\uparrow^+::*[\mathcal{A}^{acc}][1]/\varepsilon::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]]\!](\{n_T\}) \neq \{n_T\}$. Consider the following deductions:

- According to the assumption (***ii***), n_T accessible and of type e_{n+1}, then:
$\mathcal{S}[\![\varepsilon{::}e_{n+1}[\mathcal{A}^{acc}]]\!](\{n_T\})=\{n_T\}$
$\mathcal{S}[\![\varepsilon{::}e_{n+1}[\mathcal{A}^{acc}][\uparrow^+{::}*[\mathcal{A}^{acc}][1]/\varepsilon{::}e_n[\mathcal{A}^{acc}][\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]]\!](\{n_T\})=$
$\xi[\![[\uparrow^+{::}*[\mathcal{A}^{acc}][1]/\varepsilon{::}e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]]\!](\{n_T\})$.

- $\xi[\![[\uparrow^+{::}*[\mathcal{A}^{acc}]]\!](\{n_T\})=\{m_1,\ldots,m_k\}$, where $\{m_1,\ldots,m_k\}$ $(k\geq 1)$ are the accessible ancestors of n_T (if k=1 then m_1=*root*). Moreover:
$\mathcal{S}[\![\uparrow^+{::}*[\mathcal{A}^{acc}][1]]\!](\{n_T\})=\xi[\![[1]]\!](\mathcal{S}[\![\uparrow^+{::}*[\mathcal{A}^{acc}]]\!](\{n_T\}))=\{m_1\}$.

- According to the assumption (***i***) n_T has an accessible parent $m_T\in R$. From the previous deduction, this implies that $m_1\in R$.

- R=$\mathcal{S}[\![\texttt{Rewrite}(Q_n)]\!](T)=\mathcal{S}[\![\downarrow^*{::}e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]\!](T)$. According to Property 3.1, $m_1\in R$ implies that: $\mathcal{S}[\![\varepsilon{::}e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]\!](\{m_1\})=\{m_1\}$.

- We have $\mathcal{S}[\![\uparrow^+{::}*[\mathcal{A}^{acc}][1]/\varepsilon{::}e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]\!](\{n_T\})=\{m_1\}$. According to the semantics of XPath (Table 3.1), $\xi[\![[Q]]\!](\{n_T\})=\{n_T\}$ if and only if: $\mathcal{S}[\![Q]\!](\{n_T\})\neq\varnothing$. Therefore:
$\xi[\![[\uparrow^+{::}*[\mathcal{A}^{acc}][1]/\varepsilon{::}e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]]\!](\{n_T\})=\{n_T\}$.

- Summing up: $\mathcal{S}[\![\varepsilon{::}e_{n+1}[\mathcal{A}^{acc}][\uparrow^+{::}*[\mathcal{A}^{acc}][1]/\varepsilon{::}e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]]\!](\{n_T\})=$
$\xi[\![[\uparrow^+{::}*[\mathcal{A}^{acc}][1]/\varepsilon{::}e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]]\!](\mathcal{S}[\![\varepsilon{::}e_{n+1}[\mathcal{A}^{acc}]]\!](\{n_T\}))=$
$\xi[\![[\uparrow^+{::}*[\mathcal{A}^{acc}][1]/\varepsilon{::}e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]]\!](\{n_T\})=\{n_T\}$.
This contradicts our assumption that $n_T\notin\mathcal{S}[\![\texttt{Rewrite}(Q_{n+1})]\!](T)$.

Finally, we shown that for each node $n_{T_A}\in R_1$, its corresponding node $n_T\in R_2$, i.e. $R_1\subseteq R_2$. We show the reverse in the following.

b) $R_2\subseteq R_1$**.** Consider a node n_T that is referred to by the query $\texttt{Rewrite}(Q_{n+1})$ over T, i.e. n_T is of type e_{n+1} and has a parent node m_T of type e_n that is selected by $\texttt{Rewrite}(Q_n)$ over T (i.e. $m_T\in R$, R=$\mathcal{S}[\![\texttt{Rewrite}(Q_n)]\!](T)=\mathcal{S}[\![Q_n]\!](T_A)$). Assume that the node n_{T_A} is the corresponding node of n_T in T_A. This means that (***iii***) n_{T_A} has the same properties as n_T in T, i.e. n_{T_A} is of type e_{n+1} and has a parent node m_{T_A} of type e_n that is selected by Q_n over T_A. Let assume by contradiction that $n_{T_A}\notin R_1$.

According to Property 3.1, the query Q_n=$axis_1{::}e_1/\ldots/axis_n{::}e_n$ is equivalent to the query $\downarrow^*{::}e_n[axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]$, and Q_{n+1}=$axis_1{::}e_1/\ldots/axis_n{::}e_n/\downarrow{::}e_{n+1}$ is equivalent to $\downarrow^*{::}e_{n+1}[\uparrow{::}e_n/axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]$. Assuming that $n_{T_A}\notin\mathcal{S}[\![Q_{n+1}]\!](T_A)$ implies that:

$$\mathcal{S}[\![\varepsilon{::}e_{n+1}[\uparrow{::}e_n/axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]]\!](\{n_{T_A}\})\neq\{n_{T_A}\}$$

We consider the following deductions:

- $\mathcal{S}[\![\varepsilon{::}e_{n+1}[\uparrow{::}e_n/axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]]\!](\{n_{T_A}\})=$
$\xi[\![[\uparrow{::}e_n/axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]]\!](\mathcal{S}[\![\varepsilon{::}e_{n+1}]\!]\{n_{T_A}\}))$.

- According to (***iii***), n_{T_A} is of type e_{n+1}. Thus, $\mathcal{S}[\![\varepsilon{::}e_{n+1}]\!](\{n_{T_A}\})=\{n_{T_A}\}$.

- $[\uparrow{::}e_n/axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]=$
$[\uparrow{::}e_n[axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]]$.

- $\mathcal{S}[\![\uparrow{::}e_n[axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]]\!](\{n_{T_A}\})=$
$\xi[\![[axis_n^{-1}{::}e_{n-1}/\ldots/axis_2^{-1}{::}e_1/axis_1^{-1}{::}root]]\!](\mathcal{S}[\![\uparrow{::}e_n]\!]\{n_{T_A}\}))$.

- According to (***iii***), n_{T_A} has a parent node m_{T_A} of type e_n that is selected by Q_n over T_A. Thus, $\mathcal{S}[\![\uparrow::e_n]\!](\{n_{T_A}\})=\{m_{T_A}\}$.

- We conclude that: $\mathcal{S}[\![\varepsilon::e_{n+1}[\uparrow::e_n/axis_n^{-1}::e_{n-1}/\ldots/axis_2^{-1}::e_1/axis_1^{-1}::root]]\!](\{n_{T_A}\})=$ $\xi[\![[axis_n^{-1}::e_{n-1}/\ldots/axis_2^{-1}::e_1/axis_1^{-1}::root]]\!](\{m_{T_A}\})$.

- According to (***iii***), m_{T_A} is selected by Q_n at T_A implies that (Property 3.1): $\mathcal{S}[\![\varepsilon::e_n[axis_n^{-1}::e_{n-1}/\ldots/axis_2^{-1}::e_1/axis_1^{-1}::root]]\!](\{m_{T_A}\})=\{m_{T_A}\}$. This implies that: $\xi[\![[axis_n^{-1}::e_{n-1}/\ldots/axis_2^{-1}::e_1/axis_1^{-1}::root]]\!](\{n_{T_A}\})=\{n_{T_A}\}$.

- We conclude that: $\mathcal{S}[\![\varepsilon::e_{n+1}[\uparrow::e_n/axis_n^{-1}::e_{n-1}/\ldots/axis_2^{-1}::e_1/axis_1^{-1}::root]]\!](\{n_{T_A}\})=$ $\xi[\![[axis_n^{-1}::e_{n-1}/\ldots/axis_2^{-1}::e_1/axis_1^{-1}::root]]\!](\{n_{T_A}\})=\{n_{T_A}\}$.

We found that $\mathcal{S}[\![\varepsilon::e_{n+1}[\uparrow::e_n/axis_n^{-1}::e_{n-1}/\ldots/axis_2^{-1}::e_1/axis_1^{-1}::root]]\!](\{n_{T_A}\})= \{n_{T_A}\}$, this contradicts our assumption that $n_{T_A} \notin \mathcal{S}[\![Q_{n+1}]\!](T_A)$. Therefore, for any node $n_T \in R_2$, we shown that its corresponding node $n_{T_A} \in R_1$.

Finally, for the case of $q_{n+1}{=}\downarrow::e_{n+1}$, we shown that for each node of T_A referred to by Q_{n+1}, its corresponding node n_T is referred to by $\mathtt{Rewrite}(Q_{n+1})$ on T, and vice versa. This means that: $\mathcal{S}[\![Q_{n+1}]\!](T_A){=}\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T)$.

•*Case of* $\downarrow^+::e_{n+1}$. Based on the inductive assumption, $\mathcal{S}[\![Q_n]\!](T_A){=}\mathcal{S}[\![\mathtt{Rewrite}(Q_n)]\!](T){=}R$, $\mathcal{S}[\![Q_n/q_{n+1}]\!](T_A){=}\mathcal{S}[\![q_{n+1}]\!](\mathcal{S}[\![Q_n]\!](T_A)){=}\mathcal{S}[\![q_{n+1}]\!](R)$. In addition, $\mathtt{Rewrite}(Q_{n+1})$ is given by $\downarrow^*::e_{n+1}[\mathcal{A}^{acc}][\uparrow^+::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]$. According to the equivalences (1), (2), and (3) of Property 3.1 respectively, we generate the following deductions:

- $\downarrow^*::e_{n+1}[\uparrow^+::e_n]{=}\downarrow^*::e_n/\downarrow^+::e_{n+1}$.

- $\downarrow^*::e_{n+1}[\mathcal{A}^{acc}][\uparrow^+::e_n[\mathcal{A}^{acc}]]{=}\downarrow^*::e_n[\mathcal{A}^{acc}]/\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]$.

- $\downarrow^*::e_{n+1}[\mathcal{A}^{acc}][\uparrow^+::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]=$ $\downarrow^*::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]/\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]$.

- $\mathtt{Rewrite}(Q_{n+1}){=}\downarrow^*::e_{n+1}[\mathcal{A}^{acc}][\uparrow^+::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]=$ $\downarrow^*::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]/\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]$.

- $\mathtt{Rewrite}(Q_n){=}\downarrow^*::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]$.

- $\mathtt{Rewrite}(Q_{n+1}){=}\mathtt{Rewrite}(Q_n)/\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]$.

- $\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T){=}\mathcal{S}[\![\mathtt{Rewrite}(Q_n)/\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]]\!](T)=$ $\mathcal{S}[\![\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]]\!](\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T))$.

- $\mathcal{S}[\![Q_{n+1}]\!](T_A){=}\mathcal{S}[\![\downarrow^+::e_{n+1}]\!](\mathcal{S}[\![Q_n]\!](T_A)){=}\mathcal{S}[\![\downarrow^+::e_{n+1}]\!](R)$.

- $\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T){=}\mathcal{S}[\![\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]]\!](R)$.

- According to Lemma A.1, $\mathcal{S}[\![\downarrow^+::e_{n+1}]\!](R){=}\mathcal{S}[\![\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]]\!](R)$ as the accessibility predicate $\mathcal{A}^{acc}$ returns all and only accessible nodes.

We shown that $\mathcal{S}[\![Q_{n+1}]\!](T_A){=}\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T)$. We conclude that our algorithm $\mathtt{Rewrite}$ is correct for the case where q_{n+1} is defined with $\downarrow^+$ axis. Note that the case of $q_{n+1}{=}\downarrow^*::e_{n+1}$ can be done in a similar way.

•*Case of* $\varepsilon::e_{n+1}$. We should show that $\mathcal{S}[\![Q_n/\varepsilon::e_{n+1}]\!](T_A){=}\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T)$. We have $\mathcal{S}[\![Q_n/\varepsilon::e_{n+1}]\!](T_A)= \mathcal{S}[\![\varepsilon::e_{n+1}]\!](\mathcal{S}[\![Q_n]\!](T_A))= \mathcal{S}[\![\varepsilon::e_{n+1}]\!](R)$. Moreover, $\mathtt{Rewrite}(Q_{n+1})$ is

given by $\downarrow^*::e_{n+1}[\mathcal{A}^{acc}][\varepsilon::e_n[prefix^{-1}(Q_{n-1})]]$. Consider the different cases of e_n and e_{n+1} as follows:

a) $e_n{=}e_{n+1}$ or $e_{n+1}{=}*$. Since nodes of R are of type e_n, $\mathcal{S}[\![\varepsilon::e_n]\!](R){=}\mathcal{S}[\![\varepsilon::*]\!](R){=}R$. Moreover, $\downarrow::*[f_1][\varepsilon::e[f_2]]$ is equivalent to $\downarrow::e[f_1][f_2]$, and $\downarrow::e[f_1][\varepsilon::e[f_2]]$ is equivalent to $\downarrow::e[f_1][f_2]$. Thus, $\mathtt{Rewrite}(Q_{n+1}){=}\downarrow^*::e_n[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]{=}\mathtt{Rewrite}(Q_n)$, which implies that $\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T){=}\ \mathcal{S}[\![\mathtt{Rewrite}(Q_n)]\!](T){=}R$.

b) $e_n \neq *$, $e_{n+1} \neq *$, and $e_n \neq e_{n+1}$. Since R is a set of nodes of type e_n then it is clear that $\mathcal{S}[\![\varepsilon::e_{n+1}]\!](R){=}\varnothing$. Moreover, the query $\mathtt{Rewrite}(Q_{n+1})$ over any XML tree T returns no node since the nodes selected by $\downarrow^*::e_{n+1}[\mathcal{A}^{acc}]$ are of type e_{n+1}, while the predicate $[\varepsilon::e_n[prefix^{-1}(Q_{n-1})]]$ is valid at these nodes only if $e_{n+1}{=}e_n$ which is not the case. Thus, $\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T){=}\varnothing$.

c) $e_n{=}*$ and $e_{n+1} \neq *$. We have $\mathcal{S}[\![Q_n/\varepsilon::e_{n+1}]\!](T_A){=}\ \mathcal{S}[\![\varepsilon::e_{n+1}]\!](R)$. According to Property 3.1, $\mathtt{Rewrite}(Q_{n+1})$, given by $\downarrow::e_{n+1}[\mathcal{A}^{acc}][\varepsilon::*[prefix^{-1}(Q_{n-1})]]$, is equivalent to $\downarrow::e_{n+1}[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]$. Moreover, the query $\downarrow::e_{n+1}[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]{=}$ $\downarrow::*[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]/\varepsilon::e_{n+1}$. The evaluation $\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T)$ is given by: $\mathcal{S}[\![\downarrow::*[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]/\varepsilon::e_{n+1}]\!](T){=}\ \ \mathcal{S}[\![\varepsilon::e_{n+1}]\!](\mathcal{S}[\![\downarrow::*[\mathcal{A}^{acc}][prefix^{-1}(Q_{n-1})]]\!](T)){=}$ $\mathcal{S}[\![\varepsilon::e_{n+1}]\!](R)$. We conclude that $\mathcal{S}[\![Q_{n+1}]\!](T_A){=}\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T)$.

$\because \ \ \therefore \ \ \therefore \ \ \therefore \ \ \because$

Finally, for the different cases of q_{n+1} of the query Q_{n+1}, we have proven that $\mathcal{S}[\![Q_{n+1}]\!](T_A){=}\mathcal{S}[\![\mathtt{Rewrite}(Q_{n+1})]\!](T)$. This means that our rewriting approach is correct for a query of an arbitrary size. Furthermore, our rewriting algorithm `Rewrite` remains correct for any query of the fragment $\mathcal{X}_{noPred}$. □

Note.
The case of union of queries is obvious. For any queries Q_1 and Q_2 of fragment $\mathcal{X}_{noPred}$, $\mathtt{Rewrite}(Q_1 \cup Q_2){=}\ \mathtt{Rewrite}(Q_1) \cup \mathtt{Rewrite}(Q_2)$ remains correct since we have proven that our algorithm `Rewrite` is correct for any query of $\mathcal{X}_{noPred}$.

We have proven that our rewriting approach is correct for queries without predicates (i.e. $\mathcal{X}_{noPred}$). To generalize the proof for the whole fragment $\mathcal{X}$, we show in the following that our rewriting of $\mathcal{X}$ predicates, using the function `RW_Pred`, is correct.

Definition A.4. Given an access specification $S{=}(D,\mathtt{ann})$, an XML tree $T \in \mathcal{T}(D)$, and its authorized version T_A. An predicate rewriting function $\mathcal{R}$ is correct w.r.t S for a class of predicates $\mathcal{C}$ if and only if: for any predicate f of $\mathcal{C}$, any accessible node n_T of T and its corresponding node n_{T_A} of T_A, $\xi[\![[f]]\!](\{n_{T_A}\}){=}\xi[\![[\mathcal{R}(f)]]\!](\{n_T\})$. □

Recall that the same accessible node n is denoted by n_T and n_{T_A} in T and T_A respectively. A predicate f is correctly rewritten into $\mathcal{R}(f)$ if and only if either: *1)* $\xi[\![[f]]\!](\{n_{T_A}\}){=}\ \xi[\![[\mathcal{R}(f)]]\!](\{n_T\}){=}\varnothing$, or *2)* $\xi[\![[f]]\!](\{n_{T_A}\}){=}\{n_{T_A}\}$ and $\xi[\![[\mathcal{R}(f)]]\!](\{n_T\}){=}\{n_T\}$. Since $\{n_{T_A}\}$ and $\{n_T\}$ represent the same node n, then we abbreviate these two cases simply by $\xi[\![[f]]\!](\{n_{T_A}\}){=}\xi[\![[\mathcal{R}(f)]]\!](\{n_T\})$ as in Definition A.4.

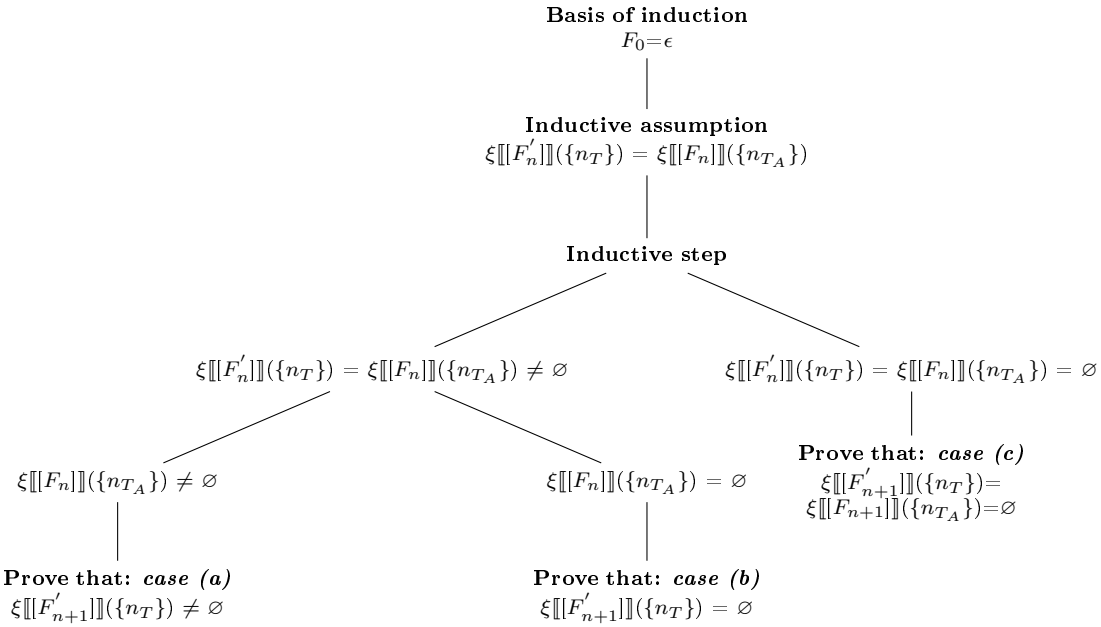


Figure A.1: Different cases to prove correctness of `RW_Pred` function.

Lemma A.3. For any predicate F of the fragment $\mathcal{X}$, the rewritten of F produced by the function `RW_Pred` is correct. □

Proof A.4. We consider first the case of simple filter F_n=$f_1/\ldots/f_n$ defined in $\mathcal{X}$ (where f_i=α_i::e_i, α_i is any axis of $\mathcal{X}$, and e_i can be $*$, *text()* function, or an element type). The other types of filters (i.e. negation, conjunction, disjunction, and text comparison) are discussed at the end of this proof. We have to prove that for any accessible node n_T of T and its corresponding node n_{T_A} in T_A, $\xi[\![[\texttt{RW\_Pred}(F_n)]]\!](\{n_T\})$=$\xi[\![[F_n]]\!](\{n_{T_A}\})$. Let us prove this by induction.

Basis of induction: We consider the simple empty predicate F_0=ϵ that is evaluated to false at any node. Thus, $\xi[\![[\texttt{RW\_Pred}(\epsilon)]]\!](\{n_T\})$=$\xi[\![[\epsilon]]\!](\{n_T\})$=$\xi[\![[\epsilon]]\!](\{n_{T_A}\})$=$\varnothing$.

Inductive assumption: We assume that our rewriting function `RW_Pred` is correct for an $\mathcal{X}$ predicate of size n, i.e. $\xi[\![[\texttt{RW\_Pred}(F_n)]]\!](\{n_T\})$=$\xi[\![[F_n]]\!](\{n_{T_A}\})$.

Inductive step: We have to show that $\xi[\![[\texttt{RW\_Pred}(F_{n+1})]]\!](\{n_T\})$=$\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})$. Based on the assumption, $\xi[\![[F'_n]]\!](\{n_T\})$=$\xi[\![[F_n]]\!](\{n_{T_A}\})$[48]. If this equality is satisfied then either the two previous evaluations (defined over n_T and n_{T_A} resp.) return an empty set of nodes or refer to the same node (i.e. a node n that is denoted n_T in T and n_{T_A} in T_A). Therefore, according to this assumption, we got three different cases to prove (***a***, ***b***, and ***c***) as depicted in Figure **??**. In the following, we prove for each cases that: $\xi[\![[\texttt{RW\_Pred}(F_{n+1})]]\!](\{n_T\})$=$\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})$.

A) Assume that $\xi[\![[F'_n]]\!](\{n_T\}) = \xi[\![[F_n]]\!](\{n_{T_A}\}) \neq \varnothing$, and $\xi[\![[F_n]]\!](\{n_{T_A}\}) \neq \varnothing$:
We have to show that $\xi[\![[F'_{n+1}]]\!](\{n_T\})$=$\{n_T\}$. Note that F_{n+1}=F_n/f_{n+1}, and

[48] We denote by F'_n the rewriting of F_n, i.e. F'_n=$\texttt{Rewrite}(F_n)$.

$\mathtt{RW\_Pred}(F_n)=f_1'[f_2'[\ldots[f_n']\ldots]]$ (where $f_i'=\mathtt{RW\_Pred}(f_i)$). According to the semantics of XPath (see Table 3.1), $\xi[\![[F_n]]\!](\{n_{T_A}\})=\{n_{T_A}\}$ implies that $\mathcal{S}[\![F_n]\!](\{n_{T_A}\}) \neq \varnothing$. In other words, for the subtree rooted at n_{T_A} in T_A, there are at least n accessible descendants of n_{T_A}, $m_{T_A}^1,\ldots,m_{T_A}^n$ such that: $n_{T_A}\ \alpha_1\ m_{T_A}^1$ and $m_{T_A}^i\ \alpha_i\ m_{T_A}^{i+1}$ $(1 \leq i < n)$, moreover, each $m_{T_A}^i$ is referred to by the sub-predicate f_i of F_n. Since n_{T_A} is the corresponding node of n_T in T_A (exactly the same node), then n_T has also n accessible descendants in T, $m_T^1,\ldots,m_T^n$ such that: $n_T\ \alpha_1\ m_T^1$, $m_T^i\ \alpha_i\ m_T^{i+1}$ $(1 \leq i < n)$, and each m_T^i is referred to by the sub-predicate f_i' of F_n'. Given the above, we show in the following that if $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})=\{n_{T_A}\}$ then $\xi[\![[F_{n+1}']]\!](\{n_T\})=\{n_T\}$ for the different cases of f_{n+1}.

Case of $\downarrow::e_{n+1}$*:* According to Property 3.1, $\xi[\![[F_n/f_{n+1}]]\!](\{n_{T_A}\})=\xi[\![[f_{n+1}]]\!](\mathcal{S}[\![F_n]\!](\{n_{T_A}\}))$, this means that f_{n+1} is evaluated over all the nodes returned by F_n. We have shown that the predicate $[F_n]$ is valid at n_{T_A} due to the existence of at least some path $n_{T_A}, m_{T_A}^1,\ldots,m_{T_A}^n$ that satisfies F_n at n_{T_A}. This means that $\mathcal{S}[\![F_n]\!](\{n_{T_A}\})$ returns at least the node $m_{T_A}^n$ referred to by f_n. Hence, $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})=\xi[\![[f_{n+1}]]\!](\xi[\![[F_n]]\!](\{n_{T_A}\}))=\xi[\![[f_{n+1}]]\!](\{\ldots,m_{T_A}^n,\ldots\})$. According to the assumption of case (**A**), $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})=\{n_{T_A}\}$, i.e. $\xi[\![[f_{n+1}]]\!](\{\ldots,m_{T_A}^n,\ldots\})=\{n_{T_A}\}$ which means that there is at least one node amongst $\{\ldots,m_{T_A}^n,\ldots\}$ that satisfies the predicate $[f_{n+1}]$, let $m_{T_A}^n$ be this node (i.e. the predicate $[\downarrow::e_{n+1}]$ is valid at $m_{T_A}^n$). The predicate $\downarrow::e_{n+1}$ over the node $m_{T_A}^n$ returns all its accessible immediate children of type e_{n+1}. Since $[f_{n+1}]$ is valid at $m_{T_A}^n$, we conclude that there is at least one accessible node $m_{T_A}^{n+1}$ of type e_{n+1} referred to by f_{n+1} that is immediate child of $m_{T_A}^n$ in T_A.

According to the assumption of case (**A**), $\xi[\![[F_n']]\!](\{n_T\})=\{n_T\}$. We have shown that there is at least a path $n_T, m_T^1,\ldots,m_T^n$ that satisfies F_n' at n_T. Since the node $m_{T_A}^n$ (the corresponding node of m_T^n) has an accessible child $m_{T_A}^{n+1}$, then m_T^n in T (referred to by f_n') must have the same node m_T^{n+1}. However, (i) m_T^{n+1} may appear in T either as immediate child of m_T^n or a descendant that is separated from m_T^n with only inaccessible nodes, in this way hiding these inaccessible nodes in T_A make $m_{T_A}^{n+1}$ appear as immediate child of $m_{T_A}^n$. Since $m_{T_A}^{n+1}$ of T_A is referred to by f_{n+1} of F_{n+1}, we should show that f_{n+1}' of F_{n+1}' refers to the same node m_T^{n+1}. We have $\mathtt{RW\_Pred}(F_{n+1})=f_1'[f_2'[\ldots[f_n'[f_{n+1}']]\ldots]]$, then f_{n+1}' (the rewritten version of f_{n+1}) is evaluated over each node returned by f_n'. As f_n refers to the node $m_{T_A}^n$, we know that f_n' refers to the node m_T^n in T. The query $\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]$ over m_T^n returns all the accessible descendants of m_T^n that are of type e_{n+1}, from (i) we conclude that $m_T^{n+1} \in \mathcal{S}[\![\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]]\!](m_T^n)$. Let R be the set of nodes resulted from the evaluation $\mathcal{S}[\![\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]]\!](m_T^n)$. The predicate $\uparrow^+::*[\mathcal{A}^{acc}]$ over the nodes of R returns for each node all its accessible ancestors, the parent is the first accessible ancestor returned. Then: $\uparrow^+::*[\mathcal{A}^{acc}][1]$ returns for each node of R, its parent node. Since $m_T^{n+1} \in R$, then (ii) the parent node of m_T^{n+1} belongs to $\mathcal{S}[\![\uparrow^+::*[\mathcal{A}^{acc}][1]]\!](R)$. Note that $[\mathtt{RW\_Pred}(f_{n+1})]=[\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]/\uparrow^+::*[\mathcal{A}^{acc}][1]=\varepsilon::*]$. We show in the following that the predicate $[\mathtt{RW\_Pred}(f_{n+1})]$ is valid at m_T^n. Since $m_{T_A}^{n+1}$ appears as immediate child of $m_{T_A}^n$ in T_A, then m_T^n is the parent node of m_T^{n+1} in T. The predicate $[\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]/\uparrow^+::*[\mathcal{A}^{acc}][1]=\varepsilon::*]$ is valid at m_T^n if there is an accessible descendant of m_T^n that has as the first accessible ancestor the node m_T^n (i.e. an accessible descendant that appears as immediate child of $m_{T_A}^n$ in T_A). From (i) and (ii), it is clear that the node m_T^{n+1}, referred to by $\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]$ and having m_T^n as parent node, satisfies the predicate $[\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]/\uparrow^+::*[\mathcal{A}^{acc}][1]=\varepsilon::*]$ at m_T^n. Given the rewritten predicate $[\mathtt{RW\_Pred}(F_{n+1})]=[f_1'[f_2'[\ldots[f_n'[f_{n+1}']]\ldots]]]$, we shown that there is at least one node m_T^n returned by f_n' for which the predicate $[f_{n+1}']$ is valid. Moreover, according to the inductive assumption, the predicate $[\mathtt{RW\_Pred}(F_n)]=[f_1'[f_2'[\ldots[f_n']\ldots]]]$ is valid. Thus, we conclude that

the predicate $[\texttt{RW\_Pred}(F_{n+1})]$ is valid at n_T, i.e. $\xi[\![[\texttt{RW\_Pred}(F_{n+1})]]\!](\{n_T\})=\{n_T\}$.

Case of $\downarrow^+::e_{n+1}$*:* By assuming that $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})=\{n_{T_A}\}$, we have to show that $\xi[\![[\texttt{RW\_Pred}(F_{n+1})]]\!](\{n_T\})=\{n_T\}$. The assumption $\xi[\![[F_n/f_{n+1}]]\!](\{n_{T_A}\})=\{n_{T_A}\}$ implies that $\xi[\![[f_{n+1}]]\!](\mathcal{S}[\![F_n]\!](\{n_{T_A}\}))= \xi[\![[f_{n+1}]]\!](\{\ldots,m^n_{T_A},\ldots\})=\{n_{T_A}\}$. The predicate $[f_{n+1}]$ (i.e. $[\downarrow^+::e_{n+1}]$) is valid over the set $\{\ldots,m^n_{T_A},\ldots\}$ implies that at least one node amongst the nodes $\{\ldots,m^n_{T_A},\ldots\}$ referred to by f_n has an accessible descendant $m^{n+1}_{T_A}$ of type e_{n+1}, let $m^n_{T_A}$ be this node. We should show that, in T, the predicate $[f'_{n+1}]$ over the node m^n_T of f'_n returns the same node m^{n+1}_T, i.e. whether $m^{n+1}_T \in \mathcal{S}[\![\texttt{RW\_Pred}(F_{n+1})]\!](\{n_T\})$. The rewritten version of F_{n+1} is given by: $\texttt{RW\_Pred}(F_{n+1})=f'_1[f'_2[\ldots[f'_n[f'_{n+1}]]\ldots]]$ where $f'_{n+1}=\downarrow^+::e_{n+1}[\mathcal{A}^{acc}]$. Since $m^n_{T_A}$ is referred to by f_n then the corresponding node m^n_T in T is referred to by f'_n of $\texttt{RW\_Pred}(F_{n+1})$. As $m^n_{T_A}$ in T_A has an accessible descendant $m^{n+1}_{T_A}$, then the corresponding node m^n_T of $m^n_{T_A}$ in T must have an accessible descendant m^{n+1}_T. According to the Lemma A.1, our accessibility predicate is correct and returns all and only accessible nodes. Thus, $\mathcal{S}[\![\downarrow^*::e_{n+1}[\mathcal{A}^{acc}]]\!](\{m^n_T\})$ returns all the accessible descendants of m^n_T, including m^{n+1}_T. We conclude that $[f'_{n+1}]=[\downarrow^*::e_{n+1}[\mathcal{A}^{acc}]]$ is valid over each node m^n_T of T returned by f'_n, and then: $\xi[\![[\texttt{RW\_Pred}(F_{n+1})]]\!](\{n_T\})=\{n_T\}$. Note that the case of $\downarrow^*::e_{n+1}$ can be done in a similar way as $\downarrow^+::e_{n+1}$.

Case of $\varepsilon::e_{n+1}$*:* $F_{n+1}=F_n/\varepsilon::e_{n+1}$, and $\texttt{RW\_Pred}(F_{n+1})=f'_1[f'_2[\ldots[f'_n[\varepsilon::e_{n+1}]]\ldots]]$. Notice that F'_{n+1} is computed simply by adding the predicate $[\varepsilon::e_{n+1}]$ into F'_n. We define the following cases:

- $e_n=e_{n+1}$ or $e_{n+1}=*$: It is clear that for each node m of type e_n referred to by f_n (resp. f'_n), the predicate $[f_{n+1}]$ given by $[\varepsilon::e_n]$ or $[\varepsilon::*]$ is valid at m. Then, $F_n[f_{n+1}]$ refers to the same nodes as F_n (resp. $F'_n[f_{n+1}]$ refers to the same set of F'_n). Then: $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})=\xi[\![[F_n]]\!](\{n_{T_A}\})=\{n_{T_A}\}$ and $\xi[\![[F'_{n+1}]]\!](\{n_T\})=\xi[\![[F'_n]]\!](\{n_T\})=\{n_T\}$.

- $e_n = *$ and $e_{n+1} \neq *$: The nodes returned by f_n of F_n in T_A (resp. those returned by f'_n of F'_n in T) may be of any type since $e_n=*$. According to the case (**A**), $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})=\xi[\![[f_{n+1}]]\!](\mathcal{S}[\![F_n]\!](\{n_{T_A}\}))= \xi[\![[f_{n+1}]]\!](\{\ldots,m^n_{T_A},\ldots\})=\{n_{T_A}\}$. This means that there is at least one node amongst $\{\ldots,m^n_{T_A},\ldots\}$ that is of type e_{n+1}, let $m^n_{T_A}$ be this node. According to the case (**A**), $\xi[\![[F'_n]]\!](\{n_T\})= \xi[\![[f'_1[\ldots[f'_n]\ldots]]]\!](\{n_T\})=\{n_T\}$. Since the node $m^n_{T_A}$ referred to by f_n is the corresponding node of m^n_T in T, then f'_n of F'_n refers to the same node m^n_T of type e_{n+1} and the predicate $[\varepsilon::e_{n+1}]$ is valid over f'_n, i.e. $\xi[\![[f'_1[\ldots[f'_n[\varepsilon::e_{n+1}]]\ldots]]]\!](\{n_T\})=\{n_T\}$. We conclude that $\mathcal{S}[\![F'_{n+1}]\!](\{n_T\})=\{n_T\}$.

- $e_n \neq *$, $e_{n+1} \neq *$, and $e_n \neq e_{n+1}$: The nodes returned by f_n of F_n in T_A and those returned by f'_n of F'_n in T are of type e_n. It is clear then that adding a simple predicate $[\varepsilon::e_{n+1}]$ to f_n (resp. f'_n) makes the predicate $[F_{n+1}]$ (resp. $[F'_{n+1}]$) false, since we try to select nodes of type e_{n+1} from a set of nodes of type e_n (i.e. nodes returned by f_n and f'_n). We conclude that $\mathcal{S}[\![F_{n+1}]\!](\{n_{T_A}\})=\mathcal{S}[\![F'_{n+1}]\!](\{n_T\})=\varnothing$.

Case of $\alpha_i::text()=c$*:* Recall that a text node is accessible if and only if its parent node is accessible. The rewriting of the text comparison predicate $[\alpha_i::text()=c]$, according to the axis α_i,

is given by $[\alpha_i{::}*[\mathcal{A}^{acc}]/\mathit{text()}{=}c]$ for $\alpha_i \in \{\downarrow^*, \downarrow^+\}$, and $[\alpha_i{::}\mathit{text()}{=}c]$ for $\alpha_i \in \{\varepsilon, \downarrow\}$. Based on the assumption of case (**A**) and for $f_{n+1}{=}\alpha_i{::}\mathit{text()}{=}c$, we should show that: $\xi[\![[F'_{n+1}]]\!](\{n_T\}){=}\{n_T\}$.

- $\alpha_i{=}\downarrow^*$ or $\alpha_i{=}\downarrow^+$: We consider only the case of $\downarrow^*$ (the remaining case is similar). We have $\xi[\![[F_n]]\!](\{n_{T_A}\}){=}\xi[\![[F'_n]]\!](\{n_T\}){=}\xi[\![[f'_1[\ldots[f'_n]\ldots]]]\!](\{n_T\}) \neq \varnothing$. We have seen that there is at least a node $m^n_{T_A}$ referred to by the sub-predicate f_n of F_n, respectively, a node m^n_T referred to by the sub-predicate f'_n of F'_n. Then the predicate $[f_{n+1}]$ (resp. $[f'_{n+1}]$) is evaluated directly over the nodes referred to by f_n (resp. f'_n), i.e. the predicate $[\downarrow^*{::}\mathit{text()}{=}c]$ is evaluated over $m^n_{T_A}$. According to the assumption of case (**A**), $\xi[\![[F_n/f_{n+1}]]\!](\{n_{T_A}\}){=}$ $\xi[\![[f_{n+1}]]\!](\mathcal{S}[\![F_n]\!](\{n_{T_A}\})){=}$ $\xi[\![[f_{n+1}]]\!](\{\ldots, m^n_{T_A}, \ldots\}){=}\{n_{T_A}\}$. This means that there is at least one node amongst $\{\ldots, m^n_{T_A}, \ldots\}$ that satisfies the predicate $[f_{n+1}]$, let $m^n_{T_A}$ be this node, i.e. $m^n_{T_A}$ has an accessible descendant node, let be $m^{n+1}_{T_A}$, that has a text node with value c. On the other hand, (i) the corresponding node m^n_T of $m^n_{T_A}$ in T must have the same node m^{n+1}_T. According to Lemma A.1, our accessibility predicate is correct, then the predicate $[\downarrow^*{::}*[\mathcal{A}^{acc}]]$ over m^n_T returns all and only accessible descendants of m^n_T in T. From (i) we conclude that m^{n+1}_T is referred to by $[\downarrow^*{::}*[\mathcal{A}^{acc}]]$, checking whether m^{n+1}_T has a text content with value c is done with the predicate $[\downarrow^*{::}*[\mathcal{A}^{acc}]/\mathit{text()}{=}c]$ that has to select, for each node m^{n+1}_T, all its accessible descendants having a text content of value c. Finally, $m^{n+1}_T \in \xi[\![[\downarrow^*{::}*[\mathcal{A}^{acc}]/\mathit{text()}{=}c]]\!](\{m^n_T\})$, this means that the predicate $[\mathtt{RW\_Pred}(\downarrow^*{::}\mathit{text()}{=}c)]$ is valid over a node m^n_T referred to by f'_n of F'_n, and then: $[F'_{n+1}]$ is valid at n_T, i.e. $\xi[\![[F'_{n+1}]]\!](\{n_T\}){=}\{n_T\}$.

- $\alpha_i{=}\downarrow$ or $\alpha_i{=}\varepsilon$: We consider only the case of $\downarrow$ (the second one is similar). According to the assumption of case (**A**), $\xi[\![[F_n/f_{n+1}]]\!](\{n_{T_A}\}){=}$ $\xi[\![[f_{n+1}]]\!](\mathcal{S}[\![F_n]\!](\{n_{T_A}\})){=}$ $\xi[\![[\downarrow{::}\mathit{text()}{=}c]]\!](\{\ldots, m^n_{T_A}, \ldots\}){=}\{n_{T_A}\}$. This means that there is at least one node amongst $\{\ldots, m^n_{T_A}, \ldots\}$ that has a text node of value c. By taking $m^n_{T_A}$ to be this node, then m^n_T in T must also have a text node with value c. This means that $[\downarrow{::}\mathit{text()}{=}c]$ is valid over the node m^n_T referred to by f'_n of F'_n, i.e. $[f'_1[\ldots[f'_n[\downarrow{::}\mathit{text()}{=}c]]\ldots]]$ is valid at n_T. We conclude that: $\xi[\![[F'_{n+1}]]\!](\{n_T\}){=}\{n_T\}$.

Finally, for all the cases of f_{n+1}, we have shown that if $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\}){=}\{n_{T_A}\}$, then $\xi[\![[F'_{n+1}]]\!](\{n_T\}){=}\{n_T\}$ which means that our rewriting function `RW_Pred` is correct for the case (**A**).

B) Assume that $\xi[\![[F'_n]]\!](\{n_T\}) = \xi[\![[F_n]]\!](\{n_{T_A}\}) \neq \varnothing$, and $\xi[\![[F_n]]\!](\{n_{T_A}\}) = \varnothing$:
We assume that the two predicate $[F_n]$ and $[F'_n]$ are valid over the nodes n_T and n_{T_A} respectively, and that (i) the predicate $[F_{n+1}]$ is not valid over n_{T_A} (i.e. $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\}){=}\varnothing$). To prove the correctness of our rewriting function `RW_Pred` in this case, it amounts to show that: $\xi[\![[F'_{n+1}]]\!](\{n_{T_A}\}){=}\varnothing$. The proof should be done for all the cases of f_{n+1} as we have done in the previous case (**A**). We show here the correctness for only the case of $f_{n+1}{=}\downarrow^+{::}e_{n+1}$ while the other cases can be done in a similar way. Taking f_{n+1} to be $\downarrow^+{::}e_{n+1}$ and assume by contradiction that (ii): $\xi[\![[F'_{n+1}]]\!](\{n_{T_A}\}) \neq \varnothing$. Note that $\mathtt{RW\_Pred}(f_{n+1}){=}\downarrow^*{::}e_{n+1}[\mathcal{A}^{acc}]$. According to the assumption (ii), we have $\xi[\![[F'_{n+1}]]\!](\{n_{T_A}\}){=}$ $\xi[\![[f'_1[\ldots[f'_n[f'_{n+1}]]\ldots]]]\!](\{n_{T_A}\})$ $\neq \{n_{T_A}\}$. This means that there is at least one node m^n_T referred to by f'_n for which the predicate $[f'_{n+1}]$ is valid, i.e. m^n_T has an accessible descendant node of type e_{n+1}, let m^{n+1}_T be this node. Moreover, m^{n+1}_T appears in T_A since it is accessible. Thus, the corresponding node

$m^n_{T_A}$ of m^n_T has the same node m^{n+1}_T denoted by $m^{n+1}_{T_A}$. According to the assumption of case (**B**), $[F_n]$ is valid at n_{T_A}, i.e. $\xi[\![[f_1/\ldots/f_n]]\!](\{n_{T_A}\}) \neq \varnothing$. Since the node m^n_T is referred to by $f^{'}_n$ in $F^{'}_n$ then its corresponding node $m^n_{T_A}$ is referred to by f_n in F_n. Moreover, we have seen that $m^n_{T_A}$ has an accessible descendant $m^{n+1}_{T_A}$ that is of type e_{n+1}. Given the above, the predicate $[f_{n+1}]=[\downarrow^*::e_{n+1}]$ is valid at the node $m^n_{T_A}$. Since the predicate $[F_n]$ is valid at n_{T_A} and $[f_{n+1}]$ is valid over at least one node returned by $[F_n]$, then we conclude that the predicate $[F_{n+1}]=[F_n/f_{n+1}]$ is valid at n_{T_A}. Thus, $\xi[\![[f_1/\ldots/f_n/\downarrow^*::e_{n+1}]]\!](\{n_{T_A}\}) \neq \varnothing$ which contradicts the assumption that we define in case (**B**) (i.e. assumption (i)). We conclude finally that the assumption (ii) that we have taken by contradiction is false, and then: $\xi[\![[F^{'}_{n+1}]]\!](\{n_{T_A}\})=\varnothing$.

C) Assume that $\xi[\![[F^{'}_n]]\!](\{n_T\}) = \xi[\![[F_n]]\!](\{n_{T_A}\}) = \varnothing$:
We have $\mathcal{S}[\![F_n]\!](\{n_{T_A}\})=\varnothing$, and $\xi[\![[F_n/f_{n+1}]]\!](\{n_{T_A}\})=\xi[\![[f_{n+1}]]\!](\mathcal{S}[\![F_n]\!](\{n_{T_A}\}))=\xi[\![[f_{n+1}]]\!](\varnothing)=\varnothing$. On the other hand, $\xi[\![[F^{'}_n]]\!](\{n_T\})=\xi[\![[f^{'}_1[\ldots[f^{'}_n]\ldots]]]\!](\{n_T\})=\varnothing$. This means that there is no node referred to by the sub-predicate $f^{'}_n$ of $F^{'}_n$. It is clear that adding any sub-predicate $f^{'}_{n+1}$ to $f^{'}_n$ does not change anything, i.e. $\xi[\![[f^{'}_1[\ldots[f^{'}_n[f^{'}_{n+1}]]\ldots]]]\!](\{n_T\})=\varnothing$ since we try to evaluate a predicate $f^{'}_{n+1}$ over an empty set of nodes returned by $f^{'}_n$. We conclude that: $\xi[\![[F_{n+1}]]\!](\{n_{T_A}\})=\xi[\![[F^{'}_{n+1}]]\!](\{n_T\})=\varnothing$.

$\because \quad \therefore \quad \therefore \quad \therefore \quad \because$

Summing up, we have proven that for any simple predicate F_{n+1} of size $n+1$, the evaluation of $[F_{n+1}]$ over the node n_{T_A} of T_A returns the same result as the evaluation of the rewritten predicate $[F^{'}_{n+1}]$ over the corresponding node n_T in T, i.e. either both the two predicates $[F^{'}_{n+1}]$ and $[F^{'}_{n+1}]$ are valid over n_T and n_{T_A} respectively, or the two predicates are false over these two nodes. In other words, for any simple predicate F of $\mathcal{X}$ of any size: $\xi[\![[F]]\!](\{n_{T_A}\})=\xi[\![[\mathtt{RW\_Pred}(F)]]\!](\{n_T\})$, which implies that our rewriting function `RW_Pred` is correct for simple predicates of $\mathcal{X}$. For the other types of predicates the proof is obvious. Consider two simple predicates F_1 and F_2, since the rewriting of these two predicates, using our function `RW_Pred`, is correct then the following predicates are correct too:

- $[\mathtt{RW\_Pred}(F_1) \wedge \mathtt{RW\_Pred}(F_2)]$ (equivalent to $[\mathtt{RW\_Pred}(F_1 \wedge F_2)]$).
- $[\mathtt{RW\_Pred}(F_1) \vee \mathtt{RW\_Pred}(F_2)]$ (equivalent to $[\mathtt{RW\_Pred}(F_1 \vee F_2)]$).
- $[\neg\ (\mathtt{RW\_Pred}(F_1))]$ (equivalent to $[\mathtt{RW\_Pred}(\neg\ F_1)]$).

We conclude finally that our rewriting function `RW_Pred` is correct for all the predicates of the fragment $\mathcal{X}$. □

Theorem A.1. The query rewriting algorithm `Rewrite` is correct for any query of the fragment $\mathcal{X}$. □

Proof A.5. The proof follows from Lemmas A.2 and A.3. Given an $\mathcal{X}$ query $Q_n=q_1/\ldots/q_n$ where: $q_i=axis_i::e_i[f_i]$ and f_i can be any filter of $\mathcal{X}$. Let $Q^{'}_n$ be a new version of Q_n where all filters are omitted, i.e. $Q^{'}_n=axis_1::e_1/\ldots/axis_n::e_n$. We have proven in Lemma A.2 that our rewriting function `Rewrite` is correct for any query of $\mathcal{X}_{noPred}$, i.e. $\mathcal{S}[\![\mathtt{Rewrite}(Q^{'}_n)]\!](T)=\mathcal{S}[\![Q^{'}_n]\!](T_A)$. Since each subquery $axis_i::e_i$ of $Q^{'}_n$ is correctly rewritten, using our function `Rewrite`, over the nodes returned by the sub-query $axis_{i-1}::e_{i-1}$, then for each sub-query $Q^{'}_i=axis_1::e_1/\ldots/axis_i::e_i$ of $Q^{'}_n$ $(i \leq n)$ we have: $\mathcal{S}[\![\mathtt{Rewrite}(Q^{'}_i)]\!](T)=\mathcal{S}[\![Q^{'}_i]\!](T_A)$. These two evaluations return the

same sets of nodes denoted R_T and R_{T_A} respectively. Lemma A.3 shows that for any accessible node n_T of T and its corresponding node n_{T_A} in T_A, and for any predicate $[f_i]$ of $\mathcal{X}$: $\xi[\![[\mathtt{RW\_Pred}(f_i)]]\!](\{n_T\})=\xi[\![[f_i]]\!](\{n_{T_A}\})$. This implies that for any node n_T of R_T and its corresponding node n_{T_A} of R_{T_A}, the previous equality is satisfied. More generally, $\xi[\![[\mathtt{RW\_Pred}(f_i)]]\!](R_T)=\xi[\![[f_i]]\!](R_{T_A})$. In other words, by substituting R_T and R_{T_A} with their values, we obtain:

- $\xi[\![[\mathtt{RW\_Pred}(f_i)]]\!](R_T)=\xi[\![[f_i]]\!](R_{T_A})$.
- $\xi[\![[\mathtt{RW\_Pred}(f_i)]]\!](R_T)=\xi[\![[\mathtt{RW\_Pred}(f_i)]]\!](\mathcal{S}[\![\mathtt{Rewrite}(Q_i')]\!](T))=$
  $\mathcal{S}[\![\mathtt{Rewrite}(Q_i')[\mathtt{RW\_Pred}(f_i)]]\!](T)=\mathtt{Rewrite}(Q_i)$.
- $\xi[\![[f_i]]\!](R_{T_A})=\xi[\![[f_i]]\!](\mathcal{S}[\![Q_i']\!](T_A))=\mathcal{S}[\![Q_i'[f_i]]\!](T_A)=\mathcal{S}[\![Q_i]\!](T_A)$.

Finally, we conclude that for any size of the query Q_n: $\mathcal{S}[\![\mathtt{Rewrite}(Q_n)]\!](T)=\mathcal{S}[\![Q_n]\!](T_A)$, thus our rewriting function $\mathtt{Rewrite}$ is correct for any query of the fragment $\mathcal{X}$. □

B

DTD Graph: algorithm and complexity

B.1 Construction of DTD Graph

1. $$\frac{\textit{Graph}\,(root \rightarrow Rg(root), \{\bigcup_{A \in Ele \setminus \{root\}} A \rightarrow Rg(A)\})}{(Ele, Rg, root)}$$

2. $$\frac{\begin{array}{l}\Big(\Sigma_\alpha \cup \{A\}, V_\alpha \cup \{v_A\}, E_\alpha \cup \{(v_A, v_\alpha)\}, \lambda_\alpha \cup \{v_A \xrightarrow{\lambda} A\}, v_A, Order_\alpha\Big); \\ where : \Big(\Sigma_\alpha, V_\alpha, E_\alpha, \lambda_\alpha, v_\alpha, Order_\alpha\Big) = \textit{Graph}\,(\alpha, P)\end{array}}{\textit{Graph}\,(A \rightarrow \alpha, P)}$$

3. $$\frac{\Big(\phi, \phi, \phi, \phi, Null, \phi\Big)}{\textit{Graph}\,(\epsilon, P)}$$

4. $$\frac{\Big(\{\texttt{str}\}, \{v_{str}\}, \phi, \{v_{str} \xrightarrow{\lambda} \texttt{str}\}, v_{str}, \phi\Big)}{\textit{Graph}\,(\texttt{str}, P)}$$

5. $$\frac{\textit{Graph}\,(B \rightarrow Rg(B), P \setminus \{B \rightarrow Rg(B)\})}{\textit{Graph}\,(B, P)\ with\ B \rightarrow Rg(B) \in P}$$

6. $$\frac{\Big(\Sigma_B, V_B, E_B, \lambda_B, v_B, Order_B\Big)}{\textit{Graph}\,(B, P)\ with\ B \rightarrow Rg(B) \notin P}$$

7. $$\frac{\begin{array}{l}\Big(\{*\} \cup \Sigma_\alpha, \{v_f\} \cup V_\alpha, \{(v_f, v_\alpha)\} \cup E_\alpha, \{v_f \xrightarrow{\lambda} *\} \cup \lambda_\alpha, v_f, Order_\alpha\Big); \\ where : \Big(\Sigma_\alpha, V_\alpha, E_\alpha, \lambda_\alpha, v_\alpha, Order_\alpha\Big) = \textit{Graph}\,(\alpha, P)\end{array}}{\textit{Graph}\,(\alpha*, P)}$$

8. $$\frac{\begin{array}{l}\Big(\cup_i \Sigma_{\alpha_i} \cup \{\odot\}, \cup_i V_{\alpha_i} \cup \{v_f\}, \cup_i E_{\alpha_i} \cup \{(v_f, v_{\alpha_1}), ..., (v_f, v_{\alpha_n})\}, \\ \cup_i \lambda_{\alpha_i} \cup \{v_f \xrightarrow{\lambda} \odot\}, v_f, \cup_i Order_{\alpha_i} \cup \{Order_{v_f}(1, v_f) = v_{\alpha_1}, ..., Order_{v_f}(n, v_f) = v_{\alpha_n}\}\Big); \\ where : \Big(\Sigma_{\alpha_i}, V_{\alpha_i}, E_{\alpha_i}, \lambda_{\alpha_i}, v_{\alpha_i}, Order_{\alpha_i}\Big) = \textit{Graph}\,(\alpha_i, P)\end{array}}{\textit{Graph}\,(\alpha_1, ..., \alpha_n, P)}$$

Figure B.1: The inductive steps in DTD graph construction.

We shall now show that for every DTD, a DTD graph can be constructed that represents all the constraints imposed by this DTD over some element types. Our DTD graph construction algorithm, termed "`Graph`", is presented in Figure B.1 as recursive inference rules. The construction is a structural induction on the expressions of the DTD (i.e., DTD productions). Given a DTD $D = (Ele, Rg, root)$, for a production $A \rightarrow Rg(A)$, algorithm `Graph` constructs first a graph rooted at A, which represents all the constraints defined between A and its children types. After, for each child type B of A, a graph is created for the production $B \rightarrow Rg(B)$. The same process is done with children types of B and their children until all descendant types of A are parsed. In this case, the final graph is the graph of A.

We describe in a nutshell the different rules of DTD graph construction shown in Figure B.1. With v_t we refer to the unique vertex labeled with type t ($t \in Ele \cup \{\texttt{str}\}$). Moreover, each time we use v_f, we refer by it to a new vertex created to represent an operator of the DTD.

Algorithm `Graph` has two parameters: a) the current production; and b) the remaining productions not already parsed (denoted P). Consider the production $A \rightarrow \alpha$ (rule (2)), we start by creating a vertex v_A (if it does not exist) representing element type A. After, the vertex root v_α of the subgraph G_α is connected to v_A to complete the description of A element type. The subgraph G_α is created as follows.

• For $\alpha = \epsilon$ (rule (3)), G_α is an empty graph $(\phi, \phi, \phi, \phi, Null, \phi)$ [49].
• For $\alpha = \texttt{str}$ (rule (4)), a graph is created with only one vertex v_{str} which is connected with v_A with the edge (v_A, v_{str}).
• For $\alpha = B$, two cases are considered. If the production $B \rightarrow Rg(B)$ is not traversed (rule (5)), then a graph is created (rooted at vertex v_B) that represents the expression $Rg(B)$. This graph is added as a subgraph of A's graph by connecting v_B into v_A with edge (v_A, v_B). In the other case, the graph of $B \rightarrow Rg(B)$ has already been constructed (i.e., the graph returned at rule (6)), then only the edge (v_A, v_B) remains to be added to complete the graph of A.

• For the expression $\alpha*$ (rule (7)), let $G_\alpha = (\Sigma_\alpha, V_\alpha, E_\alpha, \lambda_\alpha, v_\alpha, Order_\alpha)$ be the subgraph representing subexpression α. The graph of A is constructed in this case by creating a new vertex v_f (labeled with $*$), connecting it first with the vertex v_A by the edge (v_A, v_f), and with the root vertex v_α of G_α by (v_f, v_α).

• For the expression $\alpha_1, ..., \alpha_n$ (rule (8)), let $G_{\alpha_1}, ..., G_{\alpha_n}$ be the subgraphs representing subexpressions $\alpha_1, ..., \alpha_n$ respectively. The graph of A is created as follows: create a new vertex v_f labeled with $\odot$, connect v_f with v_A by the edge (v_A, v_f), and for each subgraph G_{α_i}, connect v_f with v_{α_i} by the edge (v_f, v_{α_i}). By defining the function $Order_{v_f}$ for the vertex v_f, the edges $(v_f, v_{\alpha_1}), ..., (v_f, v_{\alpha_n})$ are uniquely ordered according to the order defined between subexpressions α_i in $Rg(A)$.

Finally, rule (1) represents the star of the construction process. By invoking the algorithm with the production $root \rightarrow Rg(root)$, all the expressions of the DTD D are parsed and the graph created is the DTD graph of D.

Note that the DTD productions $A \rightarrow \alpha?$ and $A \rightarrow \alpha_1 + ... + \alpha_n$ (omitted from Figure B.1) are handled with the same principle as $A \rightarrow \alpha*$ and $A \rightarrow \alpha_1, ..., \alpha_n$ respectively.

Property B.1. Given a DTD D, its corresponding DTD graph G_D can be constructed at most in $O(|D|)$ time. □

[49] Note that an edge (v_i, v_j)=$Null$ if one of its vertex is $Null$. We have, $E \cup \{Null\}$=E.

B.1.1 DTD Graph Construction Algorithm

Our algorithm *DTDGraph* is shown in Figure B.2 that, for a given DTD $D = (Ele, Rg, root)$, computes its corresponding DTD graph G_D. We start first by creating a node n_t for each type t in D. This creation step is done only one time for the types of D. However, a node is created in G_D for each occurrence of operator "," (resp. "+", "∗", or "?") in expressions of D. Given a node n_α and an expression α, procedure *subGraph* creates a subgraph rooted at n_α and which represents the structure of expression α. The subgraph created for an element type A is called "*A element graph*", composed by all the types that can be reached from A [STZ$^+$ 7]. The algorithm *DTDGraph* traverses all the expressions of DTD D by invoking the procedure *subGraph* with node n_{root} (representing the *root* type) and the expression $Rg(root)$. In a nutshell, the procedure *subGraph*(n_α, α) proceeds as follows. The two simple cases are either when $\alpha = \epsilon$, in this case node n_α has no child; or when $\alpha =$ `str`, in this case the subgraph rooted at n_α has only one child labeled `str` and reached by the arc $n_\alpha n_{str}$. If α is an element type A, then we create an arc $n_\alpha n_A$ to connect node n_α with node n_A representing element type A. After, the A element graph is created by calling *subGraph*$(n_A, Rg(A))$. We associate boolean variable *visited*$[A]$ (initially $false$) with each element type A in D. With this variable, we ensure that each element type of D is processed only once. If $\alpha = \beta*$ (resp. $\alpha = \beta?$), then we create a new node n_* labeled with $*$, whose children (computed with *subGraph*(n_*, β)) represent the structure of expression β. After, the nodes n_α and n_* are connected with the arc $n_\alpha n_*$. Until now, node n_α has at most one child (either n_{str}, n_A, n_*, $n_?$, or no child). Thus, the list $L_{G_D}(n_\alpha)$ of its incident nodes is empty or contains only one node. The benefit of this list is shown clearly when $\alpha = \alpha_1, ..., \alpha_k$ (resp. $\alpha = \alpha_1 + ... + \alpha_k$). In this case, a new node $n_\odot$ is created with label $\odot$ and connected to node n_α with arc $n_\alpha n_\odot$ (i.e., $L_{G_D}(n_\alpha) = [n_\odot]$). After, for each subexpression α_i, a subgraph is created with root node n_{α_i} (lines 18-20). The label of this node, equals to *label*(α_i), is determined according to α_i as follows: *label*(`str`)=`str`, *label*$(A)=A$, *label*$(\beta*)=*$, *label*$(\beta?)=?$, *label*$(\beta_1, ..., \beta_k)=\odot$, and *label*$(\beta_1 + ... + \beta_k)=\oplus$. Finally, the node $n_\odot$ is connected to each node n_{α_i} by arc $n_\odot n_{\alpha_i}$. The order between these subexpressions α_i is preserved in DTD graph G_D by defining the list $L_{G_D}(n_\odot) = [n_{\alpha_1}, ..., n_{\alpha_k}]$. This order is important especially when computing a DTD D from its corresponding DTD graph G_D as explained in the next.

Property B.2. Given a DTD D, algorithm *DTDGraph* computes its corresponding DTD graph G_D in $O(|D|)$ time. □

Algorithm: *DTDGraph*

input : A DTD $D = (Ele, Rg, root)$.
output: A DTD graph $G_D = (N, A, L, lab_N, n_{root})$.

```
1  (N, A, L) ← (∅, ∅, ∅);
   /* In the following, n_t is the node created for type t                    */
2  foreach type t ∈ Ele ∪ {str} do
3  |   n_t ← newNode(t);
4  |   if (t = root) then
5  |   |   n_root ← n_t;
6  subGraph(n_root, Rg(root));
7  foreach node n ∈ N do
8  |   L ← L ∪ {L_GD(n)};
9  G_D ← (N, A, L, lab_N, n_root);
10 return G_D;
```

Procedure: *subGraph*

input: A node n_α and a regular expression α.

```
1  case α = str
2  |   A ← {n_α n_str};
3  |   L_GD(n_α) ← [n_str];
4  case α = B                                   /* case of element type B */
5  |   A ← {n_α n_B};
6  |   L_GD(n_α) ← [n_B];
7  |   if not (visited[B]) then
8  |   |   subGraph(n_B, Rg(B));
9  |   |   visited[B] ← true;
10 case α = β*
11 |   n_* ← newNode(*);
12 |   L_GD(n_α) ← [n_*];
13 |   subGraph(n_*, β);
14 case α = β?
   |   /* same principle as previous case                                     */
15 case α = α_1, ..., α_k
16 |   n_⊙ ← newNode(⊙);
17 |   L_GD(n_α) ← [n_⊙];
18 |   foreach subexpression α_i do
19 |   |   n_αi ← newNode(label(α_i));
20 |   |   subGraph(n_αi, α_i);
21 |   L_GD(n_⊙) ← [n_α1, ..., n_αk];
22 case α = α_1 + ... + α_k
   |   /* same principle as previous case                                     */
```

Procedure: *newNode*

input : A label l.
Output: A node n labeled l.

```
1  Create a new node n_l;
2  N ← N ∪ {n_l};
3  lab_N(n_l) ← l;
4  L_GD(n_l) ← [];
5  return n_l;
```

Figure B.2: DTD Graph Construction.

Bibliography

[ACC04] Stefan Andrei, Wei-Ngan Chin, and Salvador Valerio Cavadini. Self-embedded context-free grammars with regular counterparts. *Acta Inf.*, 40(5):349–365, 2004.

[BBC+00] Elisa Bertino, M. Braun, Silvana Castano, Elena Ferrari, and Marco Mesiti. Author-x: A java-based system for xml data protection. In *Data and Application Security, Development and Directions, IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security*, volume 201 of *IFIP Conference Proceedings*, pages 15–26. Kluwer, 2000.

[BBC+10] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. Xml path language (xpath) 2.0 (second edition). *W3C Recommendation. Available at: http://www.w3.org/TR/2010/REC-xpath20-20101214/.*, December 2010.

[BCCGY93] Nora Boulahia-Cuppens, Frédéric Cuppens, Alban Gabillon, and Kioumars Yazdanian. Multilevel security in object-oriented databases. In *Security for Object-Oriented Systems, Proceedings of the OOPSLA-93 Conference Workshop on Security for Object-Oriented Systems*, Workshops in Computing, pages 79–89. Springer, 1993.

[BCF07] Loreto Bravo, James Cheney, and Irini Fundulaki. Repairing inconsistent xml write-access control policies. In *Database Programming Languages, 11th International Symposium, DBPL 2007, Vienna, Austria, September 23-24, 2007, Revised Selected Papers*, Lecture Notes in Computer Science, pages 97–111. Springer, 2007.

[BCFF+10] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. Xquery 1.0: An xml query language (second edition). *Available at: http://www.w3.org/TR/xquery/.*, December 2010.

[BCFS12] Loreto Bravo, James Cheney, Irini Fundulaki, and Ricardo Segovia. Consistency and repair for xml write-access control policies. *VLDB J.*, 21(6):843–867, 2012.

[BCG+11] Iovka Boneva, Anne-Cécile Caron, Benoît Groz, Yves Roos, Sophie Tison, and Slawek Staworko. View update translation for xml. In *Database Theory - ICDT 2011, 14th International Conference*, pages 42–53. ACM, 2011.

[Bet08] Albert D. Bethke. Representing procedural logic in xml. *JSW*, 3(2):33–40, 2008.

[BF02] Elisa Bertino and Elena Ferrari. Secure and selective dissemination of xml documents. *In TISSEC*, 5(3):290–331, 2002.

[BGMS11] Angela Bonifati, Martin Hugh Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of xml views. In *14th International Conference on Extending Database Technology (EDBT)*, pages 177–188. ACM, 2011.

[BHJ13] Stefan Böttcher, Rita Hartel, and Thomas Jacobs. Fast multi-update operations on compressed xml data. In *Big Data - 29th British National Conference on Databases (BNCOD)*, volume 7968 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2013.

[BIO] Biopolymer markup language. `http://xml.coverpages.org/BIOML-XML-DTD.txt`. June 2013.

[BJS93] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Access control in object-oriented database systems - some approaches and issues. In *Advanced Database Systems*, volume 759 of *Lecture Notes in Computer Science*, pages 17–44. Springer, 1993.

[BJS99] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. A flexible authorization mechanism for relational data management systems. *ACM Trans. Inf. Syst.*, 17(2):101–140, 1999.

[BK08] Michael Benedikt and Christoph Koch. Xpath leashed. *ACM Comput. Surv.*, 41(1), 2008.

[BLS06] Denilson Barbosa, Gregory Leighton, and Andrew Smith. Efficient incremental validation of xml documents after composite updates. In *Database and XML Technologies, 4th International XML Database Symposium (XSym)*, volume 4156 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.

[BMKL] Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. Toxgene: An extensible template-based data generator for xml. *In WebDB 2002, pp. 49-54.*

[Bou10] Ronald Bourret. *XML Database Products.* `http://www.rpbourret.com/xml/XMLAndDatabases.htm`, June 2010.

[BP08] Mikolaj Bojanczyk and Pawel Parys. Xpath evaluation in linear time. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2008)*, pages 241–250. ACM, 2008.

[BP11] Mikolaj Bojanczyk and Pawel Parys. Xpath evaluation in linear time. *J. ACM*, 58(4):17, 2011.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (xml) 1.1 (second edition). w3c recommendation. *Available at: http://www.w3.org/TR/2006/REC-xml11-20060816/.*, 2006.

[BPV04] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[BS03] Stefan Böttcher and Rita Steinmetz. A dtd graph based xpath query subsumption test. *In: Database and XML Technologies, First International XML Database Symposium (XSym)*, pages 85–99. Springer, 2003.

[CD99] James Clark and Steve DeRose. Xml path language (xpath) 1.0. *W3C Recommendation. Available at: http://www.w3.org/TR/xpath/.*, November 1999.

[Cho02] Byron Choi. What are real dtds like? In *Fifth International Workshop on the Web and Databases (WebDB)*, pages 43–48, 2002.

[CIR11] Asma Cherif, Abdessamad Imine, and Michaël Rusinowitch. Optimistic access control for distributed collaborative editors. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, pages 861–868, 2011.

[Con07] Gao Cong. Query and update through xml views. In *Databases in Networked Information Systems, 5th International Workshop, (DNIS)*, volume 4777 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 2007.

[CRK+13] Jianneng Cao, Fang-Yu Rao, Mehmet Kuzu, Elisa Bertino, and Murat Kantarcioglu. Efficient tree pattern queries on encrypted xml documents. In *Joint 2013 EDBT/ICDT Conferences, Workshop Proceedings*, pages 111–120. ACM, 2013.

[DdVPS02] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. A fine-grained access control system for xml documents. *ACM Trans. Inf. Syst. Secur.*, 5(2):169–202, 2002.

[DFGM08] Ernesto Damiani, Majirus Fansi, Alban Gabillon, and Stefania Marrara. A general approach to securely querying xml. *Computer Standards & Interfaces*, 30(6):379–389, 2008.

[DZ08a] Maggie Duong and Yanchun Zhang. Dynamic labelling scheme for xml data processing. In *On the Move to Meaningful Internet Systems, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Proceedings, Part II*, volume 5332 of *Lecture Notes in Computer Science*, pages 1183–1199. Springer, 2008.

[DZ08b] Maggie Duong and Yanchun Zhang. An integrated access control for securely querying and updating xml data. In *Proceedings of the Nineteenth Australasian Database Conference (ADC)*, volume 75 of *CRPIT*, pages 75–83. Australian Computer Society, 2008.

[FCG04a] Wenfei Fan, Chee Yong Chan, and Minos N. Garofalakis. Secure xml querying with security views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 587–598. ACM, 2004.

[FCG04b] Wenfei Fan, Chee Yong Chan, and Minos N. Garofalakis. Secure xml querying with security views. In *SIGMOD Conference*, pages 587–598, 2004.

[Feg11] Leonidas Fegaras. Incremental maintenance of materialized xml views. In *Database and Expert Systems Applications - 22nd International Conference (DEXA 2011)*, volume 6861 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2011.

[FGJK06] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Smoqe: A system for providing secure access to xml. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1227–1230. ACM, 2006.

[FGJK07] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Rewriting regular xpath queries on xml views. In *ICDE*, pages 666–675. IEEE, 2007.

[FGK03] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. Xl: an xml programming language for web service specification and composition. *Computer Networks*, 42(5):641–660, 2003.

[FGMP13] Massimo Franceschet, Donatella Gubiani, Angelo Montanari, and Carla Piazza. A graph-theoretic approach to map conceptual designs to xml schemas. *ACM Trans. Database Syst.*, 38(1):6, 2013.

[FM04] Irini Fundulaki and Maarten Marx. Specifying access control policies for xml documents with xpath. In *SACMAT 2004, 9th ACM Symposium on Access Control Models and Technologies*, pages 61–69. ACM, 2004.

[FM07] Irini Fundulaki and Sebastian Maneth. Formalizing xml access control for update operations. In *SACMAT*, pages 169–174. ACM, 2007.

[FYL+09] Wenfei Fan, Jeffrey Xu Yu, Jianzhong Li, Bolin Ding, and Lu Qin. Query translation from xpath to sql in the presence of recursive dtds. *VLDB J.*, 18(4):857–883, 2009.

[GB01] Alban Gabillon and Emmanuel Bruno. Regulating access to xml documents. In *Database and Application Security XV, IFIP TC11/WG11.3 Fifteenth Annual Working Conference on Database and Application Security, July 15-18, 2001, Niagara on the Lake, Ontario, Canada*, volume 215 of *IFIP Conference Proceedings*, pages 299–314. Kluwer, 2001.

[GCV11] Haris Georgiadis, Minas Charalambides, and Vasilis Vassalos. A query optimization assistant for xpath. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT 2011)*. ACM, 2011.

[Ged] Genealogy markup language. `http://xml.coverpages.org/gedml-dtd9808.txt`. June 2013.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 95–106. Morgan Kaufmann, 2002.

[GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of xpath query evaluation. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems(PODS 2003)*, pages 179–190. ACM, 2003.

[GKP05] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[GKPS05a] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The complexity of xpath query evaluation and xml typing. *J. ACM*, 52(2):284–335, 2005.

[GKPS05b] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The complexity of xpath query evaluation and xml typing. *J. ACM*, 52(2):284–335, 2005.

[GKPS05c] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The complexity of xpath query evaluation and xml typing. *J. ACM*, 52(2):284–335, 2005.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[GMRR01] Ashish Gupta, Inderpal Singh Mumick, Jun Rao, and Kenneth A. Ross. Adapting materialized views after redefinitions: techniques and a performance study. *Inf. Syst.*, 26(5):323–362, 2001.

[GSC+09] Benoît Groz, Slawomir Staworko, Anne-Cécile Caron, Yves Roos, and Sophie Tison. Xml security views revisited. In *Database Programming Languages - DBPL 2009, 12th International Symposium*, volume 5708 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2009.

[GY03] Jonathan L. Gross and Jay Yellen. *Handbook of Graph Theory*. CRC Press, 2003.

[HK] Satoshi Hada and Michiharu Kudo. Xml access control language: Provisional authorization for xml documents. `http://www.research.ibm.com/trl/projects/xml/xacl/xacl-spec.html`. October 2000.

[HL13a] Wen-Chiao Hsu and I-En Liao. Cis-x: A compacted indexing scheme for efficient query evaluation of xml documents. *Inf. Sci.*, 241:195–211, 2013.

[HL13b] Wen-Chiao Hsu and I-En Liao. Cis-x: A compacted indexing scheme for efficient query evaluation of xml documents. *Inf. Sci.*, 241:195–211, 2013.

[JB99] Xiaoyi Jiang and Horst Bunke. Optimal quadratic-time isomorphism of ordered graphs. *Pattern Recognition*, 32(7):1273–1283, 1999.

[Jia07] Xibei Jia. *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.

[KA08] Assadarat Khurat and Joerg Abendroth. A mechanism for requesting hierarchical documetns in xacml. In *IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 202–207. IEEE, 2008.

[KCKN04] Rajasekar Krishnamurthy, Venkatesan T. Chakaravarthy, Raghav Kaushik, and Jeffrey F. Naughton. 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)*, pages 42–53. IEEE Computer Society, 2004.

[KMR05] Gabriel M. Kuper, Fabio Massacci, and Nataliya Rassadko. Generalized xml security views. In *SACMAT*, pages 77–84. ACM, 2005.

[KMR09] Gabriel M. Kuper, Fabio Massacci, and Nataliya Rassadko. Generalized xml security views. *Int. J. Inf. Sec.*, 8(3):173–203, 2009.

[LCS12] Zhen Hua Liu, Hui J. Chang, and Balasubramanyam Sthanikam. Efficient support of xquery update facility in xml enabled rdbms. In *ICDE*, pages 1394–1404. IEEE Computer Society, 2012.

[LDS+90] Teresa F. Lunt, Dorothy E. Denning, Roger R. Schell, Mark Heckman, and William R. Shockley. The seaview security model. *IEEE Trans. Software Eng.*, 16(6):593–607, 1990.

[LLHY13] Jixue Liu, Chengfei Liu, Theo Härder, and Jeffrey Xu Yu. Update xml views. *CoRR*, abs/1302.1923, 2013.

[LLLL11] Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu. Qfilter: rewriting insecure xml queries to secure ones using non-deterministic finite automata. *VLDB J.*, 20(3):397–415, 2011.

[LSAF05] Shiyong Lu, Yezhou Sun, Mustafa Atay, and Farshad Fotouhi. On the consistency of xml dtds. *Data Knowl. Eng.*, 52(2):231–247, 2005.

[Mar04] Maarten Marx. Xpath with conditional axis relations. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology*, volume 2992 of *Lecture Notes in Computer Science*, pages 477–494. Springer, 2004.

[MI12a] Houari Mahfoud and Abdessamad Imine. *A General Approach for Securely Querying and Updating XML Data.* INRIA Research Report. `http://hal.inria.fr/hal-00664975`, 2012.

[MI12b] Houari Mahfoud and Abdessamad Imine. A general approach for securely updating xml data. In *Proceedings of the 15th International Workshop on the Web and Databases (WebDB 2012)*, pages 55–60, 2012.

[MI12c] Houari Mahfoud and Abdessamad Imine. On securely manipulating xml data. In *Foundations and Practice of Security - 5th International Symposium (FPS 2012), Revised Selected Papers*, volume 7743 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 2012.

[MI12d] Houari Mahfoud and Abdessamad Imine. On securely manipulating xml data. In *Journées Bases de Données Avancées (BDA 2012)*, 2012.

[MI12e] Houari Mahfoud and Abdessamad Imine. Secure querying of recursive xml views: a standard xpath-based technique. In *WWW (Companion Volume)*, pages 575–576. ACM, 2012.

[MIR13] Houari Mahfoud, Abdessamad Imine, and Michaël Rusinowitch. Svmax: a system for secure and valid manipulation of xml data. In *Proceedings of the 17th International Database Engineering & Applications Symposium (IDEAS)*, pages 154–161. ACM, 2013.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[MN10a] Sebastian Maneth and Kim Nguyen. Xpath whole query optimization. *PVLDB*, 3(1):882–893, 2010.

[MN10b] Sebastian Maneth and Kim Nguyen. Xpath whole query optimization. *PVLDB*, 3(1):882–893, 2010.

[MNSB06] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of xml schema. *ACM Trans. Database Syst.*, 31(3):770–813, 2006.

[MSW05] Sriram Mohan, Arijit Sengupta, and Yuqing Wu. Access control for xml: a dynamic query rewriting approach. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management*, pages 251–252. ACM, 2005.

[MTKH03] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. Xml access control using static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 73–84. ACM, 2003.

[MTKH06] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. Xml access control using static analysis. *ACM Trans. Inf. Syst. Secur.*, 9(3):292–324, 2006.

[Nic12] Anisoara Nica. Incremental maintenance of materialized views with outerjoins. *Inf. Syst.*, 37(5):430–442, 2012.

[NS06] Frank Neven and Thomas Schwentick. On the complexity of xpath containment in the presence of disjunction, dtds, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[OAu] An open protocol to allow secure authorization (oauth). `http://oauth.net/`. June 2013.

[PV00] Yannis Papakonstantinou and Victor Vianu. Dtd inference for views of xml data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 35–46. ACM, 2000.

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

[Ras06] Nataliya Rassadko. Policy classes and query rewriting algorithm for xml security views. In *Data and Applications Security XX, 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec)*, volume 4127 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2006.

[RCD+08] Jonathan Robie, Don Chamberlin, Michael Dyck, Daniela Florescu, Jim Melton, and Jérôme Siméon. Extensible markup language (xml) 1.0 (fifth edition). w3c recommendation. *Available at: `http://www.w3.org/TR/2008/REC-xml-20081126/`.*, 2008.

[RCD+11] Jonathan Robie, Don Chamberlin, Michael Dyck, Daniela Florescu, Jim Melton, and Jérôme Siméon. Xquery update facility 1.0. *W3C Recommendation. Available at: `http://www.w3.org/TR/xquery-update-10/`.*, March 2011.

[SAM] Oasis security assertion markup language (saml) tc. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`. June 2013.

[SBG10] Slawek Staworko, Iovka Boneva, and Benoît Groz. The view update problem for xml. In *Proceedings of the 2010 EDBT/ICDT Workshops*, ACM International Conference Proceeding Series. ACM, 2010.

[SdV00] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design, Tutorial Lectures [revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design (FOSAD 2000)]*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer, 2000.

[SF02a] Andrei Stoica and Csilla Farkas. Secure xml views. In *Research Directions in Data and Applications Security, IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security*, volume 256 of *IFIP Conference Proceedings*, pages 133–146. Kluwer, 2002.

[SF02b] Andrei Stoica and Csilla Farkas. Secure xml views. In *DBSec*, volume 256 of *IFIP Conference Proceedings*, pages 133–146. Kluwer, 2002.

[Sha12] Professor Dr. Narasimha-Moorthy Shastry. *Integrated Healthcare IHE Pathway for the Patients: Patient Treatment Lifecycle Management (PTLM). Radiology Clinic, United Kingdom (2000)*, October 2012. `http://www.clinrad.nhs.uk/`.

[ST90] Paul D. Stachour and Bhavani M. Thuraisingham. Design of ldv: A multilevel secure relational database management system. *IEEE Trans. Knowl. Data Eng.*, 2(2):190–209, 1990.

[STZ+ 7] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. *In: Proceedings of 25th International Conference on Very Large Data Bases (VLDB)*, pages 302–314, 1999. ISBN 1-55860-615-7.

[tC06] Balder ten Cate. 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)*, pages 328–337. ACM, 2006.

[tCL09] Balder ten Cate and Carsten Lutz. The complexity of query containment in expressive fragments of xpath 2.0. *J. ACM*, 56(6), 2009.

[Tho84] Wolfgang Thomas. Logical aspects in the study of tree languages. In *CAAP*, pages 31–50, 1984.

[TKCW09] Stephen Tully, George Kantor, Howie Choset, and Felix Werner. A multi-hypothesis topological slam approach for loop closing on edge-ordered graphs. *In: Proceeding of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4943–4948, IEEE, 2009.

[TTL13] Manogna Thimma, Tsam Kai Tsui, and Bo Luo. Hyxac: a hybrid approach for xml access control. In *18th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 2013.

[UMU] Umu-xacml-editor. http://sourceforge.net/projects/umu-xacmleditor/. June 3013.

[VBH06] André Prisco Vargas, Vanessa P. Braganholo, and Carlos A. Heuser. Conflict resolution in updates through xml views. In *Current Trends in Database Technology - EDBT 2006, EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Revised Selected Papers*, volume 4254 of *Lecture Notes in Computer Science*, pages 192–205. Springer, 2006.

[VHP] Roel Vercammen, Jan Hidders, and Jan Paredaens. Query translation for xpath-based security views. *In EDBT 2006, pp. 250-263.*

[Wei11] Andreas M. Weiner. *Cost-Based XQuery Optimization in Native XML Database Systems.* PhD thesis, 2011.

[WKD04] Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 2004.

[Woo03] Peter T. Wood. Containment for xpath fragments under dtd constraints. In *Database Theory - ICDT 2003, 9th International Conference*, volume 2572 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2003.

[WTWS13] Xiaoying Wu, Dimitri Theodoratos, Wendy Hui Wang, and Timos Sellis. Optimizing xml queries: Bitmapped materialized views vs. indexes. *Inf. Syst.*, 38(6):863–884, 2013.

[XACa] Oasis extensible access control markup language (xacml) tc. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml. Version 3.0, January 3013.

[XACb] Xacml editor 6.x. https://jira.duraspace.org/browse/ISLANDORA/component/10402. June 3013.

[Xia12] Xiaofeng Xia. A conflict detection approach for xacml policies on hierarchical resources. In *IEEE International Conference on Green Computing and Communications (GreenCom)*, pages 755–760. IEEE, 2012.

[XML] W3c xml schema recommendation. http://www.w3.org/XML/Schema. Version 1.1.

[ZLC+12] Xiaojian Zhao, Huan-Bo Luan, Junjie Cai, Jin Yuan, Xiaoming Chen, and Zhoujun Li. Personalized video recommendation based on viewing history with the study on youtube. In *The 4th International Conference on Internet Multimedia Computing and Service (ICIMCS)*, pages 161–165. ACM, 2012.

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

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

Résumé. Durant ces dernières années, plusieurs travaux ont proposé des modèles de contrôle d'accès pour sécuriser l'accès en lecture aux données XML, basés seulement sur des DTDs non-récursives. Le contrôle d'accès XML considérant les opérations de mise à jour n'a pas reçu suffisamment d'attention. Dans ce travail, nous présentons un modèle général pour spécifier le contrôle d'accès aux données XML moyennant des primitives de mise à jour du *W3C XQuery Update Facility*. Notre approche pour enforcer ces spécifications de mise à jour est basée sur la notion de réécriture des requêtes (*query rewriting* en anglais). Cependant, la réécriture des requêtes dans le cas des DTDs récursives reste un probleme ouvert. Pour pallier à ce problème, nous proposons un algorithme linéaire, basé seulement sur le standard XPath, et qui permet de réécrire chaque opération de mise à jour, définie par rapport à une DTD arbitraire (récursive ou non), en une autre opération sûre afin qu'elle soit évaluée seulement sur des données XML modifiables par l'utilisateur qui a soumis l'opération. Ce travail représente le premier effort autour de la sécurisation des opérations de mise à jour XML dans la présence des DTDs arbitraires, une classe importante d'opérations de mise à jour, et un fragment riche de XPath. Nous étudions dans le reste de ce travail l'intéraction entre les droits de lecture et de mise à jour, et nous décrivons une solution qui permet de préserver la confidentialité et l'intégrité des données XML.

Mots clés: Réécriture des requêtes, Contôle d'accès XML, Mise à jour XML, Vues de sécurité XML, XPath, XQuery, Confidentialité et Intégrité.