PhD Dissertation

International Doctorate School in Information and Communication Technologies

DIT - University of Trento

GENERALIZED XML SECURITY VIEWS

Nataliya Rassadko

Advisor:
Prof. Gabriel Mark Kuper
Università degli Studi di Trento

Co-Advisor:
Prof. Fabio Massacci
Università degli Studi di Trento

February 2008
Abstract

We study the problem of efficient fine-grained access control for XML documents. Our model is similar to the multilevel security model adopted for relational databases. Namely, for every user with a certain clearance level, we provide a schema of all and only available data. This schema can be used by the user for query formulation.

Due to hierarchical structure of XML, hardly can a view be expressed via a single XPath query. Therefore, we construct a DTD view by propagating security specification from annotated elements to non-annotated ones. Our security annotations are specified in an expressive XPath fragment and may include conditions on DTD structure, XML data values and user properties. The propagation algorithm enforces security policy on the DTD and emulates propagation of the same security labels in XML document. Along with the DTD view construction, we calculate also a σ-function that represents hidden information.

User query answering may be conducted in two ways. In the first way, we materialize an XML view that conforms to the DTD view and is constructed from the initial XML by filtering out fragments matched by the σ-function. In the second way, the σ-function is incorporated into the user query so that the rewritten query can be answered by evaluation over the initial XML. The answer set is the same as if the initial query were evaluated over the materialized XML view.

We prove the correctness of algorithms for DTD view construction, XML
view materialization and query rewriting. We also discuss their time- and space- complexity that is confirmed in our experiments.

Keywords
[XML, XPath, security view, query rewriting, policy enforcement, access control]
Contents

1 Introduction 1
  1.1 The Problem .............................................. 2
  1.2 Our Contribution ......................................... 5
  1.3 Dissertation Outline .................................... 8

2 XML Security Views 11
  2.1 A Motivating Example ................................. 11
  2.2 XML and XPath ........................................... 18

3 Document Access Control 23

4 Schema Access Control 31
  4.1 Schema Access Control for Top-Down Policies .......... 31
  4.2 Security Views for Top-Down Policies ................. 33
  4.3 View Construction for Top-Down Policies ............ 42
  4.4 Schema-Level Access for Bottom-Up Policies .......... 48
  4.5 Theoretical Results .................................... 55

5 Query Rewriting Algorithm 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Theoretical Results</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>Implementation</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>Architecture</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>Class diagrams</td>
<td>88</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Core DTD</td>
<td>89</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Secure DTD</td>
<td>91</td>
</tr>
<tr>
<td>6.2.3</td>
<td>XML</td>
<td>98</td>
</tr>
<tr>
<td>6.2.4</td>
<td>XPath class diagram</td>
<td>102</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Query rewriting</td>
<td>104</td>
</tr>
<tr>
<td>6.3</td>
<td>Workflow</td>
<td>104</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Sequence diagram</td>
<td>104</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Invocation implementation</td>
<td>105</td>
</tr>
<tr>
<td>7</td>
<td>Experimental results</td>
<td>109</td>
</tr>
<tr>
<td>7.1</td>
<td>View Construction</td>
<td></td>
</tr>
<tr>
<td>7.1.1</td>
<td>Scalability</td>
<td>109</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Policy enforcement performance: schema-based vs. instance-based</td>
<td>113</td>
</tr>
<tr>
<td>7.2</td>
<td>Query Rewriting</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Discussion</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>State of the Art</td>
<td>125</td>
</tr>
<tr>
<td>8.1</td>
<td>Run-time policy</td>
<td></td>
</tr>
<tr>
<td>8.1.1</td>
<td>Provisional authorization: Kudo et al.</td>
<td>126</td>
</tr>
<tr>
<td>8.1.2</td>
<td>A rule-based XML access control model</td>
<td>127</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Client-based access control for XML documents</td>
<td>129</td>
</tr>
</tbody>
</table>
8.1.4 XML access control with policy matching tree . . . 131
8.1.5 RDF Metadata for XML Access Control . . . . . . 132
8.1.6 Role-based access control to XML databases . . . . 133

8.2 Combining access control rules
into user query . . . . . . . . . . . . . . . . . . . . . . . . . 134
8.2.1 Static analysis: Murata et al. . . . . . . . . . . . . 134
8.2.2 QFilter . . . . . . . . . . . . . . . . . . . . . . . . . 135
8.2.3 Optimizing the secure evaluation of twig queries . . 137

8.3 Structural locality
of access control . . . . . . . . . . . . . . . . . . . . . . . . 140
8.3.1 A compressed accessibility map for XML documents 140
8.3.2 Integrated CAM for XML . . . . . . . . . . . . . . . . 141
8.3.3 Document Order Labeling . . . . . . . . . . . . . . . 142

8.4 XML views . . . . . . . . . . . . . . . . . . . . . . . . . . 144
8.4.1 Author-XML . . . . . . . . . . . . . . . . . . . . . . 144
8.4.2 Fine-grained access control to XML documents: Dami-ani et al. . . . . . . . . . . . . . . . . . . . 147
8.4.3 Regulating Access to XML Documents: Gabillon et al. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
8.4.4 Access Control on XML Relationships . . . . . . 152

8.5 Cryptographically enforced
access control . . . . . . . . . . . . . . . . . . . . . . . . . 153
8.5.1 XML Pool Encryption . . . . . . . . . . . . . . . . . 153
8.5.2 Secure and selective dissemination of XML documents154
8.5.3 Conditional access for XML documents . . . . . . 155
8.5.4 Hierarchical and Role-based Access Control for XML Documents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
8.5.5 Securing XML documents with one private key . . 159
8.6 Relational Access
   Control for XML ........................................ 159
   8.6.1 XENA .............................................. 159
   8.6.2 XML access control using relational databases 160
8.7 DTD-based views ...................................... 161
   8.7.1 Secure XML views ................................. 161
   8.7.2 Secure XML querying .............................. 164
   8.7.3 A Dynamic Query Rewriting Approach .......... 167
8.8 Discussion ............................................. 168
   8.8.1 Policy .............................................. 168
   8.8.2 Model and query answering ...................... 168
8.9 Other XML access
   control support ......................................... 170
   8.9.1 Derived access control specification for XML .. 170
   8.9.2 Secure XML publishing in the presence of data inference ........................................ 171

9 Conclusion ........................................... 175

Bibliography ............................................ 179
List of Tables

3.1 Policy alternatives ........................................... 26
8.1 Existing policy frameworks ................................. 169
8.2 Existing XML access control frameworks .............. 169
# List of Figures

1.1 Schema of accessible data materialization ........................................ 5

2.1 The graph representation of the DTD document $D$ .......................... 12
2.2 Partial annotation of the XML document conforming to the DTD from Example 1 ................................................................. 13
2.3 Authorized XML view for student Vladimir Romanov ................... 15
2.4 Security annotation for competing student ................................. 17
2.5 The semantics of operators ......................................................... 21

3.1 Algorithm POLICY CLASS ...................................................... 29

4.1 Security annotation for the applicant ......................................... 32
4.2 Security view for the applicant .................................................. 34
4.3 Authorized DTD view for student Vladimir Romanov by [40] ........ 34
4.4 Algorithm MATERIALIZE ....................................................... 41
4.5 Removing qualifier $\text{ann}(\text{applications, application}) = Q[\text{student-data/id = }$\text{login}] ......................................................... 44
4.6 Algorithm ANNOTATE VIEW .................................................... 45
4.7 Algorithm BUILD VIEW .............................................................. 47
4.8 Removing qualifier $\text{ann}(\text{recomm - letter, unreliable}) = Q[q_2]$ .... 51
4.9 Algorithm QUALIFIER REMOVING ........................................... 53
4.10 Algorithm SPLIT ....................................................................... 54
4.11 Algorithm ANNOTATE VIEW .................................................... 56
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Generic parse tree</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Algorithm QUERY REWRITE</td>
<td>66</td>
</tr>
<tr>
<td>5.3</td>
<td>Procedure getReverse$\sigma$</td>
<td>69</td>
</tr>
<tr>
<td>6.1</td>
<td>An overall architecture of the system</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>Package diagram of the system</td>
<td>88</td>
</tr>
<tr>
<td>6.3</td>
<td>Class diagram of Wutka DTD structure</td>
<td>90</td>
</tr>
<tr>
<td>6.4</td>
<td>Class diagram of secure DTD structure</td>
<td>92</td>
</tr>
<tr>
<td>6.5</td>
<td>Class diagram of SiXPath XPath</td>
<td>103</td>
</tr>
<tr>
<td>6.6</td>
<td>Sequence diagram of the system</td>
<td>105</td>
</tr>
<tr>
<td>7.1</td>
<td>DTD view construction degradation for Y/N labels</td>
<td>110</td>
</tr>
<tr>
<td>7.2</td>
<td>Scalability of DTD view construction algorithm</td>
<td>111</td>
</tr>
<tr>
<td>7.3</td>
<td>Schema of the file auctions.dtd</td>
<td>113</td>
</tr>
<tr>
<td>7.4</td>
<td>Schema-based vs instance-based materialization</td>
<td>115</td>
</tr>
<tr>
<td>7.5</td>
<td>Annotation for a registered user of an auctions portal</td>
<td>116</td>
</tr>
<tr>
<td>7.6</td>
<td>Set of queries</td>
<td>117</td>
</tr>
<tr>
<td>7.7</td>
<td>Set of rewritten queries</td>
<td>118</td>
</tr>
<tr>
<td>7.8</td>
<td>Query rewriting performance</td>
<td>119</td>
</tr>
<tr>
<td>7.9</td>
<td>Experiments on query evaluation</td>
<td>120</td>
</tr>
</tbody>
</table>
Acknowledgements

First of all I would like to thank my advisors Gabriel Kuper and Fabio Massacci for their priceless support, guidance and sponsorship throughout these years. Without their ideas, comments, and friendly assistance, this dissertation would not have been ever possible. They taught me to carry out the research, exemplifying the energy, curiosity and eagerness to achieve the goals. I am very grateful for this invaluable experience and the opportunity to work together.

I would also thank my colleagues at SAP Labs in Sophia-Antipolis, where I took an internship during my PhD study. Especially, I express my thanks to Luca Compagna who gave me a chance for this stage, Azeddine Benamer who supported me with new knowledge and his friendship, and all the great people of the French Sap Labs, for making the work really amazing, and for creating a very friendly, encouraging and fruitful environment.

Special thanks go to my reviewers, Elena Ferrari and Stefano Paraboschi, who have made a great effort to read the thesis and provided useful feedback.

With my deepest gratitude I would like to dedicate this work to my family: to my parents for everything they have given to me, for their belief and encouragement; to my husband Sebastiano for giving me inspiration, understanding, and love; and to my daughter Ilaria for an example of optimism and cheerfulness.
Chapter 1

Introduction

XML [98] has become the main standard for data representation and exchange on the Web. In light of the sensitive nature of many business applications, this raises the important issue of security in XML and the selective exposure of information to different classes of users based on their access privileges.

More specifically, for an XML database $T$ there may be a number of user groups who want to query the same document. For these user groups, different access policies may be imposed, specifying which elements of $T$ the users are allowed to access. The problem of XML security is to specify and enforce these access policies. More formally, given a query $Q$, the goal is to ensure that the evaluation of $Q$ over $T$ returns only information in $T$ that the user is allowed to access. This calls for access and inference control, i.e., protecting sensitive data from direct access or indirect inference through queries by unauthorized users.

To address such security issues, there is a need of a generic, flexible security model that can effectively support multiple policies for controlling access to XML content at various levels of granularity (e.g., restricting access to entire subtrees or specific elements in the document tree based on their content or location). Perhaps even more importantly, enforcing such
access-control policies should not drastically degrade either performance or functionality of the XML query-execution engine. In addition, access-control enforcement should not complicate maintenance of consistency and integrity of the data when either the XML data or the access policies are updated.

Furthermore, XML documents are typically accompanied by a DTD [98] or an XML Schema [102] that specifies the internal structure of the data. For the same reasons that a database schema is needed for query formulation and processing for traditional databases, XML schemas are also important for XML query formulation and optimization; in addition, schemas are critical for XML data exchange and integration [8]. Thus, access control should not inhibit schema availability, i.e., the availability of necessary schema information (e.g., DTDs) specifying the structure of the accessible data.

Putting these together, an XML security model must support:

1. a simple, powerful and fine grained authorization mechanism that can control access to both content and structure;
2. efficient mechanisms for the enforcement of security policies without full look up of the underlying document;
3. schema information, characterizing exactly those elements accessible to each type of user, in the same way that a relational database offers security views to their users.

1.1 The Problem

While specifications and enforcement of access control are well understood for traditional databases [37, 61, 72, 78], the study of security for XML is less established. Early security models that have been proposed for XML
do not meet criteria 3 above and, to a lesser extent, criteria 1 and 2. In particular, cryptographically enforced access control to XML documents [64, 21] considers only protection of XML data but not a schema; different parts of the XML tree are encrypted by different keys (typically one key for a particular combination of access control rules applicable to an XML element); and these kinds of protection make querying difficult since a large amount of decryption is needed. Run-time policy evaluation scenarios [56, 21] require policy propagation from the root to the requested node. This may result in making accessibility decisions for every node test of the user query. For some queries, this may be made more efficient in the presence of DTD [68, 28]. Other optimization techniques rely on a compressed accessibility map [88, 53, 89], or combining access control policy with user queries [62]. In these cases, the user still does not have a schema of the accessible data. This problem was slightly resolved in [20, 30], where a “loosened” variant (i.e., a DTD where every forbidden element is “optional”) of the schema is provided; however, the overall security policy enforcement is performed at the document level by fully annotating the entire XML document/database, which may require an expensive view materialization (see Chapter. 7), and complicates consistency and integrity maintenance.

In addition, fixing the access control policies at the instance level without providing or computing a schema makes it difficult for the security officer to understand how the authorized view of a document for a user or a class of users actually looks like. Hence, such a solution is hardly practical for large and complex documents. On the other hand, revelation of excessive schema information might lead to security breaches: an unauthorized user can deduce or infer confidential information via multiple queries and analyze the schema even if only accessible nodes are queried.

To overcome these limitations, the notion of an XML security view was
initially proposed by Stoica and Farkas [79] and later refined by Fan et al. [40]. The basic idea is to provide a schema that describes the data that can be seen by the user, as well as a (hidden) set of XPath expressions that describes how to derive the data in the view from the original data.

A typical approach to specifying and enforcing access control for traditional databases is to define views on which security permissions should be applied (e.g., multi-level security view for a relational database [61, 72, 78, 84], discretionary access control over relational databases [10] and object-oriented databases [22]). For instance, Lunt et al. [60] showed how to use standard SQL queries to implement the SeaView multi-level secure database. Defining a relational view via an SQL query is straightforward, and the relational schema of the view is immediate. In contrast, the hierarchical structure and the dependencies (e.g., ancestor and descendants) between nodes in XML data, as well as the presence of disjunction and recursion in DTDs make it impossible to define a security view via a single query, and introduces new challenges in how to generate a view that conforms to a view DTD. The challenges were observed in [8], which showed that even for XML views of relational data, it is nontrivial to ensure that the views typecheck. This observation has been further confirmed in [4, 3], which showed that type checking of XML views of relational data, even for simple DTDs, is computationally intractable: co-NEXPTIME for extremely restricted view definitions, and undecidable for realistic views.

The intuition how to construct XML security views was found in [72] which shows how to compute a full database labelling from a partial one given by security views. From the XML viewpoint, a partial assignment of security labels to XML document nodes can also be extended to a full assignment. From this, it is easy to compute an XML analogue of relational view by means of “sanitization” operation which hides (e.g., deletes or encrypts) nodes with negative authorizations, but reveals their permitted
children \([11, 30, 44, 56]\). The resulting XML tree is called \(\text{authorized} (T_A)\). We avoid \(T_A\) calculation by means of enforcing security policy directly on XML. Instead, we employ schema level enforcement, resulting in the DTD schema of permitted data (or in other words, DTD view \(D_v\)) like in \([79, 40]\). The \(\text{materialized}\) version of XML document \((T_M)\) is constructed from the initial XML document by deleting forbidden nodes with respect to the DTD view \(D_v\) so that \(T_A\) be isomorphic to \(T_M\).

A summary of how to construct a schema for the accessible data is shown in Fig. 1.1.

### 1.2 Our Contribution

Our first contribution is an investigation of different alternatives for policy definition and enforcement at the level of an XML tree (Chapter 3). Our analysis shows that not all combinations of policy options satisfy the properties of \(\text{completeness}\) and \(\text{consistency}\), i.e. some policy settings do not result in a unique fully annotated tree. We provide a classification of policies under different options for security label propagation and con-
flict resolution. The comparison of our policy framework with those of the existing proposals is done in Chapter 8.

Next, we generalize the notion of XML security views to arbitrary non-recursive DAG DTDs and to conditional constraints defined in an expressive XPath fragment. For each view, a security specification is a simple extension of the document DTD $D$ with security annotations and security policies used to automatically obtain a full annotation from a partial one. We note that the security views proposed in [79] in some sense are smarter than ours since they can preserve semantic associations among some XML tags. For this purpose, corresponding cover stories are selected by a security officer. However, neither full DTD labelling derivation nor querying is discussed in [79].

Our third contribution deals with algorithm for view derivation. Namely, we show a generic algorithm that constructs a fully annotated DTD $D_F$ (from the partial security specification) for different policies so that $D_F$ reflects a full annotation of a corresponding XML document. From this full specification, we derive a security view $S$ consisting of a view DTD $D_v$ and a function $\sigma$ defined via XPath queries. The view DTD $D_v$ shows only the data that is accessible according to the specification. The view is provided to the users so that they can formulate their queries over the view. The function $\sigma$ is withheld from the users, and is used to extract accessible data from the actual XML documents to populate a structure conforming to $D_v$. The formal proof of correctness of view derivation algorithm is also given.

Although similar results were obtained in [40], this thesis elaborates on certain issues left open. In particular, we consider general XML DTDs defined in terms of regular expressions rather than normalized DTDs. Furthermore, in contrast to [40], we do not allow dummy element types in the definition of security views. While this complicates the derivation of
security views, it prevents possible inference or deduction of sensitive information from dummy elements in the presence of external knowledge. In addition, their algorithm for view construction in [40] only applies for top-down policy propagation, while our approach both for top-down and for bottom-up policies. Next, Fan et al. [40] did not show how to construct fully annotated DTD $D_F$. The latter can be extensively exploited, for example, in run-time policy evaluation scenarios when $D_F$ is calculated for a requesting user once for session and all further accessibility checks are done on $D_F$.

Another issue, in [40], it was claimed that view materialization is not needed since it is time-consuming and can be avoided by use of query rewriting. However, in many applications, views can remain unchanged for a long period of time and, hence, can be queried directly without query rewriting. Moreover, query rewriting may introduce complicated qualifiers, and thus evaluation of them may lead to exponential response time. Guided by these observations, we provide an algorithm for view materialization. We also show that materialized XML views conform to $D_v$ and are isomorphic to those of [30, 17]. This is our fourth contribution.

On the other hand, we also developed a query rewriting algorithm, proved its correctness and evaluated its time complexity. The idea of query rewriting is to incorporate the $\sigma$-function into the user query so that the rewritten query can be answered by evaluation over the initial XML. The answer set is the same as if the initial query were evaluated over the materialized XML view. The novelty of our query rewriting algorithm is that it processes the reverse axes. For this purpose, we introduced the notion of the reversed $\sigma$-function and provided an algorithm for its calculation. We also improved the performance of query rewriting by using the divide-and-conquer strategy rather than dynamic programming used in [40].

We created a prototype of a system for enforcing XML access control
1.3. DISSERTATION OUTLINE

CHAPTER 1. INTRODUCTION

according to our proposal. It contains implementation of algorithms for DTD view construction, XML view materialization and query rewriting and is used for the experimental evaluation of our proposal, both for view materialization and query rewriting performance. Theoretical results are confirmed during our experiments.

1.3 Dissertation Outline

The rest of the thesis is organized in the following way.

Chapter 2 introduces basic concepts of XML, DTD, and XPath. Namely, we formalize the notion of DTD graph, XML tree conforming to DTD, and XPath semantics. In this chapter, we also describe our running example.

Chapter 3 is dedicated to policy classification. We consider different options of policy propagation and study properties of policy completeness and consistency. All reasoning is held on the XML level.

Chapter 4 elaborates top-down policy propagation at the DTD level. The idea is to emulate XML-based policy enforcement presented in the previous Chapter and to construct a DTD schema of available data. This chapter also presents a method of XML view materialization from the DTD view and proves that the materialized view is isomorphic to that calculated in the case of the direct XML-based policy enforcement. Finally, we extend the results to bottom-up policy propagation and present a generic algorithm for view derivation. In addition, the chapter provides a formal analysis of the developed algorithm.

Chapter 5 considers the case when XML views remain virtual and presents an algorithm for query rewriting. Algorithm correctness and complexity are formally analyzed and proved.
Chapter 6 describes issues related to the implementation of our system: architecture, class diagrams and workflows for both view materialization and query rewriting use cases.

Chapter 7 is dedicated to experimental evaluation of our proposal. First, it measures scalability of our DTD view construction algorithm. Then it compares the performance of earlier approaches to view materialization with our method. Next, we evaluate query rewriting performance for queries of different complexity. Finally, we compare evaluation of rewritten queries over the initial XML, original queries over the materialized XLM view, and original queries over the authorized XML.

Chapter 8 overviews related work and summarizes the advantages of our proposal over existing approaches to XML access control.

Chapter 9 concludes the thesis and draws the future work.
Chapter 2

XML Security Views

2.1 A Motivating Example

We start with a running example assuming an intuitive understanding of XML documents as trees and DTDs as DAGs.

Example 1: We describe a DTD a database containing the information on applications for PhD/MS program. Each application is initiated by a student described via student-data with an element id uniquely identifying the student and representing the student’s login name. Student-data is composed of name, desired degree (PhD or MS) department, and waiver. The latter field may take values “true” or “false”\(^1\) and means that student does (does not) waive his/her right to inspect the content of the recommendation letters. The application is supported by several letters of recommendation (recomm-letter); some of them can be classified as for letter body and is provided by a separate evaluator having name, title and institution attributes. The evaluator places comments on the applicant’s skills in free-text field, which is either a PDF or a TXT file, and rates applicant’s English proficiency, and possible contributions to PhD or MS program. Letters of recommendation are reviewed by the admission com-

\(^1\)A domain of some value, like waiver, may be restricted by means of ENTITY declaration of DTD and is not considered in the current thesis.
Figure 2.1: The graph representation of the DTD document $D$

mittee and are assigned to a category favorable or unfavorable depending on the context.

The corresponding DTD graph is depicted on Fig. 2.1, where solid lines represent concatenation (i.e., AND-relation between a node and its children), dashed lines represent disjunction (i.e., OR-relation), and combined line, called nesting represent mix of relations between a node and its child nodes (in Fig. 2.1, rating has child English and either MS or PhD). Stars on lines represent zero-to-many cardinality between a node and its child of a certain type. Fig. 2.2(a) shows an XML document conforming to this DTD, where boxed leaves of the tree represent text values. Gray labels on some edges of this XML will be explained later.

The need to provide users with a schema-level security view is illustrated by the access control requirements in Example 2.

**Example 2:** An applicant can access his/her personal record located under the field student-data. Access to fields favorable and unfavorable is forbidden, while the visibility of rating and free-text is possible if the waiver is false (data-dependent access). Moreover, the applicant should
CHAPTER 2. XML SECURITY VIEWS

2.1. A MOTIVATING EXAMPLE

Figure 2.2: Partial annotation of the XML document conforming to the DTD from Example 1

\( q_1, q_2 \equiv \text{student-data/id} = \$login \)
\( q_3, q_4 \equiv \text{..../student-data/id} = \$login; \)
\( q_5, \ldots, q_{13} \equiv \text{ancestor::application/student-data/id} = \$login[/waiver = "false"]; \)

(b) The meaning of qualifiers

not be aware of the reliability of the recommendation letters as the leakage of this information to recommenders might lead to diplomatic incidents.

Finally, we require the field department to be visible in any case: either from a proper student’s application or from external one. Thus, the student will be able to see how many applications are submitted to different departments of the university. Hence, a ranking of departments can be inferred.

Corresponding qualifiers are shown in Fig. 2.2(a) as gray labels on edges. The meaning of the labels is the following: \( \mathbf{Y} \) means always accessible,
2.1. A MOTIVATING EXAMPLE

N means always forbidden, while $q_1, \ldots, q_{13}$ are encoded in XPath and presented in Fig. 2.2(b). In these qualifiers, $\$login$ is a variable that is instantiated dynamically during the login into the system. So, if $\$login$ is “dkonovalov” then $q_1, q_3, q_5 - q_{10}$ are evaluated to true, $q_2, q_4, q_{11} - q_{13}$ are evaluated to false. If the value of $\$login$ is “vromanov” the situation is reverse.

Existing cryptographical proposals like [64], [20] as well as run-time policy evaluation scenarios [56], [21], [68], [28], [62] and view-based approaches [20], [30] enforce such security constraints directly on the XML document. The DTD is usually used only for typing of the XML document with security labels which are then propagated over the entire document through corresponding document portions. These systems specify how to restrict access at the data level and how to obtain authorized view of data.

Example 3: Let us construct an authorized view for Vladimir Romanov using some other well-known access control models.

- **Author-X by Bertino et al.** [11] does not support qualifiers, so we assume that a preprocessing step of their algorithm evaluating qualifier with the following semantics: if $q$ is evaluated to true, the label is $Y$ (i.e. permitted for access), otherwise $N$ (i.e. forbidden for access). In the case of Vladimir Romanov, $q_1, q_3, q_5, \ldots, q_{10}$ are evaluated to false, and $q_2, q_4, q_{11}, \ldots, q_{13}$ are evaluated to true. Next, we assume that the propagation option is **CASCADE** [11] (i.e., top-down to all subelements) and deny takes precedence in the case of conflicts. Moreover, the approach relies on denial downward consistency property according to which subtree rooted in forbidden node should be deleted. The the view for Vladimir Romanov is shown in Fig. 2.3(a). It contains only the student’s personal information. However, if Dmitry didn’t wave his right to see the content of recommendation letters, he would not
CHAPTER 2. XML SECURITY VIEWS 2.1. A MOTIVATING EXAMPLE

(a) Bertino et al. [11]  
(b) Damiani et al. [30]  
(c) Fan et al. [40]

Figure 2.3: Authorized XML view for student Vladimir Romanov

be able to access this information anyway because of denial downward consistency property. Selective dissemination [20] is a crypto-version of the Author-X which promotes a push architecture. The difference is that negative authorizations are not supported. However, everything that is not permitted is forbidden which is, basically, local closed policy. This means that if some XML subtree rooted at node $n$ cannot be decrypted by a set of keys owned by the user, neither can any of its subtrees. Hence, the same view can be extracted as in the previous case.

• Access control processor by Damiani et al. [30]. The pre-
processing step mentioned previously is already included in the algorithm. The rule *most specific takes precedence* (which simply says that an unlabelled node takes the security label of its first labelled ancestor/descendant) is used for resolving conflicts. Finally, forbidden nodes are deleted if they do not have permitted children, otherwise only their attributes are cleaned. Such a policy enforcement results in Vladimir’s view depicted in Fig. 2.3(b). For Dmitry Konovalov, the field *unreliable* would be visible as well because it has the permitted child *recomm-letter*. Sensitive tags *favorable* and *unfavorable* are revealed as well because the content of recommendation letters are accessible.

- **Security views by Fan et al. [40].** Although [40] does not discuss XML view materialization, we tried to emulate their schema-level enforcement at the XML instance. The view for Vladimir Romanov is the same as in the first case. However, the field *department* from the application of Dmitry Konovalov would still be missing. However, let’s imagine that Vladimir Romanov does not waive his right to see the content of recommendation letters. Then, the XML view is presented in Fig. 2.3(c) (1) misses *department* information of other applications, (2) has the element *dummy1* that may suggest to the user an idea that something is hidden.

As we have seen above, none of these methods produces a correct XML view. Author-X hides too much information that should have been available for the user and sometimes reveals sensitive information in a meaningless way. The access control processor by Damiani et al., on the contrary, reveals too much information. The last methods is better, but some information is still missing due to inappropriate qualifier semantics that deletes
CHAPTER 2. XML SECURITY VIEWS

2.1. A MOTIVATING EXAMPLE

(a) Authorized XML view for student Dmitry Konovalov

(b) and for student Vladimir Romanov

Figure 2.4: Security annotation for competing student

A subtree rooted at a node where the qualifier does not hold.

**Example 4:** Fig. 2.4(a) and Fig. 2.4(b) are the authorized views retrieved from the document in Fig. 2.2(a). In particular, Dmitry Konovalov has login dkonvalov and does not waive his right to see recommendation letters supporting his application (i.e., waiver=false), while Vladimir Romanov has login vromanov and waiver=true in his case. Both users also may infer the ranking among departments of the university since their views include department elements of all the other applications.

An important question remains unanswered: what schema information should be provided to the user? To formulate and process queries, the user needs a schema describing the accessible data. One solution, suggested by Damiani et al. [30], is to loosen the original DTD (make forbidden nodes optional). In some cases, it is unacceptable to expose even the loosened DTD to final user. To illustrate this, consider two permissible XPath queries about a letter of recommendation:

\[ Q_1 : /\text{applications}/\text{application}/*/\text{evaluator} \]
\[ Q_2 : /\text{applications}/\text{application}/\text{recomm-letter}/\text{evaluator} \]

The query \( Q_1 \) finds all elements of type evaluator that are associated with
recommendation letter (including unreliable ones), while \(Q_2\) returns only evaluators of reliable recomm-letters. Although most of the unreliable data is hidden, a look at the DTD document allows one to infer which letters are considered as unreliable: the evaluators in \(Q_1\) that are not returned by \(Q_2\); thus, we have a security breach. This is because all evaluators are visible, but in different ways.

### 2.2 XML and XPath

We first review DTDs (Document Type Definitions [98]) and XPath [100] queries. It is well-known, that non-recursive DTDs may be modelled as a DAG.

**Definition 1:** A DTD \(D\) is a triple \((\text{Ele}, P, \text{root})\), where \(\text{Ele}\) is a finite set of element types; \(\text{root}\) is a distinguished type in \(\text{Ele}\) called “root”, and \(P\) is a function defining element types such that for each \(A\) in \(\text{Ele}\), \(P(A) = \alpha\), where \(\alpha\) is a regular expression, defined as follows:

\[
\alpha := \text{str} | \text{Ele} | \epsilon | \alpha + \alpha | \alpha, \alpha | \alpha^*
\]

where \(\text{str}\) is a special type denoting PCDATA, \(\epsilon\) is the empty word, and “+”, “,”, and “*” denote disjunction (or choice container), concatenation (or sequence container), and the Kleene star, respectively. We call \(A \rightarrow P(A)\) as the **DTD production rule** of \(A\). For all element types \(B\) occurring in \(P(A)\), we refer to \(B\) as a child type of \(A\) and to \(A\) as a parent type of \(B\).

In the spirit of [79], our DTD definition allows mixed containers (e.g., choice container may include sequence subcontainer, etc) and do not require any normal form as in [40], where it was claimed that any DTD may be normalized. However, no algorithm on normalization/unnormalization were provided.

\(^2\)Although we may assume that normalization can be carried via additional intermediate nodes (e.g.,
Example 5: According to Def. 1, the formal representation of the database from Example 1 is the following. The DTD $D$ is defined as $(Ele, P, db)$, where

$\begin{align*}
db &= \text{applications} \\
Ele &= \{\text{applications, application, student-data, department, degree, waiver, name, recomm-letter, evaluator, title, institution, id, letter, rating, English, MS, PhD, free-text, PDF, TXT, unreliable, reason, favorable, unfavorable}\}
\end{align*}$

and the function $P$ is the following (we omit the definition of elements whose type is $\text{str}$):

$\begin{align*}
P(\text{applications}) &= (\text{application}^*) \\
P(\text{application}) &= (\text{student-data}, \text{recomm-letter}^*, \text{unreliable}^*) \\
P(\text{student-data}) &= (\text{department, degree, waiver, name, id}) \\
P(\text{recomm-letter}) &= (\text{evaluator, letter}) \\
P(\text{evaluator}) &= (\text{name, title, institution}) \\
P(\text{letter}) &= (\text{favorable}+\text{unfavorable}) \\
P(\text{favorable}) &= (\text{rating, free-text}) \\
P(\text{unfavorable}) &= (\text{rating, free-text}) \\
P(\text{rating}) &= (\text{English, MS+PhD}) \\
P(\text{free-text}) &= (\text{PDF+TXT}) \\
P(\text{unreliable}) &= (\text{recomm-letter, reason})
\end{align*}$

An XML document is typically modelled as a node-labelled tree.

Definition 2: [40] An XML tree $T$ conforms to a DTD $D$ iff

1. the root of $T$ is the unique node labelled with $\text{root}$;

2. each node in $T$ is labelled either with an $Ele$ type $A$, called an $A$ element, or with a value of type $\text{str}$, called a text node;

3. each $A$ element has a list of children of elements and text nodes such that their labels form a word in the regular language defined by $P(A)$;

\footnote{rating has two children: English and a normalization node normalizing_node with two children PhD and MS), the role of these nodes during policy enforcement, DTD view querying and query rewriting presented in [40] is not clear.
4. each text node carries a \texttt{str} value and is a leaf of the tree.

We call \( T \) an \textit{instance} of \( D \) if \( T \) conforms to \( D \).

Next, we consider a class of XPath queries that corresponds to the CoreXPath of Gottlob et al. [48] augmented with the union operator and atomic tests and which is denoted by Benedict et al. [9] as \( \mathcal{X} \).

The XPath axes we consider as primitive are \textit{child}, \textit{parent}, \textit{ancestor-or-self}, \textit{descendant-or-self}, \textit{self}. Gottlob et al. [48] show how the semantics of such axes can be computed in polynomial time. In the sequel we denote by \( \theta \) one of the primitive axes and by \( \theta^{-1} \) its inverse. Notice that each primitive axis has its inverse within the same set of primitives. For instance, 
\[ \text{descendant-or-self}^{-1} = \text{ancestor-or-self} \].

\textbf{Definition 3}: An XPath expression in \( \mathcal{X} \) is defined by the following grammar:

\[
\begin{align*}
\langle xpath \rangle & ::= \ '/?' (\langle path \rangle | (\langle path \rangle \ '∪' (\langle path \rangle)) \ast \\
\langle path \rangle & ::= \langle step \rangle ('/' (\langle step \rangle)) \ast \\
\langle step \rangle & ::= \langle test \rangle | \langle test \rangle (\lceil (\langle qual \rangle) \rceil) \ast \\
\langle test \rangle & ::= \theta :: 'A | \theta :: '*' \\
\langle qual \rangle & ::= (\langle path \rangle) \text{\texttt{op}} c | (\langle qual \rangle) \text{\texttt{and}} (\langle qual \rangle) | (\langle qual \rangle) \text{\texttt{or}} (\langle qual \rangle) | \text{\texttt{not}} (\langle qual \rangle) | \lceil (\langle qual \rangle) \rceil :'
\end{align*}
\]

where \( \theta \) stands for an axis, \( c \) is a \texttt{str} constant, \( A \) is a label, \( op \) stands for one of \( =, <, >, \leq, \geq \). The result of the \texttt{qual} filtering is called \textit{qualifier} and is denoted by \( q \). We denote by \( \mathcal{X}_{\text{NoTest}} \) the fragment built without the \langle path \rangle \ op \ c \ test.

For the sake of readability, we ignore the difference between \texttt{xpath} and \texttt{path}; we denote both with \( p \). We also abbreviate \texttt{self} by \( \epsilon \), \texttt{child} \::\::\::\::\:: A/p with \( A/p \), \texttt{descendant-or-self} \::\::\::\::\:: A/p by \( //A/p \), and \( p = p_1/p_2 \) with \( p_2 = //p'_2 \) is written \( p \) as \( p_1//p'_2 \). The parent axis is also abbreviated as \( ../ \).

The semantics of XPath is obtained by adapting to our fragment the \( S_\rightarrow \), \( S_\leftarrow \), \( E \) operators proposed by Gottlob et al. [48] and is identical to the proposal of Benedickt et al. [9]. Intuitively \( S_\rightarrow [[p]] (N) \) gives all nodes that
are reachable from a node in \( N \) using the path \( p \). The \( S_{\rightarrow} [p] \) functions gives all nodes from which a path \( p \) starts to arrive at the queried node. The \( \mathcal{E} [q] \) function evaluates qualifiers and returns all nodes that satisfy \( q \).

For the sake of readability we overload the \( \theta \)-symbol to stand for both the semantics and the syntax of axes. So, given a set of nodes \( N \) of a document \( \mathcal{T} \) we have \( \theta(N) = \{ m \mid n \theta m \text{ for } n \in N \} \). In other words, \( \theta(N) \) returns the nodes that are reachable according to the axis from a node in \( N \). By \( \mathcal{T} (A) \) we denote the set of nodes that have element type \( A \).
By $\mathcal{T}(\ast)$ we denote all nodes of a document. By $\mathcal{O}(c)N$ we denote the function that returns all nodes of a set $\mathcal{T}(N)$ whose $\text{str}$ value is $op\ c$.

The semantics of the other operators are shown in Fig. 2.5.
Chapter 3

Document Access Control

Since we want to develop an access control enforced at DTD level emulating XML-based policy enforcement, we start to investigate document access control, policy specification and enforcement.

Definition 4: An authorization specification at instance level is a pair $(T, \text{ann})$, where $T$ is an XML tree, $\text{ann}$ is an annotation of $T$ nodes with $Y$, $N$, and $Q[q]$ expressed in XPath fragment from Def. 3.

Definition 5: Let $(T, \text{ann})$ be an authorization specification. The authorized version $T_A$ of $T$ according to the authorization specification is obtained from $T$ as follows:

1. Evaluate qualifiers top down starting from the root and replace annotations by $Y$ or $N$ depending on the result;

2. For each unlabelled node, label it with
   - the annotation of its nearest labelled ancestor; or
   - the annotation of its nearest labelled descendants applying value conflict resolution policy; or
   - the local propagation value;
3. Delete all nodes labelled with N from the result, making all children of a deleted node $v$ into children of $v$’s parent.

The annotation of the document, before deleting nodes in the last step, is called the full annotation of $T$. □

We present different policies to extend a partial annotation of the XML document to a full one. There are a number of alternatives. Although top-down propagation is considered to be the most natural in many applications [11], [30], [34], [62], there are other proposals that also consider bottom-up propagation of security labels [56], [55].

Our security model is based on a specific policy, used for determining a complete authorization specification of a document based on a partial specification. This is the most-specific-takes-precedence (MSTP) policy [37]. Different applications may have different requirements, and we now look at alternative approaches.

We can classify security policies using two orthogonal classifications that focus on completeness and consistency [37]. The first classification is based on how one handles unassigned values, while the second is based on the handling of conflicting assignments and how one restores consistency.

We are interested only in policies that are complete and consistent:

**Definition 6:** A policy is complete and consistent if every partially annotated tree can be extended to a fully annotated tree. □

To capture the variety of policy propagation and conflict resolution options we have identified the following framework:

**Local Propagation Policy (LP):** “open”, “closed”, or “none”;

**Hierarchy Propagation Policy (HP):** “topDown” (td), “bottomUp” (bu), or “none”;
CHAPTER 3. DOCUMENT ACCESS CONTROL

Structural Conflict Resolution (SC): “localFirst” (lf), “hierarchyFirst” (hf), or “none”;

Value Conflict Resolution (VC): “denialTakesPrecedence” (dtp), “permissionTakesPrecedence” (ptp), or “none”.

The LP option is similar to traditional policies for access control: “open”, if a node is not labelled N, it should be labelled by Y; “closed” means that a node not labelled Y should be labelled by N; finally, the “none” option says that a node is not labelled.

The HP option specifies how node annotation is inherited. In the case of “td”, an unlabelled node with a labelled parent inherits the label of its parent. In the case of “bu” an unlabelled node inherits the label from labelled children. The “none” option says that no hierarchy propagation is applied. Note that the “bu” case can result in conflicts, and they should be addressed by the VC Resolution Policy.

The SC option specifies whether the local or the hierarchy rule takes precedence (“lf” or “hf” respectively); while “none” means that the choice depends on the values and on the VC option. The latter specifies how to resolve conflicts for unlabelled nodes that are assigned different labels by the preceding rules: N always has precedence over Y (“dtp”); Y always has precedence over N (“ptp”), and no choice (“none”).

We list here several possible policies. These are variations of classical security policies that are used in other settings ([37]):

- either permission-takes-precedence or denial-takes-precedence together with either the closed or open policy;

- most-specific-takes-precedence with the top-down policy and the root labelled either Y or N by default.
Table 3.1: Policy alternatives

<table>
<thead>
<tr>
<th></th>
<th>HP</th>
<th>LP</th>
<th>SC</th>
<th>VC</th>
<th>additional condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>td</td>
<td>≠none</td>
<td>hf</td>
<td>*</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>td</td>
<td>none</td>
<td>*</td>
<td>*</td>
<td>root is annotated</td>
</tr>
<tr>
<td>3</td>
<td>bu</td>
<td>≠none</td>
<td>hf</td>
<td>≠none</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>bu</td>
<td>none</td>
<td>*</td>
<td>≠none</td>
<td>all leaves are annotated</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>≠none</td>
<td>lf</td>
<td>*</td>
<td>none</td>
</tr>
<tr>
<td>6</td>
<td>none</td>
<td>≠none</td>
<td>*</td>
<td>*</td>
<td>none</td>
</tr>
<tr>
<td>7</td>
<td>≠none</td>
<td>≠none</td>
<td>none</td>
<td>≠none</td>
<td>none</td>
</tr>
<tr>
<td>8</td>
<td>none</td>
<td>none</td>
<td>*</td>
<td>*</td>
<td>none</td>
</tr>
<tr>
<td>9</td>
<td>≠none</td>
<td>≠none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>10</td>
<td>bu</td>
<td>*</td>
<td>hf</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>11</td>
<td>bu</td>
<td>none</td>
<td>≠hf</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

- most-specific-takes-precedence with top-down policy and either the closed or open policy.

In the sequel, we give conditions for complete and consistent policy combinations. We represent all the possible policy options in Table 3.1, where the symbol “*” means “any”, i.e. any possible value. Note that Table 3.1 reflects all 81 possible combination of security options, since symbols * and ≠ in columns HP, LP, SC, and VC means, respectively, three and two possible values for the corresponding policy option.

**Definition 7:** A policy is called a top-down/bottom-up/local policy if it satisfies the conditions in lines 1-2/3-4/5-6 respectively of Table 3.1. □

**Proposition 1:** The top-down, bottom-up and local policies are complete and consistent. □

**Proof:**

**Top-down policy.** Assume that T is a partially annotated tree. We show that the annotation can be extended to full.
Consider condition “*” (the first line of Table 3.1) of the topDown security policy of Table 3.1.

Base case: if the root is annotated then we are done. If it is not annotated then according to the definition it can obtain its annotation from the local security policy: Y/N if local=\textit{open/closed} respectively. Thus the root annotation is defined.

Inductive case: consider an arbitrary node \( n \) with annotated parent \( p \). If \( n \) is annotated we are done. Otherwise, \( n \) obtains its annotation from the parent since structural conflict=\textit{hierarchyFirst}. Thus annotation of any node is defined.

Consider condition “root is not annotated” (the second line of Table 3.1) of topDown security policy of Table 3.1.

Base case: the root is annotated.

Inductive case: consider an arbitrary node \( n \) with annotated parent \( p \). If it is annotated we are done. Otherwise it obtains its annotation from the parent. Thus annotation of any node is defined.

**Bottom-up policy.** Assume that \( T \) is a partially annotated tree. We show that the annotation can be extended to full.

Consider condition 3 describing the bottomUp security policy of Table 3.1.

Base case: if the children are annotated then we are done. If some of them are not annotated, then according to the condition they can obtain their annotation from the local security policy: Y/N if local=\textit{open/closed} respectively. Thus annotation of all leaves is defined.

Inductive case: consider an arbitrary node \( n \) with all annotated children. If \( n \) is annotated we are done. Otherwise, \( n \) obtains its anno-
CHAPTER 3. DOCUMENT ACCESS CONTROL

tation from the children since \textit{structural conflict} = \texttt{hierarchyFirst}. However, different children can have different annotations, but \textit{value conflict} \# \texttt{nothingTakesPrecedence} can be used to define “winning” label. Thus the entire tree is annotated.

Consider condition 4 describing \textit{bottomUp} security policy of Table 3.1.

Base case: all leaves are annotated.

Inductive case: consider an arbitrary node \( n \) with all annotated children. If it is annotated, we are done. Otherwise it obtains its annotation from the children. Different children can have different annotations, but \textit{value conflict} \# \texttt{nothingTakesPrecedence} will yield a unique annotation. Thus annotation of any node is defined.

\textbf{Local policy.} Assume that \( T \) is a partially annotated tree. We show that the annotation can be extended to full.

Consider condition 5 describing \textit{local} security policy of Table 3.1. Since \textit{structural conflict} = \texttt{localFirst}, \textit{local} is enforced in the first turn.

Consider case 6 of \textit{local} security policy of Table 3.1. Since \textit{hierarchy} security policy is not defined, \textit{local} is enforced.

Thus, for each not annotated node \( n \), we enforce \textit{local} security policy that assigns either a label \( Y \) or \( N \) depending on \textit{local} policy definition.

\[ \square \]

In some cases both HP and LP policies are defined, but SC policy is “none”. Hence, we apply both HP and LP thus obtaining for each node more than one security annotation. The result is defined by means of VC policy which should defined, i.e. the conditions in line 7 of Table 3.1 should be satisfied.
CHAPTER 3. DOCUMENT ACCESS CONTROL

Algorithm Policy Class

Input: Policy combinations: \{HP, LP, SC, VC\}

Output: Policy class

1: if \((HP \neq \text{none} \land LP \neq \text{none} \land SC = \text{hierarchyFirst}) \vee (HP \neq \text{none} \land LP = \text{none})\) then

2: if \(HP = \text{topDown}\) then

3: return topDown policy;

4: else

5: if \(VC \neq \text{none}\) then

6: return bottomUp policy;

7: else

8: return unresolvable policy;

9: end if

10: end if

11: else if \((HP \neq \text{none} \land LP \neq \text{none} \land SC = \text{localFirst}) \vee (HP = \text{none} \land LP \neq \text{none})\) then

12: return local policy;

13: else if \(HP \neq \text{none} \land LP \neq \text{none} \land SC = \text{none}\) then

14: if \(VC \neq \text{none}\) then

15: return multilabel policy

16: else

17: return unresolvable policy

18: end if

19: else if \(HP = \text{none} \land LP = \text{none}\) then

20: return unresolvable policy;

21: end if

Figure 3.1: Algorithm Policy Class

Definition 8: A policy is called a multilabel policy if it satisfies the conditions in line 7 of Table 3.1.

Proposition 2: The multilabel policy is complete and consistent.

Proof: The proof follows from the completeness and consistency of the hierarchical and local policies along with value conflict resolution option...
All the other policies are classified as unresolvable. Indeed, policies following the condition 8 are incomplete because neither HP nor LP is applied which results in the fact that unlabelled nodes are not assigned any label; policies in lines 9 and 10 are inconsistent because either the “winning” label in multilabel case is not provided (line 9) or value conflict which arises in the case of “bu” policy propagation is not resolved (line 10); policy in line 11 may be either inconsistent (some XML leaves are not defined; therefore, “bu” propagation is inconsistent) or incomplete (all leaves are defined but have different labels; therefore, value conflict arises and cannot be resolved).

So far, we have defined five classes of policies: local policy, top-down policy, bottom-up policy, multilabel policy, and unresolvable policy class. These classes were identified based on particular combinations of policy options for value propagation (hierarchy or local) and conflict resolution (structural and value).

Fig. 3.1 summarizes the policy class identification procedure.
Chapter 4

Schema Access Control

4.1 Schema Access Control for Top-Down Policies

For each user group, an access specification is defined to be a partial mapping \( \text{ann} \) such that for each production \( A \rightarrow \alpha \) in \( D \) and each element type \( B \) in \( \alpha \), \( \text{ann}(A, B) \) is either \( Y \) or \( N \) or an XPath qualifier \([q] \), denoting that the \( B \) child of an \( A \) element is accessible, inaccessible, or conditionally accessible depending on \([q] \), respectively.

Example 6: In Fig. 4.1, we show an example of security specification described in Example 2: paths to unconditionally allowed (forbidden) element types from their corresponding parents are marked with \( Y(N) \), and conditionally accessible element types are marked by qualifiers \( q_1 \), \( q_2 \) and \( q_3 \) (Fig. 4.1(b)). In particular, elements unreliable and letter are forbidden to everybody, while information on department is unconditionally allowed in spite of conditional accessibility \((q_1) \) of its ancestor application that is permitted if the students’s login is the same as the id field of the underlying student node. Next, \( q_2 \) permits access to recomm-letter element if it is a descendant of a permitted application, \( q_3 \) reveals subtrees rooted at rating and free-text if the latter are descendants of a permitted application...
4.1. SCHEMA ACCESS CONTROL
FOR TOP-DOWN POLICIES

![Diagram of schema access control](image)

(a) Security annotation defined at DTD level

\[ q_1 = \text{student-data}/id = $\text{login} \]
\[ q_2 = ../../	ext{student-data}/id = $\text{login}; \]
\[ q_3 = \text{ancestor::application/student-data}[id = $\text{login}]/\text{waiver} = "\text{false}"; \]

(b) Meaning of security annotation qualifiers

Figure 4.1: Security annotation for the applicant

having waiver = “true”. In all three qualifiers, $\text{login}$ is a dynamic variable
that is assigned at run time and equals the student’s login name.

**Definition 9:** An **authorization specification** \( S \) is a pair \((D, \text{ann})\), where
\( D \) is a DTD, \( \text{ann} \) is a partial mapping such that, for each top-down edge
\((A, B)\), \( \text{ann}(A, B) \), if defined, is an annotation of the form:

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

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

Every \( \text{ann}(A, B) \) defines a **source element type** \( A \) denoted as \( s \), a **destination element type** \( B \) denoted as \( d \), and a **generator of a security label for**
\( B \) (or simply **generator**) \((A, B)\) denoted \( g \). Thus, we can write \( \text{ann}(A, B) \)
as \( \text{ann}(s, d) \) or \( \text{ann}(g) \).
Example 7: In Fig. 4.1, \( \text{ann}(\text{application}, \text{unreliable})=\text{N} \) defines a source application, destination unreliable and a generator \( (\text{application}, \text{unreliable}) \) of a security label \( \text{N} \) for unreliable.

Intuitively, labelling an edge \( (A, B) \) with an unconditional annotation is a security constraint expressed at the schema level: \( \text{Y} \) or \( \text{N} \) indicates that, in the case of top-down propagation, the corresponding \( B \) child of an \( A \) element in an XML document is always accessible (\( \text{Y} \)) or always inaccessible (\( \text{N} \)), no matter what the actual values of these elements in the document are. If \( \text{ann}(A, B) \) is not explicitly defined, then \( B \) inherits the accessibility of \( A \) or obtains a label from the default policy. On the other hand, if \( \text{ann}(A, B) \) is explicitly defined it overrides the accessibility of \( B \) obtained via propagation.

4.2 Security Views for Top-Down Policies

From a specification as presented in the previous subsection, we would like to infer a DTD view \( D_v \) which represents a schema of available data for the user. We use a set of XPath queries showing how to construct \( D_v \) from the initial DTD \( D \). We call this set of XPath expressions \( \sigma \)-function.

At this point, it would be interesting to continue the comparison of our vision of view with those of proposals in [11], [31], and [40]. Namely, the first one does not consider any schema information and reveals the intial DTD to the user as it is. Obviously, this strategy will leak too much sensitive information and hence is not reliable. Damiani’s loosened version of the DTD is the original DTD with optional cardinality “?” on every edge that was not of cardinality zero-or-many “*”. It slightly alleviates the problem of the previous approach but not enough. Finally, the DTD view constructed according to the algorithm in [40] is shown in Fig. 4.3.
4.2. SECURITY VIEWS FOR
TOP-DOWN POLICIES

CHAPTER 4. SCHEMA ACCESS CONTROL

Figure 4.2: Security view for the applicant

(a) Security DTD view

\[ XP_1 = application[q_1] \]
\[ XP_2 = letter/(favorable \cup unfavorable)/rating[q_3] \]
\[ XP_3 = letter/(favorable \cup unfavorable)/free-text[q_3] \]
\[ XP_4 = ./(\epsilon \cup unreliable)/recomm-letter[q_2] \]
\[ XP_5 = application[\neg q_1]/student-data/department \]

(b) Meaning of XPath expressions

Figure 4.3: Authorized DTD view for student Vladimir Romanov by [40]
Namely, edge \((\text{applications, department})\) is missing, two dummy elements are introduced instead of \text{favorable} and \text{unfavorable}. Finally, \text{recomm-letter} is added to the view with \(XP'_2 = (\text{unreliable} \cup \epsilon)/\text{recomm-letter}\). However, if \(\text{ann}(()\text{application, recomm-letter})\) were \(\text{N}\) then \text{recomm-letter} would have been missing from the DTD view at all although \(q_4\) allowed an access to it. This is because, in DFS traversal of the DTD graph in [40], the negative authorization would have arrived to \text{recomm-letter} before \(q_4\) from \text{unreliable}. It means that \text{recomm-letter} would have been immediately substituted with its children.

In the following, the view is provided to the users so that they can formulate their queries over the view. This means that the users can only access data via \(D_v\). At the same time, the function \(\sigma\) is withheld from the users, and is used by the system to extract accessible data from the actual XML document.

\textbf{Example 8:} Fig. 4.2 shows the view DTD \(D_v\) which represents accessible data according to Example 2, and \(\sigma\)-function expressed by means of XPath expressions \(XP_1-XP_5\). In particular, \(XP_1\) says that only nodes of type \text{application} with student’s \text{id} equal to student’s login are included in the view. \(XP_2\) and \(XP_3\) skip all forbidden elements on the path from \text{recomm-letter} to, respectively, \text{rating} and \text{free-text} that are the children of accessible \text{application} (condition \(q_3\)), \(XP_4\) extracts all \text{recomm-letter}, including unreliable ones. Finally, \(XP_5\) collects all \text{departments}, even those that are located in forbidden parts of the tree.

\textbf{Definition 10:} Let \(D\) be a DTD. A security view for \(D\) is a pair \((D_v, \sigma)\) where \(D_v\) is a DTD schema of accessible information and \(\sigma\) is a function from pairs of adjacent element types such that for each element type \(A\) in \(D_v\) and its child element type \(B\), \(\sigma(A, B)\) is an expression in \(\mathcal{X}\) defining accessibility of \(B\) from \(A\).
Drawing an analogy between relational databases and XML, we elaborate a method of XML view materialization using the $\sigma$-function so that the XML view conforms to the DTD view $D_v$.

**Example 9:** It is easy to see that the views for Dmitry Konovalov and Vladimir Romanov, presented in Example 2 conform to the DTD view of Example 8. \(\square\)

The derivation of a materialized XML view is explained in the next definition:

**Definition 11:** Let $S = (D_v, \sigma)$ be a security view. The semantics of $S$ is a mapping from documents $T$ conforming to $D$ to documents $T_V$ such that:

1. $T_V$ conforms to $D_v$

2. The nodes of $T_V$ are a subset of the nodes of $T$, and their element type is unchanged.

3. For any node $n$ of $T$ which is in $T_V$, let $A$ be the element type of $n$, and let $B_1, \ldots, B_m$ be the list of element types that occur in $P(A)$. Then the children of $n$ in $T_V$ are

$$\bigcup_{1 \leq i \leq m} S_\sigma[[\sigma(A, B_i)]]\{\{n\}\}.$$ 

and the order of nodes of type $B_i$ for each $i = 1, \ldots, m$ is the same as in $T$.

$T_V$ is called the *materialized version* of $T$ w.r.t. the view $S$. \(\square\)

A classical question for relational database research, namely whether a view produced by the MATERIALIZE algorithm is actually populated by some instances is true. Since the root of the document is always labelled $Y$, the materialized view always has at least one node. We can show that for
Chapter 4. Schema Access Control

4.2. Security Views for Top-Down Policies

XPath fragment, the algorithm is efficient. Let \( f(n, d) \) be the complexity of evaluating an XPath expression of size \( n \) on a document of size \( d \). Gottlob et al. [48] have shown that for CoreXPath (i.e., \( \mathcal{X} \) without union and test) it is \( f(|\sigma|, |T|) = O(|\sigma| \times |T|) \). We extend their result to \( \mathcal{X} \) without test and with a factor of \( T \) to the full \( \mathcal{X} \) fragment. Let \( |\sigma| \) be the size of the largest XPath expression in the range of \( \sigma \). Then:

**Lemma 1:** Every XPath query \( p \in \mathcal{X}_{\text{NoTest}} \) over a document \( T \) can be evaluated in time \( O(|p| \times |T|) \).

**Proof:** The proof follows the line of Gottlob, Koch and Pichler [48] for the CoreXPath fragment (that is without union of paths): we use the functions \( S_{\rightarrow} \), \( S_{\leftarrow} \), and \( E \) to compute a query tree which is then evaluated bottom-up to yield the desired complexity result.

For the full fragment considered here, the naive implementation of union would lead to an exponential blow up because the processing of \( p_1 \) is duplicated in \( S_{\rightarrow} [[p_1/(p_2 \cup p_3)]] (N) = S_{\rightarrow} [[p_1/p_2]] (N) \cup S_{\rightarrow} [[p_1/p_3]] (N) \).

To avoid this blow-up we use a query DAG instead of a query tree. Each path of the form \( S_{\rightarrow} [[p_1/(p_2 \cup p_3)]] (N) \) is mapped into a (single source) rooted DAG in which the root is labelled \( \cup \) with two children, one corresponding to the root of \( S_{\rightarrow} [[p_2]] (X) \) and one corresponding to the root of \( S_{\rightarrow} [[p_3]] (X) \). The shared \( X \) leaf node is the root of the \( S_{\rightarrow} [[p_1]] (N) \) node.

Formally, this is equivalent to say that \( S_{\rightarrow} [[p_1/(p_2 \cup p_3)]] (N) \) is evaluated using the symbolic rightmost lazy evaluation. In other words,

\[
\begin{align*}
\text{let } X_1 &= S_{\rightarrow} [[p_1]] (N); \\
\text{let } X_{21} &= S_{\rightarrow} [[p_2]] (X_1); \\
\text{let } X_{31} &= S_{\rightarrow} [[p_3]] (X_1); \\
\text{then } S_{\rightarrow} [[p_1/(p_2 \cup p_3)]] (N) &= (X_{21} \cup X_{31}).
\end{align*}
\]

For the evaluation of the \( S_{\rightarrow} [|(p_1 \cup p_2)/p|] \) function, a similar strategy
4.2. SECURITY VIEWS FOR TOP-DOWN POLICIES

CHAPTER 4. SCHEMA ACCESS CONTROL

can be applied:

\[
\begin{align*}
\text{let } X_1 &= \{ x \mid \mathcal{S} - [p_1] (x) \neq \emptyset \}; \\
\text{let } X_2 &= \{ x \mid \mathcal{S} - [p_2] (x) \neq \emptyset \}; \\
\text{let } X &= \mathcal{S} - [p_1] (X_1) \cup \mathcal{S} - [p_2] (X_2); \\
\text{then } \mathcal{S} - [(p_1 \cup p_2)/p] &= \mathcal{S} - [p] (X).
\end{align*}
\]

With this construction each XPath expression can be transformed in time \(O(|p|)\) into a query DAG of size \(O(|p|)\) in which each operation is a set operation that can be computed in time \(O(|T|)\) thus yielding the desired upper bound. \(\square\)

The addition of the test operation increases slightly the complexity because the computation of the \(\mathcal{O}(c)N\) operator requires the comparison of the \texttt{str} value \(c\) with the \texttt{str} value at every node of a set \(T (N)\) which may include all nodes of the tree. This yields a quadratic increase in data complexity. Once the \(\mathcal{O}(c)N\) has been computed at the appropriate leaves of the query DAG, all other operations can be done in time linear in the size of the document. Hence, the following takes place:

**Lemma 2:** Every XPath query \(p \in \mathcal{X}\) over a document \(T\) can be evaluated in time \(O(|p| \times |T|^2)\). \(\square\)

**Corollary 3:** Every valid DTD view whose annotations are in \(\mathcal{X}\), respectively in \(\mathcal{X}_{\text{NaTest}}\), can be materialized in \(O(|\sigma| \times |T|^3)\), resp. \(O(|\sigma| \times |T|^2)\), by Algorithm \textsc{Materialize}. \(\square\)

**Proof:** The first step of the algorithm takes up only \(O(|\sigma| \times |T|^3)\), resp. \(O(|\sigma| \times |T|^2)\), by using the construction in Lemma 1, resp. Lemma 2, for the evaluation of XPath queries. For the subsequent processing the number of iteration is bounded by the number of nodes in \(T\) and each step can be performed in \(O(|\sigma| \times |T|)\) steps. \(\square\)

From Lemmas 1, 2 and Corollary 3, we immediately prove the next theorem:
Theorem 4: Algorithm Materialize computes a materialized view in time $O\left(f(|\sigma|, |T|) \times |T|\right)$.

Definition 12: A valid security view is one for which the semantics are always well-defined, i.e., if for every document $T$, its materialized version conforms to the security view DTD.

Not all views are valid: wrong typing, violated cardinality constraints, and other problems could be all causes of a view to be invalid. For example, in the case of loosened DTD, authorized XML view is not valid since the loosened DTD may contain (optional) element types that in XML document should be deleted and their children should be attached to permitted parents. However, the semantics of an optional element assumes that the absence of the element means the absence of the subtree rooted at that element. Consequently, the authorized XML view does not conform to the loosened DTD. To resolve this problem, Damiani et al. [30] proposed to delete only those forbidden nodes that do not have permitted descendants. This means that, sometimes, the user is allowed to see the information that is not permitted. The views that we construct from an annotated DTD are valid (see Sec. 4.3) and reveal all and only permitted information.

Security specification and views are related as follows.

Definition 13: Let $(D, \text{ann})$ be an authorization specification, and let $S = (D_v, \sigma)$ be a security view for $D$. $S$ is data equivalent to $(D, \text{ann})$ iff for every document $T$ conforming to $D$, the materialized version $T_v$ is isomorphic to the authorized version $T_A$.

Two weaker characterizations are based on the notion of data secrecy and data availability.

Definition 14: Let $(D, \text{ann})$ be an authorization specification, and $S =$

\footnote{Sometimes these notions are also termed “consistency” and “completeness” in the literature [37] but that terminology can be misleading in our context.}
4.2. SECURITY VIEWS FOR
TOP-DOWN POLICIES

CHAPTER 4. SCHEMA ACCESS CONTROL

(D_v, σ) a security view for D.

1. S guarantees data secrecy iff for every T conforming to D, and for every node n of T, if n occurs in T_V then n must also occur in the authorized tree T_A.

2. S guarantees data availability iff for every T conforming to D, and every node n of T, if n occurs in the authorized tree T_A then n occurs in the materialized version T_V.

Intuitively a secrecy-preserving view insures that no forbidden node is leaked, whereas an availability-preserving view is a guarantee that no permitted node is withheld from legitimate users. Obviously, data equivalence implies secrecy and availability, but the converse does not hold, since a data equivalent view also “preserves the structure” of the original document.

Given a security view S = (D_v, σ) and document T conforming to a DTD D, we give an algorithm constructing T_V in Fig. 4.4.

Proposition 5: If S = (D_v, σ) is a valid view for D, then the result of Algorithm MATERIALIZE is a document T_V that is the materialized version of T.

Proof: To proof the proposition, we must show that all three conditions of Def. 11.

In lines 9-16 algorithm evaluates σ(A, B_i), i = 1, . . . , m at any non-visited node n of type A in T_V. The result of this evaluation is a set of nodes of types B_1, . . . , B_m that become children of node n. The order of children of type B_i is the same as in T for every i = 1, . . . , m. This is exactly what the third condition of Def. 11 says.

The second condition holds obviously, because the algorithm constructs T_V from T. Consequently, the nodes of T_V are a subset of T. In addition,
Algorithm **Materialize**

**Input:** a document $T$ conforming to DTD $D$, a DTD View $(D_v, \sigma)$

**Output:** a materialized view $T_V$ of $T$ or ⊥ (there is no such view)

1. Set the root of $T_V$ to be the root of $T$;
2. for all nodes $n$ of type $A$ in $T$ do
   3. let $A \rightarrow P(A)$ the corresponding rule in $D_v$
   4. for all $B$ occurring in $P(A)$ do
      5. precompute $S_{\neg [\sigma(A, B)]} ([n])$
   6. end for
3. end for
8. assign to $T_V$ the root of $T$ and mark it as unprocessed
9. while there are unprocessed nodes in $T_V$ do
10. select an unprocessed node $n$ of type $A$ with rule $A \rightarrow P(A)$ in $D_v$
11. mark the nodes in
    \[
    \bigcup_{B \text{ occurs in } P(A)} S_{\neg [\sigma(A, B)]} ([n])
    \]
    in $T$ as unprocessed children of $n$ in $T_V$
12. if a child of $n$ already occurs as a processed node in $T_V$ then
13. return ⊥ (invalid view)
14. end if
15. mark $n$ as processed
16. end while

Figure 4.4: Algorithm **Materialize**

the algorithm neither changes the names of element types nor adds new ones.

Finally, since we assume that $D_v$ is valid, and any node $n$ of type $A$ in $T_V$ has children of types $P(A)$, it is clear that $T_V$ conforms to $D_v$. Thus, the first condition holds. □
4.3 View Construction for

Top-Down Policies

We now show how to construct a security view, given a DTD document and an authorization specification. The derivation of $D_v$ has two parts: (i) computing $D_F$ such that $Y/N$ labels are assigned to every element type, and (ii) restructuring $D_F$ so that $N$-labelled elements are deleted and their permitted children are attached to their nearest permitted ancestors. In this manner, the view DTD $D_v$ shows all, and only, the accessible data. The first part may use various propagation techniques (top-down, bottom-up, among siblings, etc.) and conflict resolution rules (denial or permission takes precedence, qualifier over $Y$ takes precedence and vice versa, explicitly defined label takes precedence over propagated one, priorities among access control rules applicable to the element, etc.) or may impose some default value on unlabelled elements.

Since we put annotations on DTD edges (generators), the idea behind our algorithm is to “push” security labels from generators to destination types.

**Definition 15:** If $\text{ann}(s, d)$, is defined and equals $a$, we say that $s$ transmits (or propagates) annotation $a$ to $d$ via $g$. □

Having obtained an annotation, a destination type, in its turn, becomes a source type and may transmit its annotation to its children (new destination types) via corresponding generators that were initially unlabelled (thus preserving most specific takes precedence condition).

**Example 10:** In Fig. 4.1, there is an annotation $\text{ann}(\text{recomm-letter}, \text{letter}) = N$. It means that $\text{recomm-letter}$ is a source transmitting $N$ to a destination $\text{letter}$. The type $\text{letter}$ obtains annotation $N$, becomes a source for favorable and unfavorable, and transmits $N$ to them. □
4.3. VIEW CONSTRUCTION FOR TOP-DOWN POLICIES

Since an annotation can be presented as a qualifier, the algorithm, first of all, eliminates qualifiers. For this purpose, it expands each of them into a union of two element types: one is the original element type, which is annotated $Y$, and the other is a new type, which is annotated $N$. Since the tag of an element uniquely determines the type, it follows that new type names cannot match any nodes in a document that conforms to the original DTD. This is not a serious problem, as all these new type names will be deleted in the final security view.

Definition 16: The semantics of $\text{ann}(A, B) = Q[q]$ is to split of node type $B$ in the DTD into two nodes having the following meanings: visible node instance of type $B$ is a child of a node instance of type $A$ if $B[q]$ holds, and invisible otherwise.

Basically, Def. 16 is an emulation of security policy enforcement at XML instance: if $B[q]$ holds at the current concrete node of type $B$, this node is visible in authorized view; otherwise, it is deleted from the final view. Obviously, the expression $B[q]$ cannot be evaluated at the DTD level.

Example 11: In Fig. 4.5, we demonstrate what happens to the edge $(\text{applications}, \text{application})$ with an annotation $Q[\text{student-data/id} = \$\text{login}]$. Namely, we split application into a visible application$_Y$, which is a normal application, and invisible application$_N$, which should be deleted afterwards. Both newly created elements have the same set of parents and the same set of children. For any child, the newly created element transmits the same annotation as the old one. For each parent, we construct a $\sigma$-function describing the situation when qualifier holds for the visible node, and the situation when qualifier does not hold for the invisible. The latter is equivalent to the situation when a negation of the initial qualifier holds.

To avoid an overloading of Fig. 4.5, we don’t show the elimination of qualifiers $Q[q_2]$ and $Q[q_3]$. Regarding the last one, elements rating and free-
Figure 4.5: Removing qualifier \( \text{ann}(\text{applications}, \text{application}) = Q[\text{student-data/id} = \$login] \)

text should be split twice because of the sources favorable and unfavorable transmitting qualifier. However, there will be always two kinds of newly created nodes, visible and invisible. Hence, we can either reuse elements created previously, or merge multiple elements of the same visibility into a unique element.

After removing qualifiers, the next step expands the annotation to a “full annotation” by propagating the remaining \( Y \) and \( N \) labels. The idea is simple: if all incoming edges of some destination element have the same annotation \( Y \) or \( N \) this element becomes visible or invisible respectively and transmits this annotation to its children. It is easy to see that every XML document has a unique full annotation \([40]\). At the schema level, however, this is not the case, as there may be several “paths” in the DTD that reach the same element type, each of which results in a different annotation. We use a similar technique when we handle qualifiers, i.e., we introduce new element types, and label the original one with \( Y \) (emulation of a visible node), which is connected with parents transmitting to it \( Y \),
Algorithm Annotate View

Input: A authorization specification \((D, \text{ann})\)

Output: Fully annotated DTD \(D\)

1. Initialize \(D_v := D\) where \(\text{ann}\) is defined on \(D_v\) as on \(D\);
2. for all production rules \(A \rightarrow P(A)\) in \(D_v\) and all \(B \in P(A)\) do
3.    initialize \(\sigma(A, B) := B\)
4. end for
5. for all \(A \rightarrow P(A)\) and all \(B \in P(A)\) with \(\text{ann}(A, B) = Q[q]\) do
6.    add to \(D_v\) a new element type \(B'\) and a production rule \(B' \rightarrow P(B)\)
7.    replace \(B\) by \(B + B'\) in \(P(A)\)
8.    set \(\sigma(A, B) := B[q]\); \(\sigma(A, B') := B[\neg q]\);
9.    set \(\text{ann}_\text{data}(B) = Y\) and \(\text{ann}_\text{data}(B') = N\);
10. for all element types \(C\) occurring in \(P(B)\) do
11.    set \(\sigma(B', C) := \sigma(B, C)\);
12.    set \(\text{ann}(B', C) := \text{ann}(B, C)\);
13. end for
14. end for
15. while \(\text{ann}_\text{data}(B)\) of some element types \(B\) is undefined do
16.   if all generators \(A\) of \(B\) have defined \(\text{ann}(A, B)\) then
17.      if all \(\text{ann}_\text{data}(A) = Y\) then
18.         set \(\text{ann}_\text{data}(B) := Y\);
19.      else if all \(\text{ann}_\text{data}(A) = N\) then
20.         set \(\text{ann}_\text{data}(B) := N\);
21.      else
22.         add to \(D_v\) a new element type \(B'\) and a production rule \(B' \rightarrow P(B)\)
23.         set \(\sigma(A, B') := B\);
24.         set \(\text{ann}_\text{data}(B) = Y\), \(\text{ann}_\text{data}(B') = N\);
25.      for all element types \(C\) occurring in \(P(B)\) do
26.         set \(\sigma(B', C) := \sigma(B, C)\);
27.         set \(\text{ann}(B', C) := \text{ann}(B, C)\);
28.      end for
29.      for all generators \(A\) of \(B\) do
30.         if \(\text{ann}(A) = N\) then
31.            replace \(B\) with \(B'\) in \(P(A)\)
32.         end if
33.      end for
34.   end if
35. end if
36. end while

Figure 4.6: Algorithm Annotate View
4.3. VIEW CONSTRUCTION FOR TOP-DOWN POLICIES  

and the “copy” with \( N \) (emulation of an invisible node), which is connected to parents transmitting to it \( N \). The function \( \sigma \) between parents and newly created nodes are simply the name of the split element. This is because invisible copy will be deleted, while the visible one will be considered as the original one. The newly created nodes transmit their visibility to their children.

**Example 12:** Elements \( application_Y \) and \( application_N \) transmit, respectively, \( Y \) and \( N \) to \( student-data \). The latter should be split to visible and invisible copies, i.e. \( student_data_Y \) and \( student_data_N \), children of \( annotation_Y \) and \( annotation_N \) respectively. The \( \sigma \)-function in both cases is equal to \( student − data \). The newly created elements are connected to all children of \( student-data \) and transmit to them corresponding visibility. However, since \( \text{ann}(student − data, department)=Y \), which is more specific, \( Y \) overrides negative visibility transmitted by \( student − data_N \).

An overall algorithm for top-down policy propagation is shown in Fig. 4.6. In particular, lines 5–14 of ANNOTATE VIEW algorithm perform splitting nodes when removing qualifier (Def. 16). Steps 15–36 perform a top-down propagation with splitting (lines 21–34). The result of ANNOTATE VIEW execution is a fully annotated DTD.

**Definition 17:** DTD is called *fully annotated* if for every DTD node \( A \), there is a defined function \( \text{ann}_{data}(A) := Y \mid N \). The function \( \text{ann}_{data} \) is called *full annotation* of the DTD document.

Having obtained a fully annotated DTD, we delete all the element types that are labelled \( N \), modifying the regular expressions and the \( \sigma \) functions accordingly. This is shown in the BUILD VIEW algorithm in Fig. 4.7.
Algorithm Build View

Input: Fully annotated DTD $D$

Output: A security view $(D_v, \sigma)$

1: for all element types $B$ with $\text{ann}(B) = N$ do
2:  for all production rules $A \rightarrow P(A)$ do
3:    if $B$ occurs in $P(A)$ then
4:      for all $C$ that occurs in $P(B)$ do
5:        set $\sigma(A, C) := \sigma(A, B)/\sigma(B, C) \cup \sigma(A, C)$
6:      end for
7:    replace $B$ by $P(B)$ in $P(A)$ if $B \rightarrow P(B)$ exists and by $\epsilon$ otherwise
8:  end if
9: end for
10: each element $B_Y$ rename to $B$;

Figure 4.7: Algorithm Build View

Example 13: In Example 12, $\text{department}$ is reachable via two paths:

\[ p_1 = \text{applications/application}_Y[\text{student-data/id = $\text{login}$}] / \text{student} - \text{data}_Y/\text{department} \]
\[ p_2 = \text{applications/application}_N[\neg(\text{student-data/id = $\text{login}$})] / \text{student} - \text{data}_N/\text{department} \]

In the path $p_2$, there are two elements that should be deleted: $\text{application}_N$ and $\text{student} - \text{data}_N$. Since all permitted children should be connected to all permitted parents, an additional edge between $\text{applications}$ and $\text{department}$ is created and

\[ \sigma(\text{applications, department}) = \text{application}[\neg(\text{student-data/id = $\text{login}$})]/\text{student-data}/\text{department} \]

that is exactly $XP_5$ from Example 8. \hfill \square
In the previous section, we showed how to construct the view for the top-down policy. In the case of local policy, we suppose that \( \text{ann}(A, B) \) is an annotation between parent \( A \) and its child \( B \). Therefore, pushing of security labels is performed in a top-down manner. This approach assures that there will not be any conflicts at XML tree since every node \( B \) will have only one parent \( A \), i.e. only one generator. Hence, we can consider local policy to be subsumed by top-down policy. Note that we can push annotation bottom-up from children to a parent. However, in this case, the VC option must be defined. Consequently, local policy class will be subsumed by bottom-up policy. Finally, the multilabel policy requires application of both local and hierarchical (top-down or bottom-up) policies. Therefore, we say that multilabel policy is also subsumed by top-down and bottom-up policies and in this section, we can restrict the attention to bottom-up policy only.

First, the definition of authorization specification is extended as follows: all generators of the authorization specification are of one type: either top-down or bottom-up edges.

**Definition 18:** Element types \( A \) and \( B \) of DTD \( D \) are called adjacent if one of the following statements is true: (i) \( B \in P(A) \) of a production rule \( A \rightarrow P(A) \), or (ii) \( A \in P(B) \) of a production rule \( B \rightarrow P(B) \).

In the case (i), a DTD edge \( (A, B) \) is a top-down edge; otherwise, it is a bottom-up edge.

We then can generalize the semantics of qualifiers at the DTD level as follows:

**Definition 19:** The semantics of \( \text{ann}(s, d) = Q[q] \) is a splitting of node type
d into two ones having the following meaning: visible node instance of type d is visible if child(s, d)[q] holds, and invisible otherwise, where child(s, d) is a function that for generator (s, d) that returns a child element type ch = s ∨ d with respect to a DTD structure, i.e., for some element type U ≠ ch, U ∈ {s, d}, DTD should have a production rule U → P(U) such that ch ∈ P(U)².

From the top-down view point, the destination d is visible if d[q] (d is a child) holds. On the other hand, from the bottom-up view point, d is visible if s[q] (s is a child) holds. The last interpretation of qualifier seems strange but it is required in such a form because σ-function evaluation and view materialization is held in a top-down manner. Namely, σ(A, B) = Q means that Q should be evaluated at B; in the XML view, B children of A are extracted according to Q.

As in the top-down approach we push security annotations along edges, but from children to parents. In this case, we must take into an account, that an analogous operation in the XML tree for any destination may have multiple sources.

Example 14: Consider the DTD from Example 1.
Generators (student-data, name) and (evaluator, name) differently transmit annotation to name in the sense that a node of type name may have a parent of type student−data or evaluator. Hence, these generators cannot influence on ann_data(name) simultaneously. If we consider a bottom-up case, generators (title, evaluator) and (institution, evaluator) influence on ann_data(evaluator) simultaneously because any node of type evaluator has both children of types institution and title in any XML instance. However, this is not a case for generators (MS, rating) and (PhD, rating) influencing on ann_data(rating) non-simultaneously as long as a node of type rating has

²In symmetric way we may introduce function parent(s, d) that returns parent element type w.r.t. DTD structure for a pair (s, d)
either a child of type MS or a child of type PhD.

Suppose, we are given an authorization specification \((D, \text{ann})\). We denote the set of all generators of \(d\) as \(G(d)\).

**Definition 20:** We say that a subset of \(G(d)\) denoted as \(\overline{G}(d)\) has a simultaneous influence on \(\text{ann}_{\text{data}}(d)\) if there exists \(T\) conforming to \(D\) such that every instance of type \(d\) has a set of either outgoing or incoming edges corresponding to the set \(G(d)\). We call \(\overline{G}(d)\) a set of simultaneous influence (SSI).

As we have seen above, a node may have several SSIs. This means that different nodes of the same element type in the same XML document may have different full annotation \(\text{ann}_{\text{data}}\). Hence, we split an element w.r.t. a set of sources rather than a single source.

**Definition 21:** We say that \(d\) obtains a preliminary full annotation (PFA) from SSI \(\overline{G}(d)\), denoted \(\text{ann}_{\text{data}}(d)_{\overline{G}(d)}\), if for every \(g \in \overline{G}(d)\), \(\text{ann}(g)\) it is the same for every \(g \in \overline{G}(d)\), \(\text{ann}(g) \neq \emptyset\), and \(\text{ann}(g) \neq Q[q]\).

The notion of PFA is introduced (i) to reflect the possibility of a node obtaining either a positive or a negative full annotation, and (ii) to take into an account the fact that any node may have multiple sources. Obviously, if \(\text{ann}(g)\) is the same for all \(g \in \overline{G}(d)\) then \(\text{ann}_{\text{data}}(d)_{\overline{G}(d)} = \text{ann}(g)\). Otherwise, to resolve a conflict within an SSI, we use VC resolution option if it is defined \(^3\). Value conflict may arise only in the case of bottom-up policy class because every XML instance usually contains a node having more than one child.

In the definition of PFA, we required that \(\text{ann}(g) \neq Q[q]\). This is because the first step of the algorithm should be removing qualifier according to Def. 19.

\(^3\)If VC option is not defined, the view does not exist
Let us consider an example of removing qualifiers while propagating security annotations bottom-up. First of all, as in top-down case, $d_Y$ and $d_N$ transmit, respectively, $Y$ and $N$ to destinations $d'$ of $d$ if $\text{ann}(d, d') = \emptyset$; otherwise, they transmit $\text{ann}(d, d')$ to $d'$. As Fig. 4.4 shows, unreliable is split into unreliable$_Y$ and unreliable$_N$ both transmitting $N$ to application because of the original element unreliable transmitting $N$. Otherwise, $\text{ann}(\text{unreliable}_N, \text{application})$ and $\text{ann}(\text{unreliable}_Y, \text{application})$ should have been $N$ and $Y$ respectively.

Next, if policy is bottom-up, $d$ is substituted with $d_Y + d_N$ in production rule of every $d'$s parent, because $d$ is split conditionally under all its parents. On the other hand, this substitution is done in only one production rule $s \rightarrow P(s)$ if the policy is top-down. This is because only $s$ has a child $d$ for which $d[q]$ either holds or does not. Regarding the other sources $s'$ different from a considered $s$, there is a common rule for top-down and bottom-up propagations: $d_N$ should be connected with the sources transmitting $N$ while $d_Y$ with all the others. After refining generators transmitting
4.4. SCHEMA-LEVEL ACCESS FOR BOTTOM-UP POLICIES

CHAPTER 4. SCHEMA ACCESS CONTROL

qualifiers or nothing, we may need to split \( d_Y \) again.

**Example 15:** In Fig. 4.4, \( \text{unreliable}_Y \) is connected to both its sources, since \( \text{recomm} - \text{letter} \) transmits \( Y \) due to qualifier evaluation, \( \text{reason} \) transmits nothing. On the other hand, \( \text{unreliable}_N \) is connected only to \( \text{recomm} - \text{letter} \) that transmits \( N \). □

The \( \sigma \)-function between \( d_Y \) and \( d_N \) and their source transmitting the qualifier should be \( d[q] \) and \( d[-q] \) respectively. For the other sources \( s' \), \( \sigma = \text{child}(s', d) \). The \( \sigma \)-function for \( (d, d') \) is \( \text{child}(d, d') \).

**Example 16:** In Fig. 4.4, we show in boxes a \( \sigma \) between the newly created nodes and their sources that are children. Note that now \( \text{ann}(\text{recomm} - \text{letter}, \text{unreliable}_Y) = Y \) and \( \text{ann}(\text{recomm} - \text{letter}, \text{unreliable}_N) = N \) as in top-down case. □

The procedure for removing qualifier is depicted in Fig. 4.9. Cycle FOR starting in line 3 implements the attachment of \( d_Y \) and \( d_N \) to parents of \( d \). In lines 11 and 5, the choice \( d_Y + d_N \) substitutes \( d \) in the case of **top-down** and **bottom-up** policy respectively. Lines 14 and 15 are the beginning of subroutines to connect \( d_Y \) and \( d_N \) with source \( s \) and other sources \( p \neq s \) respectively.

Having removed qualifiers, we can define SSIs. For top-down propagation, SSIs contain only one generator (parent-child DTD edge), and the number of SSIs is equal to the number of parents in DTD graph. The situation is more complicated in bottom-up propagation. First of all, every destination element type \( d \) may have several children in the DTD graph transmitting their security labels to \( d \). Secondly, the number of SSIs and their components depend on the presence of choices \( (\alpha + \alpha) \) in \( P(A) \). More precisely, if we present every sequence \( (\alpha, \alpha) \) of \( P(A) \) as an arithmetic product \( (\alpha \times \alpha) \), and every choice \( (\alpha + \alpha) \) of \( P(A) \) as an arithmetic sum in parenthesis, then the precise number of SSIs and their configuration is, re-
Algorithm Qualifier Removing

**Input:** Partially annotated DTD with qualifiers, policy class PC

**Output:** Partially annotated DTD without qualifiers

1. for every generator \((s, d)\) such that \(\text{ann}(s, d) = Q[q]\) do
2. Create element types \(d_Y\) and \(d_N\);
3. for all destinations \(d'\) of \(d\) do
4. Connect \(d_Y\) and \(d_N\) with \(d'\):

\[
\begin{align*}
\sigma(\text{parent}(d', d_Y), \text{child}(d', d_Y)) &= \sigma(\text{parent}(d', d), \text{child}(d', d)); \\
\sigma(\text{parent}(d', d_N), \text{child}(d', d_N)) &= \sigma(\text{parent}(d', d), \text{child}(d', d)); \\
\text{ann}(d_Y, d') &= \text{ann}(d, d') = \text{ann}(d, d'), \text{if}\ \text{ann}(d, d') \neq \emptyset; \\
\text{ann}(d_Y, d') &= Y; \text{ann}(d_N, d') = N, \text{if} \ \text{ann}(d, d') = \emptyset;
\end{align*}
\]

5. if policy is bottom-up then
6. for every parent \(d'\) of \(d\) do
7. substitute \(d\) with \(d_Y + d_N\) in production rule \(d' \rightarrow P(d')\);
8. end for
9. end if
10. if policy is top-down then
11. Substitute \(d\) for \(d_Y + d_N\) in production rule \(s \rightarrow P(s)\);
12. end if
13. Connect \(d_Y\) and \(d_N\) with \(s\):

\[
\begin{align*}
\sigma(\text{parent}(s, d_Y), \text{child}(s, d_Y)) &= \text{child}(s, d)[q]; \\
\sigma(\text{parent}(s, d_N), \text{child}(s, d_N)) &= \text{child}(s, d)[\neg q]; \\
\text{ann}(s, d_Y) &= Y; \text{ann}(s, d_N) = N;
\end{align*}
\]

15. Connect \(d_Y\) (\(d_N\) respectively) with other sources \(s' \neq s\) transmitting \(Y|Q||\text{nothing}\) (\(N\) respectively)

\[
\begin{align*}
\sigma(\text{parent}(s', d_Y), \text{child}(s', d_Y)) &= \sigma(\text{parent}(s', d), \text{child}(s', d)) \\
(\sigma(\text{parent}(p, d_N), \text{child}(p, d_N)) &= \sigma(\text{parent}(p, d), \text{child}(p, d)) \text{ respectively}); \\
\text{ann}(s', d_Y) &= \text{ann}(s', d); \ (\text{ann}(s', d_N) = \text{ann}(s', d)\text{ respectively});
\end{align*}
\]

16. end for
Algorithm Split

Input: DTD element type \( d \) having generators with different annotations

1: Create element types \( d_Y \) and \( d_N \);
2: for every SSI \( \overline{G_k}(d)(k = \overline{1,n}) \) having sources \( \{s_1, \ldots, s_{m_k}\} \) and resulting in a preliminary full annotation \( Y \) or \( N \) of \( d \) do
3: Connect source \( s_i \) of every generator \( g_i \in \overline{G_k}(d), i = \overline{1,m_k}, \) respectively, with \( d_Y \) or \( d_N \) setting:
   \[
   \sigma(\text{parent}(s_i, d_Y), \text{child}(s_i, d_Y)) = \text{child}(s_i, d) = \sigma(\text{parent}(s_i, d_N), \text{child}(s_i, d_N))
   \]
   \[
   \text{ann}(s_i, d_Y) = \text{ann}(s_i, d)(= Y); \text{ann}(s_i, d_N) = \text{ann}(s_i, d)(= N);
   \]
4: end for
5: for every generator \( g' = (d, d') \) where \( d \) is a source do
6: Connect \( d_Y \) and \( d_N \) with \( d' \) setting:
   \[
   \sigma(\text{parent}(d', d_Y), \text{child}(d', d_Y)) = \text{child}(d', d) = \sigma(\text{parent}(d', d_N), \text{child}(d', d_N))
   \]
   \[
   \text{ann}(d_Y, d') = \text{ann}(d, d') = \text{ann}(d_N, d');
   \]
7: end for

Figure 4.10: Algorithm Split

spectively, the number of components and multipliers in every component
in the resulting arithmetic expression after removal of parenthesis.

Example 17: Arithmetic representation of production rule
\( \text{rating} \rightarrow ( \text{English}, (\text{MS} + \text{PhD}) ) \) is \( \text{rating} = ( \text{English} \times (\text{MS} + \text{PhD}) ) \)
which is equal to \( \text{English} \times \text{MS} + \text{English} \times \text{PhD} \) after removal of parenthesis. Therefore, in the case of bottom-up propagation, \( \text{rating} \) has two SSIs: \( S_1 = \{\text{English}, \text{MS}\} \) and \( S_2 = \{\text{English}, \text{PhD}\} \).

Next, for every SSI, we calculate a preliminary full annotation using
the VC option if necessary (e.g., in the case of bottom-up policy class).
If different SSIs deliver to \( d \) different annotations, we perform the same
splitting operation as in the case of qualifier removed.

Example 18: Suppose \( \text{ann}(\text{English}, \text{rating})=\text{ann}(\text{MS}, \text{rating})=Y, \)
ann(PhD, rating) = N. Then, following the previous example, $S_1$ results in a positive PFA (i.e., $Y$), while PFA of $S_2$ is conflicting (both $Y$ and $N$ are transmitted to rating) and depends on VC.

Finally, if all PFAs are the same, then $\text{ann}_{\text{data}}$ of the destination node is clearly defined. Otherwise, as we said above, we perform a splitting operation w.r.t. SSIs transmitting different PFAs.

**Example 19:** If in the previous example $\text{VC} = \text{denialTakesPrecedence}$, we have different PFAs: $Y$ from $S_1$ and $N$ from $S_2$. Consequently, we split rating into a visible and an invisible version. Otherwise, we assign $Y$ as $\text{ann}_{\text{data}}(\text{rating})$.

The generic splitting algorithm (valid for top-down and bottom-up policies) is shown in Fig. 4.10.

We assume that every DTD element type $e$ that is required to be initially annotated (like root or all leaves for bottom-up propagation) automatically retransmits its annotation to all generators $g = (e, d')$ such that $\text{ann}(g) = \emptyset$.

The generic algorithm ANNOTATE VIEW is shown in Fig 4.11. It starts with a preprocessing procedure which is needed only for the local policy. After preprocessing and qualifier removal steps, we use a queue: if the next considered element type $d$ has a full annotation $\text{ann}_{\text{data}}(d)$, there is no need to process it; otherwise, the algorithm returns to line 2. If all generators of $d$ are annotated, then we decide $\text{ann}_{\text{data}}(d)$. Otherwise, we place $d$ back to queue (step 19).

### 4.5 Theoretical Results

**Theorem 6:** Let $(D, \text{ann})$ be a security specification where $D$ is non-recursive. Algorithms terminate and produce valid security views.

**Proof:** First, we prove that the algorithms. Indeed, in ANNOTATE
Algorithm Annotate View

**Input:** Partially annotated DTD $D$

**Output:** Fully annotated DTD

1: Preprocessing;
2: Qualifier Removing;
3: Create empty $queue$, initialize it with all DTD element types;
4: while $queue$ is not empty do
5: $d := \text{DEQUEUE}(queue)$;
6: if $\text{ann}_{\text{data}}(d) = \emptyset$ then
7: if $d$ belongs to all generators with defined $\text{ann}$ then
8: Calculate SSIs $\{G_1(d), G_2(d), \ldots, G_n(d)\}$;
9: for every $G_i(d)$ do
10: Calculate $\text{ann}_{\text{data}}(d)_{G_i(d)}$ (applying value conflict resolution policy option if not for all $g \in G_i(d)$ $\text{ann}(g)$ is the same);
11: end for
12: if all PFAs of $d$ are the same ($Y$ or $N$) then
13: Assign any $\text{ann}_{\text{data}}(d)_{G_i(d)}$ to $\text{ann}_{\text{data}}(d)$;
14: else
15: $\text{SPLIT}(d)$;
16: end if
17: For every $d' \in P_{\text{ann}}(d)$ such that $\text{ann}(d, d') = \emptyset$, set $\text{ann}(d, d') = \text{ann}_{\text{data}}(d)$;
18: else
19: $\text{ENQUEUE}(queue, d)$;
20: end if
21: end if
22: end while

Figure 4.11: Algorithm Annotate View

VIEW, there are two sources of iteration: the first is step 2 which terminates because the number of qualifiers is finite, the second one is the cycle while starting in line 4 that extends the annotation to a “full” one where $\text{ann}_{\text{data}}$ is defined as either $Y$ or $N$ for every element type. Suppose that the algorithm never reaches the state when $queue$ is empty (i.e., does not terminate). It may happen if every element of queue is expecting a full
Consider one such an element type $e$ having a source $s_{k+1}$ such that $\text{ann}_{\text{data}}(s_{k+1}) = \emptyset$ which is also *expecting*. Inductively, we suppose that $s_{k+1}$ has an expecting source $s_{k+2}$, etc. Therefore, in the DTD graph, there exists either an infinite path or a cycle $s_{k+n}, \ldots, s_{k+2}, s_{k+1}, e, s_{k+n}$ of expecting element types that is a contradiction. Therefore, this while terminates.

The algorithm BUILD VIEW always terminates because a fully annotated DTD contains a finite set of $\mathbb{N}$-labelled nodes. Hence, the first step in BUILD VIEW always reduces the number of element types in the DTD by one.

Secondly, we show that $D_v$ is a DTD. $D_v$ would fail to be a DTD only if, for some element type $A \in D_v$, there exists $B \in P(A)$ such that $B$ was deleted in step 7 of BUILD VIEW. Since $B$ is deleted by BUILD VIEW $\text{ann}_{\text{data}}(B)$ in a fully annotated DTD $D_F$ must be equal to $\mathbb{N}$, and therefore $B$ is replaced by $P(B)$ in step 7 of BUILD VIEW. Hence, $B$ cannot occur in $P(A)$, a contradiction. As we are considering only non-recursive DTDs, we must also show that the new DTD is non-recursive. But this follows immediately, as any cycle $D_v$ can be traced back to a cycle in $D$.

Finally, we prove that the resulting security view is valid. For this purpose, we must show that $T_M$ conforms to $D_v$. To do this, we first examine $T_F$ which is the fully annotated version of $T$, and $D_F$ which is the fully annotated DTD. At this point, we would like to show that $T_F$ conforms to $D_F$, but there is a problem, namely that some of the nodes in $T_F$ should be typed by element types in $D_F$ because of removing qualifier and splitting. To have them typed appropriately, we extend the notion of typing so that the new types will also match the corresponding old type from which they are generated. Namely, we allow each new element type

---

4Without the loss of generality, we may consider the state of *queue* when all nodes having $\text{ann}_{\text{data}}$ are deleted at step 5.
4.5. THEORETICAL RESULTS

By or to type the same nodes that were typed by \(B\). With this modified definition of typing, a node in \(T_F\) that is annotated \(N\) (resp. \(Y\)) will be typed by a type in \(D_F\) that is annotated \(N\) (resp. \(Y\)). Since all the new nodes are deleted at step 7 of Build View the new definition of typing reduces to the standard definition, completing the proof.

Now we need a technical lemma that will be used to prove Theorem 8.

**Lemma 7:** Let \((D, \text{ann})\) be a security specification where \(D\) is a not-recursive DTD and \((D_v, \sigma)\) is the security view that is constructed by Algorithms Annotate View and Build View, for any sequence of element types \(B_0 \ldots B_n\) in the fully annotated \(D\) such that \((i)\) \(B_{i+1}\) is a child type of \(B_i\) for \(i = 0 \ldots n - 1\), and \((ii)\) each \(B_i\) for \(i = 1 \ldots n - 1\) is annotated \(N\), there exists an XPath expression \(p\) and \(q_1 \ldots q_n\) XPath qualifiers such that the following equation holds for all set of nodes \(N\):

\[
S_\rightarrow [[\sigma(B_0, B_n)]](N) = S_\rightarrow [[p]](N) \cup S_\rightarrow [[B_1[q_1]/ \cdots / B_n[q_n]]](N)
\]

**Proof:** The proof is by a nested induction on \(n\) and the number of iterations of step 5 of algorithm Build View.

Base case: \(n = 1\), then \(B_1\) is a child of \(B_0\). There are two cases: (1) \(\text{ann}(B_0, B_1) = N\), and (2) \(\text{ann}(B_0, B_1) = Q[q]\). In the first case, \(\sigma(B_0, B_1) = B_1\) because of initialization step of Annotate View algorithm. In the second case, before step 5 of Build View is executed, algorithm Annotate View would set \(\sigma(B_0, B_1) = B_1[q_1]\) for a suitable qualifier \(q_1\). Therefore, up to this point, the theorem holds by setting \(p = B\), \(q_1 = \emptyset\) in the first case and \(p = \emptyset\) in the second case. During step 7 of algorithm Build View it is possible that the elimination of some \(N\)-children of \(B_0\) would modify \(\sigma(B_0, B_1)\). Namely, it may happen if \(B\) has a child \(C\) which, in its turn
has a child \( B_1 \). In this case, we get

\[
S\to [[\sigma(B_0, B_1)]](N) = S\to [[\sigma(B_0, C) / \sigma(C, B_1) \cup \sigma(B_0, B_1)]](N) =
\]

\[
S\to [[\sigma(B_0, C) / \sigma(C, B_1)](N) \cup S\to [[\sigma(B_0, B_1)]](N) =
\]

\[
S\to [[\sigma(B_0, C) / \sigma(C, B_1)](N) \cup S\to [[B_1[q_1]]](N) =
\]

\[
S\to [[p_1]](N) \cup S\to [[B_1[q_1]]](N)
\]

where \( q_1 \) may be \( \emptyset \). If \( B_1 \) itself is eliminated from \( P(B_0) \) this would not change the selection function constructed so far for \( B_1 \).

For the inductive case, let \( B_0 \ldots B_n \) be the sequence of nodes and let \( B_i \) for \( i \in \{1 \ldots n - 1\} \) be the last node that is eliminated by step 5 of the algorithm \textsc{Build View}. Since the DTD is not recursive neither \( \sigma(B_0, B_i) \), nor \( \sigma(B_i, B_n) \) can be changed by this step. Without the loss of generality, let \( \text{ann}(B_i, B_{i+1}) = Q[q] \) for some \( 0 \leq i < n \). We put \( q_{i+1} = \emptyset \) if needed. Then, by evaluating the \( S\to \) operator and by induction hypothesis we get:

\[
S\to [[\sigma(B_0, B_n)]](N) =
\]

\[
S\to [[\sigma(B_0, B_i) / \sigma(B_i, B_n)] \cup \sigma(B_0, B_n)](N) =
\]

\[
S\to [[\sigma(B_0, B_i) / \sigma(B_i, B_n)](N) \cup S\to [[\sigma(B_0, B_n)]](N) =
\]

\[
S\to [[\sigma(B_i, B_n)](S\to [[p_{1,i}]](N) \cup S\to [[B_1[q_1]/ \ldots / B_i[q_i]]](N)) \cup S\to [[p_{0}]](N) =
\]

\[
S\to [[\sigma(B_i, B_n)](S\to [[p_{1,i}]](N)) \cup S\to [[\sigma(B_i, B_n)](S\to [[B_1[q_1]/ \ldots / B_i[q_i]]](N)) \cup S\to [[p_{0}]](N) =
\]

\[
S\to [[p_{1}]](N) \cup S\to [[\sigma(B_i, B_n)](S\to [[B_1[q_1]/ \ldots / B_i[q_i]]](N)) \cup S\to [[p_{0}]](N) =
\]

\[
S\to [[p_{2}]](N) \cup S\to [[\sigma(B_i, B_n)](S\to [[B_1[q_1]/ \ldots / B_i[q_i]]](N)) =
\]

\[
S\to [[p_{2}]](N) \cup S\to [[p_{3}]](N) \cup S\to [[B_1[q_1]/ \ldots / B_i[q_i]/ B_{i+1}[q_{i+1}]/ \ldots / B_n[q_n]](N) =
\]

\[
S\to [[p]](N) \cup S\to [[B_1[q_1]/ \ldots / B_n[q_n]](N)
\]

The case \( i = n \) is similar to the above one by combining the reasoning for the base case and the intermediate case case above. \( \square \)

**Remark 4.1** In this lemma, there is no condition on the labelling of either \( B_0 \) or \( B_n \) as this would make the induction hypothesis needed for the proof
not strong enough. Equally we need to quantify over all sets $N$ or the composition of two intermediate sequences during the induction step would not have an inductive hypothesis strong enough.

**Theorem 8:** Let $(D, \text{ann})$ be an authorization specification, where $D$ is non-recursive, and $(D_v, \sigma)$ is the security view constructed by algorithms **Annotate View** and **Build View**. Let $T$ be a document, $T_A$ the authorized version of $T$ and $T_V$ the materialized version of $T$ with respect to $(D_v, \sigma)$. Then $T_A$ is isomorphic to $T_V$. \hfill $\square$

**Proof:** The proof is by top-down induction on $T$. The root of $T$ is clearly in both $T_A$ and $T_V$. By induction, assume that $n$ is of element type $A$, and is in both $T_A$ and $T_V$. We must show that each child $n$ in $T_A$ is also a child of $n$ in $T_V$, and vice versa.

$\implies$ Let $m$, of type $B$, be a child of $n$ in $T_A$. Assume, first, that $m$ is a child of $n$ in the original document $T$. Consider the fully annotated DTD $(D_F, \text{ann}_{\text{data}})$. Since $n$ is in $T_A$, $\text{ann}_{\text{data}}(A) = Y$. Since $m$ is in $T_A$, it follows that $\text{ann}_{\text{data}}(B) = Y$ as well, and so element type $B$ is in $D_v$; hence it is in $T_V$. Note, that in the case of top-down propagation if $\text{ann}(A, B) = Q[q]$, then $q$ must hold at $m$.

We must show that $m$ is in $S_\rightarrow [[\sigma(A, B)]] (\{n\})$. Let $p$ be a path between $n$ and $m$ in $T$. At the step of initialization, the algorithm **Annotate View** sets $p = B$. At the step of removing qualifier, $p = B$ may be replaced $p = B[q]$. Finally, step 5 of algorithm **Build View** may add additional disjuncts to $p$. In all cases $m$ is clearly in the result.

Now consider the case where $m$ is not a child, but a descendant, of $n$ in $T$. Let $n, n_1, \ldots, n_k, m$ ($k \geq 1$) be the sequence of nodes in $T$ from $n$ to $m$, of element types $B_1, \ldots, B_k$. (Next, we suppose that $B_0 = A, B_{k+1} = B$.) Since these nodes are not present in $T_A$, each $\text{ann}(B_{i-1}, B_i)$, $1 \leq i \leq k$ must be either undefined, $N$ or $Q[q_i]$, with the qualifier in the latter case
CHAPTER 4. SCHEMA ACCESS CONTROL 4.5. THEORETICAL RESULTS

evaluated to false at \( n \). Furthermore, \( \text{ann}_{\text{data}}(B) \) must be either \( \mathbf{Y} \) or a qualifier \( Q[q] \) that is evaluated to true at \( m \), which implies that \( B \) is in \( D_v \).

To show that \( m \) is accessible from \( n \) in \( T \) via path \( \sigma(A, B) \), observe first that \( D_F \) contains artificial element types that were obtained from \( B_j \) by splitting whenever. For this part of the proof, we shall write \( B_j' \) and \( B_j \) in the case when \( \text{ann}_{\text{data}}(B_j) = \mathbf{N} \) and \( \text{ann}_{\text{data}}(B_j) = \mathbf{Y} \) respectively. Whenever \( B_i \) inherits qualifier \( Q[q_i] \) via its sources, the step of qualifier removing of algorithm \textsc{Annotate View} initially sets \( \sigma(B_i-1, B_i) \) to \( B_i[\neg q_i] \); when \( B_i \) has a generator with \( \text{ann} N \) or undefined, \( \sigma(B_i-1, B_i) \) is initially set equal to \( B_i' \) in initialization step of \textsc{Annotate View}. Finally, step 7 of \textsc{Build View} deletes elements types \( B_1', \ldots, B_k' \), replacing \( \sigma(A, B) \) by a disjunction of paths, and by Lemma 7 we get:

\[
S \rightarrow [[\sigma(A, B)]|\{n\}] = S \rightarrow [[p \cup B_1[\neg q_1]/B_2[\neg q_2]/\cdots/B_k[\neg q_k]/B]|\{n\}]
\]

with some of the \( q_i \)'s absent, when \( \text{ann}(B_{i-1}, B_i) \) is \( \mathbf{N} \) or undefined. It follows that \( m \in S \rightarrow [[\sigma(A, B)]|\{n\}] \) (i.e., \( m \) is accessible from \( n \) in \( T \) via path \( \sigma(A, B) \)), as desired.

\( \iff \) For the converse, let \( m \) be a child of \( n \) in \( T_V \). We must show that \( m \) is a child of \( n \) in \( T_A \).

From the definition of \( T_V \), \( m \) must be in the result of evaluating \( \sigma(A, B) \) at \( n \). Let \( n = n_0, n_1, \ldots, n_k, m = n_{k+1} \) \((k \geq 0)\) be the shortest path from \( n \) to \( m \) that is used in the evaluation of the \( \sigma \) function. We claim that \( n_{i+1} \) is accessible from \( n_i \) in \( T \) via the path \( \sigma(B_i, B_{i+1}) \) \((0 \leq i \leq k, B_0 = A, B_{k+1} = B)\). Indeed, \( B_i \) is deleted in step 7 of \textsc{Build View} so that \( \sigma(B_{i-1}, B_{i+1}) \) is replaced by

\[
\sigma(B_{i-1}, B_i)/\sigma(B_i, B_{i+1}) + \sigma(B_{i-1}, B_{i+1})
\]

By our induction hypothesis, \( n_{i+1} \in S \rightarrow [[\sigma(B_{i-1}, B_{i+1})]|\{n_{i-1}\}] \). If \( n_{i+1} \) were in the second disjunct above, we would have a contradiction with
the assumption that our path was the shortest. Therefore \( n_{i+1} \) must be accessible from \( n_i \) in \( T \) by the path \( \sigma(B_i, B_{i+1}) \). Therefore for \( 0 < i < k+1 \), \( \sigma(B_{i-1}, B_i) \) is

1. \( B_i \) when \( \text{ann}(B_{i-1}, B_i) \) (\( \text{ann}(B_{i+1}, B_i) \) respectively) is either \( \text{N} \) or undefined. The case \( \text{ann}(B_{i-1}, B_i) = \text{Y} \) (\( \text{ann}(B_i, B_{i-1}) = \text{Y} \) respectively) is impossible except when \( i = k+1 \), as the element type \( B_i \) is absent in \( T_V \) and, consequently, is deleted in step 7 of \textsc{Build View}.

2. \( B_i[\neg q] \) when \( \text{ann}(B_{i-1}, B_i) \) is \( \text{Q}[q] \).

In both case, it follows that \( n, n_1, \ldots, n_k, m \) is a path in \( T \). It remains to show that \( n_1, \ldots, n_k \) are deleted in \( T_A \). For nodes inheriting a qualifier via some source, this is immediate; for the other nodes it follows from the fact that the algorithm used to define a complete annotation is the same in the definition of \( T_A \) and in Algorithm \textsc{Annotate View}. \( \square \)
Chapter 5

Query Rewriting Algorithm

So far, we have proposed a way to reveal accessible data for the user by means of view materialization. The materialized view is used for evaluation of user queries. There are two ways to implement query evaluation in the presence of materialized view. The first is to deliver a calculated view to the user who will query it locally. The second possible solution involves a client-server architecture where client is the user who wants to query the XML document in the presence of available data schema. Server stores the initial XML document and must satisfy requests of client to this document: server either stores every materialized view or recalculates a corresponding view on the fly every time client issues a query. In the former case as well as in local querying, the integrity maintenance becomes unfeasible. In the latter case, the view materialization can be very time- and memory-consuming that is unacceptable in the presence of many users requesting the database simultaneously.

An alternative is to use a query rewriting. Namely, the users queries are answered without view materialization: we rewrite a user query formulated at the DTD view into an equivalent query formulated at the original DTD using hidden rules and evaluate the resulting query over the original XML database.
The idea of query rewriting for privacy preserving is well-known and was presented for in relational databases \[66\], \[74\], \[75\]. These results were later extended to XML as well \[7\], \[28\], \[40\], \[54\], \[70\], \[65\].

In our case, the idea of query rewriting is to incorporate \(\sigma\)-function in the user’s query.

The algorithm for query rewriting has two phases: query parsing and further translation of the parsed query into \(\sigma\)-functions. Query parsing phase implies that user query is represented as a tree of subqueries (parse tree) according to the grammar that we have shown in Def. 3. The graphic representation of a generic parse tree is show in Fig. 5.1. Namely, basic block is one of \(\langle xpath\rangle\), \(\langle path\rangle\), \(\langle step\rangle\), \(\langle qualifier\rangle\), or \(\langle test\rangle\) which is \(\theta::\lambda\) where \(\lambda\) is either an element type name or an asterisk \(*\); optional part means that \(\langle xpath\rangle\) may consist of one or more \(\langle path\rangle\)s that may consist of one or more \(\langle step\rangle\)s that may include zero or more \(\langle qual\rangle\)s; extended basic block is related to a qualifier of the form \(\langle path\rangle=const\); expression is related to qualifiers including and, or, and not operators with operands as other qualifiers. If \(\langle xpath\rangle\) or \(\langle path\rangle\) occur in a qualifier, they are parsed as basic blocks according to Def. 3. Leaves of the parse tree are \(\langle test\rangle\)s.
The translation of the parsed query starts from the leaves of the parse tree and moves up to the root \( \langle xpath \rangle \). In particular, for each subquery \( p \) and some element \( A \), the algorithm calculates \( QR(p,A) \) using \( \text{QUERY REWRITE}(p_i, B_j) \), where \( p_i \) is a direct subquery (child in a parse tree) of \( p \) and \( B_j \) is a node reachable from \( A \) via \( p_i \) in \( D_v \). At the same time, the algorithm calculates \( \text{reach}(p,A) \) representing the set of nodes reachable from node \( A \) via the path \( p \). To obtain a rewriting of the initial user query \( q \), we invoke \( \text{QUERY REWRITE}(q, \text{root}) \).

The algorithm 5 shows the translation procedure. Lines 1, 4, 7, 10, 20 distinguish whether the subexpression is \( \langle xpath \rangle \), \( \langle path \rangle \), node with zero or more qualifiers, qualifier \( \langle qual \rangle \), or node test respectively.

Processing of \( xpath \). The processing of union of paths is shown in lines 1-3. The algorithm identifies the set of paths composing an \( xpath \): \( P = \{p_1, \ldots, p_m\} \), where each \( p_i \) is a \( \langle path \rangle \).

Processing of \( path \). In the case of \( \langle path \rangle \) (lines 4-6), first of all, we distinguish a set of steps. We assume that this set of steps can be enumerated according to occurrence of a step in the initial XPath expression \( p \). After that, we apply devide-and-conquer-like algorithm to process \( \langle path \rangle \). Namely, we recursively split the initial \( \langle path \rangle \) into two equal sub\( \langle path \rangle \)s with the same number (\( \pm 1 \)) of steps. Then these subpaths are rewritten and joined into one path which is a rewritten version of input \( \langle path \rangle \). More precisely, we look for all the nodes reachable from \( A \) by the first subpath \( (q_1) \), i.e. \( \text{reach}(q_1, A) \) (e.g. \( v_1 \) and \( v_2 \)). Subpath \( q_2 \) is rewritten w.r.t. to \( \text{reach}(q_1, A) \). At the same time, we calculate nodes reachable by \( q_2 \) (e.g. \( w_1, w_2, w_3 \) and \( w_4 \)). The latter will be added to a set of nodes reachable from \( A \) by \( q = q_1/q_2 \).

Processing of \( step \). A step may include qualifiers (lines 7–9) that are processed in the way similar to processing \( \langle path \rangle \): first of all, algorithm handles node test \( \langle test \rangle \) (analogue of the first subpath of the input \( \langle path \rangle \))
Algorithm Query Rewrite

**Input:** a parsed subquery \( q \), a node \( A \) for which query rewriting is carried

**Output:** rewritten subquery \( q \) w.r.t. \( A \) node

1. if \( q \) is \langle xpath \rangle then
   
   // \( q = p_1 \cup p_2 \cup \ldots \cup p_n \)
   
2. \( QR(q,A) := \bigcup_{p_i} \text{Query Rewrite}(p_i,A); \)
3. \( \text{reach}(q,A) := \bigcup_{p_i} \text{reach}(p_i,A); \)
4. else if \( q \) is \langle path \rangle then
   
   // \( q = \text{step}_1/\text{step}_2/\ldots/\text{step}_m; q_1 = \text{step}_1/\ldots/\text{step}_m/\text{step}_{m/2}; q_2 = \text{step}_{m/2+1}/\ldots/\text{step}_m \)
5. \( QR(q,A) := \text{Query Rewrite}(q_1,A) \cup \bigcup_{v \in \text{reach}(q_1,A)} \text{Query Rewrite}(q_2,v); \)
6. \( \text{reach}(q,A) := \text{reach}(q,A) \cup \bigcup_{v \in \text{reach}(q_1,A)} \text{reach}(q_2,v); \)
7. else if \( q \) is \langle step \rangle then
   
   // \( q \) is test[\text{qual}_1][\text{qual}_2]\ldots[\text{qual}_k]
8. \( QR(q,A) := \text{Query Rewrite}(\text{test},A) \cup \bigcup_{\text{qual}_i} \bigcup_{v \in \text{reach}(\text{test},A)} \text{Query Rewrite}(\text{qual}_i,v); \)
9. \( \text{reach}(q,A) := \text{reach}(\text{test},A); \)
10. else if \( q \) is \langle qual \rangle then
11.   if \( q \) is \langle xpath \rangle then
12.     \( QR(q,A) := \text{Query Rewrite}(q,A); \)
13.   else if \( q \) is \langle path \rangle = c then
14.     \( QR(q,A) := \text{Query Rewrite}(q,A) = c; \)
15.   else if \( q \) is not \langle qual \rangle then
16.     \( QR(q,A) := \text{not Query Rewrite}(\text{qual},A); \)
17.   else if \( q \) is \langle qual_1 \rangle \& \langle qual_2 \rangle or \langle qual_1 \rangle or \langle qual_2 \rangle then
18.     \( QR(q,A) := \text{Query Rewrite}(\text{qual}_1,A) \text{ op } \text{Query Rewrite}(\text{qual}_2,A); \)
     // where op is either and or or
19.     end if
20. else if \( q \) is \langle test \rangle then
21.     if \( \theta \) is ‘child’, ‘parent’, ‘descendant-or-self’, or ‘ancestor-or-self’ then
22.         \( QR(q,A) := \bigcup_v QR(\text{path}(A,v,A)); \)
23.         \( \text{reach}(q,A) := \{v\}, \) where \( v \) is an element \( \lambda \) reachable by axis \( \theta; \)
24.     else if \( \theta \) is ‘attribute’ then
25.         if \( A \) has attribute \( \lambda \) then
26.             \( QR(q,A) := q; \)
27.         end if
28.     end if
29.     end if
30. end if
31. return \( QR(q,A); \)

---

**Figure 5.2:** Algorithm Query Rewrite

66
and, then, every filter \(\langle \text{qual} \rangle\) (analogue of the second subpath of the input \(\langle \text{path} \rangle\)). The difference with the processing of \(\langle \text{path} \rangle\) is that a set of node reachable by \(q\) from \(A\) is defined by rewriting of \(\langle \text{test} \rangle\) w.r.t. \(A\).

**Processing of qualifiers.** Qualifier processing (in lines 10-18) is either an \(\langle \text{xpath} \rangle\), i.e. without operators (line 11); or \(\langle \text{path} \rangle\) compared with a constant; or boolean combination of qualifiers, i.e. with one (line 15) or two (line 17) operands. In the first two cases, the algorithm simply performs a rewriting as for \(\langle \text{xpath} \rangle\) and \(\langle \text{path} \rangle\) subquery respectively. In the latter two cases, we process a qualifier for unary *not* and binary *and* or *or* functions respectively.

**Processing of node test.** The processing of node test \(\langle \text{test} \rangle = \theta :: \lambda\) produces a path in terms of \(\sigma\) from each element \(A\) of \(D_v\) to \(\lambda\) according to axis \(\theta\). We have to mention that \(\lambda\) is not only a DTD element type name but also a wildcard \(*\). Therefore, we use \(\bigcup_{B \in \lambda}\) in the formulas below. Thus, we have the following possibilities in rewriting construction:

1. \(\theta\) is child: \(QR_{(\text{test}, A)} = \bigcup_{B \in \lambda} \sigma(A, \lambda)\), where \(B = \lambda\) if \(\lambda\) is a DTD element type name, and \(B\) is any node that is in relation \(\theta\) with \(A\) if \(\lambda\) is a wildcard \(*\);

2. \(\theta\) is parent: \(QR_{(\text{test}, A)} = \bigcup_{B \in \lambda} \sigma^{-1}(\lambda, A)\) (the definition of \(\sigma^{-1}(B, A)\) goes below);

3. \(\theta\) is descendant-or-self (ancestor-or-self): \(QR_{(\text{test}, A)}\) is assigned a union of all paths from \(A\) to \(\lambda\) (from \(\lambda\) to \(A\)) in \(D_v\) that are rewritten w.r.t. \(\sigma\) (\(\sigma^{-1}\))-function as it described above.

Now, we clarify the meaning of \(\sigma^{-1}(B, A)\). \(\sigma(B, A)\) is a collection of paths from \(B\) to \(A\) in the initial DTD \(D\) such that \(B\) is a parent of \(A\) in \(D_v\): \(\sigma(B, A) := \bigcup_{i=1}^{k} p_i\) where each

\[
p_i = c_{i_1}[f_{i_1}]/c_{i_2}[f_{i_2}]/\ldots/c_{i_n-1}[f_{i_{n-1}}]/c_{i_n}[f_{i_n}]
\]
with $c_{i_1}$ as the child of $B$ (since $p_i$ is applied to $B$), $c_{i_n}$ is $A$, each $c_i$ is a
child of $c_{i-1}$, $f_i$ is a filter expression for the node $c_i$. Then $\sigma^{-1}(B, A)$ is
defined as follows:

**Definition 22:** The reverse $\sigma^{-1}(B, A)$ of non empty $\sigma(B, A)$ is $\sigma^{-1}(B, A) := \\
\bigcup_{i=1}^{k} p_i^{-1}$ where each

$$p_i^{-1} = \text{self}:: c_{i_n}[f_{i_n}] / \text{parent}:: c_{i_n-1}[f_{i_n-1}] / \ldots$$

$$/ \text{parent}:: c_{i_1}[f_{i_1}] / \text{parent}:: B[\text{preRewrite}(\uparrow^*, B, \text{root})]$$

is applied to the node $A$.

An expression $\text{self}:: c_{i_n}[f_{i_n}]$ ensures that $\sigma^{-1}(B, A)$ is applied to a permitted $A$ node. Analogously, $\text{parent}:: B$ is filtered by [$\text{preRewrite}(\uparrow^*, B, \text{root})$]
to guarantee that the user cannot reveal any additional information like in the following example:

**Example 20:** Suppose, $D_v$ contains a fragment $B \rightarrow (C, A), C \rightarrow (D, A)$;
the user has access to $B$ and $A$, but $C$ and $D$ are visible only under some condition $Q_C$. The query $//A/parent::C/D$ may leak sensitive information if we do not restrict $C$ with an expression $[Q_C]$.

The algorithm of $\sigma^{-1}(B, A)$ calculation directly follows from Def. 22. We report it in Fig. 5.3.

The closest approach to query rewriting is presented by Fan et al. in [40], [42]. The main differences are:

1. Our algorithm derives a security view without any dummy element types which may be a source of sensitive information leakage. Therefore, the $\sigma$-function used in our query rewriting has different semantics.

2. An extended XPath fragment has $\text{parent}$ and $\text{descendant-or-self}$ axes.

3. Fan et al. use dynamic programming so that $QR_{(q,A)}$ is calculated for every DTD element type $A$; while we perform a rewriting of $q$ w.r.t.
CHAPTER 5. QUERY REWRITING ALGORITHM

Procedure getReverseσ

Input: elements A (child of B), B (parent of A) of D_v (as string), i.e. \( \sigma(B, A) = \bigcup_{i=1}^{k} p_i \) in D

Output: \( \sigma^{-1}(B, A) \)

1: reverse := ∅
2: for each \( p_i \) do
3: \( p := \text{parent} :: B' \);
4: if \( B \neq \text{root} \) then
5: \( p := p[\text{preRewrite}(↑ *, B, \text{root})] \);
6: end if
7: while \( p_i \neq \emptyset \) do
8: step := \( p_i\).getFirstStep();
9: \( p := \text{parent} :: step/p ; \)
10: \( p_i := p_i\).getRemainingSteps();
11: if \( p_i \neq \emptyset \) then
12: \( p := \text{self} :: step/p ; \)
13: // Otherwise, we have reached \( c_{i_n} \) which is A
14: // and should not be attached to \( \sigma^{-1}(B, A) \)
15: end if
16: end while
17: reverse := reverse ∪ \( p_i \);
18: end for
19: return reverse

Figure 5.3: Procedure getReverseσ

to a subset of relevant element types A of DTD in a recursive manner. Thus, memory is saved (in [40], memory can be always evaluated as \( \Omega(m \times |D|) \), where \( m \) is a number of all steps in the path including those in qualifiers. In our approach, there are cases when \( \Theta(m) \) of memory is required).

4. Moreover, we use divide-and-conquer approach that is faster than simple linear lookup proposed in [40] (takes \( O(n^2) \) time)

5. However, we do not consider recursive DTDs, while Fan at al. [40]
5.1. THEORETICAL RESULTS

6. Finally, we formally prove the correctness of our query rewriting algorithm in the next Section.

5.1 Theoretical Results

To prove the correctness of Query Rewrite algorithm we have to show that a result set that is obtained by evaluating user query over XML view is the same as a result set computed from evaluation of rewritten query over the original XML tree. More formally:

\[ S \rightarrow [q] (\{n\}_{T_A}) = S \rightarrow [q'] (\{n'\}_T) \]

where \( q \) is the original user query, \( q' \) is a rewritten query; \( n \) (\( n' \)) is a set of nodes from which evaluation of \( q \) (\( q' \)) starts in \( T_A(T) \).

Lemma 1: [child] If the XPath fragment contains only child then the algorithm is correct. \( \square \)

Proof: A user query has the form \( q = label_1/label_2/\ldots/label_{n-1}/label_n \), where \( label_i, i = 1, n \) are element names in \( D_v \). The algorithm decomposes \( q \) into a set of steps \( L = \{label_1, label_2, \ldots, label_{n-1}, label_n\} \). For each element \( label_i, i = 2, \ldots, n \) of \( L \), algorithm calculates the rewriting of a path \( label_{i-1}/label_i \) (\( \text{root}/label_1 \) if \( i = 1 \)). We claim that the rewriting of \( A/B \) is \( \sigma(A, B) \). To prove this, consider a node \( n \) of type \( A \) and its child \( m \) with type \( B \) in the authorized version of data (\( T_A \)). Now we show that for any node \( n \) of type \( A \):

\[ S \rightarrow [B] (n_{T_A}) = S \rightarrow [p] (n_T) \]

where \( p \) is a path between \( n \) and \( m \) in \( T \), \( p = \sigma(A, B) \). In other words, the node \( m \) accessible by the path \( B \) from the node \( n \) in \( T_A \) is accessible by the path \( p \) applied to the node \( n \) in \( T \).
CHAPTER 5. QUERY REWRITING ALGORITHM 5.1. THEORETICAL RESULTS

Let us consider any $m$ from $S_\rightarrow [[B]](\{n\}_{T_A})$, i.e. $m$ is a child of $n$ in $T_A$. Assume first, that $m$ is also a child of $n$ in the original document $T$. Consider the fully annotated DTD ($D_F$, $\text{ann}_{\text{data}}$). Since $n$ is in $T_A$, $\text{ann}_{\text{data}}(A) = Y$. Since $m$ is in $T_A$, it follows that $\text{ann}_{\text{data}}(B) = Y$. It means that either $\text{ann}(A, B) = Y$ or $\text{ann}(A, B) = Q[q]$. In the latter case, $q$ must hold at $m$.

- If $\text{ann}(A, B) = Y$ then $\sigma(A, B) = B$. On the other hand, since $m$ is a child of $n$ in $T$ then the path from $n$ to $m$ can be expressed by the XPath expression $p = B$. Thus, for every $m$ of type $B$, which is a child of $n$ of type $A$ such that $\text{ann}(A, B) = Y$, in $T$ we have $S_\rightarrow [[B]](\{n\}_{T_A}) \subseteq S_\rightarrow [[p]](\{n\}_T)$ (i.e. $m$ is necessarily in $S_\rightarrow [[p]](\{n\}_T)$), where $p = B = \sigma(A, B)$.

Now we prove the reverse. We show this fact by a contradiction. Suppose that there exists $m'$ of type $B$ and $m' \in S_\rightarrow [[p]](\{n\}_T)$ such that $m' \notin S_\rightarrow [[B]](\{n\}_{T_A})$. Then, either $\text{ann}(A, B) = Q[q]$ or $\text{ann}(A, B) = N$. But this contradicts with the assumption that $\text{ann}(A, B) = Y$. Therefore, we conclude that $S_\rightarrow [[B]](\{n\}_{T_A}) = S_\rightarrow [[p]](\{n\}_T)$.

- If $\text{ann}(A, B) = Q[q]$ then we set $\sigma(A, B)$ equal to $B[q]$ while removing qualifiers in Annotate View. Then, in Materialize), for the node $n$ of type $A$ we select from $T$ only those children $m$ of type $B$ that satisfy the condition $q$. This means that any node $m \in S_\rightarrow [[B]](\{n\}_{T_A})$ satisfies the condition $q$ in $T$. The latter means that the path $p = B[q]$ is the path between $n$ and $n$ in $T$, i.e. $m \in S_\rightarrow [[B[q]]](\{n\}_T) = S_\rightarrow [[\sigma(A, B)]](\{n\}_T)$.

Now we prove that $S_\rightarrow [[p]](\{n\}_T)$ contains only the elements of the set $S_\rightarrow [[B]](\{n\}_{T_A})$. Assume by contradiction that there exist $m'$ of type $B$ and $m' \in S_\rightarrow [[p]](\{n\}_T)$ such that $m' \notin S_\rightarrow [[\text{label}_i]](\{n\}_{T_A})$. Therefore, either $m'$ does not satisfy $q$ in $T$ or $\text{ann}(A, B) = N$. In
5.1. THEORETICAL RESULTS CHAPTER 5. QUERY REWRITING ALGORITHM

the former case, $m' \notin S_\rightarrow [|B|] (\{n\}_T)$ which contradicts our assumption about $m'$. The latter case contradicts the assumption that \texttt{ann}(A, B) = Q[q]. Therefore, we conclude that $S_\rightarrow [|B|] (\{n\}_{T_A}) = S_\rightarrow [|p|] (\{n\}_T).

We have determined the correctness of the algorithm when $m$ is a child of $n$ in $T$. Now, assume that $m$ is not a child but a descendant of $n$ in $T$. Let $n = n_0, n_1, \ldots, n_k, n_{k+1} = m$ ($k \geq 1$) be the sequence of nodes in $T$ from $n$ to $m$, of element types $A = B_0, B_1, \ldots, B_k, B_{k+1} = B$. Since the nodes $n_j, 0 < j < k + 1$ are not present in $T_A$, each $\texttt{ann}(B_{j-1}, B_j)$ ($1 \leq j \leq k$) must be either undefined, $N$ or $Q[q]$, with the qualifier in the latter case evaluating to false at $n_j$. Furthermore, $\texttt{ann}(B_k, B_{k+1})$ must be either $Y$ or a qualifier $Q[q]$ that evaluates to true at $m$, which implies that $B$ is in $D_v$.

To show that $m$ is in $S_\rightarrow [|\sigma(A, B)|] (\{n\}_T)$, observe first that $D_F$ contains element types $B_j'$ (for some $j$) whenever $\texttt{ann}(B_{j-1}, B_j)$ is undefined or is a qualifier. Whenever $\texttt{ann}(B_{j-1}, B_j)$ is $Q[q]$, the step removing qualifiers in \textsc{Annotate View} initially sets $\sigma(B_{j-1}', B_j')$ to $B_j[\neg q]$; when $\texttt{ann}(B_{j-1}, B_j) = N$ or is undefined, $\sigma(B_{j-1}', B_j')$ is initially set equal to $B_j$ in \textsc{Annotate View}. Finally, \textsc{Build View} deletes elements types $B_1', \ldots, B_k'$, replacing $\sigma(A, B)$ by a disjunction of paths. It follows that $m \in S_\rightarrow [|\sigma(A, B)|] (\{n\}_T)$, as desired.

Now we prove that $S_\rightarrow [|p|] (\{n\}_T)$ contains only the elements of $S_\rightarrow [|B|] (\{n\}_{T_A})$, where $p = \sigma(A, B)$. If, on the contrary, there exists $m'$ of type $B$ and $m' \in S_\rightarrow [|p|] (\{n\}_T)$ such that $m' \notin S_\rightarrow [|\text{label}_i|] (\{n\}_{T_A})$ then either $m'$ does not satisfy the path $p$ in $T$ or $\texttt{ann}(A, B) = N$. In the former case, $m' \notin S_\rightarrow [|p|] (\{n\}_T)$ which contradicts our assumption about $m'$. The latter conclusion is a contradiction with the assumption that $\texttt{ann}(A, B) = Q[q]$ or $Y$. Therefore, we conclude that $S_\rightarrow [|B|] (\{n\}_{T_A}) = S_\rightarrow [|p|] (\{n\}_T)$.

As we saw, the correct rewriting for an XPath expression $A/B$ (or equivalently $A/child::\text{label}_i$) is $\sigma(A, B)$ if $B$ is a child of $A$ in $D_v$. If $B$ is not
a child of $A$, the rewriting of $A/B$ does not exist, since $\sigma(A, B)$ is empty, and therefore should be set as $\emptyset$ and any user query containing substring $A/B$ is not valid.

Having constructed the rewritings for all subpaths of the form $label_{i-1}/label_i$, algorithm QUERY REWRITE backtracks. Hence, if we constructed a correct rewriting for $p_{1,k} = label_1/\ldots/label_k$ and for $p_{k+1,n} = label_{k+1}\ldots/label_n$, where $1 < k < n$, the rewriting is correct for the path $p = label_1/\ldots/label_n$.

Please note, that in the case $k = n-1$, $p_{k+1,n} = label_n$. and equal to rewriting of $p_{1,k}$ w.r.t. some node of type $A$ followed by a rewriting of $p_{k+1,n}$ w.r.t. nodes of types reachable by $p_{1,k}$ starting from $A$. More formally,

$$QR(p,A) = QR(p_{1,k},A)/QR(p_{k+1,n},reach(A,QR(p_{1,k},A)))$$

Let us prove it by induction.

**Basis of induction:** $QR(label_1/label_2,root) = QR(label_1,root)/QR(label_2,reach(root,QR(label_1,root)))$ is a correct rewriting. Since we do not consider $\ast$, this means that $reach(root,QR(label_1,root)) = label_1$ and the following must hold:

$$S \rightarrow [||label_1/label_2|| \{w\}_{T_A}] = S \rightarrow [||QR(label_1,root)/QR(label_2,label_1)|| \{w\}_{T}]$$

where $w$ is a node of type $root$.

Indeed, let $v$ be a node of type $label_1$ and a child of $w$, $u$ be a node of
type \textit{label}_2 and a child of \textit{v}. Then we have that:

\[
S \rightarrow [\text{\textit{label}}_1/\text{\textit{label}}_2] (\{w\}_{T_A}) = \]

[According to the semantics of XPath, where XPath is \textit{label}_1/\textit{label}_2]

\[
S \rightarrow [\text{\textit{label}}_2] (\{S \rightarrow [\text{\textit{label}}_1] (\{w\}_{T_A})\}_{T_A}) = \]

[According to the above proven correctness of \textit{A/B} rewriting]

\[
S \rightarrow [\text{\textit{label}}_2] (\{S \rightarrow [\sigma(\text{\textit{root}}, \text{\textit{label}}_1)] (\{w\}_{T})\}_{T_A}) = \]

\[
[S \rightarrow [\sigma(\text{\textit{root}}, \text{\textit{label}}_1)] (\{w\}_{T}) \text{ is a node } \text{\textit{v}} \text{ of type } \text{\textit{label}}_2 \text{ in } T] \]

[According to the semantics of XPath where XPath is \textit{\sigma(\text{\textit{root}}, \text{\textit{label}}_1)}/\textit{\sigma(\text{\textit{label}}_1, \text{\textit{label}}_2)} ]

\[
S \rightarrow [\sigma(\text{\textit{root}}, \text{\textit{label}}_1)/\sigma(\text{\textit{label}}_1, \text{\textit{label}}_2)] (\{w\}_{T}) = \]

[according to \textit{QUERY REWRITE}, this is exactly]

\[
S \rightarrow [QR(\text{\textit{label}}_1, \text{\textit{root}})/QR(\text{\textit{label}}_2, \text{\textit{label}}_1)] (\{w\}_{T})
\]

**Inductive assumption** Assume that for \textit{w} of type \textit{root}

\[
S \rightarrow [\text{\textit{label}}_1/\ldots/\text{\textit{label}}_n] (\text{\textit{w}}_{T_A}) = S \rightarrow [QR(\text{\textit{label}}_1/\ldots/\text{\textit{label}}_n, \text{\textit{root}})] (\text{\textit{w}}_{T})
\]

**Inductive step** We show that

\[
S \rightarrow [\text{\textit{label}}_1/\ldots/\text{\textit{label}}_n/\text{\textit{label}}_{n+1}] (\text{\textit{w}}_{T_A}) = S \rightarrow [QR(\text{\textit{label}}_1/\ldots/\text{\textit{label}}_n, \text{\textit{root}})/QR(\text{\textit{label}}_{n+1}, \text{\textit{label}}_n)] (\text{\textit{w}}_{T})
\]

Indeed

\[
S \rightarrow [\text{\textit{label}}_1/\ldots/\text{\textit{label}}_{n+1}] (\{w\}_{T_A}) = \]

\[
S \rightarrow [\text{\textit{label}}_{n+1}] (\{S \rightarrow [\text{\textit{label}}_1/\ldots/\text{\textit{label}}_n] (\{w\}_{T_A})\}_{T_A}) = \]

\[
S \rightarrow [\text{\textit{label}}_{n+1}] (\{S \rightarrow [QR(\text{\textit{label}}_1/\ldots/\text{\textit{label}}_n, \text{\textit{root}})] (\text{\textit{w}}_{T_A})\}) = \]

\[
S \rightarrow [QR(\text{\textit{label}}_1/\ldots/\text{\textit{label}}_n, \text{\textit{root}})/\sigma(\text{\textit{label}}_{n+1}, \text{\textit{label}}_n)] (\{w\}_{T}) = \]

\[
S \rightarrow [QR(\text{\textit{label}}_1/\ldots/\text{\textit{label}}_n, \text{\textit{root}})/QR(\text{\textit{label}}_{n+1}, \text{\textit{label}}_n)] (\text{\textit{w}}_{T})
\]

\[\square\]

**Remark 5.1** Inductive step of the Lemma 1 may be proven not only for \textit{label}_{n+1} but for a path of an arbitrary length. Therefore, dividing \langle \textit{path} \rangle at
CHAPTER 5. QUERY REWRITING ALGORITHM

5.1. THEORETICAL RESULTS

step\(_{[m/2]}\) is correct. Furthermore, the correctness of the algorithm remains valid independently on the place of division. However, dividing at two equal subpaths simply improves the performance of query rewriting since parse tree has a lesser depth.

**Lemma 2:** [child, parent] If XPath fragment contains \{child, parent\} then the algorithm is correct. □

**Proof:** Suppose that some \(step_i\) of the user query has axis parent, i.e. it is \(step_i = parent :: label_i\). Namely, \(p = label_1/\ldots/label_{i-1}/parent :: label_i/label_{i+1}/\ldots/label_n\). We assume that we apply a divide operation of our divide-and-conquer algorithm at step \(i-1\). Hence, we have to show that \(QR_{(p,root)} = Q_1/Q_2\) where \(Q_1 = QR_{(p_{i-1},root)}\), \(Q_2 = QR_{(parent::label_i/p_{i+1,n},label_{i-1})}\). The correctness of \(Q_1\) follows from Lemma 1. Namely, \(S_\rightarrow [\{p_{1,i-1}\}] (root_{T_A}) = S_\rightarrow [\{Q_1\}] (root_T)\) and is equal to \(\{n\}\), the last set of nodes \(n\) of type \(label_{i-1}\) (i.e., \(reach(root, p_{1,i-1})\)) is equal both in \(T_A\) and in \(T\). Therefore, \(p_{i,n}\) should be rewritten w.r.t. \(label_{i-1}\). Now we consider \(Q_2\). We start from evaluation of \(p_{i,n}\) in \(T_A\) w.r.t. \(\{n\}\), set of nodes of type \(label_{i-1}\):

\[
S_\rightarrow [\{parent :: label_i/label_{i+1}/\ldots/label_n\} (\{n\}_{T_A})]
\]

[According to XPath semantics]
\[
S_\rightarrow [\{label_{i+1}/\ldots/label_n\}] (S_\rightarrow [\{parent :: label_i\} (\{n\}_{T_A})])_{T_A}
\]
\[
S_\rightarrow [\{label_{i+2}/\ldots/label_n\}] (S_\rightarrow [\{label_{i+1}\}] (S_\rightarrow [\{parent :: label_i\} (\{n\}_{T_A})]))_{T_A}
\]

At this step, we have to show that \(QR_{(parent::label_i,label_{i-1})} = \sigma^{-1}(label_i, label_{i-1})\) is the correct rewriting, i.e.

\[
S_\rightarrow [\{parent :: label_i\}] (\{n\}_{T_A}) = S_\rightarrow [\{\sigma^{-1}(label_i, label_{i-1})\}] (\{n\}_{T})
\]

Let us consider any \(m\) of type \(label_i\) such that \(m \in S_\rightarrow [\{parent :: label_i\}] (\{n\}_{T_A})\), i.e. \(m\) is a parent of \(n\) in \(T_A\). Assume first, that \(m\) is also a parent of \(n\) in an original tree \(T\). Consider the fully annotated DTD schema \((D_F, \text{ann}_{data})\).
Since $n$ is in $T_A$, $\text{ann}_{\text{data}}(\text{label}_{i-1})=Y$. Since $m$ is in $T_A$, $\text{ann}_{\text{data}}(\text{label}_i)=Y$. It means that either $\text{ann}(\text{label}_i, \text{label}_{i-1})=Y$ or $\text{ann}(\text{label}_i, \text{label}_{i-1})=Q[q]$. In the latter case, $q$ must hold at $n$.

- If $\text{ann}(\text{label}_i, \text{label}_{i-1})=Y$, then there exists $\sigma(\text{label}_i, \text{label}_{i-1}) = \text{label}_{i-1}$ (it is established at the step of initialization of algorithm \textsc{Annotate View} and is never changed after) which should be applied to $\text{label}_i$. The calculation of $\sigma^{-1}(\text{label}_i, \text{label}_{i-1})$ results in the expression $\text{parent} :: \text{label}_i[QR(\text{path}(\text{root}, \text{label}_i), \text{root})]$ which should be applied to node $\text{label}_{i-1}$, where $\text{path}(A, B)$ is a path from $A$ to $B$ expressed in $\{\downarrow\}$ in $D_v$. On the other hand, path from $n$ to $m$ can be expressed as $\text{parent}::\text{label}_i$ and $\text{label}_i$ should be a permitted parent in $T$, i.e., it should be extended with the qualifier $[QR(\text{path}(\text{root}, \text{label}_i), \text{root})]$. The necessity of this qualifier was discussed above. In a nutshell, $\text{label}_i$ may be both permitted and forbidden depending on the position in the XML tree. Hence, $\text{label}_i$ is visible in $D_v$ and, accessible under certain conditions in $T$. However, if instance of $\text{label}_i$ is not accessible, it may still be used in the rewriting of the user query fragment that precedes to $\text{step}_i$. Consequently, a smart user can try to access a forbidden $\text{label}_i$ via $\text{parent}$ axis. However, with the above mentioned extension, the user will fail.

Therefore, for every node $m$ of type $\text{label}_i$, which is a parent of node $n$ of type $\text{label}_{i-1}$ and $\text{ann}(\text{label}_i, \text{label}_{i-1})=Y$, we have that $S_- [\text{parent} :: \text{label}_i] (\{n\}_{T_A}) \subseteq S_- [\sigma^{-1}(\text{label}_i, \text{label}_{i-1})] (\{n\}_{T})$ (i.e. $m$ is necessarily in $S_- [\sigma^{-1}(\text{label}_i, \text{label}_{i-1})] (\{n\}_{T})$).

Now we prove that $S_- [\sigma^{-1}(\text{label}_i, \text{label}_{i-1})] (\{n\}_{T})$ contains only elements of the set $S_- [\text{parent} :: \text{label}_i] (\{n\}_{T_A})$. We show this fact by a negative proof, i.e we suppose that there exist $m'$ of type $\text{label}_i$ and $m \in S_- [\sigma^{-1}(\text{label}_i, \text{label}_{i-1})] (\{n\}_{T})$ (i.e. $m'$ is a parent of $n$ in $T$)
such that $m' \notin S_\rightarrow [[parent :: label_i]] (\{n\}_{T_A})$ (i.e. $m'$ is not a parent of $n$ in $T_A$). As a consequence, either $\text{ann}(label_i, label_{i-1}) = Q[q]$ or $\text{ann}(label_i, label_{i-1}) = N$. But the latter is a contradiction with the assumption that $\text{ann}(label_i, label_{i-1}) = Y$. Therefore, we conclude that $S_\rightarrow [[parent :: label_i]] (\{n\}_{T_A}) = S_\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]] (\{n\}_{T})$.

- If $\text{ann}(label_i, label_{i-1}) = Q[q]$ then we set $\sigma(label_i, label_{i-1}) = label_{i-1}[q]$ at the step of elimination of qualifiers in algorithm Annotate View. At the process of building $T_A$ (algorithm Materialize), we select only those children $n$ of type $A$ that satisfy condition $q$. It means that for every node $n$ satisfying condition $q$ in $T_A$, $S_\rightarrow [[parent :: label_i]] (\{n\}_{T_A})$ contains nodes $m$ of type $nN$, i.e. $m$ is in the set $S_\rightarrow [[parent :: label_i]] (\{n\}_{T}) = S_\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]] (\{n\}_{T})$, where $n$ satisfies the condition $q$ in $T$.

Now we prove that $S_\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]] (\{n\}_{T})$ contains only the elements of the set $S_\rightarrow [[parent :: label_i]] (\{n\}_{T_A})$ and $n$ satisfies condition $q$ in $T$. On the contrary, we assume that there exists $m'$ of type $label_i$ and $m' \in S_\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]] (\{n\}_{T})$ (i.e. $m'$ is a parent of $n$ in $T$) such that $m' \notin S_\rightarrow [[parent :: label_i]] (\{n\}_{T_A})$ (i.e. $m'$ is not a parent of $n$ in $T_A$). Therefore, either $n$ does not satisfy the condition $q$ in $T$ or $\text{ann}(nN, A) = N$. In both cases, we have an obvious contradiction with the initial assumption that $\text{ann}(label_i, label_{i-1}) = Q[q]$ and $n$ satisfies condition $q$ in $T$. Therefore, we can conclude that $S_\rightarrow [[parent :: label_i]] (\{n\}_{T_A}) = S_\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]] (\{n\}_{T})$.

Heretofore, we have determined that the algorithm is correct when $m$ is a parent of $n$ in $T$. Now let us assume that $m$ is not a parent but an ancestor of $n$ in $T$. Let $m = n_0, n_1, \ldots, n_k, n_{k+1} = n$ ($k \geq 1$) be the sequence of nodes in $T$ from $m$ to $n$ of element types $label_i = B_0, B_1, \ldots, B_k, B_{k+1} = label_{i-1}$. Since the nodes $n_j, 0 < j < k + 1$ are not present in $T_A$, each
ann\((B_j, B_{j-1})\) (1 \geq j \geq k) must be either undefined, N, or Q[q], with the qualifier evaluating, in the latter case, to false at \(n_j\). Furthermore, ann\((B_k, label_{i-1})\) must be either Y or qualifier Q[q] that evaluates to true at \(n\), which implies that \(label_{i-1}\) in \(D_v\) and its instance is in \(T_A\).

To show that \(m\) is in \(S\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]]\) \(\{n\}_T\), observe first that \(D_F\) contains element types \(B_{j}'\) (for some \(j\)) whenever ann\((B_{j-1}, B_j)\) is either undefined or a qualifier. In the latter case, the step of qualifiers elimination in the algorithm ANNOTATE VIEW initially sets \(\sigma(B_{j-1}', B_j')\) to \(B_j[q]\) (respectively, \(\sigma^{-1}(B_{j-1}', B_j') = parent::B_{j-1}\) should be applied to node of type \(B_j\) satisfying condition \(q\) in \(T\), i.e. if \(B_j\) occurs in \(\sigma^{-1}(label_i, label_{i-1})\), it has be accompanied by qualifier \([q]\)); when ann\((B_{j-1}, B_j)\) = N or is undefined, ann\((B_{j-1}, B_j)\) is set to \(B_j\) at the step of initialization of ANNOTATE VIEW (respectively, \(\sigma^{-1}(B_{j-1}, B_j) = parent::B_{j-1}\) should be applied to node of type \(B_j\)). Finally, algorithm BUILD VIEW deletes element types \(B_1', \ldots, B_k'\) replacing \(\sigma(label_i, label_{i-1})\) by a disjunction of paths (the construction of \(\sigma^{-1}(label_i, label_{i-1})\) goes in the reverse direction using \(\sigma^{-1}\) of deleted edges; reverse direction means that rather than append deleted information to the existing expression, algorithm append existing expression to the deleted information). It follows that \(m \in S\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]]\) \(\{n\}_T\) as desired.

Now we prove that \(S\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]]\) \(\{n\}_T\) contains only the elements of the set \(S\rightarrow [[parent :: label_i]]\) \(\{n\}_{T_A}\). If, on the contrary, there exist \(m'\) of type \(label_i\) and \(m' \in S\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]]\) \(\{n\}_T\) (i.e. \(m'\) is an ancestor of \(n\) in \(T\)) such that \(m' \notin S\rightarrow [[parent :: label_i]]\) \(\{n\}_{T_A}\) (i.e. \(m'\) is not a parent of \(n\) in \(T_A\)) then either \(m'\) does not satisfy path \(\sigma^{-1}(label_i, label_{i-1})\) in \(T\) or ann\((label_i, label_{i-1})\) = N. In the former case, \(m' \notin S\rightarrow [[\sigma^{-1}(label_i, label_{i-1})]]\) \(\{n\}_T\) that is a contradiction with our assumption about \(m'\). The latter conclusion is a contradiction with the assumption that ann\((label_i, label_{i-1})\) = Q[q] or Y. Therefore, we conclude
that $S \rightarrow [\text{parent} :: label_i] (\{n\}_T)$

$S \rightarrow [\sigma^{-1}(label_i, label_{i-1})] (\{n\}_T)$.

As we saw, the correct rewriting for an XPath expression $label_{i-1}/\text{parent}::label_i$ is $\sigma^{-1}(label_i, label_{i-1})$ if $label_{i-1}$ is a child of $label_i$ in $D_v$. If it is not so, the rewriting of $label_{i-1}/\text{parent}::label_1$ does not exist, since $\sigma(label_i, label_{i-1})$ does not exist, and therefore should be set as $\emptyset$ and any user query containing substring $label_{i-1}/\text{parent}::label_i$ should be considered as erroneous.

Now we return to expression:

$S \rightarrow [\text{label}_{i+2}/\ldots/\text{label}_n] (\{S \rightarrow [\text{parent} :: label_i] (\{n\}_T)\})_{T_A} =$

$S \rightarrow [\text{label}_{i+2}/\ldots/\text{label}_n] (\{\sigma^{-1}(label_i, label_{i-1})\})_{T_A} =$

[According to Lemma 1]

$S \rightarrow [\sigma^{-1}(label_i, label_{i-1})/QR(label_{i+1}/\ldots/label_n, label_i)] (\{n\}_T)$

Here $\sigma^{-1}(label_i, label_{i-1})/QR(label_{i+1}/\ldots/label_n, label_i) = QR(p_{i,n}, label_{i-1})$ which is $Q_2$.

Finally $QR(p_{1,n},\text{root})$ is obtained by merging $Q_1$ and $Q_2$, which is, being proven by induction as in Lemma 1, a correct rewriting.

Lemma 3: [child, parent, $\cup$] If XPath fragment contains $\{\text{child, parent, } \cup\}$ then the algorithm is correct.

Proof: User query is a chain of steps. Some steps represent children, e.g. $\text{child} :: label_i$, or parents, e.g., $\text{parent} :: label_j$, others are union of paths formulated in the XPath fragment $\{\downarrow, \uparrow, \cup\}$, e.g. instead of $label_x$, user query $q$ may contain step $p_1 \cup p_2 \cup \ldots \cup p_m$, where each $p_k, 1 \leq k \leq m$ is a chain of children and union of paths. As in the previous case, algorithm transforms user query into set of steps. If a step is a node test (i.e. $label$), we process it as in the case of the XPath fragment $\{\downarrow, \uparrow\}$. Otherwise, if a step is a union of paths, algorithm identifies the set of these paths $P = \{p_1, \ldots, p_m\}$ and initiates recursive transformation of each element of $P$ into a set of steps. The proof of the lemma is similar to that of the previous
one. The difference is that at some point algorithms tries to rewrite either $label_i/\bigcup_{j=1}^m p_j$ or $\bigcup_{j=1}^m p_j/label_i$. However, before proving that a rewriting for such an expression is correct, we will show that $QR(\bigcup_{j=1}^m p_j, A)$, is equal to $\bigcup_{j=1}^m QR(p_j, A)$. More formally, we have to show that

$$S \rightarrow \left[ |\bigcup_{j=1}^m p_j| \right] (\{w\}_{TA}) = \bigcup_{j=1}^m S \rightarrow [|p_j|] (\{w\}_{TA})$$

where $w$ is a node of type $A$.

Indeed, if the union of paths contains nested unions, we continue parsing. At some stage of parsing nested unions of paths, we reach the point when union of paths contains only those paths that are expressed in the XPath fragment \{↓, ↑\}. For such paths $p_i$, the expression $S \rightarrow [|p_i|] (\{w\}_{TA})$, according to Lemma 1, is equal to $S \rightarrow [|QR(p_j, A)|] (\{w\}_T)$, where $QR(p_j, A)$ is a rewriting of an expression $A/p_i$. Therefore,

$$\bigcup_{j=1}^m S \rightarrow [|p_j|] (\{w\}_{TA}) = \bigcup_{j=1}^m S \rightarrow [|QR(p_j, A)|] (\{w\}_T)$$

On the other hand, the expression $S \rightarrow \left[ |\bigcup_{j=1}^m p_j| \right] (\{w\}_{TA})$ represents a set of nodes accessible from node $w$ of type $A$ by $\bigcup_{j=1}^m p_j$ in $T_A$. The same set of nodes may be obtained if we make a union of sets $S_j = S \rightarrow [|p_j|] (\{w\}_T)$, i.e. sets of nodes accessible from node $w$ of type $A$ by $p_j$ in $T_A$. Since we suppose that $p_j$ is an XPath expression in fragment \{\}, $S_j = S \rightarrow [|QR(p_j, A)|] (\{w\}_T)$ according to the Lemma 1. Summarizing all the written above, we may conclude that $rw(\bigcup_{j=1}^m p_j, A) = \bigcup_{j=1}^m QR(p_j, A)$. Therefore

$$S \rightarrow \left[ |\bigcup_{j=1}^m p_j| \right] (\{w\}_{TA}) = \bigcup_{j=1}^m S \rightarrow [|QR(A,p_j)|] (\{w\}_T)$$

Now we consider the backtracking of a recursion, i.e. merging stage. Recall that merging happens after rewritings are recursively calculated for right and left subpaths of the original path. For a path $step_1/\ldots/step_n$, $step_k$ is a union of paths for some $k \in [1..n]$. We have to show that

$$S \rightarrow [|step_1/\ldots/step_n|] (\{w\}_{TA}) = S \rightarrow [|QR(step_1/\ldots/step_n,root)|] (\{w\}_T)$$
where $w$ is a node of type root.

This immediately follows from the property proven above in this lemma and from inductive unfolding of $S_\rightarrow [[step_1/.../step_n]] \{w \}_T$ according to XPath semantics.

**Lemma 4:** If XPath fragment contains \{child, parent, *\} then the algorithm is correct.

**Proof:** User query $q$ is a chain of node tests with parent or child axis. However, some step of the chain may have * instead of node name, e.g. $label_i$. As above, algorithm transforms user query into a set of node names used in $q$ including *. The processing of an XPath expression $A/\ast$ occurs in the following way.

1. Find all nodes $B_1, B_2, \ldots, B_n$ that are children or parents (depends on axis before *) of $A$;

2. Transform expression $A/\ast$ into $A/(\theta :: B_1 \cup \theta :: B_2 \cup \ldots \theta :: cupB_n)$;

3. Perform rewriting of the new expression.

The correctness of algorithm for expression constructed in XPath fragment \{↓, ↑, *\} was proved in Lemma 3.

The previous Lemma immediately results in

**Corollary 5:** If XPath fragment contains \{child, parent, ∪, *\} then rewritten query produced by the algorithm QUERY REWRITE extracts from the initial data tree $T$ the same answer set as the answer set of the user query $q$ over authorized tree $T_A$.

**Lemma 6:** If XPath fragment contains \{child, parent, ∪, *, descendant-or-self\} then the algorithm is correct.

**Proof:** Consider a user query $p$ where some $step_i$ is descendant-or-self :: $label_i$. As in the proof of Lemma 2, we consider the rewriting $Q_{(p, root)}$.
5.1. THEORETICAL RESULTS

CHAPTER 5. QUERY REWRITING ALGORITHM

and prove that it is equal to $Q_1/Q_2$, where $Q_1 = QR_{(p_{1,i-1}, \text{root})}$, $Q_2 = QR_{(p_{i,n}, \text{reach}(\text{root}, Q_1))}$. The correctness of $Q_1$ then follows from Lemmas proven above. Regarding $Q_2$, we apply formulas of XPath semantics and find out that:

$$S_\rightarrow \left[ [\text{descendant-or-self}:: \text{label}_i/\text{step}_i/\ldots/\text{step}_i] \left( \{n\}_{T_A} \right) \right] = S_\rightarrow \left[ [\text{step}_{i+2}/\ldots/\text{step}_n] \left( \left\{ S_\rightarrow \left[ [\text{step}_{i+1}] \left( \left\{ S_\rightarrow \left[ [\text{descendant-or-self}:: \text{label}_i] \left( \{n\}_{T_A} \} \right) \right] \right) \right] \right) \right) \right]_{T_A}$$

where $n$ is a node of type $\text{label}_{i-1}$ if $\text{label}_{i-1}$ is not *. Otherwise, it is a set of nodes $S_\rightarrow [[Q_1]](\text{root})$ of different types $\text{reach}(\text{root}, Q_1)$ in $T$ and, equivalently, $\text{reach}(\text{root}, p_{1,i-1})$ in $T_A$.

For every element $A \in \text{reach}(\text{root}, Q_1)$, we construct a rewriting of $\text{descendant-or-self}:: \text{label}_i$ in the following manner: for every element type matching $\text{label}_i$ ($\text{label}_i$ may be *), we construct a union of all possible paths from $A$ to $\text{label}_i$ in $D_v$. Since these paths are expressed as a sequence of children, $QR_{(\text{descendant-or-self}:: \text{label}_i, A)}$ is correct according to Corollary 5. Hence $Q_2$ is a correct rewriting. Finally, we obtain the rewriting of the initial query by merging $Q_1$ and $Q_2$ which is also correct because of Corollary 5.

\[
\text{Lemma 7:} [\text{child}, \text{parent}, \cup, *, \text{ancestor-or-self}] \text{ If XPath fragment contains }
\{\text{child}, \text{parent}, \cup, *, \text{ancestor-or-self}\} \text{ then the algorithm is correct.} \]

\[
\text{Proof: } \text{Consider a user query } p \text{ where some step}_i \text{ is ancestor-or-self :: label}_i. \text{ As in the proof of Lemma 2, we consider the rewriting } Q_{(p, \text{root})} \text{ and prove that it is equal to } Q_1/Q_2, \text{ where } Q_1 = QR_{(p_{1,i-1}, \text{root})}, \text{ } Q_2 = QR_{(p_{i,n}, \text{reach}(\text{root}, Q_1))}. \text{ The correctness of } Q_1 \text{ then follows from Lemmas proven above. Regarding } Q_2, \text{ we apply formulas of XPath semantics and find out that:}
\]

$$S_\rightarrow [[\text{ancestor-or-self}:: \text{label}_i/\text{step}_i/\ldots/\text{step}_i] \left( \{n\}_{T_A} \right) \right] = S_\rightarrow [[\text{step}_{i+2}/\ldots/\text{step}_n] \left( \left\{ S_\rightarrow \left[ [\text{step}_{i+1}] \left( \left\{ S_\rightarrow \left[ [\text{ancestor-or-self}:: \text{label}_i] \left( \{n\}_{T_A} \} \right) \right] \right) \right] \right) \right) \right]_{T_A}$$

where $n$ is a node of type $\text{label}_{i-1}$ if $\text{label}_{i-1}$ is not *. Otherwise, it is a set of nodes $S_\rightarrow [[Q_1]](\text{root})$ of different types $\text{reach}(\text{root}, Q_1)$ in $T$ and, equivalently, $\text{reach}(\text{root}, p_{1,i-1})$ in $T_A$. 

82
CHAPTER 5. QUERY REWRITING ALGORITHM 5.1. THEORETICAL RESULTS

For every element \( A \in \text{reach}(\text{root}, Q_1) \), we construct a rewriting of \( \text{ancestor-or-self} :: \text{label}_i \) in the following manner: for every element type matching \( \text{label}_i \) (\( \text{label}_i \) may be \( * \)), we construct a union of all possible paths from \( A \) to \( \text{label}_i \) in \( D_v \). Since these paths are expressed as a sequence of accessible parents (i.e., each parent should be extended with a qualifier \([QR(path(root,\nu),\text{root})]\), where \( \nu \) is an element type on the path from \( A \) to \( \text{label}_i \)), \( QR(\text{ancestor-or-self}::\text{label}_i,A) \) is correct according to Corollary 5. Hence \( Q_2 \) is a correct rewriting. Finally, we obtain the rewriting of the initial query by merging \( Q_1 \) and \( Q_2 \) which is also correct because of Corollary 5.

From Lemmas 6 and 7 immediately follows

**Corollary 8:** \([\text{child}, \text{parent}, \cup, *, \text{descendant-or-self}, \text{ancestor-or-self}] \) If XPath fragment contains \{↓, ↑, ∪, *, ↓ ∗, ↑ ∗} then the algorithm is correct. □

**Theorem 9:** \([\text{child}, \text{parent}, \cup, *, \text{descendant-or-self}, \text{ancestor-or-self}, [],] \) If XPath fragment contains \{\text{child, parent, } ∪, *, \text{descendant-or-self, ancestor-or-self,=[],}]\} then rewritten query produced by the algorithm QUERY REWRITE extracts from the initial data tree \( T \) the same answer set as the answer set of the user query \( q \) over authorized tree \( T_A \). □

**Proof:** User query \( q \) is a chain of children. Each step may be extended by a set of qualifiers. As we said above, node with filters is processed in the analogous mode as \( \langle \text{path} \rangle \) expression, i.e. processing of node test (analogue of the left subpath), processing of every predicate (analogue of the right subpath), joining of processed parts. Consider that our path is \( \text{step}_1/.../\text{step}_k[\text{filter}]/.../\text{step}_n \). We have to prove that \( QR(\text{step}_1/.../\text{step}_k[\text{filter}]/.../\text{step}_n,\text{root}) = Q_1[Q_f]/Q_{\text{end}} \), where \( Q_1 = QR(\text{step}_1/.../\text{step}_k,\text{root}) \); \( Q_f = QR(\text{filter},\text{reach(root,Q_1)}) \); \( Q_{\text{end}} = QR(\text{step}_{k+1}/.../\text{step}_n,\text{reach(root,Q_1)}) \).

The correctness of \( Q_1 \), \( Q_f \), and \( Q_{\text{end}} \) follows from lemmas above. The correctness of the overall rewriting follows from unfolding according XPath
5.1. THEORETICAL RESULTS CHAPTER 5. QUERY REWRITING ALGORITHM

semantics.

At this point we are interested in boolean qualifiers. According to the semantics of XPath on Fig. 2.5, or in qualifier means a union of disjuncts, i.e. $\theta :: \lambda[\text{qual}_1 \text{ or } \text{qual}_2]$, can be represented as $\theta :: \lambda[\text{qual}_1] \cup \theta :: \lambda[\text{qual}_2]$. Using the semantics of union, the latter can be rewritten as $\theta :: \lambda[\text{qual}_1 \cup \text{qual}_2]$. The resulting filter represents a union of paths if both $\text{qual}_1$ and $\text{qual}_2$ is an expression of type $\langle \text{path} \rangle$. Therefore, the correctness is proved.

The node with and-expression, i.e. $\theta :: \lambda[\text{qual}_1 \text{ and } \text{qual}_2]$, can be rewritten as $\theta :: \lambda[\text{qual}_1][\text{qual}_2]$ (see the semantics of and operator on Fig. 2.5). Therefore, this case is reduced to the observations regarding general processing of node test with qualifiers.

Finally, rewriting of qualifiers with comparison operators, where $\langle \text{path} \rangle$ is the left part of comparison, while the right part of comparison is constant (literal or number), has the form:

$$QR(\theta::\lambda[\text{path op } c],A) = QR(\theta::\lambda,A)/\bigcup_{v \in \text{reach}(\theta,A)}[QR((\text{path op } c),v)] =$$

$$QR(\theta::\lambda,A)/\bigcup_{v \in \text{reach}(\theta,A)}[QR(\text{path},v) \text{ op } c]$$

The correctness of the last expression follows from the semantics of evaluation of qualifiers with operator $\text{op}$. □

**Theorem 10:** The time complexity of algorithm Query Rewrite is $O(|D_v|^2 \times \log m + 1)$, where $m$ is the number of steps in the user query, including those that are in qualifiers. □

**Proof:** If a set of steps of $\langle \text{path} \rangle$ can be represented as an array then we can divide that array in two equal subarrays (one contains the left subpath, another - the right one), we can recursively rewrite subpath corresponding to those subarrays and join them in $O(n)$ time, where $n$ is a number of nodes reachable by the left subpath from root node. If the path contains the reverse parent axes, the inversion of a corresponding $\sigma$-function will take $O(k)$, where $k$ is the number of steps in $\sigma$-function. Hence, the overall
complexity will be no more than $O((n \times k \times \log m + 1)$, where $m$ is a number of steps in path including those in qualifiers plus root from which the query starts. Since, $n$ and $k$ do not exceed $|D|$, the complexity of algorithm does not exceed $O(|D_v|^2 \times \log (m + 1))$.

**Remark 5.2** If a query does not contain reverse axes, the complexity of query rewriting is $O(|D_v| \times \log (m + 1))$.

An example of DTD for which the upper bound of the previous theorem is achievable is the following: root has $s$ children $A_i, i = 1, \ldots, s$. Each $A_i$ has $s$ children $B_j, j = 1, \ldots, s$. Security annotation is the following: $\text{ann}(\text{root}, A_i) = \mathbb{N}, i = 1, \ldots, s$, and $\text{ann}(A_i, B_j) = \mathbb{Y}, i = 1, \ldots, s; j = 1, \ldots, s$. Then, $D_v$ is root $\rightarrow B_j, j = 1, \ldots, s$ with $\sigma(\text{root}, B_j) = \bigcup_{i=1}^{s} A_i/B_j$ of the length $2 \times s$. Now assume that $s = |D|/2$. The query $*$ will be rewritten as follows: since in $D_v$, there are $|D|/2$ element types reachable from root (they are $B_j, j = 1, \ldots, |D|/2$), and for each of them $\sigma(\text{function})$ is reversed in $O(2 \times |D|/2)$. Therefore, overall complexity is $O(|D|^2)$. Since $m = 1$, logarithmic factor is 1.
5.1. THEORETICAL RESULTS CHAPTER 5. QUERY REWRITING ALGORITHM
Chapter 6

Implementation

We have developed a Java library that implements our algorithms of policy classification, policy propagation, view construction and query rewriting. It also implements functions of query evaluation both for view materialization and query rewriting approaches.

6.1 Architecture

Our client-server architecture is shown in Fig. 6.1, where rectangular boxes represent processes, cylinders stands for repositories, arrows (with labels) mean communication links (transmitting specific data).

![Figure 6.1: An overall architecture of the system](image-url)
More precisely, the client is the user who issues XPath queries to the server. In its turn, the server stores the XML files, their DTD schemas with annotations, policies for annotation propagation, and user profiles.

The user should be recognized by an authentication module that is outside the scope of this thesis. Existing authentication mechanisms [77] are adequate for this purpose. Once the user is authenticated and classified, he/she may make authorization requests.

As soon as the server receives a user query, an authorization process takes place, i.e., either the XML view is materialized or the query is rewritten. After that, the query is answered by evaluation over either the XML view or the original XML document, respectively. Finally, query evaluation module returns an answer set to the user.

6.2 Class diagrams

Our library uses different external packages. The package diagram is shown in Fig. 6.2, where arrows show import and usage dependencies among packages. The following packages are used in our system:

1. **WutkaDTD** is borrowed from [93]. It implements a DTD parser that reads any DTD file, scans it and constructs a DTD structure compliant with the specification of DTD [98], i.e., it can represent any DTD,
including non-normalized ones.

2. **SecureDTD** extends Wutka DTD with security properties, i.e., it contains classes related to security annotation definition, parsing, assignment, and propagation. It includes classes related to policy according to which annotation is propagated. It also implements \( \sigma \)-function that is modified along with the process of the DTD view creation. Finally, it implements algorithms **POLICY CLASS**, **BUILD VIEW**, **ANNOTATE VIEW**.

3. The **XML** package is used to deal with XML parsing, validation, and XML view materialization, i.e. algorithm **MATHEMATICIZE** is implemented by the classes belonging to this package.

4. **XPath** is a package related to XPath parsing and representation as an object according to XPath (Def. 3). We borrowed this package from SiXPath project [106].

5. **Query** contains classes that reuse the objects created by the classes of the previous package for rewriting queries. It implements algorithm **QUERY Rewrite**.

6. **EUtil** contains common constants and a class with an auxiliary static function that instantiates the variable \$login with the login name of the user.

### 6.2.1 Core DTD

The class diagram of the Wutka DTD structure is shown in Fig. 6.3. Arrows headed with black rows represent aggregation dependencies, while arrows ending with white triangle represent inheritance relations.
Basically, an object of \texttt{DTDParser} is invoked for reading a DTD document. That object scans the DTD document symbol by symbol with an object of \texttt{Scanner} class and stores the DTD as an object of \texttt{DTD} class.

A DTD object consists of \textit{elements} and \textit{items} organized as \texttt{Hashtable} and \texttt{Vector} respectively to store DTD elements and DTD containers (i.e., sequences or choices of element names, and simple DTD elements). In its turn, DTD element is represented as an object of \texttt{DTDElement} having \texttt{name}, \texttt{attributes}, and \texttt{content}. The last one is of \texttt{DTDItem} class, represents

Figure 6.3: Class diagram of Wutka DTD structure
the right part of the production rule $A \rightarrow P(A)$, and has only one field *cardinal* of DTDCardinal class that is used to reflect cardinality of children of $A$. There are four predefined objects of DTDCardinal class: NONE, OPTIONAL, ONEMANY, and ZEROMANY.

Each DTD attribute belongs to DTDAttribute class and has *name*, *defaultValue* and *decl*. The last one is of DTDDec1 class that corresponds to a possible declaration of attribute, i.e., FIXED, IMPLIED, REQUIRED, VALUE (i.e., CDATA).

As we said above, a DTD container represents an object of DTDItem class. The last one has five subclasses: DTDPCDATA, DTDEmpty, DTDAry, DTDName, and DTDCContainer. The first three are self-explainatory. DTDName is the smallest DTD container consisting of exactly one DTD element name, and it is used to construct more complex DTD containers of DTDCContainer class. In its turn, any DTDCContainer object may be a part of other DTDCContainer object, and so on. Since DTDName and DTDCContainer are subclasses of DTDItem, they may have their own cardinalities. Finally, each container can be sequence, choice, or mixed content. This is implemented by three subclasses of DTDCContainer: DTDSquence, DTDCChoice, DTDMixed.

### 6.2.2 Secure DTD

Package org.dtdsec enriches the core DTD package with security elements: policy, annotations, $\sigma$-function, policy propagation, etc. The class diagram of this package is show in Fig. 6.4.

We encode both policy and security annotations in an XML file, that conforms to the following DTD:

```xml
<!DOCTYPE security_configuration [
  <!ELEMENT security_configuration (policy, annotations)>]
<!ELEMENT policy (hierarchy_propagation, local_propagation, hierarchy_conflict, value_conflict)>
<!ELEMENT hierarchy_propagation (#PCDATA)>
<!ELEMENT local_propagation (#PCDATA)>
```
Figure 6.4: Class diagram of secure DTD structure
CHAPTER 6. IMPLEMENTATION 6.2. CLASS DIAGRAMS

An example of an annotation corresponding to such a DTD is:

```
<security_configuration>
  <policy>
    <hierarchy_propagation>topDown</hierarchy_propagation>
    <local_propagation>closed</local_propagation>
    <hierarchy_conflict>hierarchyFirst</hierarchy_conflict>
    <value_conflict>denialFirst</value_conflict>
  </policy>
  <annotations>
    <annotation>
      <nodeFrom>people</nodeFrom><nodeTo>person</nodeTo>
      <label>@id=$login</label>
    </annotation>
  </annotations>
</security_configuration>
```

An object representing the policy is of class Policy. Security annotations are stored as objects of DTDAnnotation class which is basically an array list of DTDAnnotationItem objects. The constructors of both classes parse their own part of the XML that encodes policy and annotations. For example, for Policy object construction, Policy parser queries nodes policy/*/text(). The result is compared with 12 values of all policy propagation options (they are encoded as private static final constants) and corresponding bytes (also constant) are assigned to variables with the prefix sp (see Fig. 6.4). Finally, identifyClass() is called for the policy object. It implements the POLICY CLASS algorithm and assigns a decided constant with the prefix iPC to the variable policyClass. Class Policy contains other procedures with the prefix is and get. They return true if a corresponding option does not equal none, and the value of the policy option respectively. We note that a PolicyException is thrown if the XML contains unrecognizable values of the policy propagation options (different from those defined in Chapter 3), or if the policy is unresolvable.
The DTDAnnotation parser simply recognizes the content of XML fields nodeFrom, nodeTo, annotation, constructs from them an object of DTDAnnotationItem class and adds it to an array list of annotations.

Both the policy and annotations are parts of the DTD. However, we could not simply implement a secure DTD as a subclass of Wutka DTD, because the Wutka DTD parser (including scanner and tokenizer) would have been rewritten from scratch. Therefore, we wrote wrappers for DTD, DTD elements, DTD items, and DTD containers. A wrapper of a class is a class that aggregates a wrapped class and contains other attributes and operations that can have access to attributes of wrapped class. For example, SecureDTD aggregates Wutka DTD, but it also includes objects of Policy, DTDAnnotation, Sigma classes. The functionality of SecureDTD is as follows:

getDTD() returns the core Wutka DTD;

getSecureElementByName(String name) returns SecureElement having name name from Hashmap elements;

applyPartialAnnotation() assigns items of DTDAnnotation to edges of SecureDTD;

annotateView() is an implementation of ANNOTATE VIEW;

buildView() is an implementation of BUILD VIEW;

getSigma() returns a Sigma object representing $\sigma$-function of the DTD;

cloneSecureDTDElement(SecureDTDElement element) returns SecureDTDElement having the same core Wutka DTDElement, but negative ann$_{data}$, that is propagated to its destinations; this function is common for the next two ones;
removeQualifier(SecureDTDElement element) connects the original element and the cloned SecureDTDElement with their sources and destinations according to algorithm QUALIFIER REMOVING and returns the new SecureDTDElement;

split(SecureDTDElement element) connects the original element and the cloned SecureDTDElement with their sources and destinations according to algorithm SPLIT and returns the new SecureDTDElement;

isChildOf(DTDElement destination, DTDElement source) is used in the previous two functions and implements function child(d, s), where destination d and source s are DTD objects;

isPFAMonotone(byte[] pfas) returns true if all PFAs the same (Y or N); otherwise, it returns false.

As we said above, SecureDTD object has a hash map of SecureDTDElement items. Each SecureDTDElement is a wrapper for DTDElement, i.e. it aggregates DTDElement object, and contains other security specific fields, e.g., hash maps of incoming and outgoing generators (hm_incoming and hm_outgoing), full annotation ann_data. Each generator is implemented as a pair of SecureDTDElement and String annotation. The functionality of this class is the following:

getName() returns the name of DTD element;

getDTDElement() returns a wrapped DTDElement object;

getFullAnnotation() return ann_data that is an implementation of ann_data;

setFullAnnotation(byte ann) assigns a full annotation value to ann_data;

isExpecting() returns false if an annotation of any generator is not defined; otherwise, returns, true;
calculateSSIs() returns an array list of SSIs, each of which is an array list of element names;

calculatePFAs() returns an array list of PFAs each of which corresponds to an SSI calculated by the previous function;

addSecLable2InGenerator(SecureDTDElement source, String label) adds label to a generator with the source source;

addSecLable2OutGenerator(SecureDTDElement destination, String label) adds label to a generator with the source destination;

getSources() returns an array list of sources;

getDestinations() returns an array list of destinations;

removeInGenerator(SecureDTDElement source) removes a generator with a source source from hm_incoming;

removeOutGenerator(SecureDTDElement destination) removes a generator with a destination destination from hm_outgoing;

getDestinationAnn(SecureDTDElement destination) returns an annotation transmitted to destination;

getSourceAnn(SecureDTDElement destination) returns an annotation transmitted from source;

cloneDTDElement() clones the current SecureDTDElement;

addDuplicate2Container() adds a cloned DTDElement to a container of DTDElement wrapped by the current SecureDTDElement; used in algorithm QUALIFIER REMOVING;
substituteInContainer(String name) substitutes DTDElement having
   name with its clone in container of DTDElement wrapped by the cur-
  rent SecureDTDElement; used in algorithm SPLIT;

removeFromContainer(DTDElement element) removes element from
   container of DTDElement wrapped by the current SecureDTDElement;

hasChild(SecureDTDElement element) returns true if the current
   SecureDTDElement has a child element; otherwise, returns false;

hasChildren() returns false if it is a leaf element; otherwise, returns
   true.

Classes SecureDTDItem and SecureDTDContainer are abstract, they
   cannot be instantiated, but they provide some useful functions for accessing
   and modifying DTDItem and DTDContainer if corresponding objects
   are passed to them as parameters. Hence, the former are metawrappers
   for the latter. Namely, SecureDTDItem allows to do the following:

cloneDTDItem(DTDItem item) creates an object clone of item;

getSubitems(DTDItem item) returns an array list of direct subitems
   that constitute item;

hasChild(DTDItem item, DTDElement element) returns true if element
   is a child of DTD element with a content item;

calculateBottomupSSIs(DTDItem item) returns an array list of SSIs
   in the case of bottom-up policy; recall, that this requires removal of
   parentheses as in arithmetic expression;

substituteElementInContent (DTDElement element, DTDItem incon)
   substitutes N-labeled element with its production rule in item incon
   and returns the resulting content of DTDItem type.
Class SecureDTDContainer has many similar functions. The difference is that SecureDTDItem makes calls to SecureDTDContainer if metawrapped DTDItem is an instance of DTDContainer:

**getSubitems(DTDContainer container)** returns an array list of direct subitems that constitute *item*;

**hasChild(DTDContainer container, DTDElement element)** returns true if *element* is a child of DTD element with a content *container*;

**addClonedItem(DTDContainer container, String name)** substitutes any occurrence of *d* with $d_Y + d_N$ in *container* and returns a modified container; used in QUALIFIER REMOVING;

**substituteWithDup(DTDContainer container, String name)** substitutes a particular occurrence of *d* with $d_N$ in *container* and returns a modified container; used in SPLIT;

**substituteElementInContent(DTDElement element, DTDContainer incon)** substitutes *N*-labeled element with its production rule in item *incon* and returns the resulting content of DTDContainer type;

**removeFromContainer(DTDContainer content, DTDElement element)** removes *element* from *content* and returns the resulting DTDItem (not DTDContainer, because the result of removing may be DTDName or DTDEmpty).

### 6.2.3 XML

The package contains two classes that deal with XML parsing and transformation. The first class implements the SAX parsing interface [92] in the case of annotations encoded as explained above:

```java
public void startElement(String uri, String localName, String rawName, Attributes attributes) {
    if (rawName.equals(DTDAnnotation.nodeFrom)) {
        //System.out.print("nodeFrom ");
```
 Basically, procedure startElement recognizes one of three possible elements of our annotation encoding. Procedure characters reads bytes between start and end tags in buffers related to nodeFrom, nodeTo, and label tags. Finally, endElement compiles DTDAnnotationItem object, adds it to annotations, and zeros all buffers. The whole process continues until annotation-related tags exist.

Another XML related class deals with DOM structure of the XML document and implements MATERIALIZE.

The first our attempt to implement the algorithm MATERIALIZE was based on DocumentBuilder class of Xerces [94]. In a nutshell, we created an ELEMENT object called root representing XML/DTD root, and invoked a recursive procedure applyAccessibleChildrenTo with parameters root and XML root (i.e., the first root is an element type, the second XML
root is a context node). For every $D_v$ child of the first parameter, this procedure retrieves $\sigma$-function, evaluates it w.r.t. every context node (the second parameter). More precisely, it is done by the following line in source code:

```java
NodeList evalSet = XPathAPI.selectNodeList(ctx, sigma);
```

where `XPathAPI` is Xalan [91] Apace implementation of XPath evaluation.

After that, for every element of `evalSet`, the procedure creates an object of `Element` type and attaches it to the first parameter. Additionally, for every attached node, the procedure calculates a set of attributes and text nodes.

```java
NamedNodeMap attrs = ctx.getAttributes();
NodeIterator texts = XPathAPI.selectNodeIterator(ctx, "child::text()");
```

where both `attrs` and `texts` should be iterated; for every iteration element, objects of corresponding type should be created and attached to a node from the evaluation set. Finally the procedure performs invocation of itself with parameters (1) element type of context node and (2) set of context nodes.

As an experimental evaluation showed, such an implementation, as well as simple flushing of evaluation set into file, was extremely time- and memory- consuming because of the complicated structure of `Element` type of Xerces and evaluation set retrieved by `XPathAPI` of Xalan. To deal with this drawback, we decided to generate an XSLT stylesheet [99] on the basis of the $\sigma$-function and to use `Transformer` class of the same Xerces:

```java
TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer =
    tFactory.newTransformer(new StreamSource(xsltFile));
transformer.transform(new StreamSource(xmlFile),
    new StreamResult(new FileOutputStream(xmlName)));
```

For this purpose, we created an XSLT fragments
responsible for creation of XML root element, and retrieval of attributes and text element of the context node respectively. In addition, the procedure `applyAccessibleChildren` was changed so that its parameters became `previousSigma`, and `nodeTo`. Hence, the first invocation for the XML root is done with $D_v$ root for both parameters. After that, for every $D_v$ child of `nodeTo`, a corresponding $\sigma$-function and an element type accessible by this $\sigma$-function are retrieved. This information is used to construct the following XSLT fragment:

Finally, a new recursive invocation is done with parameters (1) extracted $\sigma$, and (2) element type defined by the extracted $\sigma$. Such a translation of $\sigma$ into XSLT template followed by the evaluation of the latter is much faster than DOM implementation.
6.2.4 XPath class diagram

We used an external library of SiXPath project [106] to parse a user query. The feature of this library is that the user query is represented as an array list of steps. For comparison, Xalan [91] parser represents XPath as a table of symbols. Every symbol is accompanied with an additional information as type of construct it is occurring (e.g., node test, predicate, etc), the length of the construct, table index of the end of the construct. Such a representation is, probably, optimal for query evaluation, but it is extremely complicated for query rewriting.

Another XPath parser SAXON [104] constructs a recursive representation of XPath: first_step/rest_of_path. This structure is not efficient for query rewriting since we cannot apply our divide-and-conquer algorithm. It is also less time-efficient during parsing.

The detailed class diagram of SiXPath is shown in Fig. 6.5.

The user’s query is parsed into an object of LocationPath class which contains an array list of Step objects. Each Step object is a parsed representation of the expression axis :: nodetest[predicate1] ... [predicate_n], where axis value is stored into axis variable according to constants of Axis class, nodetest is parsed into NodeTest object, and set of predicates is parsed as an array list of FilterExpr objects. Since nodetest is either label or *, NodeTest has two subclasses: NameTest and NodeType respectively. Note that the latter captures not only *, but also XPath expressions as text(), node(), etc.

FilterExpr may contain any other kind of XPath expression: FunctionCall, Number, Literal, Variable, LocationPath, and any boolean/arithmetic combination of these expressions.

Finally, package de.fzi.XPath contains a subpackage de.fzi.XPath.Parser with classes implementing parsing of XPath string into an object of LocalPath
Figure 6.5: Class diagram of SiXPath XPath
6.2.5 Query rewriting

Query rewriting package contains two classes: one that implements \( \text{reach}(A, p) \) table and functions that modify its content. The object of this class is aggregated by the class that implements QUERY REWRITE algorithm given that XPath is parsed by SiXPath parser into structure explained in the previous section.

6.3 Workflow

6.3.1 Sequence diagram

The communication between different components is shown in Fig. 6.6. Namely, the scenario starts when the user issues a query. The server calls Wutka DTD parser that returns an object of DTD class. This object is then passed to the constructor of SecureDTD class. Next, the annotation file is parsed and policy class is defined. The objects of Policy and DTDAnotation are recorded in the previously constructed SecureDTD object. After that, algorithms ANNOTATE VIEW and BUILD VIEW are invoked inside this object resulting in \( D_v \) and \( \sigma \)-function.

At this step we have two possibilities: to follow the use case of either XML view materialization or query rewriting. In the first case, we apply MATERIALIZE algorithm that returns the XML over which the user query is evaluated directly. In the second case, the user query is parsed by the SiXParser and passed to the query rewriter along with the \( \sigma \)-function of SecureDTD object. The query is rewritten and represented as a string that should be evaluated over the original XML.

In both cases the evaluation is hold by Xalan evaluator that needs XPath
as string and DOM root of the XML that should be queried. XPath string is parsed by Xalan parser into a table of symbols (see Sec. 6.2.4) that is adopted for efficient query evaluation. Finally, the answer set is returned to the user.

### 6.3.2 Invocation implementation

Query evaluation depends on the sequence of library function invocations. In this section, we describe the usage of our Java library in both view materialization and the query rewriting use cases.

First of all, the user should annotate the DTD and define policy propagation according to grammar of Sec. 6.2.2. Next, the DTD file should be
parsed by the following code, that prepares not only a core DTD structure, but also security-related features such as σ-function, generators and so on:

```java
// Parse DTD file into object of class "DTD"
SecureDTDParser sec_parser = new SecureDTDParser(new File(dtd_file));
SecureDTD m_dtd_sec = sec_parser.getSecureDTD();
```

Here `dtd_file` is a path to the DTD file.

Then, policy and annotations should be parsed into corresponding Java objects. This information is located in `ann_file`.

```java
// Parse annotation file into
// (i) object of class Policy
Policy policy = new Policy (ann_file);
// (ii) object of class DTDAnnotation
DTDAnnotation dtd_ann = new DTDAnnotation(ann_file, DTDAnnotation.SAX);
```

Note, that the annotation labels are parsed by the SAX parser [92]. Since SAX parser is a streaming parser, it doesn’t use as much memory as the DOM parser.

Initial labeling of a Java object corresponding to the DTD with a Java object corresponding to the annotation is invoked by the string:

```java
// Apply partial annotation to Secure DTD structure
m_dtd_sec.applyPartialAnnotation(dtd_ann, policy);
```

Finally, DTD view construction is performed after calling implementation of algorithms **ANNOTATE VIEW** and **BUILD VIEW**.

```java
// Apply GENERIC algorithm AnnotateView (i.e. depends on policy)
m_dtd_sec.annotateView();
// Apply algorithm BuildView
m_dtd_sec.buildView();
```

At this point, the user may either materialize XML view and query it directly, or rewrite the query and evaluate it over the original XML.

The first option is as follows. We create `validator` of **DOMValidator** class which generates DON structure of XML tree that is being sanitized:

```java
DOMValidator validator = new DOMValidator(xml_file, true);
```
CHAPTER 6. IMPLEMENTATION 6.3. WORKFLOW

The next string sanitizes the XML parsed at the previous step. Namely, the procedure \texttt{getSanitizedXMLXSLT} generates XSLT file “\texttt{extract.xslt}” according to the procedure explained above. For this purpose, DTD root element name and \( \sigma \)-function are passed as parameters. After, a standard Xerces transformation is applied, and the result with the extension “\texttt{res}” is generated.

\begin{verbatim}
validator.getSanitizedXMLXSLT(m_dtd_sec.getDTD().rootElement.getName(), m_dtd_sec.getSigma(), "extract.xslt", xml_file+".res");
\end{verbatim}

\textbf{DOMValidator} object will store a DOM structure of the materialized view. This structure is used by Xalan XPath evaluator for query evaluation:

\begin{verbatim}
Document doc = validator.getDocument();
Element root = doc.getDocumentElement();
\end{verbatim}

Recall, that before query evaluation, \$login\ should be substituted with a login name of the user. This is hardcoded by method \texttt{eliminateLoginVar} in the static class \texttt{EUtil}. The resulting query is passed to Xalan processor along with the root of the XML view, and the evaluation takes place:

\begin{verbatim}
query = EUtil.eliminateLoginVar(query);
NodeList nl = XPathAPI.selectNodeList(root, query);
\end{verbatim}

Here \texttt{query} is an XPath string typed by the user. The result of this evaluation is a \texttt{NodeList} that should be parsed by Xerces library.

In the query rewriting use case, instead of creating \texttt{DOMValidator} object, we have to create a \texttt{QueryParser} object, and to pass a query string to it:

\begin{verbatim}
QueryParser parser = new QueryParser();
Object xpath = parser.parse(query);
\end{verbatim}

The result is a generic \texttt{Object} object that is \texttt{net.sf.saxon.expr.Expression} object in reality and represents user XPath string as an array of steps. The latter is used for query rewriting that is invoked as follows:
// Create an instance of rewriter having access to sigma
QueryRewriter qrewriter = new QueryRewriter(m_dtd_sec);
String rewritten_query = qrewriter.rewrite(xpath, 
m_dtd_sec.getDTD().rootElement.getName()).

The last string is the exact invocation of algorithm QUERY_REWRITE with parameters xpath and root. The result is the rewritten query that may be evaluated directly over the initial XML. For this purpose, we create DOMValidator object and call XPathAPI.selectNodeList.
Chapter 7

Experimental results

This section consists of three parts. The first measures scalability and performance of DTD view construction. The second part shows advantages of schema-based policy enforcement over instance-based one. Finally, the last part evaluates the performance of query rewriting algorithm. It also compares three approaches to evaluation: rewritten query over the initial XML, non-rewritten query over view materialized and XML access control processor proposed by Damiani et al. [30].

All experiments were performed on a work station with 4 CPU Intel Xeon CPUs of 3.20GHz, 4GB of RAM, and RedHat Linux ES 3 as an operating system.

7.1 View Construction

Evaluation

7.1.1 Scalability

We selected 5 DTDs from http://www.xml.org/xml/schema/ of different sizes: 24, 79, 211, 802, 2303 edges. For every DTD, we generated 100 files containing 1, 2, etc. up to the number of edges with Y and N labels assigned randomly. For every such a file, we constructed a corresponding
7.1. VIEW CONSTRUCTION
EVALUATION

CHAPTER 7. EXPERIMENTAL RESULTS

Figure 7.1: DTD view construction degradation for Y/N labels

DTD view and measured wall-time required for the construction. Scatter-plot diagrams are shown in Fig. 7.1, where X-axis shows number of Y+N labels in security policy, Y-axis is a wall-time, in milliseconds, required to
construct a DTD view. Most of the scatter plots demonstrate the same behavior: increasing at the beginning and stabilizing (constant time required for view construction) to the end. The increase is expected: the more security labels we have, the more splitting and deleting operations may be required. In particular, there may be an effect of “cascading” splitting, when a subgraph of the DTD is split at every node that is below the node that was split first. For example, if element type \textit{recomm-letter} in the DTD from our motivating example is split and no other security label goes below that node, the whole subgraph rooted at \textit{recomm-letter} will be split in a cascading manner because both the original element type and its copy are connected to the children of the original element type while transmitting different security annotations (Y and N respectively).

However, at some point, DTD view construction time becomes a nearly constant, i.e., it does not increase with the growth of the number of security labels. Moreover, the time variation becomes smaller. This may be explained by the fact that the number of splitting operations may be
reduced if *most specific* labels overwrite propagated labels with the opposite annotation. If in example *recomm-letter* is split, but there are other security labels assigned to edges below that element type, it becomes clear that cascading splittings does not take place for all its descendants.

At this point, we should mention a particular scatter plot \(^1\) in Fig. 7.1(d). At the beginning there are two branches. The lower branch has the same behavior as in the other scatter plots. The upper branch has values that are two times larger than the lower one, but eventually it merges with the lower branch. Such a strange behavior is explained by a complicated structure of the DTD, i.e., with multiple destinations having many sources. In this case, the lower branch corresponds to cascading splittings in a subgraph that has a smaller number of multi-source element types, while the upper branch represents label propagation in more complicated parts of DTD which require extremely time consuming cascading splits in the presence of a small number of security labels. However, as we have seen above, cascading splits diminish if we have more security labels. As the scatter plot on Fig. 7.1(d) shows, this also holds for graphs with complicated structure.

Finally, the scatter plot for scalability is shown in Fig. 7.2. It is a union of all scatter-plots from Fig. 7.1 restricted to the number of security labels less than 150. We can see that if the number of DTD edges increases by factor of 1000, DTD view construction time increases only by factor of 10-15. Thus, we may conclude that our method is scalable. Moreover, the degradation of performance does not depend on the number of DTD edges or security labels.
7.1. VIEW CONSTRUCTION EVALUATION

7.1.2 Policy enforcement performance: schema-based vs. instance-based

We now compare the performance of naïve view-based XML materialization (corresponds to Def. 5). For this purpose, we used the XMark benchmark [95] that provides the DTD schema auctions.dtd which de-

---

The DTD 837institutional.dtd describes health care claim related to standards for electronic transmission, privacy, and security of health information.
scribes an auction scenario. It defines 75 elements describing a list of auction items, information about bidders, sellers, buyers, etc. The schema of auctions.dtd is shown on Fig. 7.3.

XMark also provides a generator of XML documents of different sizes that conform to auctions.dtd. The number and type of elements in the resulting document depends on a parameter called factor. The significant feature of XMark is the generation of each unique XML document for one factor value. We generated 20 different XML documents with factor 0.0i0 and 0.0i5, where $i = 0, 1, \ldots, 9$. The size of these XML files varies from 1MB to 9.5MB.

For each generated XML document, we applied MATERIALIZE algorithm invoked for every DTD view constructed for auctions.dtd above. In total: 7900 views with a different number of security labels in the initial annotation. The scatter-plot of such a materialization is shown in Fig. 7.4(a). Here we have to note that, in order to make experiments repeatable \(^{2}\), we used a standard Xalan XPath evaluator by Apache which leads to exponential blow up. Although there are provided polynomial algorithms of XPath evaluation [50], their implementation is done as a C++ tool [49] rather than a Java library.

We tried to run the same set of experiments for a naïve materialization, where annotations are propagated directly on XML document. To run 100 experiments only for the case of a single security label in the initial DTD annotation took around 30 hours. The result is shown in Fig. 7.4(b). Note a significant difference between two approaches from the viewpoint of “compactness” of materialization time. In other words, the view-based materialization time for the same XML document does not vary much from experiment to experiment. On the other hand, naïve materialization time

\(^{2}\)Since we had random annotations in our experiments, we cannot use C++ tool (not java-library) to evaluate $\sigma$-functions on particular nodes of XML.
has an enormous variation. This variation is caused by the presence of qualifiers of the form \[parent::parentName\] that are inevitable in the presence of destinations with multiple sources (which is true for auctions.dtd).

There is a big gap between the performance of naïve and view-based materialization due to such qualifiers. Sometimes, the naïve materialization can be faster than view-based in trivial cases, e.g., \(Y\) or \(N\) label assigned to
7.2 Query Rewriting
Algorithm Evaluation

To evaluate our query rewriting approach, we use the XMark benchmark [95] described above. The framework of our experimental schema includes a single security annotation with qualifiers depending on the user’s profile, XPath queries of different difficulties and multiple XML documents of different size.

Security annotation. Fig. 7.5 shows annotations for every registered user of auctions portal (i.e. user that can participate in auction).

More precisely, the first rule (1) prohibits access to the information about items and their physical location in regions. This rule is overridden so that items located in North America are visible for people from the United States. This rule is overridden so that items located in North America are visible for people from the United States or Canada.

```
1: ann(site, regions) = N
2: ann(regions, namerica) = Q[site/people/person[@id = $login]/address/country/text() = 'UnitedStates' or site/people/person[@id = $login]/address/country/text() = 'Canada']
3: ann(namerica, item) = Q[location/text() = 'United States' and shipping/text() = 'Will ship only within country']
4: ann(open_auction, seller) = Q[person = $login]
5: ann(open_auction, bidder) = Q[personref/@person = $login]
6: ann(bidder, increase) = Y
7: ann(closed_auction, price) = Q[parent :: closed_auction/(buyer/personref or seller)/@person = $login]
8: ann(closed_auction, buyer) = Q[personref/@person = $login]
9: ann(person, person) = Q[@id = $login]
10: ann(person, profile) = Q[business/text() = 'Yes']
11: ann(profile, business) = N
```

Figure 7.5: Annotation for a registered user of an auctions portal

a leaf, but the difference between results even in such cases is not more than 3-5 seconds which can be considered as negligible from the view point of the overall performance comparison which is around 16-20 seconds for view-based materialization and more than 100 seconds for naïve materialization in the best case.
CHAPTER 7. EXPERIMENTAL RESULTS

7.2. QUERY REWRITING

ALGORITHM EVALUATION

\[ q_1 : \text{regions/africa} \]
\[ q_2 : \text{namerica/item} \]
\[ q_3 : \text{people/person/profile/business} \]
\[ q_4 : \text{open_auctions/open_auction[initial/text() <'50' and current/text() >'100']/increase} \]
\[ q_5 : \text{people/person/*} \]
\[ q_6 : \text{open_auctions/open_auction/*/*} \]
\[ q_7 : \text{people/person/profile[@income > 85000]/parent::person/name} \]
\[ q_8 : \text{people/person/name/parent::person/address/parent::person/profile[@income > 85000]/parent::person/name} \]

Figure 7.6: Set of queries

United States or Canada (policy 2) and items from United States that can be shipped outside the country are visible for everyone (policy 3).

Next, information about the sellers and the bidders of any open auction (4 and 5) is visible if the viewer is the seller or the bidder himself. In the same way, information on buyers is protected in closed auctions (8). All bids increasing the price of any particular auction item is publicly visible (7).

Rule 9 says that a person may get personal information only about himself/herself, i.e. person id should be equal to user login. The latter is expressed by a dynamic variable $login$ which instantiation is hardcoded, i.e. if a qualifier contains the substring $login$, it should be replaced by the login name passed as a program parameter. A personal profile, instead, can be publicly available if the field business has text value Yes. However, business, its turn, is not visible.

**Queries.** We have defined a set of queries (see Fig. 7.6) to test both the algorithm QUERY REWRITE and the advantages of query evaluation w.r.t. different use cases (evaluation of rewritten query over the initial XML vs. evaluation of the original query over the materialized view).

Queries $q_1$ and $q_2$ try to access the items in the African and North American regions respectively. $q_3$ should always return the empty set because of policy rule 11. Query $q_4$ looks for auction bids on item with the particular
7.2. QUERY REWRITING
ALGORITHM EVALUATION

CHAPTER 7. EXPERIMENTAL RESULTS

\[ q'_2 \colon \text{regions/namerica}/\text{site/people/person[@id = $login]/address/country/text()} = 'UnitedStates' \text{ or } /\text{site/people/person[@id = $login]/address/country/text()} = 'Canada' /\text{item}[location/text()] = 'United States' \text{ and shipping/text()} = 'Will ship only within country']

\[ q'_4 \colon \text{open_auctions/open_auction}[\text{initial/text()} < '50' \text{ and current/text()} > '100'] /\text{bidder}[\text{not(personref/@person = $login)}] /\text{increase} \]

\[ q'_5 \colon \text{people/person[@id = $login]/(address|watches|phone|name|creditcard|emailaddress)}[\text{profile}[business/text()] = 'Yes'/\text{homepage}] \]

\[ q'_7 \colon \text{people/person[@id = $login]/profile[business/text()] = 'Yes'}/\text{income} > 85000] /\text{self::profile}[business/text()] = 'Yes'] /\text{parent::person}[site/people/person[@id = $login]]/name \]

Figure 7.7: Set of rewritten queries

start and current prices. \( q'_5 \) and \( q'_6 \) demonstrate the use of asterisk \( * \). The reverse parent axis is introduced in \( q'_7 \) that asks for the person name whose income is more than 85K. The result of this query evaluation should be the same as of query \( q'_8 \) that includes three reverse parent axes.

Some rewritten queries are depicted in Fig. 7.7. More precisely, in \( q'_2 \) before namerica, a missing element name region goes. Field namerica itself is extended with a qualifier which is \( \sigma(\text{regions}, \text{namerica}) \). In the same way, after item, \( \sigma(\text{namerica}, \text{item}) \) is appended. The rewriting of \( q'_4 \) shows that the user will receive increases that were posed by other users, while his own increase values are located under field bidder; the latter is visible since its person reference is equal to the user’s login. In \( q'_5 \), the asterisk \( * \) is rewritten into the union of visible nodes that are children of people. The rewriting of \( q'_6 \) is similar. In \( q'_7 \), we note that profile is extended with a security qualifier \( \sigma(\text{person}, \text{profile}) \) in addition to a user-defined filter on it. The second line of \( q'_7 \) shows the reverse axis rewriting. This part is inserted in \( q'_8 \) three times. Finally, \( q'_1 \) and \( q'_3 \) are rewritten to null because the user cannot see either africa (policy rule 1) or business (policy rule 11).

**XML documents.** We used 20 XML documents generated for experiments on view materialization.

**Performance results.** For each of these 20 XML document, we ran evaluation of each query \( q_j, j = 1, 8 \) from the viewpoint of 10 users \( (login = \)
7.2. QUERY REWRITING

ALGORITHM EVALUATION

Query rewriting performance results are shown in Fig. 7.8. In all experiments DTD view construction time was between 520 and 620 milliseconds. In the query rewriting part, the measured time includes query parsing and running of QUERY REWRITE algorithm. We used the SiXPath [106] processor to parse XPath queries into their tree representation. The processor represents steps of a path in an array, so that we can apply divide-and-conquer techniques that improves the efficiency of query rewriting. From Fig. 7.8, it can be easily seen that more complicated queries require more time. On the other hand, even if the user issues a query that contains time-consuming elements like * or parent axes, query rewriting performance degrades insignificantly, e.g., around 10 milliseconds between the easiest and the hardest query in our test case. Hence, we may consider query rewriting performance as constant.

Query answering time was evaluated by a Xalan [91] XPath evaluator. Unfortunately, XML Task Force evaluator [49] which is shown in [48]
7.2. QUERY REWRITING
ALGORITHM EVALUATION

CHAPTER 7. EXPERIMENTAL RESULTS

Figure 7.9: Experiments on query evaluation
as the most efficient and scalable XPath evaluator, could not be used in our test framework because it does not accept queries that contain union such as queries $q'_5$, $q'_6$. The overall result performance is shown in Fig. 7.9 for queries $q_i$, $i = 1, 8$ and each of 20 XML files. Namely, Fig. 7.9(a) shows our experience with XML Access Control Processor (XMLACP) developed by Damiani et al. [30]. Query evaluation includes a preliminary construction of an authorized view which is extremely time consuming, since security annotations are propagated on the XML tree. Moreover, security annotation is, basically, a pair $\langle xp, lbl \rangle$, where $xp$ is an XPath expression defining a node being labelled, $lbl$ is $Y$ or $N$. The step of initial labelling takes time which grows exponentially with the growth of the initial XML document size.

We have to note that XMLACP approach does not delete forbidden XML nodes if they have permitted descendants. Therefore user queries should include such forbidden steps that may reveal sensitive information. We constructed such queries from the output of Query Rewrite but without qualifiers of the initial security annotation of DTD. Another issue is related to a set of authorized views. Their number can be enormous (e.g., we can derive 126 personal views from an XML document computed with factor 0.050, i.e. of the size around 500Kb; this number grows exponentially with increasing XML size) but their sizes is about 10% less than the initial XML document. Therefore, storing all individual views is not feasible even if do not consider integrity maintenance problems.

Materializing views according to our proposal in Fig. 7.9(b) is much more efficient in time performance. Furthermore, space required for view storing is also reduced since we delete all intermediate $N$-labelled nodes.

Finally, the evaluation of rewritten queries over the initial XML (Fig. 7.9(c)) is even better: it is at least two times faster than the previous case and does not require additional storage for a materialized view.
Unfortunately, we could not retrieve other existing XML securing systems, e.g., Author-X [11] (it is lost and cannot be recuperated because of the storing computer crash [107]), Lock-X [27] (for the same reason [108]), SMOQE [41] (because additional experiments should have been conducted [109]).

7.3 Discussion

We discuss different approaches to query evaluation.

Option 1: Materialize XML views on the fly by a naïve method. In this approach, we store only DTD and security annotations corresponding to different classes of users. Security labels are expressed as \( \langle \text{xpath}, Y|N \rangle \) (see [11], [21], [30], [44], [47], [56], [62]) where \( \text{xpath} \) is an XPath expression that after evaluation returns all the nodes to which \( Y \) or \( N \) label should be assigned. Simple XML labelling can be extremely time consuming because of (i) multiple XPath evaluations, and (ii) the presence of qualifiers in XPaths.

Option 2: Store materialized XML views. This option is very attractive for query evaluation performance: the user simply submits a query which is evaluated over a materialized XML view. The big disadvantages of this approach are (i) integrity maintenance of all views in the case of updates in the original XML; (ii) XML view is usually 20-40 % smaller than the original XML document. In the presence of a large number of user classes, there will require a large amount of storage. The situation may be aggravated in the case of views that contain personal data. Every user has a different view.

Option 3: Store DTD views and materialize XML views on the fly. As we have seen above, view-based materialization is usually slower than a naïve approach. In all our experiments, the overall time for view-based materialization is at 20 seconds. Certainly, this result will
grow exponentially with an increase of XML size. However, if we apply polynomial algorithms of XPath evaluation, this measure will be decreased significantly. And thus, we can consider this option as acceptable even in the case of interactive web-applications. To this end, we have to remember about space consumption which is reduced drastically because (i) DTD view is much smaller than XML view (while $D_v+\sigma$ is more or less of the same size as the initial DTD document), (ii) the same DTD view can express conditions that allows to extract different personal data for each single user.

Option 4: Construct DTD view and materialize XML view on the fly. Recall that DTD view construction can be considered to be of constant time. Hence, as in Option 1, we store a DTD with a set of security annotations which requires very little space. On the other hand, DTD view construction plus XML materialization by MATERIALIZE algorithm on the fly has an acceptable performance.

Option 5: Construct DTD view on the fly and rewrite query. This is the most time- and space-efficient approach among all the others.
Chapter 8

State of the Art

Generally, access control is defined by a set of rules: \((subject, object, action, sign)\). In the following we assume that \(subject\) is the user that wants to access \(object\) in order to perform an \(action\) over it. This is allowed or denied according to \(sign\) which is \(Y/N\), \(+/-\), “grant”/“deny”, etc. In the following we consider \(subject\) as an output of an authentication processor. An object is usually a fragment of the XML tree. An \(action\) (or \(privilege\)) is related to reading or modification of \(object\)’s content. In addition, \(subjects\), \(objects\), and \(action\)’s may be organized in hierarchies. Therefore, \(propagation\) and \(conflict resolution\) rules should be defined as well.

8.1 Run-time policy

evaluation over XML tree

The common scenario for access control models presented in this section is that an access decision point matches the user’s query against all access control rules run-time during its evaluations. This matching may be done only for the last node test or for each step of the XPath expression. The challenge of the current group is to formulate policy in a way easy to look-
8.1. RUN-TIME POLICY
EVALUATION OVER XML TREE

8.1.1 Provisional authorization: Kudo et al.

The central notion of the approach proposed in [56] is a **provisional authorization** which has the following meaning: a subject can be allowed to perform some action on an object within some context if an obligation is fulfilled. An object is targeted as an XPath expression, and the smallest portion of object granularity is a node. It means that attribute values, text nodes and datum of node tags can be protected independently. An action is one of the following: read (user is allowed to read any element-related data), write (to update the content of nodes), create (to create a new subtree under a defined element), delete (to delete the target element and subtree rooted in it). A context is any argument for a specific action, for example time or place of request initiation. An obligation (provisional action in [56]) is a set of additional functions to be performed even if the access is not granted. The following actions are examples of obligations: log, verify, encrypt, transform, write, create, delete.

The confidential data that the user is not allowed to read or modify is transformed into a secure representation by either generating a pruned document (if the user wants to perform read action) or applying some cryptographic operations (if the user wants to perform modifying actions like write, create, delete). These transformational operations are called **security transcoding**. Note that if a forbidden element has permitted descendants, it is included into the pruned version without its attributes. In other word, denial downward consistency [68] (DDC) takes place.

Provisional authorization rules are expressed in XML access control language (XACL) [55] and constitute a policy. The latter can include additional properties as propagation (no, up, down), conflict resolution (denial takes precedence, permission takes precedence, or none) and default pol-
The policy evaluation algorithm first selects only those access control rules that can be matched to the authorization request, taking into account the triple object, subject, condition. As the result, an authorization decision list is obtained. The next step is to apply the policy properties of propagation, conflict resolution and default value. Note that in order to derive a permission to a requested unlabeled node (i.e., a node that does not carry a permission explicitly), the algorithm is invoked recursively for parent (in the case of “down” propagation) or children (in case of “up” propagation) until the explicit permission declaration is met. The resulting decisions are appended to the authorization decision list as the decisions for the original target element. Finally, if the authorization decision list contains more than one decision, all decisions obtained via propagation are eliminated and only one of the remaining decisions is selected in an arbitrary way.

Suppose that the number of access control rules is $|ACP|$, $|T|$ is a size of the XML tree. Then the complexity of the algorithm for query answering is $O(|ACP| \times |T|)$. Here, for every relevant XML node, we have to perform a full look-up of access control list in order to find relevant access control rules. Since the whole tree may be involved into consideration (e.g., when we propagate security label from children to the root), the upper bound is $O(|ACP| \times |T|)$. Note that a full annotation of some XML tree part (from a node to its ancestor or descendants with explicitly defined permissions) is constructed during runtime policy evaluation.

### 8.1.2 A rule-based XML access control model

An RDF-based proposal for run-time enforcement of access control was presented in [6]. Namely, XML-based rule language called *XML Declarative*
Description (XDD) is employed to define authorizations and policy. In general, XDD is used to represent some information stored in an XML document as XML Clauses consisting from XML Expressions. The latter may contain variables that are used to define other XML expressions or their sequence and can be of the following types: XML expressions, string, tag or attribute name, pairs of attributes and their values, some partial structures. Every XML Clause has a head which is an XML Expression representing some fact (e.g., authorization rule or policy) and a body (i.e., a set of XML expressions or first order logic predicates on XML Expressions) which is a constraint on the head.

An object is defined as a triple \( \langle \text{level}, \text{target}, \text{path} \rangle \) where \( \text{level} \in \{\text{instance, schema}\} \), \( \text{target} \) is a name of \( \text{level} \) file, \( \text{path} \) is an XPath expression identifying an element or attribute in \( \text{target} \); \( \text{privilege} \in \{\text{read, write}\} \).

A propagation policy is of two types: cascade (top-down to descendants) or no-prop (i.e., no propagation). A conflict resolution policy is denial takes precedence if most specific policy is not able to resolve conflicts. Finally, the default policy is either closed or open.

Every authorization rule is encoded into XDD by means of RDF. As we said above, XDD exploits variables. In particular, variables are used to represent subject and object in Authorization clause. The body of an Authorization clause can be, for example, a method of instantiation of variables in the head (e.g., subject \( \text{\$s:user} \) is from “Doctor” group) or modelling the semantics of (top-down) cascading authorizations (e.g., every child and attribute of an element with propagation type cascade or derived_from_cascade can be assigned an authorization with a type derived_from_cascade if it does not have an explicitly assigned authorization). The latter option shows that there is a possibility to model a bottom-up cascading as well.

Next, Policy clause is used to model different kinds of conflict resolu-
8.1. RUN-TIME POLICY EVALUATION OVER XML TREE

tion rules (e.g., denial takes precedence, instance takes precedence over schema, descendant takes precedence) and default policy (open or closed) for different pairs \( \langle \text{subject}, \text{privilege} \rangle \), where both \text{user} and \text{privilege} can be defined by XML Expressions.

Finally, Access Evaluation clauses are used to calculate a set of granting/denying authorizations to access the object defined by the XML Expression. The conflicts between these sets are resolved by the associated Policy clauses. The latter is matched against access evaluation request \( \langle \text{subject}, \text{object}, \text{privilege} \rangle \) so that \text{subject} and \text{privilege} of the Policy clause are instantiated, conflicts among different authorizations are resolved. The decision on access is represented as \$s:decision variable in Access Evaluation clause which is instantiated according to conflict resolution.

This proposal is very similar to the previous one in the sense that for any request, the whole access control list should be looked-up as many times as propagation from the nearest ancestor or descendant of the requested node should take place. Hence, the complexity of access decision algorithm is also \(|ACP| \times |T|\). The difference between proposals is that we may define that top-down propagation is not applied for node attributes but is held for node children, etc.

8.1.3 Client-based access control for XML documents

The proposal of [21] considers the case when the access control is moved to clients, e.g., secure tokens and smart cards that are used as trust components in different mobile devices (e.g., PC, PDA, cellular phone) participating in applications dealing with sensitive information (e.g., certification, authentication, electronic voting, e-payment, healthcare, digital right management, etc.). Although, we classified the proposal in [21] as run-time policy enforcement, it tackles the problem of cryptographically enforced access control (Sec 8.5). Namely, the client is responsible for decrypting
the input document, checking its integrity and evaluating the access control policy corresponding to a given (document, subject) pair. For this purpose, access control policy, key(s) to decrypt the document can be permanently hosted by the client, refreshed or downloaded via a secure channel from different sources (trusted third party, security server, parent or teacher, etc). However, the scenario not only delivers the authorized view to the user, but also allows to query the XML document without view materialization. For efficient performance of the operations, access control rules are dissociated from encryption and evaluated in a streaming manner. Consequently, the proposal considers streaming documents, i.e., event based parser (e.g., SAX [92]) is used.

Access control rule has a form \(\langle \text{sign, subject, object} \rangle\), where, object is an XML element identified by an XPath expression of the fragment \{child, descendant-or-self, [ ], *, node test\}.

The set of access control rules are propagated along the XML tree in a top-down manner with the following conflict resolution rules: denial-takes-precedence and most-specific-takes-precedence. In addition, the default policy is closed. Finally, DDC is also applied.

Every access control rule is presented as a non-deterministic finite automata (NFA) [52]. The latter distinguishes between navigational and predicate tokens of parsed access control rules (i.e., tokens that are related to location step and to predicate of XPath respectively). Since XPath expression for access control rules may contain predicates, which are difficult to manage in a streaming fashion, the papers proposes to detect and to skip them at parsing time, and to reassemble later the relevant pending parts at the right place in the final result. Thus, tokens can be pending or active. The Authorization Stack registers the related navigational tokens at a given depth in the document. Along with the tokens, the stack stores also signs of the considered tokens (positive active, positive pending, neg-
8.1. RUN-TIME POLICY EVALUATION OVER XML TREE

In [71], it was proposed to match the user query against a special structure called a *Policy Matching Tree* (PMT) representing access control policy of [68]. In this approach, *action* is either local or recursive (top-down propagated) *read*, *object* is an XML node expressed in fragment \{child, descendant-or-self, *, [[]]\}, and qualifier cannot go immediately after a node test with descendant-or-self axis.

In a nutshell, a set of access control rules is converted into a PMT which has the following structure: outgoing edges from the root of the PMT are used to match *action* (*read*); nodes of the first level (children of the root) represent the concept of *subject* and their outgoing edges are labeled by a value of *userId*, *group*, and *role*; nodes of the second level express the start of *object* matching. Subtrees rooted at those nodes are similar to an NFA with the finite states representing access decision +/− or UNDEFINED (interpreted as “−” because of the default *deny* policy) applicable only to
the node or to the subtree rooted at node accessible by this NFA. Since an object definition may include a predicate at some node test, the finite states may contain links to predicate NFAs involved in object definition. The predicates’ NFAs are attached to the root node. Hence, predicates are evaluated by the retrieval processor, which can calculate the location of the node filtered by the predicate, at the very end of the user’s query matching. Thus, a sort of static analysis is performed. The negative result of predicate evaluation takes precedence over positive result of object matching.

[71] shows that PMT generation time and PMT space complexity are linear with respect to the number of access control rules; the time complexity of matching user queries against the PMT is less than $2(K+1)|P|^{1-\lambda}(\ln V + \ln(K + 1))$ where $K$ is the number of nodes in PMT, $V$ is the maximum of outgoing edges from any node, $P$ is the number of predicates, and $\lambda = \frac{\ln V}{\ln V + \ln(K+1)}$.

8.1.5 RDF Metadata for XML Access Control

The proposal in [51] considers not only node-level security granularity, but also access control for associations among nodes. The authors propose to express such access control rules in RDF [110].

Access control rules are specified explicitly for objects and are not propagated. The default policy is closed and the conflict resolution rule is denial takes precedence.

The method distinguishes between simple security object, which is a usual XML node, and association security object which is a subtree rooted at a simple security object. Both kinds of security objects are expressed by RDF according to special syntax.

The overall architecture of the system can be described as follows: user’s query is evaluated over XML store; after that, an answer set is checked for security violations (access control rules are looked through); if they happen,
the query is rejected; otherwise, answer set is attempted to be extended with the answer history of the current user; if the extended answer set contains violations in associations, the query is rejected; otherwise, history is updated and answer set is returned. The extension of the answer set with the history relies on the notion of XML keys [23, 24], though it can also employ integrity constraints and statistical correlation.

The query language was not specified in [51].

8.1.6 Role-based access control to XML databases

The conception of role-based access control (RBAC) for XML was studied in [82]. Each subject belongs to some role with a set of privileges assigned to it, all roles are organized in role-graph. A privilege is a pair object, access, where object is defined by an XPath expression, access mode is read, update and extend. Note that access modes can also be organized hierarchically, e.g., update implies read. Finally, hierarchical structure of XML also assumes hierarchical relations among objects.

The privileges may be propagated in all hierarchies. Namely, minor roles propagate their privileges to senior roles. If an access mode is granted on some XML node, so is on descendants of that node. The same holds for implied access modes. However, negative privileges (i.e., it is forbidden to perform some action on some node) should be clearly stated in a special table of privileges, and they are not propagated in access mode hierarchy. In addition, negative authorizations are implemented as a set of constraints that are propagated neither in subject nor in access modes hierarchy but. Both negative and positive authorizations are always propagated in object hierarchy.

The algorithm on access decision identifies all privileges that are granted to the role of the requesting user except those ones that are forbidden by the constraints. The algorithm is based on a look-up of all three hierarchies.
For each privilege, it is also necessary to check if it is not negative in the table of privileges or if it is not included in some constraint definition.

### 8.2 Combining access control rules into user query

Proposals considered in this section examine the DTD and policy before evaluating query over the XML data. Such a preprocessing step tries to deduce accessibility decision based on structural properties of the DTD and the query. However, run-time evaluation is still required in many cases.

#### 8.2.1 Static analysis: Murata et al.

[68] is the first attempt to speed up runtime access control checking for XML documents. *Static analysis*, based on the access control policy and document schema (if the latter is available), decides whether a query expression may retrieve forbidden data or does not examine such data at all. The key idea for this static analysis lies in using automata for representing and comparing query expressions, access control policies and (optionally) schemas.

[68] mentions that access control policy can be represented via OASIS standard XACML [105]. The action is `read`. Objects are formulated in the XPath fragment `{child, descendant-or-self, label, *, [ ]}`. Each individual label, positive or negative, may be local or may be propagated to node descendants.

Other rules of policy framework include: (i) denial takes precedence in the case of conflicts due to propagation; (ii) denial is a default assignment for a node with undefined permission; (iii) *denial downward consistency* denies an access to the whole subtree rooted at a forbidden element.
The user formulates queries in XQuery that is based on XPath fragment described above.

The framework for static analysis includes the creation of schema automaton (if schemas exist), access control policy automaton, and query automaton. Since access control policies and XPath expressions of XQuery query may include predicates, the notions of overestimated and underestimated automaton are introduced. The first one constructs an approximation of the query by eliminating its predicates, the second one substitutes steps having predicates with an empty set, assuming that paths with predicates cannot be satisfied. The further analysis evaluates the constructed automata and relations between them. If the query is allowed, the language of its automaton is a subset of the language of access control policy automaton intersected with language of schema automaton (if the latter exists). If the query is denied, the intersection of language of query automaton and language of access control policy automaton, which is in its turn intersected with language of schema automaton if the latter exist, is an empty set. If query is neither granted nor denied (intersection is not empty), it is statically indeterminate and requires run-time policy evaluation.

Experimental analysis showed that in the absence of DTD schema, 40% of queries do not require the runtime check and 10% of queries can be rewritten so that parts always denied are eliminated from the initial XQuery. In the presence of DTD schema, these figures are even more promising: 65% and 25% respectively.

8.2.2 QFilter

In [62], Luo et al. elaborates on certain issues left open in [68]: efficient security check for XPath (i.e., determination if an XPath query is completely granted or completely denied), and query rewriting to avoid run-time pol-
8.2. COMBINING ACCESS CONTROL RULES INTO USER QUERY

For this purpose, a generalized NFA [39] is constructed from the set of access control rules (ACR). The difference from access control NFA presented in [21] is that the NFA in [21] is constructed for every access control rule separately, while in [62] a single NFA is constructed for all access control rules with the same prefix. Moreover, input and output for the NFA in [21] is a streaming XML. The input of the generalized NFA is a user query $Q$ and the output is a modified query $Q'$ with the following semantics:

- **Accept** ($Q' = Q$) if $Q$ asks only for permitted elements;
- **Deny** ($Q' = \{}$) if $Q$ asks only for forbidden elements;
- **Rewrite** ($Q' = QFilter(Q, ACR)$) if $Q$ asks for both permitted and forbidden elements.

QFilter policy is the same as in the previous approach: each particular label may be propagated locally or recursively, denial takes precedence in conflicts, the default policy is closed, and denial downward consistency holds.

Queries are formulated in the same XPath fragment as in the previous approach.

The following is the *primitive semantics* of safe answers

$$Q' = Q \cap (ACR^+ \setminus ACR^-)$$

i.e., the answer to $Q$ should contain all the nodes that satisfy permissive ACR rules ($ACR^+$) except for those that satisfy prohibitive ACR rules ($ACR^-$). Therefore, separating QFilter for $ACR^+$ and $ACR^-$, the rewritten query can be expressed by the following formula

$$Q' = QFilter(Q, ACR^+) \setminus QFilter(Q, ACR^-)$$
8.2. COMBINING ACCESS CONTROL RULES INTO USER QUERY

\( QFilter(Q, ACR) \) tries to match \( Q \) against the NFA constructed for \( ACR \) substituting \( * \) with any element accepted by any direct subsequent state of the NFA in DFS traversal. // are substituted with subpaths accepted by any subsequent states of the NFA in BFS traversal.

In comparison with [68], Accept and Deny queries are detected faster because the containment check of two automata in the former is much slower than mapping of regular expression to an automaton in the latter. Moreover, time-consuming run-time evaluation of other queries is avoided by incorporating ACR into \( Q \) so that the resulting \( Q' \) does not contain any parts violating access control rules.

Later, QFilter was extended to a distributed environment called a brokerage system [59]. In particular, brokers (i) know the information about object distribution and (ii) are responsible for policy enforcement.

Recently, frequent policy updates were considered for QFilter in [7]. Their study was performed in the context of incremental adaptation of materialized views similar to the problem of reusing views from cache [86]. The problem was then was reduced to the problem of XPath containment.

8.2.3 Optimizing the secure evaluation of twig queries

Another kind of optimization is described in [28] by Cho et al. The paper focuses on reduction of run-time recursive checks while evaluating twig queries, i.e. tree pattern [5] queries which allow parent-child, ancestor-descendant relations and predicates.

An XML security model is defined as a multilevel-security model [61, 76] where each user/subject and resource/object is assigned a security level. Object can be accessed by subject if the latter has a security level no less than that assigned to the former.

The actual value of security level is specified as an attribute of XML tag and may vary from instance to instance. Therefore, the minimal granule of
protection in XML document is an element from which security annotation
is propagated to all attributes and text values. The assignment of security
level is regulated at the DTD level ("coarse" conditions) and further should
be propagated to all document instances:

1. security label *must* be defined: it is called mandatory, it cannot be
   overridden, and its default value may be specified at DTD schema;

2. security label *may* be defined: it is called optional, specified at in-
   stance level and can override inherited labels;

3. security label *cannot* be defined: it is called forbidden and is specified
   via inheritance).

The default security policy is open, it means that if some nodes cannot be
annotated (e.g., root annotation is forbidden) then they are accessible to
every subject.

The secure evaluation of twig queries is done by means of a recursive
check (RC), i.e. security label of a query node is computed by top-down
propagation from the nearest annotated ancestor and compared with the
subject’s confidentiality level. However, sometimes checks at some nodes
can be skipped (no check or NC) or simply fulfilled by examining manda-
tory labels locally (local check or LC). For example, in parent-child tree
patterns, mandatory labels require LC, forbidden and optional ones with
no mandatory or optional ancestors invoke NC and LC respectively, in other
cases RC should be accomplished. Although the rewriting of more general
queries is basically the same, the security checks may depend on cardinality
("one", "?", "+", or "*") of DTD edges (see [28] for details).

The implementation of such security checks involves adding extra se-
curity predicates to every query node test with LC or RC checks. In par-
ticular, the following predicates were proposed in [28]: LC predicate is
CHAPTER 8. STATE OF THE ART

8.2. COMBINING ACCESS CONTROL RULES INTO USER QUERY

\[ \text{SecurityLevel} \leq \text{user.level} \text{ OR NOT SecurityLevel}, \quad \text{RC predicate is} \]
\[ \text{[self-or-nearest-ancestor(.)} \leq \text{user.level]} \]

In the following, we estimate time complexity of the algorithm that assigns different kinds of security check for every node test \(|Q|\) in user query \(Q\). In particular, the algorithm looks for paths between DTD nodes corresponding to node tests \(nt_i\) and \(nt_{i-1}\) (i.e., preceding location step), which requires \(O(|D|)\) operations, where \(D\) is the DTD document, and checks the security label of nodes of these paths to reason about security check type for \(nt_i\). Further optimization that considers cardinality adds an additional factor of \(O(|D|)\). Hence, the total time complexity is \(O(|Q| \times |D|^2)\).

The approach does not reveal the structure of DTD and XML documents to the user. The user may receive some information in the form of a set of nodes corresponding to the query that are accessible with respect to the user’s rights. Moreover, the user is not supplied with a schema according to which he/she can organize queries, and it is not clear if such a schema may be constructed. Also, specifying security levels on XML tags can cause problems with data integrity maintenance, e.g. when a new class of users is created.

The approach was implemented in LockX [27], a system for specifying and efficient querying secure XML documents.

Regarding the policy, it may be fit to our framework as follows. Mandatory and optional specification are \(Y|N\) and *, respectively. Label with local propagation of forbidden specification is a label defined via HP (td) propagation, and the default policy is “open”.

139
8.3 Structural locality of access control

A set of proposals discussed in the current section focuses on speeding up the determination of accessibility of individual XML data items during the run-time policy evaluation. In a nutshell, the considered scenario is the following: a full annotation of XML document is analyzed for the purpose of structural locality accessibility identification. The result of this analysis is a set of Compressed Accessibility Maps (CAMs), i.e., the minimal partial labeling of XML such that a full labeling can be univocally derived. The minimality means that no other labeling of the smaller size exists.

8.3.1 A compressed accessibility map for XML documents

The proposal [88] of Yu et al. constructs a compressed accessibility map (CAM) for each pair \langle user, action \rangle. In particular:

- A security annotation of an XML document has the format \((d\ast, s\ast)\), where \ast is either + or −; \(s−\) and \(s+\) mean inaccessible and accessible node, respectively; \(d−\) and \(d+\) mean inaccessible and accessible descendants, respectively. Such a labeling can be obtained from a full annotation (Y or N labels for every node) by bottom-up (or a post-order tree traversal) procedure which time complexity is proportional to the number of nodes in the database tree.

- A structural locality of accessibility is a grouping of data items according to similar accessibility properties. This is a descendant accessibility property: descendants of accessible node are accessible. On the other hand, ancestor accessibility property can also be considered: ancestors of accessible node are accessible within unit region. In every unit region, the algorithm identifies redundant nodes, i.e. nodes whose la-
beling may be inferred from descendants or ancestors, and eliminates them. The latter operation can also be done in the time linear to the number of nodes in the XML tree.

- A compressed accessibility map contains the accessibility of “crucial” nodes, i.e. the nodes that were not eliminated by the previous operations. Consequently, the accessibility of other nodes can be uniquely derived from such nodes. Hence, CAM can be interpreted as a minimal annotation for XML documents. Furthermore, CAM is used to effectively answer whether a specified node is accessible in time proportional to the product of the depth of the node in the XML tree and logarithm of the CAM size [88].

The limitation of this model is that optimality of the CAM cannot be preserved incrementally under data item updates. It means that in the worst-case, a single small change can cause a complete re-labeling of an entire database tree. However, near-optimality (optimal size+1) can be preserved [88].

8.3.2 Integrated CAM for XML

[53] proposes an extension of CAM called Integrated CAM (ICAM). The CAM labels are generalized in the following way: they are of the form \((s^*, d^*)\), where \(s\) and \(d\) have the same meaning as for CAM (i.e., node self and node descendants respectively), and \(^*\) is used for encoding of action (called “operation” in [53]) that is permitted for the user to perform. All actions are organized in a hierarchy that also includes null action standing for no action is permitted for the user. The action hierarchy allows us to infer accessibility for covered actions (e.g., write covers read, hence if write is permitted, so is read. null action is covered by all the other actions). Thus, CAMs for implied actions overlap with CAMs for covering
actions. Consequently, the integration of CAMs with overlapping node sets can result in an ICAM that is smaller than the total size of the CAMs.

In a nutshell, ICAM construction has the following steps: for every action $x$, CAM$_x$ that is a CAM for the operation $x$; traversing XML tree in post-order (hence, the time complexity is $O(|T|)$), apply level merging rule (LMR) for node if it is not redundant in at least on CAM$_x$, or delete that node otherwise. Since all actions are organized in a hierarchy, i.e. a partial ordering is defined over the set of actions, LMR selects two smallest permitted actions for $s$ and $d$ part of label respectively that cover all the actions permitted, respectively, to $s$ and $d$ parts of nodes that cannot be deleted from ICAM.

The lookup of ICAM is leveraged by (i) a special XML tree numbering scheme, (ii) a logical table structure of ICAM containing pointers from every ICAM node to the corresponding node children, and (iii) an index on the XML tree and and 2-layer hash indexes on the logical table structure of ICAM. Then the total time complexity is proportional to the depth of the requested node in the XML tree and logarithmically to the size of the ICAM.

8.3.3 Document Order Labeling

Another revision of CAMs is presented in [89]. The proposed Document Order Labeling (DOL), exploits not only the structural locality of the access control data but also correlations among the access rights of different users.

The proposal of [89] considers not only “vertical” locality (i.e., accessibility propagation from ancestors to descendants), but also “horizontal” (proceeding from the assumption that some siblings have the same annotation). For this purpose, an XML tree is represented as a sequence of tree nodes in document order, i.e., enumeration of tree nodes in the pre-order.
Compression can be performed in three ways:

- **One-dimensional: compression on objects.** For a single user, the method stores only the transition nodes (i.e., nodes whose accessibility differs from the accessibility of its parent in document order) with their accessibility label (+ or −).

  Experimental evaluation shows that CAM is 20-25% more compact than DOL. However, it was observed that the total space required for CAM may be greater. This is because each CAM node stores a reference to a document node, pointers to the node’s children in the CAM, and the access control information itself. In contrast, DOL stores only an access control code for transition nodes.

- **Two-dimensional: compression on objects and subjects.** For multiple users, the stored information is (i) a *code book* which is a table with columns \{code, subject₁, ..., subjectₙ\} where *code* is code of a distinct access control list (which occurs in the considered labeling of the XML tree) that is a set of accessibility labels (+ or −) for the subjects subject₁, ..., subjectₙ of the corresponding columns of the code book; (ii) pairs \((t, code)\), where \(t\) is a transition node and *code* is a code of its access control list.

  For an efficient access lookup, document order is divided into pages with the same number of nodes and a header containing *code* for the first element of the page and a flag which shows whether the page contains transition nodes after the first element.

- **Three-dimensional: compression on objects, subjects, and actions.** Access control correlation may also exist between different action modes [53]. However, the method proposed in [89] can perform a compression independently of the existence of a hierarchy between actions. For this purpose, a three-dimensional cube
subject × object × action can be considered in two ways: (i) as a set of subject × action matrices (organized in a code book) for different objects or (ii) as a set of vectors for every pair (object, action) with the number of bits equal to the number of subjects. In the first case, each matrix contains accessibility information for a particular object. Hence, a transition node has a different access control matrix than its predecessor. Therefore, we can store only transition nodes pointing to a corresponding matrix in the code book.

In the second case, every vector stores the complete access control information for a specific object under a specific action. Hence, a transition object does not agree with its predecessor on the accessibility for any user under any action mode. Therefore, we can store, for each transition object, pointers for each action mode.

Two types of updates are considered: changing node accessibility and updating the set of subjects. It is shown in [89] that both operations can be performed efficiently and correctly.

8.4 XML views

8.4.1 Author-\(\mathcal{X}\)

In [11, 16, 19], Bertino et al. describe a Java-based system for XML data protection called Author-\(\mathcal{X}\) which elaborates on open issues outlined in [17]. Subjects are expressed either by user id or in terms of credentials [85], which can be organized in classes called credential types. Moreover, security policy can define credential expressions (i.e. constraints on attributes, composite constraints expressed via boolean operations) which are exploited to determine the valid authorization. For these purposes, an XML-based language for specifying subject credentials and security policies
was developed and presented in [15].

Specification of protection objects can be done in different modes: a set of documents corresponding to some DTD, particular XML document, XML portions (e.g., elements, attributes and links) corresponding to DTD portions, a specific XML element in a specific XML.

The set of privileges includes *browsing* and *authoring* actions. More precisely, *browsing* implies *read* (examining the content of an XML portion), *navigate* (exercising IDREF attributes if an element with referred ID is accessible) and *browse_all* (includes both *read* and *navigate* privileges), while *authoring* defined via *write* (allows to modify and delete) and *append* (subsumed by *write*).

The *propagation* option can be *CASCADE* (propagate authorization rule to all direct and indirect subelements of object), *FIRST_LEVEL* (propagate only to direct subelements), and *NO_PROPAGATION*. In [20], the propagation options were defined, respectively, as $\ast$, $n$, and $0$, where $n$ means the propagation of the policy of the current element to its subelements which are at most $n$ levels down in the document/DTD hierarchy; $\ast$ and $0$ are equivalent to *CASCADE* and *NO_PROPAGATION*.

Finally, both positive and negative authorizations are admitted. The resolution of conflicts is resolved by *most specific takes precedence* rule (XML-level authorizations takes precedence over DTD-level ones, authorizations on descendants takes precedence over those on ancestors), *denial takes precedence* and default *deny* rules.

The views are constructed as follows. First of all, the system selects a set of DTD-based and XML-based authorizations related to the requesting user. After that, the selected authorizations are sorted so that authorizations having precedence appear at the end of the list of authorizations. Next, authorizations are applied and propagated in the XML, the conflicts are resolved. Finally, every forbidden element with is deleted. Hence, the
pruned version is provided to user.

Now, we evaluate the complexity of the algorithm. The selection step is performed in time $O(|ACR|)$, where $|ACR|$ is the number of access control rules. The sorting step is performed in time $O(|ACR|^2)$ by inserting every access control rule in the sorted list of access control rules. The propagation step takes, in the worst case, $O(|ACR| \times |T|^2)$ time where $|T|$ is the number of XML nodes in the initial XML. More precisely, for every access control rule and every XML object referred by this access control rule, an authorization sign is assigned to this object and to all its descendants if \texttt{CASCADE} propagation takes place. Finally, the pruning step requires $O(|T|)$ time. Summing, we get $O(|ACR| \times |T|^2)$ for the whole algorithm.

Considering the method of view computation, it cannot be guaranteed that this view provides all and only accessible data. This is because, conflict resolution rules are implemented by sorting access control rules. However, if we consider an example where one negative access control rule is defined at the DTD level for some element and another positive access control rule is defined at the XML level for some ancestor of this element (propagation option is \texttt{CASCADE}), then, we cannot identify the order of these two rules since no strong ordering rule is provided.

Having been provided with the view, the user can send a \textit{request}, i.e., formulate his/her queries and perform allowed authoring operations. Note that the view is computed every time the request is issued. In [18], two strategies for request evaluation are considered: \textit{top-down} and \textit{bottom-up}. The former checks authorizations starting from the DTD root and goes down through the hierarchical structure of DTD. The latter starts from the requested protection objects at the most specific granularity level of DTD and goes up until the appropriate authorization is found or the DTD root is reached. For XML documents, the same considerations can be applied.
makes sense to choose bottom-up strategy when the policy contains many negative DTD authorization rules. Otherwise, the top-down strategy is preferable.

The model of Author-$\mathcal{X}$ was enriched by temporal component and distributed/cooperative updates in [13]. In addition, a push architecture, which exploits encryption techniques, was also developed for Author-$\mathcal{X}$ [18].

### 8.4.2 Fine-grained access control to XML documents: Damiani et al.

A number of early works of Damiani et al. (see [36, 33, 32]) were dedicated to access control for XML documents. Evolution of fine-grained access control to XML documents started in [33] with general ideas and discussions of authorization granularity, policy enforcement and implementation. Next, [32] presented a formal definition of access control rule and an algorithm for computation of view. More detailed discussion of access control processor was provided in [36]. Finally, a full-fledged solution was presented in [30].

Authorization rules can be defined not only at the XML level but also at the DTD schema. And in this case, DTD annotation is used to type the XML document with security labels.

Subjects are represented as triples $<\text{user-id, IP-address, symbolic-address}>$ thus allowing authorization of both local (identified by a user account, $\text{identity, user-id}$) and remote requesters (IP-address or symbolic address of the user origination). Subject definition can possibly be extended with the fourth component role-id as in [36]. User identities can be organized in groups. On the other hand, IP-address and domains can be generalized by location patterns which are specified by using a wild card $*$ instead of a specific number or
name (or sequence of them). All three components of subject representation can be ordered into hierarchies: user-group hierarchy, IP-hierarchy and symbolic-address-hierarchy. Authorization subject hierarchy is defined as Cartesian product of listed above hierarchies.

Objects are addressed with XPath expressions of the fragment \{child, descendant-or-self, *, [], label, and, or, last(), position()\}. Element accessibility is then propagated to its attributes and text nodes.

The elaboration of the approach is conducted for the read action. However, other actions are considered as well. In particular, insert can be performed for authorized XML element, and the inserted element inherits the labeling of its parent; the delete operation is allowed only for permitted elements; update is committed on a permitted element if the result of operation execution results in a permitted element.

Sign of authorization can be either positive or negative. Type of authorization is defined by the following factors:

- propagation of authorization (defined locally for every node): local (propagated to element’s attributes only) or recursive (propagated to sub-elements and their attributes);

- instance-level (defined on XML document) or schema-level (defined on DTD schema);

- strength: soft or hard.

As a result, an access control model has 8 authorization types: LDH (local hard), RDH (recursive hard), L (local), R (recursive), LD (local at schema level), RD (recursive at schema level), LS (local soft), RS (recursive soft). Here authorization types are listed in priority from highest to lowest.

An authorization obtained via propagation can be overridden by an authorization on a more specific object. The same rule is true for propagation of schema-level authorization to an instance: instance authorization
takes precedence. However, in cases when schema-level authorization takes precedence over document-level, users are allowed to specify DTD authorization as *hard* and XML authorization as *soft*. In case of conflict between authorization rules or undefined authorization, the rule *denial takes precedence* is applied to an object.

The requester’s view of the document is computed in two steps: *tree labelling* and *tree pruning*. The tree labelling procedure starts with defining relevant access control rules, which regulate access of requester to the document, and assigning a vector of eight initial labels to each object matched by selected access control rules. Each component of the vector can carry either + or − or ε (meaning that for the current object no authorization of the current type is defined). The next step defines the “winning” label by applying rules *most specific subject takes precedence*. In the case when the conflict remains unresolved, *denial takes precedence* rule is applied. Next, the “winning” label is propagated down until *more specific object* occurs. Finally, the default *closed* policy is applied to nodes that were unable to obtain a label via the previous operations.

The tree pruning phase deletes all subtrees of a labelled XML tree that have only negative labels. In the case that descendants of a forbidden node have a positive label, the pruned view contains that forbidden node without attributes (*structural rule* in [21]). The requester is also supplied with a “loosened” version of the initial DTD schema. In the simplest approach, it means that pruned elements and attributes are marked as optional.

The algorithm for view construction consists of two tasks: (i) identification of authorization objects and (ii) evaluation of node labels. The first task can be performed efficiently [50] in time $O(|T|^2 \times |Q|^2)$ and space $O(|T| \times |Q|^2)$ for every access control rule, where $|T|$ is the size of the XML document and $|Q|$ is the size of XPath expression defining an object in the current access control rule. The second task requires $O(|ACR|)$ for initial-
ization phase (\(|ACR|\) is the number of access control rules associated with the XML document) and \(O(|T|)\) for label propagation and conflict resolution given that the number of authorizations of a particular type associated with every node is small.

Application of the proposed approach was considered in securing SOAP e-services [34, 35].

Later in [38], a temporal aspect was added to the model. The reasoning was held from the point of view of two perspectives: temporal data perspective (i.e., how data changes are reflected in XML) and temporal authorization perspective (i.e., validity of access control rule and satisfaction of the user’s request with respect to the time of issuing).

For the temporal data perspective satisfaction, each XML leaf (i.e. attributes and text nodes) has two annotations: bitemporal and historical. Bitemporal annotation consists of a transaction time interval, which represents evolution of data and is supplied by the system (when the node is created or modified), and a valid time interval, which indicates when the data is valid and supplied by the user. On the other hand, historical annotation is used to reflect changes and represented as a pair (old value, bitemporal annotation).

Subject and object have the usual meaning. The set of actions consists of read, insert, delete, and update operations which are accompanied with assignment and modification of bitemporal and historical labels. Authorization rule also contains a temporal condition which expresses a temporal constraint within which request can be satisfied. Finally, the user request, besides standard subject identification and object reference, contains a reference time (i.e., version of XML document to be observed with respect to a referred time) and request time (the current time of a request). Enforcement of an access control is by the view computation algorithm introduced above, but performed on a snapshot of an XML document (i.e.,
CHAPTER 8. STATE OF THE ART

8.4. XML VIEWS

XML document with respect to reference time).

8.4.3 Regulating Access to XML Documents: Gabillon et al.

The initial approach was introduced in [44], where authorization rules are stored in XML Authorization Sheet (XAS). Subjects are user identifiers that can be organized into groups. A special XML Subject Sheet (XSS) is used to describe existing users and groups. The latter can be ordered into a hierarchy. Hence, subject in XAS can be identified by XPath expression. Objects are expressed in XPath fragment \{child, descendant-or-self, *, [], label, $variable\}, where $variable is a name of variable that is instantiated (from user properties or context) during evaluation of access control rules. Negative and positive authorizations are admitted. The action is read. Authorizations are propagated only to text children and attributes of an element (i.e., only local propagation is considered).

The notion of priority is used to resolve conflicts between positive and negative authorizations that match the same subset of objects and subjects. Higher-priority rule takes precedence. Among several conflicting authorization rules with the same priority, the last one in authorization sheet takes precedence. This is true also for rules applicable to the same subject but located in more and less specific groups. The use of priorities is justified by natural applicability to XSLT processor (like Apache Xalan [91] or Oracle XSLT processor [99]), i.e. XAS can be easily converted to XSLT that is then used to filter out forbidden subtrees of XML document. The default semantics of security policy is closed. Denial downward consistency (see [68]) is enforced as well.

The algorithm first identifies access control rules related to the requesting user (this operation can be performed in time \(O(|XAS|)\), where \(|XAS|\) is the number of access control rules). Next, for each node \(O(|T|)\) of the source document, the algorithm finds corresponding policy rules (all XAS
rules can be scanned, hence time complexity is $O(|XAS|)$ and resolves conflicts between them according to rules described above (which can be done in $O(|XAS|)$ operations because it is necessary to select authorizations with the highest priority and apply denial takes precedence). Hence, the resulting time complexity is $O(|XAS| + |T| \times |XAS|)$.

This model was revisited in [45]. Instead of the authorization sheet associated with the whole XML document, the proposal introduced an access control list associated with each node of a source tree. The access control list consists of several authorization attributes of the form $<\text{subject, privilege}>$. At the same time each authorization attribute can belong to several nodes.

Association of authorization attributes with the nodes of source document resulted in the possibility to support dynamic trees. It means that user is allowed to perform not only read operations but also write. The latter can take one value from the set \{delete, insert, update\} which means, respectively, that the user is allowed to delete the node and its subtree, to insert a new subtree node, and to update a value of a node. Granted write privileges implies granted read privilege.

The algorithm removes all the nodes (with subtrees) that do not have privilege read granted to user. The result is used to control write operations. Since, the version in [45] is subsumed by that of [44], the same complexity results apply.

### 8.4.4 Access Control on XML Relationships

[43] considers not only ancestor-descendant relations but also siblings. In addition, a protection of associations among ancestors and descendants is proposed. Namely, relationship authorizations are defined over the XML view calculated by any method presented in the current section above. A relationship authorization is a tuple $\langle \text{Anc, Desc, Path - visibility, Sibling} \rangle$, \ldots
where Anc and Desc specify ancestor-descendant associations to be protected; Path – visibility describes the method of such a protection: path cloning, path elements depersonalization, path shortcutting; Sibling represent a set of siblings association with which should be preserved. Policies on resolving conflicts targeting the same Path – visibility and Siblings are provided.

8.5 Cryptographically enforced access control

8.5.1 XML Pool Encryption

The work present in [46] discusses a cryptographical method of hiding of forbidden XML nodes. The proposal is motivated by the fact that XML Encryption [101] is able to operate (encrypt/decrypt) only with the whole subtrees in the XML tree, while very often, there is a need to hide a certain node leaving its descendants visible. To deal with such a problem, an XML Encryption Pool was introduced in [46].

The basic idea is to encrypt each node separately with a unique node key. These encrypted nodes are removed from their original positions in the XML document into a pool of encrypted nodes. The keys to nodes permitted for the user constitute a key pool which is delivered to the user. Thus, the user’s view is a pruned XML document where forbidden nodes are completely eliminated and their permitted children are attached to their permitted ancestor.

The main problem is to make decrypted nodes be aware of their position in the view. The solution is to traverse the XML tree by DFS assigning with every node two labels (“left” and “right”) denoting the value of a counter during, respectively, the first and the second visit of this node.
Here counter is incremented by a random positive integer value. For the leaf node, the counter is incremented twice.

Having decrypted permitted nodes, the user reconstructs the XML view using the labels assigned to the node during DFS. In particular, every pair (“left”, “right”) can be considered as a range [“left” and “right”]; the spanning range denotes an ancestor node for those nodes which ranges are spanned. For example, a node with the largest range [“left”, “right”] is the root. The randomization of the counter obfuscates the fact of missing ancestors in a pruned XML tree.

8.5.2 Secure and selective dissemination of XML documents

A cryptographic method of protecting XML documents arose from view-based the Author-\(\mathcal{X}\) system and was motivated by the need in push architecture [20]. The difference is that only positive authorizations take place. Moreover, because of deny default policy, everything that is not permitted is forbidden.

The process of crypto-enforcement of access control policy is as follows. First of all, every element (and attribute) of the XML document is marked by policies applicable to it. These marking partitions the document into groups of XML elements with the same set of applicable policies. A distinct key is generated for each group. Finally, a well-formed encryption is constructed based on the partitioning and the set of corresponding keys. The most time consuming operation is marking since it takes \(O(|ACR| \times |T|^2)\), where \(|ACR|\) is the number of access control rules, \(O(|T|)\) is a size of XML \(T\). All the other steps require \(O(|T|)\) time.

Later works contain extensions of [20]. In particular, [12] introduces temporal dimension into access control rules and provides a method for generating \(|ACR|\) (instead of \(2^{|ACR|}\) in [20]) different keys. To make access control rules compliant with those of Author-\(\mathcal{X}\), negative policies, as well
as updates in a set of access control rules, were introduced in [25]. Finally, authentic third-party distribution from Owner to Client via Publisher was provided in [14, 26]. More precisely, [14] concentrates on ensuring authenticity, integrity and completeness of the answer to a query of the Client in the presence of an untrusted Publisher which can make an attempt to reduce or to alter the answer set. Merkle hash function is used to enforce authenticity. This function associates a hash value to every XML element in a bottom-up manner. Later, confidentiality aspects, when the Publisher may return a superset of the data permitted for the Client, were introduced in [26] which exploited the key generation technique of [12] and XML encryption of [20]. In addition, query answering over encrypted XML documents was proposed. In a nutshell, user is supplied with a query template (i.e., XML document without element/attribute content but with partitioning information and identifiers of policies applicable to every XML element) and a view (i.e., an authorized XML view extracted from the query template). The view is then used for query formulation. Before being submitted, the user’s front-end encrypts every location step of user’s query using keys derived from policy identifiers. Next, the encrypted user query is completed with the intermediate steps that represent pruned forbidden elements. Finally, partitioning information substitutes constants in predicates. The resulted query is sent to the Publisher and can be easily evaluated by the latter.

### 8.5.3 Conditional access for XML documents

Miklau and Suciu extends the Bertino’s idea of secure and selective dissemination of XML documents with the notion of conditional access control rules [63], which generalizes the term “subject” so that an authorization is based not on a network identifier or user name, but on the knowledge which is presented to the user.
8.5. CRYPTOGRAPHICALLY ENFORCED ACCESS CONTROL

The access control policy is expressed by a flexible language of conditional access control rules (CARs) in the following form:

\[ r ::= C : (\{B\} \rightarrow \{F\}) \]

where \( C \) is a single XPath expression specifying a context, \( B \) is a set of bindings representing the user’s knowledge, and \( F \) is a set of free values or accessible XML nodes.

The initial version of cryptographically enforced conditional access for XML document was presented in [63]. The idea is based on techniques of encrypting relational tables. Each context node \( x \) is replaced by the collection of encrypted tables \( T_r(x) \), plus metadata about the encoded bound and free values and the paths to them. Here \( T_r(x) \) is a binary table constructed from pairs \((b, f)\), where \( b \) is a concatenation of bound data values, \( f \) is a concatenation of free data values, \( r \) is a conditional access control rule with \( x \) matched as context node. The tables are represented in XML as list of rows containing data of two (for binary tables) columns.

The effect of CAR is the following: within context node \( c \), if bound value \( b \) is provided, then the subtree rooted at free value \( f \) is accessible, except for the inner context \( c' \) that may be specified by another rule. However, it is not discussed the case when \( f \) and \( c' \) coincide, i.e. an access conflict happens.

In [64], the approach was revisited. Conditional access control rules are now expressed as queries using XQuery extended by KEY and TARGET clauses representing bindings and free variables respectively. There are three kinds of bindings or keys in the proposed model: exchange keys that are produced by the data owner and simply protect a sub tree rooted at nodes associated with the key, inner keys that are generated by the system for the normalization process and represent metadata nodes to protect exchange keys and data value keys that are all the values of text nodes,
element tags and attribute names of the XML document, i.e. bindings in the previous model [63].

The query engine evaluates the policy and produces a model of “protection” over the XML tree where each node of the XML tree is associated with (i) a guard formula \( \sigma \)

\[
\sigma ::= true \mid false \mid k \mid \sigma \lor \sigma' \mid \sigma \land \sigma'
\]

which represents a combination of keys needed by the user in order to access the node; (ii) and a necessity formula, which is a disjunction of all guard formulas in the path from the root to the required node in the document.

The initial protection model can be optimized (e.g., multiple formula usage can be reduced to a single one, nested protections can be eliminated, etc.) and normalized (i.e., every node becomes protected by an atomic guard formula). The resulting protection model is subjected to an encryption procedure exploiting the W3C Recommendation on XML Encryption and Processing [101].

The user holds both a copy of the protected data instance and a set of keys. The latter allows the user to query the former. Finally, to improve a performance of the process of querying, the query processor is guided by means of an XQuery extension: \( \text{access}(\text{tag}, k_1, k_2, \ldots) \) that means “XML tag \text{tag} is protected by keys \( k_1, k_2, \ldots \).”

Other issues covered by [64] include: deciding consistency of policies (i.e., whether a protection exists), security properties, and experimental results.

The formal proof of secrecy-preserving property of the introduced protection is provided by Abadi et al. in [1].
8.5.4 Hierarchical and Role-based Access Control for XML Documents

Unlike previous papers, in [29] an object is defined by means of XPath filters [97] which use set operations to operate document subtrees.

The idea of key derivation has the same initialization phase like in [12], i.e., elements with the same set of access control rules are protected by the same key. However, keys are not generated, but key hierarchy is identified: a key corresponding to a set of policies $S$ is a child of a key corresponding to a set of policies $S' \subset S$. In addition, a master key is added as a parent of all those keys that do not have a parent. Then, for every position in the key hierarchy, a corresponding key is derived according to the proposal of Akl and Taylor [2] so that every user assigned to a security label $l$ can decrypt any object with security label $l' \leq l$. Thus, associating every key with a particular role, principles of role-based access control are satisfied.

Although the model was developed for “push” architecture, authors show how it can be utilized in “pull” scenario. In particular, they propose to use XSLT [99] to XML nodes with attributes representing accessibility. Then, XSLT is used again to calculate user’s view of the document, where only XML nodes with accessibility attribute that divides user’s key are visible.

In contrast with [12, 64], such a model delivers only one key (of smaller size, compared with [12]) to the user. Moreover, a proposed key derivation technique does not allow to deduce a key for a role of a higher clearance if two users of lower clearance role cooperate, i.e., “no read up” property is assured.

As it was declared in [29], conflicts between policies were not considered in the paper. Another limitation is that the model is applied only to nested subtrees; otherwise, overlapping of accessible regions for different
roles results in proliferating number of keys for a single role.

8.5.5 Securing XML documents with one private key

Another improvement of Bertino’s push model is considered in [90] by Zhang et al. The idea lies in the usage of an asymmetric encryption schema which allows to map one public key to several private keys. This public key is actually secret and is used for encryption. Private keys are also secret and are used for decryption. Moreover the encryption key is independent on the number of users associated with it. An over schema of public and private key generation is based on secure broadcasting proposed in [67].

The benefits of the proposal are following: one user needs only one private key; credentials or presence/absence of some users in the system does not affect other all the other users; the secure channel for key distribution is not needed.

8.6 Relational Access Control for XML

The proposals of this group exploit relational storage for XML representation. Their authors claim that query evaluation in relational databases is faster due to indexation and optimization techniques inherent to RDBMS.

8.6.1 XENA

[80] shows how to enforce security policy when XML is stored as relational databases. For this purpose, XML sEcurity eNforcement Architecture (XENA [81]) was developed. In XENA, the mapping between XML relations and relational tables is performed by XStorM [83].

Access control rules, their propagation and conflict resolution rules rely
on existing policy models for XML (e.g., [31], [11]). In addition, each rule has a special attribute that may have values hard, soft, and owner. Hard and soft rules are specified by an administrator, hard rules have the highest priority, while soft rules are of the lowest priority. Ownership rules are between the hard and low ones and specified over a particular fragment of the XML by the user if the user is the owner of that XML fragment. This last kind of access control rules is introduced with the purpose of modeling distributed XML access control. Namely, the owner can see the owned XML fragment and assign access control rules to it. Access to the rest of the XML is specified by other ownership rules or administrator’s hard and soft rules.

User query is parsed and analyzed w.r.t. access control rules. For example, if the user cannot view certain elements, access controller will take out those element from the user query. After analysis, the user query is translated into an SQL query only that information that is permitted for the user. The constructed SQL query is evaluated over relational representation of the XML. The result set is checked again against access control rules to filter out those answers that were not handled before, and the answer is transformed into an XML tree which is delivered to the user.

Even though the performance of query answering is improved due to relational storage, policy enforcement takes place at the XML document and, hence, slows down an overall evaluation process.

8.6.2 XML access control using relational databases

The proposal by Lee et al. [58] stores XML in relational tables constructed by XRel [96] and representing information on paths in the XML. Next, the current proposal also benefits from the relational storage during policy evaluation. Namely, XPath-based policies (e.g., [31], [11]) are converted into relational tables. The paper does not give well-defined query answering
proposals but rather outlines research directions in this area. However, the authors are prone to static analysis of an SQL query that is equivalent to the user’s XPath query.

8.7 DTD-based views

All the previous approaches enforce a security policy directly on the XML: either during query evaluation or while constructing the XML view. The proposals of this group reason at the DTD level.

8.7.1 Secure XML views

Stoica et al. in [79] propose producing single-level views of an XML document with multilevel security labels [61, 76] attached to every its element. For this purpose, semantic correlation between tags, modification of initial data structure and using cover stories are exploited. Thus, the concepts of Minimal Semantic Conflict Graph (MSCG) and Multi-Plane DTD Graph (MPG) are introduced.

Namely, MSCG is used to preserve semantic associations among some XML tags in partial views. The edges between related tags are labelled depending on the context. These labels serve as a cover story during the process of partial view retrieval, thus avoiding semantic conflicts and illegal inferences. For example, if we want to preserve a semantic association between physician and patient name but eliminate an existing named relation, e.g., clinical trial, military trial or ordinary trial, in some XML security view, we have to create an MSCG with nodes physician and patient name connected by means of edge with a label emergency record.

In its turn, MPG derives from the DTD a correspondence between the set of tags and the set of security labels. The MPG structure is constituted of several security levels called planes, each conforming to one security
label. Every security plane has an associated security space which is the set of dominated planes, i.e. planes that correspond to lower security labels. MPG has two reduced forms: the first and the second ones. An MPG is in the first reduced form if for any security space $SP$ it does not contain edges intersecting (going out or in) $SP$. An MPG in the second reduced form contains only those edges which are directed from a dominated plane to a dominating plane. In both cases, retrieval of a partial view implies ignoring the nodes and edges that are outside of the security space.

The algorithm to reduce MPG to the first reduced form is based on a sequence of eight procedures:

1. remove all edges $(A, B)$ such that $A$ and $B$ belong to the same security plane and all single disconnected nodes from MSCG: removed edges are redundant since their semantic association is preserved in MPG;

2. remove from MSCG all the nodes that are outside the given security space in the MPG: removed nodes have higher classification and, hence, are not included in the view;

3. create in the highest plane of the desired security space of MPG a new parent node $P$ for two semantically related nodes $A$ and $B$ (i.e., MSCG contains edge $(A, B)$ with label $P$) that are inside the same security space, and for any parent $Q$ of $A$ or $B$ outside the desired security plane replace the connection between $Q$ and $A$ or $B$ with the edge $(Q, P)$;

4. remove from MSCG edges $(A, B)$ between nodes $A$ and $B$ that have a common parent $P$ within the desired security space such that there is at least one path from $P$ to either $A$ or $B$ contained in the desired security space: edge $(A, B)$ is redundant;

5. remove from MPG the paths going outside the desired security space
if there exists a path inside the desired security space between the same nodes;

6. shortcut in MPG paths outside the desired security space that have the first and the last nodes within the desired security space, by directly connecting these nodes and deleting the edges going out of and into the desired security space;

7. remove from MPG outgoing edges from the desired security space if they are not part of a path that begins and ends in the desired security space;

8. remove from MPG incoming edges \((A, B)\) to the desired security space if they are not part of a path that begins and ends in the desired security space, and connect the root of the desired security space with \(B\).

The first procedure is invoked only once at the beginning. After that, procedures 2-8 are called until no modification can be performed. Hence, the complexity of the algorithm that calculates a security view for one specific security label is \(O(|D|^2 + |MSCG|)\), where \(|D|\) is the size of the initial DTD graph. Indeed, procedure 3 removes at least one edge (from \(Q\) to \(A\) or \(B\)) and adds at least 3 edges \(((Q, P), (P, A), \text{and } (P, B))\) such that \((Q, P)\) will be eliminated by procedure 8 and edges \((P, A)\) and \((P, B)\) will belong to security view). All the other procedures eliminate at least one edge from the MSCG. Summarizing, we should invoke all the procedures at least \(O(|D| + |MSCG|)\) times until no edge can be eliminated. On the other hand, some procedures require to find all paths between all nodes in MSCG. This operation can be done by depth-first search in \(O(|D|)\) operations. Hence, the first component of the sum is squared. Finally, the required space is \(O(|D|)\), because we only need to store DTD the graph.
8.7.2 Secure XML querying

A refined version of security views [79] was proposed by Fan et al. in [40].

This novel scenario of access control enforcement can be described by the following sequence of steps:

1. system administrator defines access specification for each class of users;
2. system perform automatic derivation of a sound and complete security view of DTD for each access specification;
3. the user, provided with the corresponding security DTD view, poses query over that view; a query over this view schema is rewritten to an equivalent query over the initial schema and optimized;
4. the rewritten query is evaluated over XML and the result returned to the requester.

This strategy provides (i) an inherent inference control according to the given policy and security specification, (ii) no “denial of service” answers, and (iii) an effective query mechanism avoiding view materialization.

In the first step, an access specification is defined over a subset of DTD edges \((A, B)\) and can have one of the values \(Y\|N\|q\), where \(Y\) means that the user is allowed to see the nodes of type \(B\) that are children of nodes of type \(A\), \(N\) forbids revealing of children \(B\) of \(A\), \([q]\) represents a conditional access to children \(B\) of \(A\). If an annotation of \((A, B)\) is not explicitly defined, \(B\) can obtain it from \(A\) by inheritance. However, an explicit annotation overrides inherited one. The root has a default \(Y\) annotation. Such an approach guarantees that the accessibility of each node in XML document corresponding to annotated DTD is uniquely defined w.r.t. security specification.

We must note, that approach by Fan et al. [40] cannot express conditions \(q_2\) and \(q_3\) from our running example, because neither parent nor
The security view of step 2 is defined as a pair $V = (D_v, \sigma)$, where $D_v$ exposes to the user the schema compliant with the user’s rights, $\sigma$ is invisible to the user and is a function constructed from XPath queries. For example, $\sigma(A, X) = A/B[q]/X$ means that node $B$ which satisfies condition $[q]$ should be skipped and its child $X$ should be appended as a child to node $A$. The algorithm of security view derivation either shortcuts inaccessible nodes (if they are elements of sequence production rule $\alpha \rightarrow B_1, \ldots, B_n$) or renames them to “dummy” (if they are elements of choice production rule $\alpha \rightarrow B_1 + \ldots + B_n$). The time complexity of the algorithm is $O(|D|^2)$, where $|D|$ is the size of the DTD document. We also need $O(|D|^4)$ memory to store the information about the $\sigma$ functions which are sets of $O(|D|^2)$ elements (all possible edges in DTD) requiring $O(|D|^2)$ space (the
longest path in DTD graph). The key feature of the derived security view is that it preserves the structure and semantics of the accessible parts of the original DTD document.

Here we have to underline that our view derivation algorithm is different from that of [40] where no splitting is introduced either at the stage of qualifier elimination or at the stage of different annotations transmitted to the same source. Instead, an unlabelled element type obtains the first label that arrives to it through DFS search. This completely compromises the emulation of policy enforcement at the instance level. Consequently, the algorithm of view construction presented in [40] is incorrect. In addition, dummy elements are introduced, and their meaning is not clear.

It is necessary to stress that XML views are never materialized. Instead, they remain virtual. The retrieval of the accessible information is performed at step 3. The requester’s query \( p \) based on \( D_v \) over the corresponding virtual instance \( T_v \) is transformed into an equivalent query \( p_t \) based on the initial schema \( D \) with the corresponding instance \( T \) so that the response is the same in both cases (i.e. \( p(T_v) = p_t(T) \)). The query rewriting algorithm is based on dynamic programming. In a nutshell, the user query is represented as a parse tree, and for each element of this parse-tree the rewriting is performed with respect to any DTD view node. Hence, the algorithm’s time complexity is \( O(|T_p| \times |D_v|^2) \), where \( |T_p| \) is the size of the query parse tree, \( |D_v|^2 \) as a rewriting subpaths *for every DTD view element* (hence, we have to find a rewriting between any pair of \( D_v \) elements according to the algorithm depicted in [40]).

The main differences with our query rewriting algorithm are: the algorithm derives a security view without any dummy element types which may be a source of sensitive information leakage. Therefore, the \( \sigma \)-function used in our query rewriting has different semantics. An extended XPath fragment has \( parent \) and \( descendant-or-self \) axes. Next, Fan et al. use dynamic
programming so that $QR_{(q,A)}$ is calculated for every DTD element type $A$; while we perform a rewriting of $q$ w.r.t. to a subset of relevant element types $A$ of DTD in a recursive manner. Finally, since we employ divide-and-conquer technique, our algorithm is faster: $O(|D_v| \times \log (m + 1))$ (without the reverse axes) vs. $O(|T_p| \times |D_v|^2)$ which is $O(|m| \times |D_v|^2)$ in the best case when there are no $\ast$, descendant-or-self axes, and unions.

8.7.3 A Dynamic Query Rewriting Approach

An absolutely different view specification was provided in [65]. More precisely, view specification language (SSX) is not a labeling as in previous research, but it is a sequence of schema transformation primitives: deletion of element/subtree, creation of a new element, copying of a subtree into other location, renaming of a node. It is clear that some elements of the sequence can logically depend on others, i.e., permutations among them are not allowed. Hence, security administrator should be careful while defining a security view.

An internal representation of view specification, called Security Annotated Schema (SAS), is generated to facilitate query rewriting algorithm. In a nutshell, SAS is an extension of the initial schema with newly created elements and additional annotations specifying whether current element type should be (conditionally) removed or how new element types should be populated in XML instance. Corresponding SAS construction algorithm provided in the paper is designed for XML Schema [102].

Because of logical dependency among security specification primitives that cannot be violated, query rewriting algorithm transforms user’s XPath query into an equivalent XQuery expressions that returns not only queried node but also permitted subtree rooted at the queried node. For this purpose, several rewriting rules depending on query (whether it is with qualifiers, or with axis descendant-or-self or without them) and annotations
of SAS are introduced.

8.8 Discussion

8.8.1 Policy

The mapping between existing policy frameworks and our proposal is summarized in Table 8.1. Note that our \( Y(N) \) label corresponds to “grant” (“deny”), + (−) of other models.

In Table 8.1, some values have an extension “ind.”. It means, that policy is applied on an individual node. For example, some nodes may propagate their annotation down to the descendants, others may not.

The method of Stoica et al. [79] is not introduced in Table 8.1, since the method assumes a fully annotation of DTD from the beginning. We also skip groups dealing with CAM and cryptography, since the first do not enforce policy but rather find more compact policy representation, the second reveal related information only if the user has keys for it. However, the priority of both methods is to propagate accessibility top-down with MSTP.

We can also note that the earlier proposals included DDC. More recent proposals repudiate this option since it is too restrictive.

8.8.2 Model and query answering

Table 8.2 shows a comparison of existing XML access control models. The column “XML view” answers the question if the model provides an XML view to the user. Namely, “Yes(weak)” means that the view is computed (either during run-time policy evaluation or by a materialization procedure) but reveals too much information, e.g., forbidden nodes with permitted descendants. On the other hand, “Yes (strong)” represents methods that
Table 8.1: Existing policy frameworks

<table>
<thead>
<tr>
<th>Method</th>
<th>HP</th>
<th>LP</th>
<th>SC</th>
<th>VC</th>
<th>MSTP</th>
<th>DDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kudo et al. [56]</td>
<td>*</td>
<td>≠none</td>
<td>hf</td>
<td>*</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Anutariya et al. [6]</td>
<td>*</td>
<td>*</td>
<td>hf</td>
<td>*</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>Bouganim et al. [21]</td>
<td>td</td>
<td>close</td>
<td>hf</td>
<td>dtp</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Qi et al. [71]</td>
<td>td ind.</td>
<td>close</td>
<td>none</td>
<td>dtp</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Gowadia et al. [51]</td>
<td>none</td>
<td>close</td>
<td>none</td>
<td>dtp</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Wang et al. [82]</td>
<td>td</td>
<td>close</td>
<td>hf</td>
<td>dtp</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>Murata et al. [68]</td>
<td>td ind.</td>
<td>closed</td>
<td>none</td>
<td>dtp</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Luo et al. [62]</td>
<td>td ind.</td>
<td>closed</td>
<td>none</td>
<td>dtp</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cho et al. [28]</td>
<td>td ind.</td>
<td>open</td>
<td>none</td>
<td>dtp</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>XENA [80]</td>
<td>td ind.</td>
<td>closed</td>
<td>none</td>
<td>dtp</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Lee et al. [58]</td>
<td>td ind.</td>
<td>closed</td>
<td>none</td>
<td>dtp</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Gabillon et al. [44]</td>
<td>td ind.</td>
<td>* ind.</td>
<td>none</td>
<td>priority, order</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Damiani et al. [31]</td>
<td>td ind.</td>
<td>closed</td>
<td>none</td>
<td>dtp</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bertino et al. [11]</td>
<td>td ind.</td>
<td>closed</td>
<td>none</td>
<td>dtp</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Fan et al. [40]</td>
<td>td</td>
<td>none</td>
<td>≠lf</td>
<td>none</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Our method</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 8.2: Existing XML access control frameworks

<table>
<thead>
<tr>
<th>Method</th>
<th>XML view</th>
<th>DTD view</th>
<th>Query asking over</th>
<th>Query answering by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kudo et al. [56]</td>
<td>Yes (weak)</td>
<td>No</td>
<td>initial XML</td>
<td>PE</td>
</tr>
<tr>
<td>Murata et al. [68]</td>
<td>No</td>
<td>No</td>
<td>initial XML/DTD</td>
<td>rewriting, PE</td>
</tr>
<tr>
<td>Gabillon et al. [44]</td>
<td>Yes (strong)</td>
<td>No</td>
<td>XML view</td>
<td>XPath evaluation</td>
</tr>
<tr>
<td>Damiani et al. [31]</td>
<td>Yes (weak)</td>
<td>Yes (loosened)</td>
<td>DTD view</td>
<td>XPath evaluation</td>
</tr>
<tr>
<td>Bertino et al. [11]</td>
<td>Yes</td>
<td>No</td>
<td>XML view</td>
<td>XPath evaluation</td>
</tr>
<tr>
<td>Cho et al. [28]</td>
<td>No</td>
<td>No</td>
<td>initial XML/DTD</td>
<td>rewriting, PE</td>
</tr>
<tr>
<td>Stoica et al. [79]</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Fan et al. [40]</td>
<td>No</td>
<td>Yes</td>
<td>DTD view</td>
<td>safe rewriting</td>
</tr>
<tr>
<td>Our method</td>
<td>Yes</td>
<td>Yes</td>
<td>DTD/XML view</td>
<td>XPath evaluation, safe rewriting</td>
</tr>
</tbody>
</table>

compute too restrictive views because of DDC.

The next column shows the presence of DTD view. Next, query asking shows the sources of information that are assumed to be used by the user to formulate queries. Obviously, those methods that are described by the
word “initial” leak information and, hence, unreliable.

The last column shows the method of query evaluation. “PE” means that policy is evaluated run-time along with query evaluation. “Rewriting” means that the query somehow optimized before being evaluated. XPath evaluation means that query is directly evaluated over XML or XML view without policy a concurrent policy analysis. Finally, “safe rewriting” means that policy is incorporated into the user query so that the query returns the result set of all and only permitted information.

As Table 8.2 shows, only the last two methods are reliable and efficient. However, our method is more flexible than by Fan et al. [40] since we provide both XPath evaluation over materialized view and safe query rewriting and evaluation over the initial XML.

8.9 Other XML access control support

8.9.1 Derived access control specification for XML

The focus of Goel et at. in [47] is on security policy specification rather than on enforcement. Access control rules defined at the schema level can be used for deriving new access control rules including content-based constraints of requested and other documents, environmental information like time and place of request initiator, information about possessed privileges. As a result, hardcoding rules for each document and its parts is no more necessary. This feature is especially valuable for systems with dynamic XML documents.

System can have prespecified set of privileges expressed, for example, in XACL [55] form \((\text{object}, \text{subject}, \text{action})\). XQuery is used to derive a dynamic rule set \((\text{object}', \text{subject}', \text{action}')\) applicable to the user \((\text{subject}')\) re-
quest (to do action' on object'), where a special predicate \texttt{access(subject, object)} can be used to define action'. Even more, object' can be defined for other XML documents, and subject' may have a recursive structure (e.g., manager may have a manager etc.).

An example of a derived rule: doctor is able to see patient’s treatment if he is able to see the diagnosis. Another example: one can maintain some part of XML document if one can read that part. More generally, the system allows to permit/deny access to some part of XML document if the user has an access to the other part of XML document (knowledge-based access control), if the request time or the transaction environment satisfies some condition (context-based access control), if some XML part satisfies some condition (content-based access control), if user properties satisfy some condition (credential-based access control). Finally, an extended set of access control rules can be enforced by some of the existing access control mechanisms, for example? Bertino’s Author-X [11, 16, 19], Damiani’s XACP [32], etc.

\subsection{Secure XML publishing in the presence of data inference}

Schema transformation [40, 57, 73] can still suffer from the leakage of sensitive information when the user exploits common knowledge to infer more information from the schema of available data. For example, patients of the same ward have the same disease. Therefore, the hidden diagnosis of some patients of some ward can be easily guessed basing on the not hidden diagnosis of other patients of the same ward.

Secure XML publishing in the presence of data inference is addressed in [87] by Yang and Li. They consider common knowledge as XML constraints that have the common form "conditions"→"facts", i.e. "facts" take place if "conditions" on XML documents are satisfied. There are three types of constraints distinguished in [87]:

child constraint: \( n \rightarrow n/n' \), i.e. node of type \( n \) must have a child of type \( n' \);

descendant constraint: \( n \rightarrow n//n' \), i.e. node of type \( n \) must have a descendant of type \( n' \);

functional dependency: \( p/p_1 \rightarrow p/p_2 \), i.e. path \( p_1 \), which follows path \( p \), implies the existence of path \( p_2 \), which follows path \( p \) as well.

A set of constraints can be used to derive some additional information on a partial document \( P \). The maximum information which can be obtained from different sequences of constraints \( C \) constitutes \( M \), a maximal inferred document on \( P \) using \( C \), which contains any possible inferred information. The construction of \( M \) can be done iteratively over set of constraints \( C \): we apply constraints one-by-one until new information can be inferred (i.e., new document is considered as XQuery query which can be mapped to the old document).

More formally, sensitive nodes are described by regulating queries expressed in XQuery. A set of regulating queries defines a partial view \( P \) that is available for the user. Although all formalism of [87] is related to XML documents, we can consider regulating queries as an analogue of \( \sigma \) function in [40, 57, 73]. If evaluation of some regulating query \( A_i \) over \( M \) results in non-empty answer set, we say that \( P \) causes an information leakage. Valid partial document does not cause any information leakage.

The algorithm of computing valid partial document can be sketched as follows:

1. apply elimination of sensitive data according to regulating queries;

2. compute maximal inferred document and check if this document reveals some sensitive data in the presence of constraint set. If not, the
constructed document is a valid document with maximum amount of information to be delivered to the user;

3. otherwise, select constraints that are responsible for information leaks and remove all data that can be mapped by these constraints.

Since multiple partial valid documents can be constructed, application-specific requirements should be considered as well. For instance, edges labeled with unconditional \( Y \) should be included in the resulting valid document. More formally, AND/OR graph [69] can be used to represent how a goal (valid partial document) can be reached by solving subproblems (satisfying regulating queries and unconditional labeling).
Chapter 9

Conclusion

We have studied the problem of consistent and efficient enforcement of access control to XML documents. We analyzed the early works in which policy enforcement is hold on the XML document directly. Those approaches are highly inefficient since even the assignment of security labels to the XML requires evaluation of a large number of XPath expressions. The situation is often aggravated by the fact that this evaluation takes place for every user query. Although many further proposals tried to analyze user query w.r.t. structural properties of DTD and policy, to make policy look-up more efficient, the idea of run-time policy evaluation over XML was still unfeasible.

The current thesis proposes an efficient way to enforce a security policy at the DTD level. We developed an algorithm to calculate a schema of all and only available information that may be observed by the user. On the other hand, we maintain functions that are used by the system to preserve information about secured parts of the DTD.

Since we put our security labels on DTD edges, rather than on XML nodes specified by XPath, labelling is fast and efficient. The propagation of security annotation pushes labels from edges to nodes and then propagates them, emulating propagation in an XML tree. We can handle both top-
down and bottom-up propagation. This makes our approach flexible and customizable.

The XML view materialization is, basically, an operation of filtering out of the original XML those its parts that are forbidden according to the previously constructed functions about secured information. This results in an XML view that conforms to the DTD view. To the best of our knowledge, our proposal is the first that allows to the user to have a consistent pair of DTD view and XML view. Other methods skip either XML view materialization or DTD view calculation. Further, we proved that our approach is correct in the sense that a materialized XML view is isomorphic to an authorized XML view.

Due to the well-known limitations of materialized views, we developed an efficient algorithm of query rewriting. In this case, we incorporate hidden information into the original query formulated for the DTD view. The result query is equivalent to the original query in the sense that the result sets of both are the same. However, the rewritten query is evaluated over the original XML since it is formulated according to the structure of the original DTD.

Our experimental results show the effectiveness of our proposal w.r.t. the previous approaches based on XML view materialization via policy enforcement directly on the XML tree. The advantage in performance is due to fast labelling and DTD-based propagation of labels. On the other hand, performance of the original queries over the materialized XML view as compared to the rewritten queries over the original XML is not significantly different. This is because of complex XPath expressions constituting \( \sigma \)-function that should be evaluated either during XML view materialization or during evaluation of the rewritten query. However, query rewriting helps avoid limitations related to materialized views, e.g., integrity maintenance, storage capacity etc.
CHAPTER 9. CONCLUSION

Directions for future work include extension of our results to recursive DTDs. We also plan to investigate the definitions of security views and authorization specifications by supporting more complex XML Schema [102] instead of DTDs, query rewriting algorithm to XQuery [103]. Another possible extension is annotation propagation among siblings, rule-based propagation etc. The observation concerning leakage of information in the case of multiple queries is left for the future work as well. Other direction is related to investigation of an impact of ID/IDREFs on security views. For this purpose, we have started to investigate referential integrity and polyinstantiation issues in multilevel relational databases. Results of this study will allow us to approach the problem of access control for distributed and federated XML documents. Finally, we would like to find other applications of the proposed algorithms. In particular, we are working on the notion of business process views and workflow views. The need in such views may be motivated by separation-of-duties requirement fulfilment. Another possible application of the current thesis would be a generation of dynamic web-page and dynamic web-site structure depending on permissions of the user. We believe that our proposal will find many other useful and interesting extensions and applications.
Bibliography


[33] Ernesto Damiani, Sabrina de Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Design and implementation of an


[47] Siddhartha K. Goel, Chris Clifton, and Arnon Rosenthal. Derived
access control specification for XML. In *Proceedings of the 2nd ACM
Workshop On XML Security (XMLSEC’03)*, pages 1–14. ACM Press,
2003.

XPath queries. In *Proceedings of the 28th Conference on Very
Large Data Bases (VLDB’02)*, 2002.

[49] Georg Gottlob, Christoph Koch, , and Reinhard Pichler. XML task

[50] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient al-
gorithms for processing XPath queries. *ACM Trans. Database Syst.*, 

[51] Vaibhav Gowadia and Csilla Farkas. RDF metadata for XML access
control. In *Proceedings of the 2nd ACM Workshop On XML Security

[52] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Lan-

[53] Mingfei Jiang and Ada Wai-Chee Fu. Integration and efficient
lookup of compressed XML accessibility maps. *IEEE Transactions
on Knowledge and Data Engineering (TKDE)*, 17(7):939–953, July
2005.

[54] Yaron Kanza, Alberto O. Mendelzon, Renee J. Miller, and Zheng
Zhang. Authorization-transparent access control for XML under the
non-Truman model. In *Advances in Database Technology - EDBT
2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 222–
239, 2006.


[107] Personal communication with Elena Ferrari.

[108] Personal communication with SungRan Cho.

[109] Personal communication with Wenfei Fan.
