SPEEDING UP XML QUERYING

Satisfiability Test & Containment Test of XPath Queries in the Presence of XML Schema Definitions

Dissertation

by

Jinghua Groppe

Lübeck, Germany, July 2008

Jinghua Groppe
Institut für Informationssysteme
Universität zu Lübeck
Ratzeburger Allee 160
D-23538 Lübeck
Germany
E-Mail: Jinghua.Groppe@ifis.uni-luebeck.de

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
der Technisch-Naturwissenschaftlichen Fakultät
der Universität zu Lübeck
in Deutschland

Dekan:	Prof. Dr. Enno Hartmann
Gutachter:	Prof. Dr. Volker Linneman
	Prof. Dr. Walter Dosch

Tag der Promotion: 9. Juli 2008

My intelligence comes from my mother
My interest in study comes from my mother
My character comes from my mother

ACKNOWLEDGEMENTS

I would like to express my thanks to all of the people, who once gave me help, support, encouragement and happy times in my study, in my live and in my work not only in our PhD work. However, it is a pity that I cannot mention each of them.

I would like to thank Professor Volker Linneman for supervising my dissertation. Without the supervision of Professor Linneman, I could not have had my promotion examination today.

I would especially like to thank Professor Franz Rammig in the University of Paderborn in Germany. Professor Rammig opened to me the gate of becoming a scientist, and paved a way for my dissertation. Without Professor Rammig, I could not have finished my dissertation today.

I also want to thank our colleges in the institute of information system of the University of Lübeck in Germany: Professor Linneman, Angela, Anke, Jana, Sven, Christoph and Nils, for the invaluable presents, they give me when I pass the promotion examination. The doctorate cap, I will keep forever.

I also want to thank our ex-colleges, especially the members in *Advanced Design System* group in C-lab/Paderborn University in Germany, for a pleasurable and fruitful work in this group.

I also want to thank my friends in Paderborn: Dr. Yuhong Zhao, Dr. Hongbi Zhang, Dr. Bo Bu and Miss Hui (Hellen) Liang for the friendship, the help and the happy time they gave me.

I would once again like to thank Dr G. Dick van Albada in the University of Amsterdam. Dr. Albada not only supervised my master thesis, but also helped to bring about a quick finish of various management procedures such that I can begin the work as a scientist in the University of Paderborn in time. I also want

to thank Dr Kamil Iskra and Dr Zhiming Zhao once again for the help, which they gave me in my master study.

Many thanks to my family for the love, which it has been giving me no matter where I am and no matter what I do.

Many thanks as well to my parents-in-law and my sister-in-law and her family for their wonderful presents.

Special thanks to my husband Sven and my son Nils. Sven gives me a family and a happy life such that my heart is not lonely and does not wander. In the three years of from finding out the topic to pass the promotion examination, Sven plays with our son in many weekends and evenings such that I can work on papers; behind me Sven lifts his “hurry up” stick over my head such that I have to run up my dissertation. Because of Nils, I do not almost have any own time after picking him up from the kindergarten. However, my son bestows my work and life so much meaning. Let us work hard to build an eternal peaceful and beautiful world for Nils and for all the children.

Lübeck, Germany, July 2008 Jinghua Groppe

ABSTRACT

This dissertation develops approaches to testing the satisfiability and the containment of XPath queries in the presence of XML Schema definitions in order to speed up XML querying.

XML provides a simple yet powerful mechanism for information storage, processing and delivery, and is a widely used standard data format. XPath is a basic language for querying XML data, and is embedded into many W3C standards, e.g. XQuery, XLink, XML Schema, XForm and Schematron, for addressing XML data. Therefore, XPath optimization plays a key role in speeding up XML query processing. The satisfiablity test and containment test of XPath are two important issues in XPath optimization.

An unsatisifable XPath query selects every time an empty result. Therefore, the application of the satisfiability test can avoid the unnessesary submission and the unnecessary evaluation of unsatisfiable queries, and thus can save querying costs. In programming languages, which embed XPath, like XOBE [Kempa and Linnemann 2003a], the satisfiability test can enable an efficient development of more robust applications by avoiding extensive tests and runtime failures caused by unsatisfiable queries. The satisfiability test can also speed up the execution of codes by the pre-computation of an empty result at compile time. Furthermore, the XPath satisfiability test plays an important role in other applications, e.g. XML access control [Fan et al. 2004], type-checking of transformations [Martens and Neven 2004] and XPath-based index update [Hammerschmidt et al. 2005].

The containment of XPath is another key factor for XPath evaluation. XPath containment can be used to minimize XPath expressions to speed up query evaluation. When using views to answer queries, the containment test is the underlying technique to decide if a new query can be answered using the results of previous queries. Using views to answer queries can significantly improve the performance of XPath processing, and reduce the communication and query costs by significantly decreasing shipped data, since part of query

evaluation has bee done when computing the cache, and since the partial or even the whole answer to the new query is already available at client side. XPath containment can also find its applications in inferring the keys of XML Schema and in testing the satisfiability of XPath queries.

Since the high complexity of XPath queries, it is not trivial to develop efficient approaches to checking XPath satisifiability and to checking XPath containment when schemas, especially recursive schemas, are in presence. [Choi 2002] shows that recursive schemas are often used in the real world. The existing solutions to XPath satisfiability consider only some subsets of XPath axes and non recursive schemas. In this thesis, we propose an approach to XPath satisfiability in the presence of XML Schema definitions, and support all XPath axes, and recursive as well as non-recursive schemas. Since XPath containment has a high complexity under constraints, there is lack of work on practical solutions to this issue. In this work, we develop an approach to checking XPath containment under constraints of XML Schema definitions.

Furthermore, we develop a data model for XML Schema and an XPath-XSchema evaluator based on the data model. We as well suggest an approach to rewriting and optimization of XPath expressions according to schemas. Our XPath-XSchema evluator evaluates XPath queries on an XML Schema definition, in order to check satisfiability and containment of XPath expressions with respect to the schema. We present a complexity analysis of our XPath-XSchema evalutor, which proves that our approach is efficient at typical cases. We present an experimental analysis of our satisfiability tester, which proves the optimization potential of avoiding the evaluation of unsatisfiable queries. We prove the correctness of our approach to XPath containment, and analyze the complexity of our approach. We develop a prototype of our containment tester and the experimental results show the efficiency of our approach.

Contents

Chapter 1 Introduction

In this chapter, we motivate our proposed approaches, describe our scientific contributions and explain the overall organization of the dissertation.

1.1 Motivation

As XML becomes increasingly popular as a language for data storage, automatic exchange and processing, larger and larger as well as more and more data are stored using XML. Table 1.1 presents several XML datasets with large data sizes. Therefore, speeding-up query processing of XML data becomes increasingly important. In this work, we focus on the satisfiability test and the containment test of XPath queries with respect to XML Schema definitions.

Table 1.1: Examples of large XML datasets

XML datasets	Data size	Reference
Computer science bibliography	389 Megabytes	[Trier 2007]
Protein sequence database	683 Megabytes	[Washington 2008a]
SwissProt knowledgebase	419 Megabytes (compressed)	[UniProtKB 2008a]
Tremble knowledgebase	2.64 Gigabytes (compressed)	[UniProtKB 2008b]
UniRef50 database	730 Megabytes (compressed)	[UniRef 2008a]
UniRef90 database	1.09 Gigabytes (compressed)	[UniRef 2008b]
UniRef100 database	1.34 Gigabytes (compressed)	[UniRef 2008c]

1.1.1 XPath

XPath [W3C XPath1.0 1999][W3C XPath2.0 2003] is a language for addressing the information in an XML document by navigating through elements and attributes in the XML document. As well as being a standalone XML query language, XPath is also embedded in other XML languages, e.g. XSLT, XQuery, XLink and XPointer, for specifying node sets in XML documents.

XQuery [W3C XQuery1.0 2004][W3C XQuery1.0-XPath2.0 2004] is an XML query language, which uses XPath expressions to identify and extract elements and attributes from XML documents. XQuery 1.0 and XPath 2.0 share the same data model and support the same functions and operators. XPath 2.0 is a subset of XQuery 1.0.

The XST Transformations language (XSLT) [W3C XSLT1.0 1999][W3C XSLT2.0 2003] is used to transform an XML document into another XML document, or another type of document, like HTML [W3C HTML4.01 1999] or XHTML [W3C XHTML1.0 2002]. XSLT uses XPath expressions to define parts of the source document, which should match one or more predefined templates.

The XML Pointer language (XPointer) [W3C XPointer1.0 2001] defines how individual parts of an XML document are addressed and is an extension and customization of XPath. XPoint uses XPath to build URI references that reference parts of XML documents. Based on XPath features, URI references can address individual points and elements as well as lists of elements, attributes or character data.

The XML Linking Language (XLink) [W3C XLink1.0 2001] is an XML markup language used for creating hyperlinks in XML documents. The URI references in XLink can contain an XPointer, which in turn contains an XPath expression. XPath is also used in other W3C specifications such as XML Schema [W3C Schema1 2004] and XForms [W3C XForms1.0 2007].

Therefore, XPath is an important construct in these W3C's standards, and optimization of XPath is fundamental for speeding up XML querying in all the instances of these languages. Automatic optimization techniques have been developed and have been used for decades in database management systems for the deductive (e.g [Bancilhon et al. 1986], [Levy at el. 1995] and [Behrend 2003]) and relational (e.g. [Jarke and Koch 1984], [Ioannidis 1996] and [Chaudhuri 1998]) worlds. Different from the query languages for relational databases, XPath supports complex navigational paths and qualifiers. It is not

trivial to develop efficient XPath evaluators and all major XPath engines have a high runtime complexity [Gottlob et al. 2002]. Therefore, there is a need to logically optimize XPath queries. The satisfiability test and the containment test of XPath queries are two important issues in logical XPath optimization.

1.1.2 Satisfiability of XPath queries

An XPath query is satisfiable if there is an XML document on which the evaluation of the XPath query returns a non-empty result; an XPath query is unsatisfiable if the evaluation of the query on any XML document returns every time the empty answer. Therefore, using the satisfiability test can avoid the submission and unnecessary evaluation of an unsatisfiable query, and thus can save processing time and users' cost. In addition, the XPath satisfiablity test is important for consistency problems, e.g. XML access control [Fan et al. 2004], type-checking of transformations [Martens and Neven 2004], and XPath-based index update [Hammerschmidt et al. 2005].

The satisfiability test of XPath queries also plays an important role in processing of programming languages, which embed XML constructors and XML query languages, e.g. XOBE [Kempa and Linnemann 2003]. XOBE is a Java-based programming language to process XML data conforming to a given schema, and is developed by the Institute of Information Systems of the University of Lübeck in Germany. XOBE embeds XPath to access and select XML data. Application of the XPath satisfiability test to the programming of the embedding languages can generate more efficient and more robust programs. If an XPath expression is detected as unsatisfiable, some computations can be already done at compile time, and thus greatly speed up program processing at run-time. If an unsatisfiable XPath expression is not allowed, then the satisfiability test can help to find errors at compile time to avoid run-time failures and extensive tests. The runtime tests are hard to be exhaustive, so even an extensive test can not guarantee the complete correctness. Therefore, the satifisfability test of XPath queries enables a fast development of reliable XOBE applications.

Therefore, many research efforts focus on the satisfiability test of XPath queries with or without respect to schemas, e.g. [Benedikt et al. 2005], [Groppe and Groppe 2006a], [Groppe and Groppe 2006b], [Groppe and

Groppe 2006c] [Groppe and Groppe 2008], [Groppe and Linnemann 2008], [Hidders 2003], [Kwong and Gertz 2002] and [Lakshmanan et al. 2004].

1.1.2.1 XPath satisfiability in the presence of schemas

In the absence of schemas, the satisfiability test can detect two kinds of errors in XPath queries:

- The first kind of errors is that the structural properties of XPath queries are inconsistent with the XML data model
- the second kind of errors is that the constraints from an XPath expression itself are inconsistent with each other

For example, the XPath query Q1=/following-sibling::a is unsatisfiable, because Q1 contains one of the first kind of errors, i.e. the root node / has no sibling node according to the XML data model. The query Q2=//person/age is tested as a satisfiable XPath query without respect to schemas. However, according to a given schema, e.g. the schema (see the appendix benchmark.xsd) in [Franceschet 2005], the element person does not have a child age. Thus, Q2 is unsatisfiable with respect to the schema.

The XPath query Q3=a[@v>2][@v<1] is unsatisfiable since Q3 contains the second kind of errors, i.e. @v>2 is contrary to @v<1. Q4=//catgraph/*[parent::*[not(edge)]] is satisfiable because Q4 conforms to the XML data model, and contains no visible conflicting constraints. However, if Q4 is rewritten to /site/catgraph/edge[parent::catgraph[not(edge)] according to a given schema, e.g. the one (see the appendix benchmark.xsd) in [Franceschet 2005], and is further optimized to /site/catgraph[not(edge)]/edge by eliminating reverse axes, then Q4 is unsatisfiable with respect to the schema. (We call Q4 is a query with *hidden* conflicting constraints.)

Therefore, we can detect more errors in XPath queries if we additionally consider schema information. We focus on the satisfiability test of XPath queries in the presence of the schemas formulated in the XML Schema language [W3C Schema1 2004] [W3C Schema2 2004].

Since XPath supports a number of navigational axes and complex qualifiers, it is not trivial to develop efficient satisfiability testers of XPath queries, when the constraints from schemas have to be considered. [Benedikt et al. 2005] theoretically shows that the complexity of XPath satisfiability depends

on the considered subsets of XPath queries and schemas, varying from PTIME to undecidable when the considered subsets of XPath and schemas increase. They also show that for some subsets of XPath, the complexity of the satisfiability problem is much lower without respect to schemas than with respect to schemas, and the presence of recursive schemas and negation in XPath queries finally lead to the undecidability of XPath satisfiability.

Existing approaches to XPath satisfiability support only partial features of the XPath language and schemas. [Kwong and Gertz 2002] and [Lakshmanan 2004] support some subsets of XPath axes and allow non-recursive schemas. We develop an approach to XPath satisfiability, which supports all the XPath axes, recursive as well as non-recursive schemas and negation operation in XPath. The satisfiability test for the XPath subset supported by our approach in the presence of the schemas supported by our approach is undecidable [Benedikt et al. 2005]. Therefore, we present an incomplete, but fast satisfiability tester, i.e. if our tester returns *unsatisfiable*, then we are sure that the XPath query is unsatisfiable, but if our tester returns *maybe satisfiable*, then the XPath query may be satisfiable or may be unsatisfiable.

1.1.3 Containment of XPath queries

Given two XPath queries Q1 and Q2, if for any XML document the result of applying Q2 is a subset of the result of applying Q1, then Q1 contains Q2, denoted as $Q1 \supseteq Q2$. XPath containment plays an important role in query optimization and other applications, e.g.

- in XPath minimization [Amer-Yahia et al. 2001] [Furfaro and Masciari 2003] [Wood 2001]: An XPath expression can not be minimal unless no part is contained by other parts of the XPath expression. Since the size of XPath expressions is a determinant of XPath processing performance, minimizing XPath expressions can speed up XPath querying.
- in using views to answer queries [Balmin et al. 2004][Xu and Özsoyoglu 2005]: *Views* store the results of previously answered queries in order to answer succeeding queries faster by reusing these results. The result of a query cannot be used to answer a new query unless the new query is contained by the answered query. Using views to answer queries is important

in many applications. In the context of query optimization, it can speed up query processing since parts of query evaluation have been done when computing views. This has a special benefit for the query languages like XPath, which have high complexity of computation. In the context of querying over network and client-server architectures, it can reduce the communication and query cost by significantly decreasing shipped data since parts or even all of the answer to the query is already available at the client. Using views to answer queries also plays a key role in database design and data integration.

- in inferring the keys of an XML Schema definition: If an XPath expression Q is defined as a key for an XML Schema definition, then all the XPath expressions, which are contained by Q, are also the keys of the schema.
- in testing the satisfiability of XPath: all the XPath expressions, which are contained by an *unsatisfiable* XPath query, are unsatisfiable, too; all the XPath expressions, which contain a *satisfiable* XPath query, are satisfiable, too.
- In database programming languages, which embed XML constructors and XML query languages: application of containment tests can improve the execution of programs by minimizing the XPath expressions and using the results of previous queries to answer new queries.

Therefore, many contributions deal with the problem of XPath containment with or without respect to constraints, e.g. [Deutsch and Tannen 2001], [Miklau and Suciu 2004], [Neven and Schwentick 2003], [Schwentick 2004] and [Wood 2003].

1.1.3.1 XPath containment under the constraints of schemas

The XPath containment under schemas has a high computational complexity compared with the XPath containment without schemas. For example, Q1=a[b] does not contain Q2=a without respect to any constraints. However, if a constraint specifies that b must occur if a occurs, then Q1 is equal to a semantically, denoted as Q1≡a, and thus Q1⊇Q2 under this constraint. If Q3=a/b[c] and Q4=a[b/d]/b, then Q3 does not contain Q4 without respect to any constraints.

However, if a schema defines that b can occur at most once then Q4≡a/b[d], and if the schema also specifies that if d occurs then c must occur, then Q3⊇Q4 with respect to the schema. The containment of XPath under schema constraints, even for XPath queries with only *child* axis and predicates, denoted as XPath$^{\{/,\ []\}}$, is coNP-complete [Neven and Schwentick 2003][Wood 2001]. [Wood 2003] identifies that containment of XPath$^{\{/,\ []\}}$ is intractable under schemas. The intractability of the containment for XPath$^{\{/,\ []\}}$ under schemas comes from the fact that inferring even some simple constraints from some schemas seems to be intractable [Wood 2001].

Existing works on XPath containment under integrity constraints and DTDs mainly theoretically study the complexity and decidability of XPath containment. Since XPath containment has a high complexity under constraints, there is lack of contributions to practical solutions to check XPath containment. Therefore, in this thesis, we fill this gap by developing a practical algorithm to test XPath containment in the presence of constraints formulated in XML Schema definitions. Our approach can support the reverse axes and the axes depending on the document order. The complexity and decidability of XPath containment, when these axes are allowed, are still unknown. Since the high complexity of containment test and the intractability of inferring even some simple constraints from a schema, we present a fast but incomplete approach to XPath containment in the presence of schemas. Given two XPath queries Q1 and Q2, our containment tester returns that Q1 contains Q2, or that Q1 *maybe* does not contain Q2.

1.1.4 XML Schema

The schema languages for XML define the XML documents by specifying the structure, semantics and data types of documents. XML documents are not necessarily associated with a schema. However, XML documents together with its schema become self-descriptive, and can generate more robust applications. Therefore, many XML documents are provided with a schema, and many applications and tools to process XML data require that all processed documents follow a given schema like [Sun 2001], [Microsoft 2001], [Exolab 2001] and [Oracle 2001]. Therefore, we study the satisfiability test and the containment test of XPath queries in the presence of schemas.

There is a number of XML schema languages available, like DTDs [W3C XML1.0 2004], XML Schema [W3C Schema1 2004] [W3C Schema2 2004], RELAX NG [OASIS 2001], Schematron [ISO Schematron 2006], XML-Data [W3C XML-data 1998], Examplotron [Vlist 2003] and DSD [Moeller 2002]. There is also a number of articles introducing and comparing different XML schema languages, e.g. [Lee and Chu 2000], [Vlist 2001] and [Wikipedia 2007].

DTDs, XML Schema and Relax NG are considered as the primary XML schema languages, each of which has its own advantages and disadvantages. RELAX NG is an OASIS RELAX NG Committee specification, released in May 2001. RELAX NG has many advantages of XML Schema and DTDs, and it also has some own advantages, e.g. it can define the *document* element of XML documents. However, the use and the support to RELAX NG are not widespread compared with the other two.

W3C's DTDs [W3C XML1.0 2004] and XML Schema [W3C Schema1 2004] [W3C Schema2 2004] are two widely used and supported XML schema languages. DTD is the earliest schema language for XML, and widely supported. DTDs are compact and highly readable and can be defined inline. However, DTDs are primarily structural in nature. DTDs have limited support for defining the type of data, and do not have ability to specify specific and precise data types above and beyond character data. Therefore, DTDs can not meet the requirement in describing XML structures and contents more precisely.

As well as imposing the constraints of structure and semantics on XML documents as DTDs do, the XML Schema language provides powerful capabilities for specifying more concrete data types on elements and attributes, most of which are not expressible in DTDs. The XML Schema language provides a large number of built in simple types and allows deriving new types for values of elements and attributes, which are only specified to be character data in DTDs. Thus, if the type of values of elements or attributes in an XPath query does not conform to constraints specified in the XML Schema definition, the XPath query selects an empty set of nodes for any valid XML document. For example, the query meeting[@date='01-05-*06*'] does not retrieve anything if the type of the attribute date is declared to have the format DD-MM-*YYYY*. Therefore, the powerful data-typing facilities supported by XML Schema provide another dimension for the satisfiability test of XPath queries. Since XML Schema can express more restrictions than a DTD, a DTD can be

easily transformed into an XML Schema representation, but in general, an XML Schema definition cannot be transformed into a DTD without loosing information.

Furthermore, the schemas written in the XML Schema language are XML documents, but the syntax of DTDs is completely different. Therefore, XML Schema can leverage various tools that have been built around XML, but DTDs can not.

1.2 Contributions

The main contributions of this thesis include:

- We develop a data model for the XML Schema language, which identifies the navigational paths of XPath queries on an XML Schema definition by mapping the *parent-child*, *preceding-sibling* and *following-sibling* relations in instance XML documents to the corresponding relations in an XML Schema definition. This model is the basis of evaluating XPath queries on XML Schema definitions
- We develop an XPath-XSchema evaluator, which evaluates XPath queries on an XML Schema definition based on the data model of the XML Schema language, and returns a set of schema paths. The schema paths integrate the constraints imposed by the schema, and thus provide means to check the satisfiablity and containment of XPath queries with respect to schemas. Our approach supports all XPath axes and recursive as well as non-recursive schemas. We define the formal semantics of the XPath-XSchema evaluator and analyze the complexity of our approach, which proves the efficiency of our XPath-XSchema evaluator at the typical cases. We develop a prototype of the XPath-XSchema evaluator, which shows the efficiency and usability of our evaluator.
- We develop a satisfiability tester of XPath queries to speed up query processing. Based on the schema paths of queries, our satifiability tester filters the XPath queries, (a) which do not conform to the constraints of semantics, structure, data type and occurrence imposed by an XML Schema definition, and (b) which contain visible and hidden conflicting constraints. We develop a prototype of our XPath satisfiability tester, which demonstrates

the potential of XPath optimization by filtering unsatisfiable XPath queries. A speedup up to several orders of magnitude for XPath processing is possible compared with common XPath evaluators.

- We suggest a practical solution to checking the XPath containment under the constraints of schemas. Our approach tests the containment of XPath queries in terms of the normalized schema paths of these queries. The normalized schema paths integrate the most constraints of schemas, which impact the containment test. We prove the correctness of our containment tester and analyze the complexity of the approach. Our experimental results show that our approach has low overhead.
- We develop an approach to rewriting of XPath expressions in order to optimize XPath queries, which eliminates the redundant parts and wildcard node-tests, eliminates reverse axes and recursive axes wherever possible.

1.3 Organization

This thesis is organized as follows:

- Chapter 1 presents the motivation, contributions and organization of this work. This chapter also describes the test system and data, which are used in the experiments in this thesis.
- Chapter 2 introduces the XML technologies, which are related to our work.
- In Chapter 3, we develop a data model for the XML Schema language to identify the navigational paths of XPath on an XML Schema definition. We provide a formal description of XML Schema and the formal semantics of the data model. This data model is the basis of our XPath-XSchema evaluator.
- Chapter 4 presents our XPath-XSchema evaluator, which evaluates XPath queries on XML Schema definitions based on the data model of the XML Schema language developed in Chapter 3. Our XPath-XSchema evaluator checks whether or not an XPath query conforms to the constraints in a given schema, and computes a set of schema paths, which integrates the constraints from the schema. The satisfiability test, containment test and

rewriting of XPath queries are built on schema paths. In this chapter, we describe the data structure of schema paths, define the formal semantics of the XPath-XSchema evaluator, and analyze the complexity of our approach.

- The rewriting of XPath queries is presented in Chapter 5, including the functions that map schema paths to XPath expressions and the rules to optimize the mapped XPath queries.
- Chapter 6 describes our satisfiability tester of XPath queries, and presents the approach to filtering the XPath queries, which contain invisible conflicting constraints, by rewriting the given XPath queries according to the schema. In this chapter we present a comprehensive performance analysis on detecting of the unsatisfiable XPath queries, which do not conform to the constraints in an XML Schema definition and which contain invisible conflicting constraints.
- Chapter 7 presents our approach to the containment of XPath queries under the constraints of schemas. Our approach checks the XPath containment in terms of the normalized schema paths of the queries. Chapter 7 describes the mechanisms to normalize schema paths and check the containment of schema paths. In Chapter 7, we also prove the correctness of our approach, analyze the complexity of our approach, and present the performance analysis of our prototype.
- Chapter 8 describes the related work and Chapter 9 ends up with the conclusions and future work.

1.4 Test system and data

The test system for all experiments is an Intel Core 2 CPU T5600 processor (where we disable one CPU), with 1.83 Gigahertz and 2 Gigabytes RAM, Windows XP as operating system and Java VM version 1.6.0. We use two popular XML engines, the XQuery evaluators Saxon version 8.0 (see [Kay 2004]) and Qizx version 0.4pl (see [Franc 2004]), to evaluate the XPath queries on XML data.

XPathMark (see [Franceschet 2005]) is a popular benchmark for XPath, and contains a set of queries, which covers the main aspects of the language XPath 1.0 [W3C XPath1.0 1999]. These queries have been designed for XML documents generated under XMark (see [CWI 2003]), which is a popular benchmark for XML data management.

Therefore, we use the XPathMark benchmark (see [Franceschet 2005]) as the source of our experimental data, and generate XML data from 0.116 Megabytes to 11.597 Megabytes by using the data generator of [Franceschet 2005]. For the experiments, an XML Schema definition benchmark.xsd (see the appendix benchmark.xsd) is manually adapted according to the DTD benchmark.dtd of the XPathMark benchmark (see [Franceschet 2005]) and the instance documents in order to integrate as many constructs of XML Schema as possible and in order to specify more specific data types for values of elements and attributes, which are all declared as #PCDATA in benchmark.dtd.

Chapter 2 XML Technologies

In this chapter, we introduce the basic technologies related with our work. The introduction of the XML technologies does not attempt to be exhaustive. Instead, we focus on the basic features of the XML technologies. We refer the interested readers to the corresponding specifications for the details of these XML technologies.

2.1 XML

The eXtensible Markup Language (XML) [W3C XML1.0 2004] is a general-purpose markup language for the storage of data in order to facilitate sharing of structured or semi-structured data across different systems, particularly via the internet.

2.1.1 XML history and virtues

In the 1970's, Charles **G**oldfarb, Ed **M**osher and Ray **L**orie developed the GML language, which is the initials of the tree author names, to enhance technical documents with structural tags. Later it became the Standard Generalized Markup Language (SGML) and was adopted by the ISO in 1986. Since SGML is quite complex and thus is difficult to use, in the late 1990 a group of people including Jon Boask, Tim Bray, James Clark and others suggested XML, the eXtensible Markup Language, as a simplified subset of SGML. On 10 February 1998, XML became a W3C recommendation. Since then, W3C set up many important standards around XML such as XML Schema [W3C Schema1 2004] [W3C Schema2 2004], XSLT [W3C XSLT1.0 1999], XPath [W3C

XPath1.0 1999] [W3C XPath2.0 2003], XQuery [W3C XQuery1.0 2004], XPointer [W3C XPointer1.0 2001], etc.

In only several years, XML became widespread accepted: a large number of software vendors have adopted the standard, a large number of information systems use XML to store data and a large number of applications process XML data. Therefore, it is strongly believed that XML will be the most common technology for all data manipulation and data transmission tasks.

The success of XML is thanks to its advantages. XML has a large number of advantages, including

- XML can represent almost any kind of information
- XML is machine-processable and human-readable
- The XML syntax is very simple and strict, and thus is very easy to learn and to use
- XML is free and extensible: it allows users to define their own tags and document structures
- XML with a schema, e.g. a DTD or an XML Schema definition, is designed to be self-descriptive

2.1.2 XML documents

An instance of the XML language is called an XML *document*. The logical component of an XML document is an *element*. An element can contain elements and *character data*, which are the content of the element. The *character data* of the element can be regarded as the value of the element. Elements that do not have content are called *empty elements*. An element can also bear extra information attached to it called *attributes*, which describe properties of the element. Attributes are made up of a name and a value. There is a distinguished element called *document element*, which contains all other elements in a document.

XML uses *markup* to represent the logical structures of a document. An XML document consists exclusively of *markup* and character data. Markup has the form of *start-tags*, *end-tags* and *empty-element tags*. XML start-tags are made up of the less-than (<) symbol, the name of the element, and a greater-than (>) symbol, e.g. <date>. The attributes of an element are included

in the start-tag of the element, e.g. <data format='DD-MM-YYYY'>, where format is an attribute name and DD-MM-YYYY is its value. XML end-tags consist of the string "</", the same name as in the start-tag, and a greater-than (>) symbol, e.g. </date>.

The empty-element tag is used to represent elements without contents, and consists of the symbol '<', the name of the element, and the string "/>", e.g. <apple/>. An empty element with attributes is represented as e.g. <apple class='1'/>. Elements with content are represented by a start-tag, the content, and an end-tag. For example, <date format='DD-MM-YYYY'> 01-09-2007 </date> and <thesis type='doctorate'> <title> Speeding Up XML Querying </title> </thesis>.

Example 2.1: Figure 2.1 presents an example XML document article.xml, which conforms to the XML Schema definition of Figure 2.2.

```
(N1)   <bib>
(N2)       <article id='A'>
(N3)           <title> My second article </title>
(N4)           <year> 2007 </year>
(N5)           <journal> Well-Known Journal (WKJ) </journal>
(N6)           <reference>
(N7)               <article id='A'>
(N8)                   <title> My first article </title>
(N9)                   <year> 2006 </year>
(N10)                  <journal>  Well-Known Journal (WKJ) </journal>
(N11)                  <reference/>
                   </article>
               </reference>
           </article>
       </bib>
```

Figure 2.1: Example XML document article.xml conforming to the XML Schema definition of Figure 2.2

2.2 XML Schema

XML documents are not necessarily accompanied by a schema. However, having a schema, XML documents become self-descriptive. A number of XML schema languages are developed such as DTDs [W3C XML1.0 2004], XML Schema [W3C Schema1 2004][W3C Schema2 2004], RELAX NG [OASIS 2001], Schematron [ISO Schematron 2006], XML-Data [W3C XML-data 1998], Examplotron [Vlist 2003] and DSD [Moeller 2002], the first two of which are widely used and supported XML schema languages.

The XML 1.0 specification [W3C XML1.0 2004] includes a schema language for defining XML document structures, called as *Document Type Definitions* (DTDs). While DTDs are widely supported and used, the requirement of more precision in describing the structure and content of documents in order to generate more robust XML applications outgrows the capabilities of DTDs. In order to provide an XML schema language with more capabilities, W3C's XML Schema Working group spent two years developing *XML Schema*, which consists of three parts: XML Schema Part 1: Structure [W3C Schema1 2004], XML Schema Part 2: Datatypes [W3C Schema2 2004], and XML Schema Part 0: Primer [W3C Schema0 2004]. The XML Schema was approved as a W3C Recommendation on 2 May 2001 and a second edition was published on 28 October 2004.

2.2.1 XML Schema definitions

XML Schema (see [W3C Schema1 2004] and [W3C Schema2 2004]) is an XML schema language for defining a class of XML documents, called *instance documents* of the schema. A schema, which is formulated in the XML Schema language, is referred to as an *XML Schema definition* (or *XSchema* or *XSD* as short name), which is itself an XML document. An XSD defines the structure of the instance documents, the vocabulary (e.g. the names of elements and attributes), and the data types and occurrence constraints of elements and attributes.

XSD uses an element element to declare an element; the name of the element is given in the name attribute of element; the type of the element is speci-

fied in the type attribute of element. The attributes minOccurs and maxOccurs specify the occurrence constraints of the element in instance XML documents. For example, the element declaration <element name='title' maxOccur=1> defines an element named title, which can occur at most once. An element is required to appear when the value of minOccurs is 1 or greater. XSD uses attribute elements to declare attributes with the name specified in the name attribute and the type specified in the type attribute of attribute, e.g. <attribute name='number' type='int'/>. Attributes may appear once or not at all. Attributes are declared with a use attribute to indicate that the attribute is required, optional, or even prohibited, e.g. <attribute name='number' use='optional'/>. The fixed attribute is used in both attribute and element declarations to specify the value of an attribute or an element to be a particular value, e.g. <attribute name='number' fixed=10/> and <element name='title' fixed="Speeding Up XML Querying">.

In XSD, the elements that contain sub-elements and/or attributes have complex types; the elements that contain only text data have simple types and attributes always have simple types. SimpleType elements describe the specific types of the elements of simple type. ComplexType elements describe the contents of elements of complex type.

XSD provides a number of built-in simple types and allows defining new simple types. A new simple type is derived from an existing simple type, which is the *base* type of the derived type and which may be either built-in or derived. The new simple type is obtained by restricting the values of its base type. The simpleType element defines and names the new simple type; the restriction element indicates the base type and identifies the facets that restrict the range of values of the base type.

The complexType element declares that the content of an element contains sub-elements but does not contain text data when the mixed attribute is not set to true; complexType declares that the content of an element contains sub-elements and text data when the mixed attribute is set to true. The complexContent element declares the content of an element similar to the complexType element. XSD uses the simpleContent element to declare the content of an element that contains attributes and only text data.

XSD defines several model groups: group, choice, sequence and all groups for grouping elements; attributeGroup defines groups of attributes. The choice group allows only one of its elements to appear in an instance; the sequence group stipulates that its elements must appear in an instance according to the order they occur in the group; the elements in an all group can appear in any

order. The group, choice and sequence model groups can be arbitrarily nested; an attributeGroup can contain other attributeGroup groups. The occurrence constraints also apply to the model groups.

XSD also provides mechanisms to declare a new element and a new group by referencing an existing definition of the element and the existing group, e.g. <element ref='title' minOccurs='1'>, which declares an element title defined in e.g. <element name='title'>.

Definition 2.1 (***XML Schema language supported***): we support a significant subset of the XML Schema language, defined in the following EBNF rules.

XSchema ::= <schema> (simpleTypeD | complexTypeD | groupD | attributeGroupD | elementD | attributeD)∗ </schema>.
simpleTypeD ::= <simpleType (name=*Name*)?> restrictionSimpleTypeD </simpleType>.
restrictionSimpleTypeD ::= <restriction base=*Name*> facet∗ </restriction>.
facet ::= <minExclusive value=*Value* /> | <minInclusive value=*Value* /> | <maxExlusive value=*Value* /> | <maxInclusive value=*Value* /> | <totalDigits value=*Value* /> | <fractionDigits value=*Value* /> | <length value=*Value* /> | <minLength value=*Value* /> | <maxLength value=*Value* /> | <enumeration value=*Value* /> | <whiteSpace value=*Value* /> | <pattern value=*Value* />.
complexTypeD ::= <complexType (mixed=*Boolean*)? (name=*Name*)?> (simpleContentD | complexContentD | ((groupD | allD | choiceD | sequenceD)? (attributeD | attributeGroupD)∗)) </complexType>.
simpleContentD ::= <simpleContent> (restrictionSimpleContent | extensionSimpleContent) </simpleContent>.
complexContentD ::= <complexContent (mixed= *Boolean*)?> (restrictionComplexContent | extensionComplexContent) </complexContent>.
restrictionSimpleContent ::= <restriction base=*Name*> facet∗ (attributeD | attributeGroupD)∗ </restriction>.
extensionSimpleContent ::= <extension base=*Name*> (attributeD | attributeGroupD)∗ </extension>
restrictionComplexContent ::= <restriction base= 'anyType'> (attributeD | attributeGroupD)∗ </restriction>.
extensionComplexContent ::= <extension base=*Name*> ((groupD | allD | choiceD | sequenceD)? (attributeD | attributeGroupD)∗) </restriction>
groupD ::= <group (maxOccurs=(*nonNegativeInteger* | 'unbounded'))? (minOccurs=*nonNegativeInteger*)? (name=*Name* | ref=*Name*)?>

(allD | choiceD | sequenceD)? </group>.
attributeGroupD ::= <attributeGroup (name=*Name* | ref=*Name*)?>
(attributeD | attributeGroupD)∗ </attributeGroup>
allD ::= <all maxOccurs= '1' minOccurs=('0' | '1')> elementD∗ </all>
choiceD ::= <choice (maxOccurs=(*nonNegativeInteger* | 'unbounded'))?
(minOccurs=*nonNegativeInteger*)?> (elementD | groupD | choiceD | sequenceD)∗
</choice>.
sequenceD ::= <sequence (maxOccurs=(*nonNegativeInteger* | 'unbounded'))?
(minOccurs=*nonNegativeInteger*)?> (elementD | groupD | choiceD | sequenceD)∗
</sequence>.
elementD ::= <element (fixed=*string*)? (maxOccurs=(*nonNegativeInteger* | 'unbounded'))?
(minOccurs=*nonNegativeInteger*)? (Name=*Name* | ref=*Name*)?
(type=*Name*)?> (simpleTypeD | complexTypeD)? </element>
attributeD ::= <attribute (fixed=*string*)? (name=*Name* | ref=*Name*)? (type=*Name*)?
(use=('optional' | 'prohibited' | 'required'))?> simpleTypeD? </attribute>

where Name is a string, Boolean is a boolean value, i.e. true or false , nonNegativeInteger is a non negative integer, string is a character string, and Value is a value, e.g. a number or a string.

Example 2.2: Figure 2.2 presents an XML Schema definition bib.xsd, which defines a class of documents describing the information on articles. This schema is designed to contain as much language constructs as possible, and meanwhile to be as simple as possible and thus to be easy to read.

```
(D1) <schema>

(D2)   <complexType name='articleType' mixed='true'>
(D3)     <sequence>
(D4)       <element name='title' maxOccurs=1 type='string'/>
(D5)       <element ref='year' maxOccurs=1/>
(D6)       <choice>
(D7)         <element name='journal' maxOccurs=1 type='string'/>
(D8)         <element name='conference' maxOccurs=1 type='string'/>
           </choice>
(D9)       <element name='reference' minOccurs=0 maxOccurs=1 type='referenceType'/>
         </sequence>
(D10)    <attribute name='id' type='string' use='required'/>
       </complexType>
```

```
(D11)  <complexType name='referenceType'>
(D12)    <sequence>
(D13)      <element='article' minOccurs=0 maxOccurs='unbounded' type='articleType'/>
         </sequence>
       </complexType>

(D14)  <element name='bib'>
(D15)    <complexType>
(D16)      <all>
(D17)        <element name='article' minOccurs=0 maxOccurs='unbounded'
                    type='articleType'/>
(D18)        <element name='name' minOccurs=0 maxOccurs=1 type='string'/>
           </all>
         </complexType>
       </element>

(D19)  <element name='year'>
(D20)    <simpleType>
(D21)      <restriction base='int'>
(D22)        <length value=4/>
(D23)        <minExclusive=1900/>
           </restriction>
         </simpleType>
       </element>

     </schema>
```

Figure 2.2: An XML Schema definition bib.xsd

2.3 XPath language

W3C developed XPath (see [W3C XPath1.0 1999] and [W3C XPath2.0 2003]) as a query language for XML data. XPath models an XML document as a tree of nodes and addresses a node-set of an XML tree. XPath gets its name from its basic feature, the use of path expressions, which provides the capability to navigate through the hierarchical structure of an XML tree. An XPath expres-

sion is evaluated on an XML tree and yields an unordered collection of nodes without duplicates.

2.3.1 XPath data model

XPath models an XML document as a tree. XML document trees can be compared to family trees, so XPath uses the genealogical taxonomy to describe the hierarchical structure of an XML document tree, referring to children, descendants, parents and ancestors. An XML document tree consists of nodes. There are seven types of nodes, where we are primarily interested in the most important node types, which are the root, element, attribute and text node types.

- **root nodes**

An XML document tree only has one root node, which is the top of the hierarchy that represents the XML document. The root node has a sole child, which corresponds to the document element.

- **element nodes**

Element nodes correspond to elements in an XML document. Each element node has an element node as parent, except the document element that has the root node as its parent. The children of an element node are element nodes, comment nodes, processing instruction nodes and text nodes.

- **attribute nodes**

Attribute nodes correspond to the attributes of an element. If an element node bears attribute nodes, the element node is the parent of these attribute nodes, but these attribute nodes are not children of the parent element. Elements never share attribute nodes.

- **text nodes**

Character data is grouped into text nodes. As much character data as possible is grouped into each text node, and thus a text node never has an immediately following or preceding sibling that is a text node. Text nodes do not have any children.

Comment nodes and processing instruction nodes correspond to comments and processing instructions in the XML document. Namespace nodes keep track of the set of namespace prefix/URI pairs.

Element nodes have an ordered list of child nodes. The nodes are ordered according to *document order*, which corresponds to the order in which the first character of the XML representation of each node occurs in the XML document representation.

Example 2.3: Figure 2.3 is a tree representation of the XML document bib.xsd in Figure 2.2. In this figure, we only present the attribute nodes of the element node D2 and the attribute nodes of other element nodes are left out for simplification of representation.

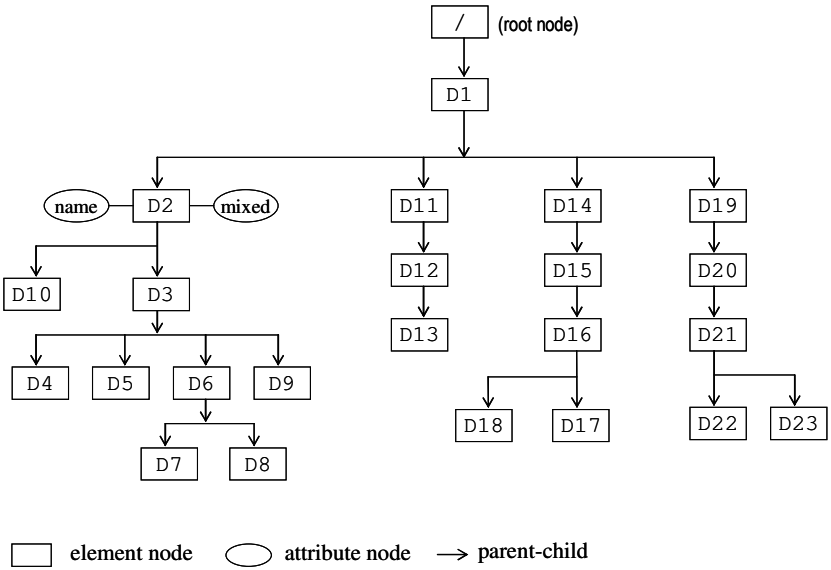

Figure 2.3: The tree representation of the XML document bib.xsd in Figure 2.2, where we only present the attribute nodes of the element node D2.

2.3.2 XPath expressions

An instance of the XPath language is called an XPath *expression*. The syntax of XPath expressions is similar to the familiar path expressions used in URLs and in Unix and Windows Systems to locate directories and files. The semantics of an XPath expression is defined in terms of the semantics of its sub-expressions. The smallest expression is called a *location step* a::n[q_1]...[q_i], which consists of an *axis* a and a *nodetest* n with or without *predicates* q_1, …, q_i. An axis selects a set of nodes by navigating through the tree structure relevant to a *context* node; a node test filters by node types and labels the nodes specified by axis; the nodes selected by a node test are further filtered by the predicates. Table 2.1 presents the axes defined in XPath and demonstrates the nodes selected by these axes. In Table 2.1, the column Example uses bib.xsd in Figure 2.2 as the considered XML document and Figure 2.3 is the tree representation of Figure 2.2; the column C indicates the context node and the column Result indicates that the result node set of applying the considered axis to the context node.
Table 2.2 lists the node tests of XPath.

Table 2.1: Axes of XPath

Axis	Node-set selected	Example	
		C	Result
self	context node	D1	{D1}
child	child nodes	D1	{D2, D11, D14, D19}
descendant	descendant nodes: the transitive closure of the child axis	D19	{D20, D21, D22, D23}
descendant-or-self	context node and descendant nodes	D19	{D19, D20, D21, D22, D23}
parent	parent node	D13	{D12}
ancestor	ancestor nodes: the transitive closure of the parent axis	D13	{D12, D11, D1, /}
ancestor-or-self	context node and ancestor nodes	D13	{D13, D12, D11, D1, /}
following	nodes occurring after the context node in document order, but excluding the descendants	D15	{D19, D20, D21, D22, D23}

	of the context node		
preceding	nodes occurring before the context node in document order, but excluding the ancestors of the context node.	D6	{D5, D4, D10}
following-sibling	children of the context node's parent, occurring after the context node in document order	D5	{D6, D9}
preceding-sibling	children of the context node's parent, occurring before the context node in document order	D6	{D5, D4}
Attribute	the attribute nodes of the context node	D2	{name, mixed}

Table 2.2: Node tests of XPath

Node test	Nodes selected
label	element or attribute nodes with the given label
*	element or attribute nodes
text()	text nodes
node()	any nodes

A predicate is a Boolean expression, which can be an XPath expression or another expression, e.g. a logical expression or a relational expression. The Boolean expressions are applied to each node selected by the node test. If they are evaluated to true, the tested node is selected, otherwise filtered out. When an XPath expression is used as a predicate, it evaluates to true if it selects any nodes at all; it is false if it does not select any nodes.

Location steps in XPath expressions are separated by the token '/', and the nodes selected by a location step are the context nodes of the next location step. If an XPath expression starts with the slash (/), the XPath expression navigates from the root node of the document and is called an *absolute* XPath expression. Otherwise, the XPath expression navigates from a given context node, and is a *relative* XPath expression.

The XPath language also defines several abbreviations, which is more compact and allows XPath expressions to be written and read more easily than the

full syntax. In the abbreviated syntax of the XPath language, the axis child is left out, e.g. child::a = a; the symbol @ represents the axis attribute, e.g. attribute::b = @b; the symbol // represents /descendant-or-self::node()/, e.g. //c=/descendant-or-self::node()/c; the symbol · selects the context node, e.g. a[·/b].

Example 2.4: Figure 2.4 presents an example of an XPath query Q. Q selects the parent nodes reference of the nodes article, which are some descendant nodes of the document node bib. The location step descendant::article has two predicates. The first predicate qualifies that the nodes article must have children year. The second predicate qualifies that the nodes article cannot have children editor, or the nodes article may have children editor, but the nodes article cannot have bib nodes as ancestor nodes.

```
/bib/descendant::article[year][not(self::node()[editor]/ancestor::bib)]/parent::reference
```

Figure 2.4: An XPath Query Q

Definition 2.2 (*XPath subset supported*): The syntax of the supported XPath subset in this work is defined in EBNF as follows:

```
expression  e ::=  e|e | /e | e/e | e[q] | a::n.
predicate   q ::=  e | e o C | e=e | q and q | q or q | not(q) | (q).
axis        a::=  child | attribute | descendant | self | following | preceding |
                  parent | ancestor | DoS | AoS | FS | PS.
nodetest    n ::=  label | ∗ | node() | text().
operator    o ::= = | < | > | ≥ | ≤ | ≠.
```

where label is an element or attribute name and C is a literal, i.e. a string or a number. Furthermore, we write DoS for descendant-or-self, AoS for ancestor-or-self, FS for following-sibling and PS for preceding-sibling. □

2.4 XOBE: An XML-embedding language

There is a number of programming languages, which embed XML documents and XML query languages, such as XOBE [Kempa and Linnemann 2003a][Kempa and Linnemann 2003b], $XOBE_{DBPL}$ [Schuhart et al. 2006][Schuhart 2006], XDuce [Hosoya et al. 2000], HaXML [Wallace und

Runciman 1999] und XMLambda [Shields und Meijer 2001]. In this section, we demonstrate using XOBE how our work can optimize the programming of XML-embedding languages.

XOBE is programming language for XML-based applications, developed by the institute of information system of the University of Lübeck in Germany. XOBE extends the object-oriented programming language Java to process XML fragments, which conforms to a given schema. XML fragments are represented as *XML objects* in XOBE. Every element declaration and every type definition in the schema correspond to an implicit XML object class, and instances of an element declaration or a type definition of the schema are XML objects.

In order to access data and sub-fragments in XML objects in XOBE programs, XOBE incorporates XPath. In XOBE, the context node of an XPath expression is denoted by an XML object variable; the list of nodes returned by an XPath expression is regarded as an XML object. Example 2.5 demonstrates the concept of XML objects and the usage of XPath expressions in an XOBE method update. This example also illustrates how the application of satisfiablity tests of XPath queries can speed up XOBE programming.

Example 2.5: The XOBE method update declares XML objects, e.g. bib and article, which correspond to the declarations of elements bib and article in the XML Schema definition bib.xsd of Figure 2.2. The method updates the element bib by deleting certain selected elements.

```
bib update(bib c) {
   bib.article ba;
   article sc;
   ba = c//article[parent::reference][@id] ;
   for (int i=0; i<ba.getLength(); i++) {
      sc = ba.get(i);
      if sc[year<1900]  remove(sc/year);
   }
   return c;
}
```

The XOBE method update generates an updated bib-object. The XPath expression c//article[parent::reference][@id] in update returns a list of nodes article, which are the descendants of the node bib and fulfills the constraints of two

predicates. The size of the list is determined by the method getLength. The method update removes a child year of an article, if the child year has a value less than 1900. According to the schema in Figure 2.2, the value of the element year must be greater than 1900. Therefore, if we apply the satisfiability test at compile time, we can already know that the result of the method update is the input c. Furthermore, we can rewrite the query c//article[parent::reference][@id] to c/article//reference/article[parent::reference][@id] and then eliminate the reverse axis parent and the redundant part @id (since the attribute id is required) according to the schema bib.xsd in Figure 2.2. Therefore, our approach can optimize the query c//article[parent::reference][@id] to c/article//reference/article to speed up querying XML data, and thus speed up program execution if the update of bib does take place.

Chapter 3 Data Model for XML Schema

Based-on the data model for the XML language given by [Wadler 2000], we develop a data model for XML Schema for mapping the *parent-child*, *preceding-sibling* and *following-sibling* relations in instance XML documents to the corresponding relations in an XML Schema definition in order to identify the navigational paths of XPath queries on an XML Schema definition. Based on the data model, we suggest an approach to evaluating XPath queries on an XML Schema definition in order to check if an XPath query conforms to the constraints in the schema and to integrate the constraints in the schema into XPath queries, which is presented in Chapter 4.

3.1 Motivation

In order to test the satisfiability and containment of XPath queries in the presence of a schema, we check whether or not the XPath queries conform to the constraints imposed in the schema, and integrate the constraints from the schema into XPath queries.

XPath queries select information by navigating through the hierarchical structure of an XML tree, e.g. from a parent element to its child elements. We call the paths identified by XPath queries the navigational paths of XPath queries on instance XML documents. The navigational paths are visited by a common XPath evaluator, when it evaluates XPath queries on instance XML documents in order to retrieve the nodes specified by the XPath queries.

A schema defines a class of XML documents by specifying the hierarchical structure of the XML tree. Therefore, checking if an XPath query conforms to the constraints in a schema is to navigate through the schema tree to find a corresponding part, which describes the hierarchical structure of the XML tree specified by the XPath query. We call the paths identified by the XPath query the navigational paths of the XPath query on the XML Schema definition. The navigational paths are visited by our XPath-XSchema evaluator (see Chapater 4), when it evaluates the XPath query on the XML Schema definition in order to check if the XPath query conforms to the constraints in the schema.

However, in an XML Schema definition, the node D1 that declares the element N1 and the node D2 that declares the element N2 have typically not the same relation as N1 and N2 have in an instance XML document. For example, the node (N6) is the parent of the node (N7) in the instance XML document article.xml in Figure 2.1; in the XML Schema definition bib.xsd in Figure 2.2 the node (D9) that declares (N6) is not a parent of the node (D13) that declares (N7). Therefore, the navigational paths of an XPath query on instance XML documents are different from its navigational paths on an XML Schema definition.

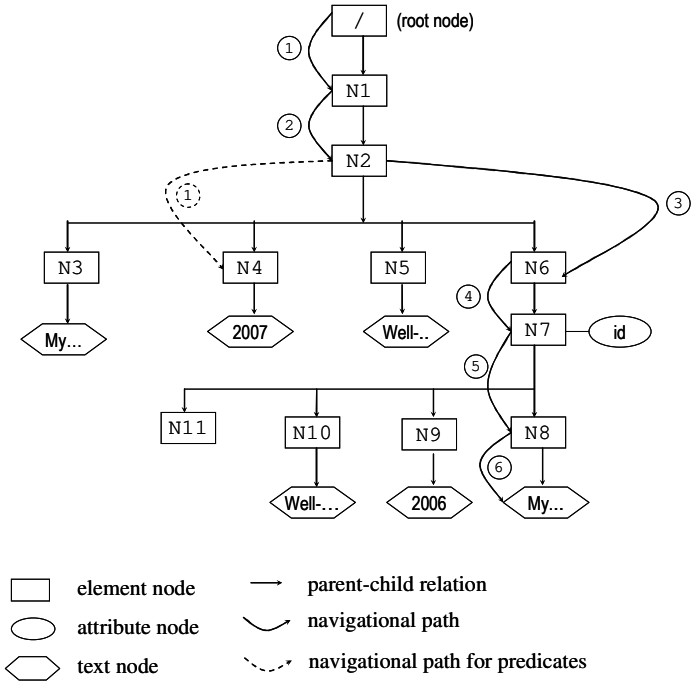

Figure 3.1: The navigational paths of the query /bib/article[year]/reference/article/title/text() on the tree representation of the instance XML document article.xml in Figure 2.1

Figure 3.1 and Figure 3.2 present the navigational paths of the XPath query /bib/article[year]/reference/article/title/text() on the tree representation of the instance document article.xml in Figure 2.1 and the XML Schema definition bib.xsd in Figure 2.2. The two figures show that the navigational paths of XPath queries on an XML Schema definition are more complicated than the navigational paths of XPath queries on the instance XML document. In this

chapter, we develop a data model for the XML Schema language to identify the navigational paths of XPath queries on an XML Schema definition.

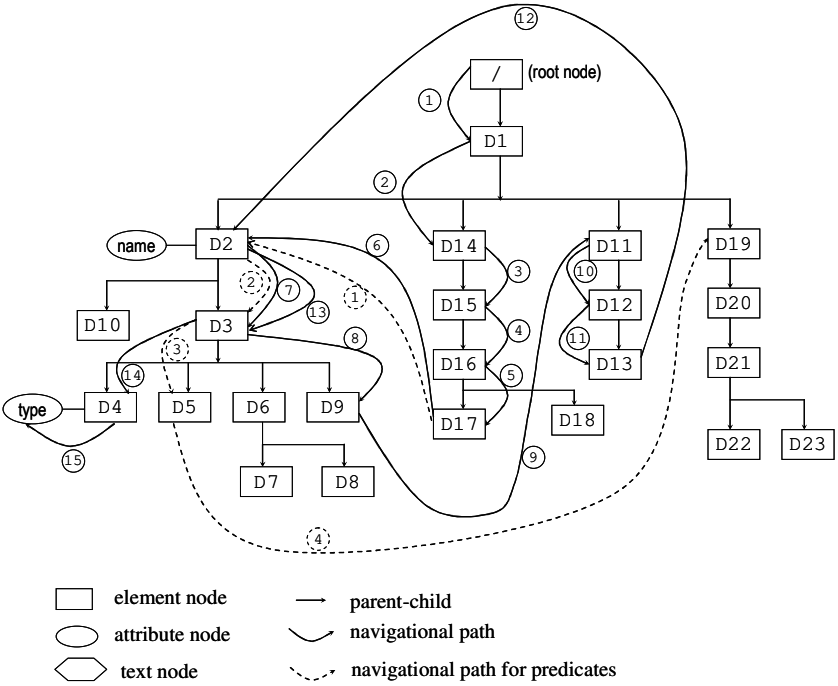

Figure 3.2: The navigational paths of the query /bib/article[year]/reference/article/title/text() on the tree representation of the XML Schema definition bib.xsd in Figure 2.2, where not all the attribute nodes are presented for simplification of presentation

3.2 Notations

The following notations on sets, relationships and sequences are used to model the XML Schema language, and are also used to model the schema path (see Chapter 4). Set(T) (or Sequence(T) respectively) indicates the type of a set (or of a sequence respectively) the entries of which are of type T. We write $\varnothing$ for the empty set, $\in$ for membership and $\cup$ for the union of sets. We express the signature of a function f by $f: T_1 \rightarrow T_2$, where T_1 is the type of the domain and T_2 is

the type of the co-domain. Note that a type T can be a simple type, e.g. an XML node (Node), an XPath expression (XPath) or a node test (NodeTest). Furthermore, T can be a type of a set the entries of which are of a type T_1, i.e. Set(T_1), a type of a sequence the entries of which are of a type T_1, i.e. Sequence(T_1), or the cross-product of two or more types, e.g. $T_1 x T_2$. The transitive closure f^+ and reflexive transitive closure f^* of a function $f: T \rightarrow Set(T)$ are defined as follows:

$f^n(x) = \{ z \mid y \in f^{n-1}(x) \wedge z \in f(y) \}$, where $f^0(x) = \{x\}$ and $f^1(x) = f(x)$ and $n \geq 1$
$f^+(x) = \cup_{n=1}^{\infty} f^n(x)$
$f^*(x) = \cup_{n=0}^{\infty} f^n(x)$

We write $(x_1, ..., x_m)$ (where $m \geq 1$) for a sequence of entries $x_1, ..., x_m$. We use the operator + to concatenate two sequences, e.g. $(x_1, ..., x_m) + (y_1, ..., y_n) = (x_1, ..., x_m, y_1, ..., y_n)$. Let s be a sequence, then we write |s| for the length of s, i.e. the number of entries in s, and write s[k] for the *k*-th entry of the sequence s (where $k \in \{1,..., |s|\}$). Thus, s[1] (if s is not empty) indicates the first entry of s and s[|s|] indicates the last entry of s, s[|s|-1] (where $|s| \geq 2$) indicates the pre-last entry of s, and so on. We use the following notations to describe an attribute node of an element node, e.g. we write D4@<type> or D4@<type='string'> for the attribute node type='string' of the node D4=<element name='title' maxOccurs='1' type='string'/> in Figure 2.2. Furthermore, we also call a node in an XML Schema definition an XSchema node.

3.3 Concepts

An XML Schema definition is a set of nodes of type Node. There are four specific Node types iElement, iAttribute, iText and iRoot in an XML Schema definition, which are associated with *instance element*, *instance attribute*, *instance text* and *instance root* nodes of the XML Schema definition. An instance element node declares an element of XML documents; an instance attribute node declares an attribute; an instance text node indicates a text node; an instance root node indicates the root of an XML document. Accordingly, we define four functions with signature Node→Boolean to test the type of a node: isiElement, isiAttribute, isiText and isiRoot, which return true if the type of the given node is of type iElement, iAttribute, iText or iRoot respectively, otherwise false.

Definition 3.1 (*instance nodes*): The *instance nodes* of an XML Schema definition are

- <schema...> (which is the *instance root* node of type iRoot)
- <element name=N...> (which is an *instance element* node of type iElement),
- <attribute name=N...> (which is an *instance attribute* node of type iAttribute),
- attribute node type=T of nodes <element type=T...>, which we denote as @<type=T> (which is an *instance text* node of type iText, if T is a built-in simple type),
- <simpleType...> (which is an *instance text* node of type iText),
- <complexType mixed='true'...> (which is an *instance text* node of type iText),
- <simpleContent...> (which is an *instance text* node of type iText), and
- <complexContent mixed='true'...> (which is an *instance text* node of type iText).

Definition 3.2 (*instance child nodes*): Let N1 and N2 be two XSchema nodes of type *iElement*. If the element defined in N2 can appear in an instance XML document as a child of the element defined in N1, then N2 is an instance child node of N1.

Taking the XML Schema definition bib.xsd in Figure 2.2 as an example, D17 is an instance child node of D14; D19 is an instance child of D17 and D13; D8 is an instance child of D13 and D17; D13 is an instance child node of D9.

Definition 3.3 (*instance text nodes*): Let N1 be an XSchema node of type *iElement*, and N2 be an XSchema node of type *iText*. If N2 is an attribute node of N1 or a node that is used to define the type of the element declared in N1, then N2 is an instance text node of N1.

In Figure 2.2, the attribute node @<type='string'> of D4 is an instance text node of D4; D2 is an instance text node of D17 and D13; D20 is an instance text node of D19.

Definition 3.4 (*instance attribute nodes*): Let N1 be an XSchema node of type *iElement*, and N2 be an XSchema node of type *iAttribute*. If the attribute defined in N2 can appear in an instance XML document as an attribute of the element defined in N1, then N2 is an instance attribute node of N1.

In Figure 2.2, D10 is an instance attribute node of D17 and D13.

Definition 3.5 (*instance parent nodes*): Let N1 be an XSchema node of type *iElement*, and N2 be either an instance child node or an instance text node or an instance attribute node of N1, then N1 is the instance parent node of N2.

Definition 3.6 (*instance sibling, instance preceding sibling and instance following sibling nodes*) : Let N1 be an XSchema node of type *iElement* or *iText*, and N2 be an XSchema node of type *iElement* or *iText*. If the element that is

defined in N1 or the text whose data type is defined in N1 can appear in valid XML documents as a preceding sibling or a following sibling respectively of the element that is defined in N2 or the text whose data-type is defined in N2, then N1 is an instance preceding sibling node or an instance following sibling node respectively of N2. If N1 is an instance preceding sibling node or an instance following sibling node of N2, then N1 is an instance sibling of N2.

In Figure 2.2, D2 is an instance preceding sibling node of D4; D8 is an instance following sibling node of D4; D19 is an instance proceeding sibling node of D7, D8 and D9.

Definition 3.7 (*succeeding nodes*): A node N2 in an XML Schema definition is a *succeeding node* of a node N1 in the XML Schema definition if

- N2 is a child node of N1, or
- N1=<element type=N...> and N2=<simpleType name=N...> with the same N, or
- N1=<attribute type=N...> and N2=<simpleType name=N...> with the same N, or
- N1=<element type=N...> and N2=<complexType name=N...> with the same N, or
- N1=<element ref=N...> and N2=<element name=N...> with the same N, or
- N1=<attribute ref=N...> and N2=<attribute name=N...> with the same N, or
- N1=<group ref=N...> and N2=<group name=N...> with the same N, or
- N1=<attributeGroup ref=N> and N2=<attributeGroup name=N> with the same N, or
- N1=<restriction base=N> and N2=<simpleType name=N...> with the same N, or
- N1=<extension base=N> and N2=<simpleType name=N...> with the same N, or
- N1=<extension base=N> and N2=<complexType name=N...> with the same N.

Definition 3.8 (*preceding nodes*): Node N1 in an XML Schema definition is a *preceding node* of a node N2 in the XML Schema definition if N2 is a *succeeding node* of N1.

Figure 3.3 presents the succeeding nodes for each node on the tree presentation of the XML Schema definition bib.xsd in Figure 2.2, where the attribute nodes are left out.

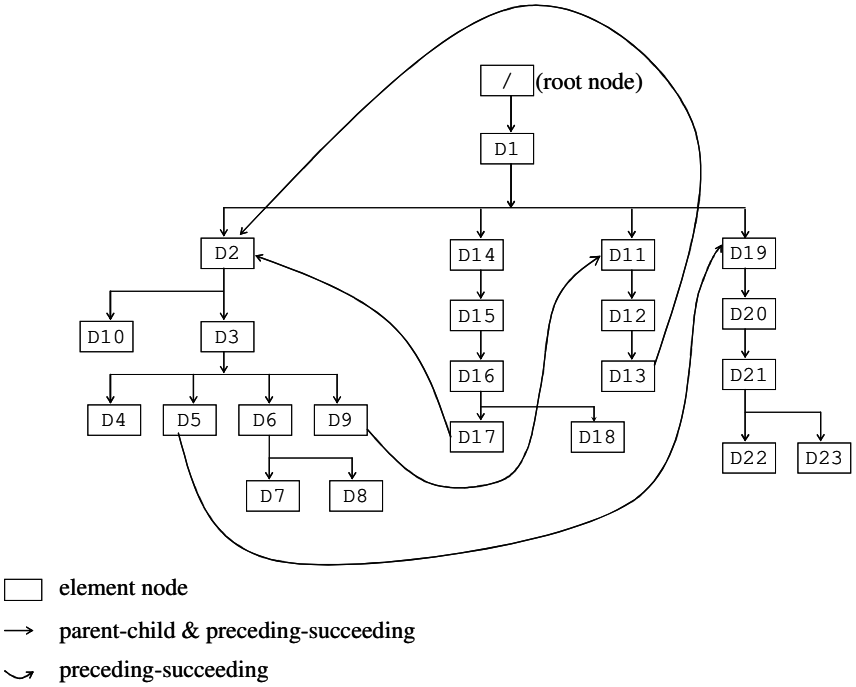

Figure 3.3: The succeeding nodes of each node in the tree of the XML Schema definition bib.xsd in Figure 2.2, where attribute nodes are left out

3.4 Functions

Figure 3.4 defines the data model of the XML Schema language, which consists of a group of functions. These functions relate an XSchema node to a set of XSchema nodes or to a set of sequences of XSchema nodes, or relate a sequence of XSchema nodes to a set of sequences of XSchema nodes, represented in comprehension notation (see [Wadler 1999]).

- child(N) = {N1 | N1 is a child node of N}
- succeeding(N) = {N1 | N1 is a *succeeding* node of N}
- iChild-helper(N) = $\cup_{i=0}^{\infty} S_i$, where S_0 = {(N)},

S_i = { y+(N1) | y∈ S_{i-1} ∧ N1∈ succeeding(y[|y|]) ∧
¬isiElement(N1) ∧ ¬isiAttribute(N1)}

- iChild(N)={ y+(N1) | (y=(N) ∧ isiRoot(N) ∧ N1∈ child(N) ∧ isiElement(N1)) ∨
(y∈ iChild-helper(N) ∧ N1∈ succeeding(y[|y|]) ∧ isiElement(N1)) }

- iAttributeChild(N) = { y+(N1) | y∈ iChild-helper(N) ∧
N1∈ succeeding(y[|y|]) ∧ isiAttribute(N1) }

- iText-helper(N) = $\cup_{i=0}^{\infty}$ R_i, where R_0 = {(N)},
R_i = { y+(N1) | y∈ R_{i-1} ∧ N1∈ succeeding(N') ∧ ¬isiText(N') ∧
¬isiAttribute(N') ∧ N'= y[|y|] ∧ N'≠<complexType> ∧ (
N'≠<element type=T> ∨
(N'=<element type=T> ∧ ¬built-in(T))) }

- iTextChild(N) = { y | (y∈ iText-helper(N) ∧ isiText(y[|y|])) ∨ (
(y=z+(N1) ∧ z∈ iText-helper(N) ∧ isiText(N1) ∧ N'= z[|z|] ∧ ¬isiText(N') ∧
((N'=<element type=T…> ∧ N1=N'@<type>) ∨
(N'=<complexType> ∧ N1∈ succeding(N')))) }

- iPS(x) = { y | (y∈ iChild(x[1]) ∨ y∈ iTextChild(x[1])) ∧
y[|y|]≠@<type=T> ∧ y[y]≠<simpleType> ∧ y[y]≠ <simpleContent> ∧ (
(y[|y|]=<complexType mixed='true'> ∨
y[|y|]=<complexContent mixed='true'> ∨
x[|x|]=<complexType mixed='true'> ∨
x[|x|]=<complexContent mixed='true'>)
∨
(x=y ∧ ∃i∈ {2, 3, ..., |x|}: attribute(x[i], 'maxOccurs')>1)
∨
(∀i∈ {1, …, k}: x[i]=y[i] ∧ x[k+1]≠y[k+1] ∧ k<min(|x|, |y|) ∧ (
x[k]=<all>
∨
∃i∈ {2, 3, ..., k}: attribute(x[i], 'maxOccurs')>1
∨
(y[k+1]<<x[k+1] ∧ ∀i∈ {2, 3, ..., k}: (
x[i]=<sequence maxOccurs=1> ∨
x[i]=<choice maxOccurs=1> ∨
x[i]=<group maxOccurs=1> ∨
(x[i]≠<sequence> ∧ x[i]≠<choice> ∧ x[i]≠<group> ∧ x[i]≠<all>)) ∧
x[k]≠<choice>)))}

- iFS(x) = { y | (y∈ iChild(x[1]) ∨ y∈ iTextChild(x[1])) ∧

y[|y|]≠@<type=T> ∧ y[y]≠<simpleType> ∧ y[y]≠ <simpleContent> ∧ (
(y[|y|]=<complexType mixed='true'> ∨
y[|y|]=<complexContent mixed='true'> ∨
x[|x|]=<complexType mixed='true'> ∨
x[|x|]=<complexContent mixed='true'>)
∨
(x=y ∧ ∃i∈{2, 3, ..., |x|}: attribute(x[i], 'maxOccurs')>1)
∨
(∀i∈{1, ..., k}: x[i]=y[i] ∧ x[k+1]≠y[k+1] ∧ k<min(|x|, |y|) ∧ (
x[k]=<all>
∨
∃i∈{2, 3, ..., k}: attribute(x[i], 'maxOccurs')>1
∨
(x[k+1]<<y[k+1] ∧ ∀i∈{2, 3, ..., k}: (
x[i]=<sequence maxOccurs=1> ∨
x[i]=<choice maxOccurs=1> ∨
x[i]=<group maxOccurs=1> ∨
(x[i]≠<sequence> ∧ x[i]≠<choice> ∧
x[i]≠<group> ∧ x[i]≠<all>)) ∧
x[k]≠<choice>)))))}

Figure 3.4: A data model of XML Schema for identifying the navigational paths of XPath queries on XML Schema definitions

The function child: Node → Set(Node) relates an XSchema node to all its child nodes; the function succeeding: Node → Set(Node) relates an XSchema node to all its *succeeding* nodes; the function preceding: Node → Set(Node) relates an XSchema node to all its *preceding* nodes. Let us take the XML Schema definition bib.xsd in Figure 2.2 as an example. child(D1)={D2, D11, D14, D19}; child(D18)=∅. succeeding(D2)={D3, D10}; succeeding(D5)={D19}; succeeding(D17)={D2}. preceding(D18)={D16}; preceding(D2)={D1, D17, D13}; preceding(D11)={D1, D9}.

iChild: Node → Set(Sequence(Node)), which is defined to find the instance child nodes with type *iElement* of an XScheme node N, relates the XSchema node N to a set of XSchema node sequences, i.e. if y∈iChild(N), then y[1]=N and y[|y|] is an instance child node of N. Other nodes in y are the intermediate nodes visited when searching for y[|y|] of y[1], i.e. ones that belong to both succeeding$^+$(y[1]) and preceding$^+$(y[|y|]). Some of them may be the declaration nodes of model groups, which control the occurrence number of y[|y|], and the occurrence order of y[|y|] with respect to its instance sibling nodes in an instance

XML document. In Figure 2.2, iChild(D14)={(D14, D15, D16, D17), (D14, D15, D16, D18)}; iChild(D9)={(D9, D11, D12, D13)}; iChild(D7)=∅.

iAttributeChild: Node → Set(Sequence(Node)), which is defined to find the instance attribute nodes of an XSchema node N, relates the node N to a set of node sequences, i.e. if y∈ iAttribueChild(N), then y[1]=N and y[|y|] is an instance attribute node of N. Other nodes in y are the intermediate nodes visited when searching for y[|y|] of y[1], i.e. ones that belong to both succeeding$^+$(y[1]) and preceding$^+$(y[|y|]). In Figure 2.2, iAttributeChild(D13)={(D13, D2, D10)}; iAttributeChild(D17)={(D17, D2, D10)}.

The auxiliary function iChild-helper: Node → Set(Sequence(Node)) helps iChild(N) and iAttributeChild(N) to find the corresponding nodes, and returns all the node sequences visited before the instance child nodes and instance attribute nodes of the XSchema node N. In Figure 2.2, iChild-helper(D17)={(D17), (D17, D2), (D17, D2, D3), (D17, D2, D3, D5), (D17, D2, D3, D6)}; iChild-helper(D9)={(D9), (D9, D11), (D9, D11, D12)}; iChild-helper(D19)={(D19), (D19, D20), (D19, D20, D21), (D19, D20, D21, 22), (D19, D20, D21, D23)}.

iTextChild: Node → Set(Sequence(Node)) is defined to find the instance text nodes of an XSchema node N, and relates the node N to a set of node sequences. Let y∈ iTextChild(N), then y[1]=N and y[|y|] is an instance text node of N. The nodes between y[1] and y[|y|] are the intermediate nodes visited when searching for y[|y|] of y[1], i.e. the nodes that belong to both succeeding$^+$(y[1]) and preceding$^+$(y[|y|]). In Figure 2.2, iTextChild(D13)={(D13, D2)}; iTextChild(D17)={(D17, D2)}; iTextChild(D4)={(D4, D4@<type='string'>)}; iTextChild(D19)={(D19, D20)}; iTextChild(D9)=∅.

The XML data model defines that an element of simple type must have and only has a text node, and that an element of complex type can either have one or more text nodes or have no text node at all. XML Schema specifies whether or not an element of complex type has text nodes, but does not specify the number of the text nodes. Therefore, we only need to take care whether or not an XSchema node has instance text nodes, and we only need to find one instance text node but not all the instance text nodes of an XSchema node. We achieve these goals by using the auxiliary function iText-helper: Node → Set(Sequence(Node)).

If N of iText-helper(N) declares an element of simple type, then N must have instance text nodes, which are either the attribute node type=T of N if T is a built-in simple type, or the nodes <simpleType...> in succedding$^+$(N). In Figure 2.2, the attribute node type='string' of D4 is the instance text node of D4; D20 is an instance text node of D19.

If N declares an element e of complex type, then there must exist a node of a complex type declaration, i.e. D=<complexType...>, which is used to define the type of the element e, i.e. D is a node in succeeding$^+$(N). If D contains the construct mixed='true', then D is an instance text node of N. If D is not an instance text node of N, but D has a child node of <simpleContent...> or <complexContent mixed='true'...>, then the child node of D is the instance text node of N. If D does not have such a child, then N does not have instance text nodes. In Figure 2.2, each of D14 and D17 defines an element of complex type. D14 does not have instance text nodes; D17 has an instance text node D2.

Let y∈iText-helper(N), then y[1]=N, and y[|y|] is either an instance text node or a node visited before the instance text node, or the node <complexType...> or a node visited before <complexType...>, or a node visited before an instance attribute node. The auxiliary function built-in(T) in iText-helper(N) tests whether or not the type T is a built-in simple type. In Figure 2.2, iText-helper(D4)={(D4)}; iText-helper(D13)={(D13), (D13, D2)}; iText-Helper(D17)={(D17), (D17, D2)}; iText-helper(19)={(D19), (D19, D20)}; iText-helper(D14)={(D14), (D14, D15)}.

Different from the XML data model, where a node has only a parent node, in XML Schema definitions, a node may have several instance parent nodes, e.g. in Figure 2.2, D4 has two instance parent nodes D13 and D17. Thus, the function iPS: Sequence(Node) → Set(Sequence(Node)) for finding the instance preceding sibling nodes and the function iFS: Sequence(Node) → Set(Sequence(Node)) for finding the instance following sibling nodes relate a sequence x of nodes to a set of sequences of nodes. The first node in x is the instance parent node of the last node of x. Let y be a node sequence in iPS(x), then y[1]=x[1], and y[|y|] is both an instance child node or an instance text node of y[1] and an instance preceding sibling node of x[|x|]. In Figure 2.2, iPS(D14, D15, D16, D18)={(D14, D15, D16, D17)}; iPS(D17, D2, D3, D5, D19)={(D17, D2), (D17, D2, D3, D4)}; iFS(D13, D2, D3, D6, D7)={(D13, D2), (D13, D2, D3, D9)}; iFS(D13, D2, D3, D5, D19)={(D13, D2), (D13, D2, D3, D6, D7), (D13, D2, D3, D6, D8), (D13, D2, D3, D9)}.

In order to help understand the functions iPS(x) and iFS(x), we first look at the relation between an instance text node and its instance sibling nodes. Since XML Schema does not specify the position of the instance text nodes of a node N that defines an element e of complex type, we assume that a text child of the element e may appear before or after other children of the element e in any instance XML document. If y[|y|]=<complexType mixed='true'...> or y[|y|]=<complexContent mixed='true'...>, then y[|y|] is an instance text node of y[1] that defines an element of complex type. Thus, the instance text node is an instance following-sibling and an instance preceding-sibling of its instance siblings and itself.

However, if an element e of complex type has attributes and the text child but has no element children, then the text child is the only child of e. If y[|y|]=<simpleContent...>, then y[|y|] is an instance text node of y[1] that defines such elements as e. Therefore, the instance text node <simpleContent...> does not have any instance sibling. Similarly, the text child of an element of simple type is the only child of the element, so the instance text node of a node that defines an element of simple type has no instance sibling nodes. If y[|y|]=@<type=T> or y[y]=<simpleType...>, then y[|y|] is an instance text node of y[1] that defines an element of simple type, and thus y[|y|] has no instance preceding and following sibling nodes.

Furthermore, XML Schema specifies the following rules on the occurrence order of sibling elements in instance XML documents:

- The elements defined in an *all* model group can occur in any order in an instance XML document
- The elements defined in a *sequence* model group can occur in the order in an instance XML document as the order they are defined in the XML Schema definition
- The elements defined in a *choice* model group cannot occur simultaneously in an instance XML document
- However, if a *sequence* group or an *choice* group can occur more than one time, then the elements in these groups can occur in any order in an instance XML document

According to the XML Schema language, we identify that a node N2=y[|y|] is an instance preceding sibling node of the instance node N1=x[|x|], i.e. y is a node sequence in iPS(x), if one of the following conditions holds:

- N2 is an instance text node of x[1], and x[1] defines an element that contains also sub-elements.
- N2 is an instance child node of x[1] if N1 and N2 are contained in an *all* model group (e.g. D18 is an instance preceding sibling of D17), or
- N2 is an instance child node of x[1] if there is at least a model group, which either directly or recursively contains both N1 and N2, is declared with maxOccurs>1, or
- N2 is an instance child node of x[1], and N2 is visited before N1 in the XML Schema definition, if all the model groups, which either directly or recursively contain both x and y, consist of only *sequence* and *choice* groups, which are declared with maxOccurs=1 explicitly or implicitly (e.g. D4 is an instance preceding sibling of D9 since D4 is visited before

D9, but D9 is not an instance preceding sibling of D4). However, N2 is not an instance sibling node of N1, if N1 and N2 are contained in a common *choice* group, and either N1 or N2 must be directly contained in the *choice* group (e.g. D7 is not an instance sibling of D8).

We now give some comments on the details in the functions iPS(x) and iFS(x). x[|x|] and y[|y|] have some common ancestor nodes, some of which may be the model groups that either directly or recursively contain x[|x|] and y[|y|], e.g. D13, D2 and D3 are the common ancestor nodes of D4 and D19. The common ancestor nodes are the nodes from x[1] to x[k] if $\forall i \in \{1, ..., k\}$: $x[i]=y[i] \wedge x[k+1] \neq y[k+1] \wedge k<\min(|x|, |y|)$, where the function min(|x|, |y|) returns the minimum of |x| and |y|. Among these common ancestor nodes, x[1] is the instance parent node of x[|x|] and y[|y|], and thus the possible model group nodes in these common ancestor nodes are the nodes from x[2] to x[k]. If x[|x|]=y[|y|] , then x=y. In this case, whether or not x[|x|] is a sibling node of itself relies on the occurrence constraints of x[|x|]. If x[|x|] can occur more than one time, i.e. $\exists i \in \{2, 3, ..., |x|\}$: attribute(x[i], 'maxOccurs')>1, then x[|x|] is either a preceding sibling node or a following sibling node of itself (e.g. D13 is both a preceding sibling and following sibling of itself).

XML Schema stipulates that an *all* group must appear as the sole child at the top of a content model (e.g. D16), and the content model of an *all* group consists of element declarations, i.e. <all...> elementD* </all> (e.g. D17 and D18). Therefore, if x[k]=<all>, then x[|x|] and y[|y|] are contained in an *all* group, and thus the element declared in x[|x|] may appear before or after the element declared in y[|y|] in any valid XML document (e.g. D17 is both a preceding sibling and following sibling of D18). If there is at least one node in (x[2], ..., x[k]) defined with maxOccurs>1, i.e. $\exists i \in \{2, 3, ..., k\}$: attribute(x[i], 'maxOccurs')>1, then the element declared in x[|x|] may appear before or after the element declared in y[|y|] in any valid XML document. If (x[2], ..., x[k]) does not contain an *all* group and each model group in the sequence is defined with maxOccurs=1, then the element declared in x[|x|] and the element declared in y[|y|] appear in an XML instance document in the same order as the visited order of the node x[|x|] and the node y[|y|]. The visited order is defined by the order in which y[k+1] and x[k+1] appear in the XML Schema definition (e.g. y[|y|]=D19 is an preceding sibling of x[|x|]=D7 since y[k+1]=D5 occurs before x[k+1]=D6). Let N1 and N2 be two nodes in an XML Schema definition, then N1<<N2 indicates that N1 appears before N2 in the XML Schema definition. However, if x[k] is the node <choice>, then the element defined in x[|x|] and the element defined in y[|y|] cannot appear simultaneously in any XML instance document (e.g. D7 and D8 are not instance siblings). Therefore, y[|y|] is not an instance sibling node of x[|x|], and

thus y is not a node sequence of iPS(x). The auxiliary function attribute(N, attributeName) returns the value of the attribute attributeName in node N, e.g. attribute(N, 'maxOccurs') retrieves the value of the attribute with the name maxOccurs in node N.

Figure 3.5 defines the function NT: Node × NodeTest → Boolean, which checks whether or not an instance XSchema node N conforms to a node test of XPath to filter the schema node according to the node test of XPath.

- NT(N, ∗) = isiElement(N) ∨ isiAttribute(N)
- NT(N, label) = (isiElement(N) ∧ attribute(N, 'name')=label) ∨
  (isiAttribute(N) ∧ attribute(N, 'name')=label)
- NT(N, text()) = isiText(N)
- NT(N, node()) = true

Figure 3.5: Functions that test an instance XSchema node against a node test of XPath

Chapter 4 XPath-XSchema Evaluator

A common XPath evaluator is typically constructed to evaluate XPath queries on instance XML documents. Our approach evaluates XPath queries on an XML Schema definition based on the XML Schema data model developed in Chapter 3 in order to

- check whether or not XPath queries conform the constraints given in the schema for testing the XPath satisfiability with respect to schemas, and
- integrate the constraints in the schema into XPath queries for checking the XPath containment and rewriting XPath queries under schemas, and for further testing the XPath satisifiability.

Therefore, we name our XPath evaluator *XPath-XSchema* evaluator.

A schema S defines a class of XML documents by specifying constraints, e.g. the relation between elements-elements and between elements-attributes. An XPath query Q describes a part of an XML document by specifying some of the constraints. If a constraint described by Q can not be mapped to a corresponding one in S, then XML documents described by Q do not exist, and thus Q is unsatisfiable. Therefore, evaluating Q on S is to check whether or not there is a corresponding constraint in S for each constraint in Q. For example, Q=/a/b[c] describe three constraints: (1) the element a is a child of the root node, (2) the element b is a child of a, and (3) b should have a child c. For (1), we search for the nodes N1 in S, which declare an element a, and also declare that the element a is a child of the root node in an XML document. If N1 is found, we proceed to the constraint (2) and search for the nodes N2 in S, which declare an element b, and also declare that the element b occurs as a child of the element a declared by N1. After N2 is found, we check the constraint (3) in a similar way. If a constraint is not resolved, the evaluation aborts and Q is unsatisfiable with respect to S.

4.1 Schema paths

Instead of computing the node set of XML documents specified by an XPath query, our XPath-XSchema evaluator computes a set of schema paths to the possible resultant nodes, when the XPath query is evaluated by a common XPath evaluator on instance XML documents. If an XPath query cannot be evaluated completely, the schema paths for the XPath query are computed to the empty set of schema paths.

4.1.1 Definition

The schema paths are actually a log of the process of searching for the relevant nodes described by the XPath query from a given XML Schema definition, i.e. the navigational paths of the XPath query on the XML Schema definition. Therefore, the major construct of the schema paths is the navigational paths.

In order to better understand the definition of schema paths in Definition 4.1, we first outline how the XSchema-XPath evaluator searches for relevant nodes in an XML Schema definition based on the XML Schema data model to construct the schema paths. Similar to a common XPath evaluator, our approach starts the search at the root node of the XML Schema definition. The search continues from an XML Schema node N1 typically to a *succeeding* node N2 of the node N1 in the case of a forward axis, or to a *preceding* node N2 of the node N1 in the case of a reverse axis. The search passes the nodes in the XML Schema definition, which are not *instance nodes*. The search continues until an *instance* node specified by the current location step is retrieved, which is a relevant schema node logged in the schema paths.

A schema is a recursive schema, if this schema defines one element as its own child or descendant. For example, in the XML Schema definition bib.xsd in Figure 2.2, the element article defined in the node D13 has one child element reference defined in the node D9, which has one child element article defined in D13. Therefore, the element article defined in D13 is one descendant of its own.

In the presence of recursive schemas, it may occur that the XSchema-XPath evaluator revisits a node of the schema without any progress in the processing of the query. We call this a *loop*. For the purpose of detecting a loop, the schema path needs to log the information of the part of the XPath ex-

pression, which has been processed. The schema paths of the XPath expressions in a predicate, which are computed in the same way, are attached to the context node of the predicates. We also need a parameter in the schema path to indicate the relation between expressions in a predicate. In an XML Schema definition, an instance node might have several instance parent nodes in that multiple elements might contain some identical sub-elements and each element is declared only one time. Since we cannot retrieve the parent nodes unambiguously from only the XML Schema definition, we need to log the information of the parent nodes in the schema path.

Definition 4.1 (*Schema paths*): A schema path the type of which we denote by schema_path is a sequence of pointers to either the schema path records <XP, S, a, z, lp, f>, or the schema path records <o, f>, or schema path records <e> where

- XP is an XPath expression,
- S is a set of sequences of XSchema nodes,
- a is a label and a ∈ {child, parent, FS, PS, self, attribute},
- z is a set of pointers to schema path records,
- lp is a set of schema paths,
- f is a set of sets of schema paths,
- e is a predicate expression self::node() op C, where C is a literal, i.e. a number or a string, and op is a operator and op∈{=, <, > , ≥ , ≤ , ≠}, and
- o is a keyword and o ∈ {=, or, and, not}.

Let Q be an XPath query, which is the input of our XPath-XSchema evaluator, and $Q=XP_e/XP_c/XP_r$, where XP_e is the part, which has been evaluated; XP_c is the part, which is being evaluated; XP_r is the part, which has not been evaluated so far by the XPath-XSchema evaluator. In a schema path record, XP is dependent on XP_e. XP is needed for the detection of loop schema paths. S is a set of sequences of XSchema nodes and is computed according to the data model of XML Schema described in Chapter 3. The last node N_l in each sequence s of S is an instance node, which is visited by the XPath-XSchema evaluator when evaluating XP_c, and which is also a context node to compute the following nodes. The first node N_f of s is an instance parent node of N_l, and other nodes in s are ones that are visited when searching for N_l of N_f, some of which may be the nodes of model groups and are useful for consistency checking of occurrence constraints. a is a label associated with the schema node N_l, indicating an

XPath axis, i.e. child, parent, FS, PS, self or attribute, from which the node N_l is generated. a is needed for rewriting.

The field z in a schema record R is a set of pointers to the schema path records in which the last schema node of the node sequences is the instance parent node of the last schema node of the node sequences of the record R. Note whenever an instance XSchema node is the first node of a loop, the node has more than one possible instance parent node, and thus there are several sequences of nodes and pointers in a schema path record. lp represents loop schema paths; f represents the sets of schema paths computed from one or more predicates that test the last node of S, which is the context node of the predicates. The schema paths can consist of predicate expressions, e.g. {(<self::node()=100>)}. o represents operators like =, or, and and not to indicate the operation on the schema paths of predicates.

A point that should be emphasized is that a schema path consists of a sequence of pointers to the records rather than a sequence of the records, and this is important for the form of loop schema paths. As we will describe later on, often more than one schema path will be computed from an XPath query. When a schema path is computed, a loop that is involved in the path might not be detected until other schema paths are computed. Using pointers, we only need to modify the records to include the loop paths without any modification of the schema paths.

4.1.2 Example

Example 4.1: Our XPath-XSchema evaluator evaluates the XPath query Q in Figure 2.4 in Chapter 2, i.e. Q = /bib/descendant::article[year][not(self::node()[editor]/ancestor::bib)]/parent::reference (which is also repeated in Figure 4.1 in this chapter for the convenience of readers) on the XML Schema definition bib.xsd of Figure 2.2, and computes the schema paths presented in Figure 4.2. Figure 4.3 is the navigational paths of Q on tree representation of bib.xsd, where we only present the nodes in bib.xsd, which consist of the navigational paths. The navigational paths are visited by our XPath-XSchema evaluator, when it evaluates Q on bib.xsd. Figure 4.4 is the graphical representation of Figure 4.2, in which we only present the last node of the node sequences in a schema path

record rather than the entire record for simplicity of presentation and readability.

```
/bib/descendant::article[year][not(self::node()[editor]/ancestor::bib)]/parent::reference
```

Figure 4.1: The query Q of Figure 2.4 is repeated here for the convenience of readers

```
R1→ {(</, {(D1)}, -, -, -, ->,
R2→   </bib, {(D1, D14)}, child, {R1}, -, ->,
R3→   </bib/article, {(D14, D15, D16, D17)}, child, {R2}, -, ->,
R4→   </bib/article, {(D17, D2, D3, D9), (D13, D2, D3, D9)}, child, {R3, R5},
  R6→   {(</bib/article, {(D13, D2, D3, D9)}, child, {R5}, -, ->,
  R5→     </bib/article, {(D9, D11, D12, D13)}, child, {R4}, -, ->)}, ->
R5→   </bib/article, {(D9, D11, D12, D13)}, child, {R4}, -,
  R7→       {{(<-, {(D9, D11, D12, D13)}, self, {R4}, -, ->,
  R8→         <year, {(D13, D2, D3, D5, D19)}, child, {R7}, -, ->)},
  R9→       {(<not,
    R10→              ∅>)}} >,
R11→  <Q, {(D17, D2, D3, D9), (D13, D2, D3, D9)}, parent, {R3, R5}, -, ->}
```

Figure 4.2: Schema paths of query Q in Figure 4.1 computed on the schema bib.xsd in Figure 2.2

In order to help readers understand this example, we first outline how the XPath-XSchema evaluator searches for relevant nodes from bib.xsd to construct the schema paths of Q. Our evaluator first searches for the instance root node from bib.xsd. D1 is defined as the instance root node, and is the first relevant schema node logged in the schema paths of Q.

Since the first location step bib of Q selects the child elements bib of the root node of an XML document, our evaluator searches among *child* nodes of the instance root node D1 in bib.xsd for the *instance child* nodes, which define the elements with name bib, i.e. <element name='bib'...>. The child nodes of D1 are D2, D11, D14 and D19. D14 is the specified node and the corresponding node sequence visited is (D1, D14).

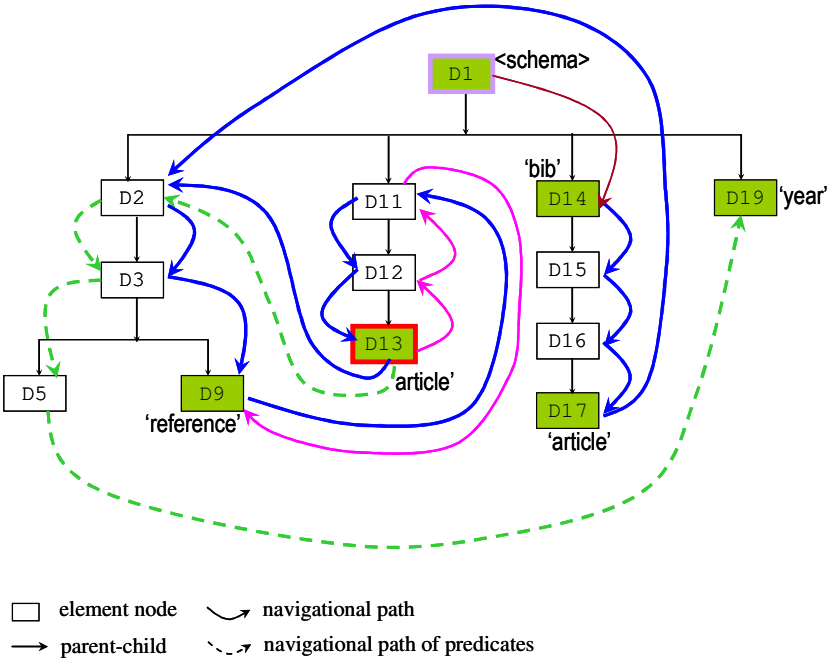

Figure 4.3: The navigational paths of Q of Figure 4.1 on the tree representation of bib.xsd of Figure 2.2, where we only present the nodes in bib.xsd, which consist of the navigational paths.

Afterwards, the search continues from an XSchema node to its *succeeding* nodes when processing forward axes. The search passes the non-*instance* nodes in the XML Schema definition, and continues until finding an *instance* node relevant to the current location step. The node sequence visited is logged.

The next location step descendant::article of Q selects all the descendant nodes article of the context nodes, and thus we search from D14 for all instance descendant element nodes <element name='article'...> of D14, i.e. we search the instance child nodes of D14, and the instance child nodes of the instance child nodes of D14, and so on.

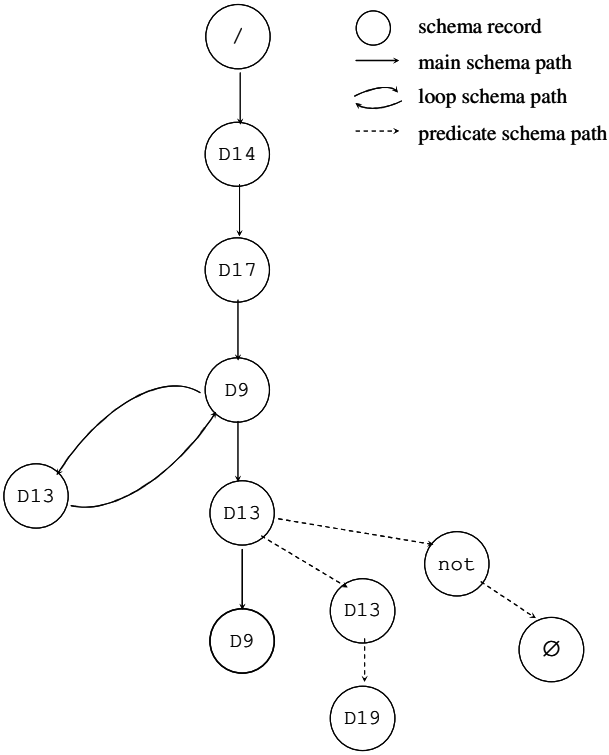

Figure 4.4: Graphical representation of the schema paths in Figure 4.2, where we only present the last node of the node sequences of the records of the schema paths

We first search the *succeeding* nodes of D14, which are D15. D15 is not an instance node, and thus we proceed to search for the succeeding nodes of D15. D15 has a succeeding node D16. Since D16 is not an instance node either, we continue the search from D16. D17 and D18 are two succeeding nodes of D16, and are also two instance element nodes. Among them D17 is a specified instance child node. Since D18 is not a specified instance node, neither it has any descendant nodes, the search along D18 is aborted. The search continues from D17.

The succeeding nodes of D17 are D2. D2 is not a specified instance node, neither it is an instance element node (while it is an *instance text* node) (see

iChild(N) in Figure 3.1), and thus we continue from D2. D2 has two succeeding nodes D3 and D10. D10 is an instance *attribute* node and thus the search along D10 aborts. The succeeding nodes of D3 are D4, D5, D6 and D9. Since D4 is not a specified instance node, nor it has any descendant nodes, the search along D4 aborts. The descendant nodes of D5 are D19, D20, D21, D22 and D23, none of which is a specified instance node by descendant::article, so D5 will not contribute to the search of resultant nodes.

Similarly, the search along D6 will be aborted too, since the succeeding nodes D7 and D8 of D6 are not the specified instance nodes, nor they have any descendant nodes. Although D9 is not a specified instance node, D9 has descendant nodes, and thus we continue from D9.

The search along D9 has the result that the instance child nodes of D9 are D13 (and the corresponding node sequence is (D9, D11, D12, D13)). An instance child node of D13 is D9 (and the corresponding node sequence is (D13, D2, D3, D9)), which has D13 as the instance child node. Thus, we are stuck in a loop! We do not continue the search along D9, when a loop is detected, i.e. our evaluator revisits an XSchema node (D9 in this example) when evaluating a location step (descendant::article in this example). The resultant nodes selected by descendant::article are D17 and an infinite number of D13, because D13 is in a loop.

descendant::article has two predicates [year] and [not(self::node()[editor]/ancestor ::bib], the context nodes of which are D17 and D13. The schema paths of predicates are computed in the same way, and are attached to the context node of the predicates. Furthermore, D17 is logged with the instance parent D14; D13 is logged with the instance parent D9. Therefore, when we evaluate the last location step parent::reference, we only check the information of parent nodes in the context record to find the resultant instance parent node, e.g. D9 is the specified instance parent node in this example.

Figure 4.5 describes the call graph of evaluating the query Q in Figure 4.1 over the XML Schema definition bib.xsd in Figure 2.2 in order to compute the schema paths (see Figure 4.2) of Q. In this figure, we write Q[i] to indicate the *i*-th location step in Q, e.g. Q[2] indicates the second location step descendant::article[year][not(self::node()[editor]/ancestor::bib)]. In the following we give a step-wise description on the computation of the schema paths of Q, and the general approach to computation of schema paths is presented in Section 4.2.

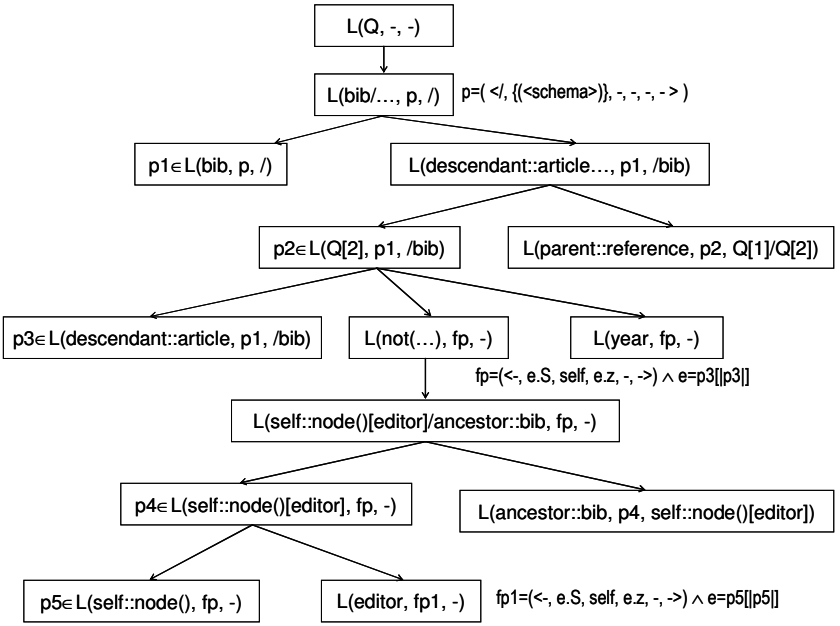

Figure 4.5: The call graph of the XPath-XSchema evaluator when evaluating Q, where Q[i] indicates the *i*-th location step in Q

Our XPath-XSchema evaluator first evaluates the very first part / of Q, and computes the first schema path record R1→ </, {(D1)}, -, -, -, -> (see line 1 of Figure 4.6 in Section 4.2.1). The schema paths computed so far are L={(R1)}. In order to evaluate the first location step bib, our evaluator uses L(child::bib, (R1), /) (see L(child::n, p, xp) of Figure 4.7), where child::bib is the XPath expression that is evaluated on bib.xsd in this function; / is the XPath expression that has been evaluated so far when the function is called. bib selects the *instance* child node D14 of the instance root node D1, and the node sequences visited are {(D1, D14)} computed from iChild(D1) (see Figure 3.1). Thus, the second schema path record is computed R2→ </bib, {(D1, D14)}, child, {R1}, -, ->. The schema paths computed so far are L={(R1, R2)}.

Afterwards, our evaluator uses L(descendant::article, (R1, R2), /bib) (see L(descendant::n, p, xp) of Figure 4.7) to evaluate the next location step descendant::article. L(descendant::article, (R1, R2), /bib) first calls L(child::node(), (R1, R2),

/bib). The selected instance child nodes of D14 from the record R2 are D17 and D18, and the corresponding node sequences are (D14, D15, D16, D17) and (D14, D15, D16, D18). Now the following schema paths L are computed so far.

L = {p1, p2}, where
p1 = (R1, R2, </bib/article, {(D14, D15, D16, D18)}, child, {R2}, -, ->)
p2 = (R1, R2, </bib/article, {(D14, D15, D16, D17)}, child, {R2}, -, ->)
= (R1, R2, R3)

Since D18 is not a resultant node of the location step descendant::article, p1 is not a resultant schema path of the current location step (filtered out by using L^r(self::article, p1, /bib)). D18 does not have any descendant nodes either, so the schema paths of the branch is computed to empty (by using L(child::node(), p1, /bib)). D17 is a resultant node of descendant::article, and thus we have the first resultant schema path of descendant::article, i.e. G={p2}={(R1, R2, R3)}, by using L^r(self::article, p2, /bib). Since D17 has descendant nodes, we now continue from p2. L(child::node(), p2, /bib) is called again, and the instance element and text nodes selected by child::node() are D2, D4, D19, D7, D8 and D9. Since we can not find any resultant nodes along D2, D4, D19, D7 and D8, the schema paths along these branches will be computed to empty. Although D9 is not a resultant node of descendant::article, D9 has an instance child node D13. Therefore, we can construct a new schema path record from D9, i.e. R4→ </bib/article, {(D17, D2, D3, D9)}, child, {R3}, -, ->, and thus the new schema paths are L={(R1, R2, R3, R4)}. From D9, we proceed to its instance child nodes D13, and get the new schema paths L={(R1, R2, R3, R4, R5)}, where R5→ </bib/article, {(D9, D11, D12, D13)}, child, {R4}, -, ->. Since D13 is a resultant node of descendant::article, we now have two resultant schema paths, i.e. G = {(R1, R2, R3), (R1, R2, R3, R4, R5)}.

The search continues from D13 by recursively calling L(child::node(), p, xp) and other functions. We get the following schema paths

R1→ {(</, {(D1)}, -, -, -, ->,
R2→ </bib, {(D1, D14)}, child, {R1}, -, ->,
R3→ </bib/article, {(D14, D15, D16, D17)}, child, {R2}, -, ->,
R4→ </bib/article, {(D17, D2, D3, D9)}, child, {R3}, -, ->
R5→ </bib/article, {(D9, D11, D12, D13)}, child, {R4}, -, ->
R6→ </bib/article, {(D13, D2, D3, D9)}, child, {R5}, -, ->
R7→ </bib/article, {(D9, D11, D12, D13)}, child, {R6}, -, ->
R8→ </bib/article, {(D13, D2, D3, D9)}, child, {R7}, -, ->
…)}

A loop occurs when evaluating descendant::article, i.e. D13 is an instance child node of D9 and D9 is an instance child node of D13. The loop is already detected when we determine the record R6, because we detect that the record R6 and a previous record R4 have the same last node D9 in the node sequence field S and the same XPath expression in the XP field. We do not continue the computation from R6, and the records after R6 are presented here for the purpose of demonstration. When a loop is detected, the loop part is integrated into the field of loop schema paths in the record, where the last schema node of the node sequences is the initial node of the loop, i.e. R4 is modified as follows:

R4→ </bib/article, {(D17, D2, D3, D9), (D13, D2, D3, D9)}, child, {R3, R5},
R6→ {(</bib/article, {(D13, D2, D3, D9)}, child, {R5}, -, ->,
R5→ </bib/article, {(D9, D11, D12, D13)}, child, {R4}, -, ->)}, ->

Note since the node D9 in the record R6 is the initial node of the loop, the record R6 becomes the first record in the loop schema path. Thus, the resultant schema paths of descendant::article are:

L=G={P1, P2}, where

P1= (R1, R2, R3)
P2 = (R1, R2, R3, R4, R5)

R1→ </, {(D1)}, -, -, -, ->
R2→ </bib, {(D1, D14)}, child, {R1}, -, ->
S3→ </bib/article, {(D14, D15, D16, D17)}, child, {R2}, -, ->
R4→ </bib/article, {(D17, D2, D3, D9), (D13, D2, D3, D9)}, child, {R3, R5}
R6→ {(</bib/article, {(D13, D2, D3, D9)}, child, {R5}, -, ->
R5→ </bib/article, {(D9, D11, D12, D13)}, child, {R4}, -, ->)}, ->
R5→ </bib/article, {(D9, D11, D12, D13)}, child, {R4}, -, ->)}

We present the detection of loops and the construction of loop schema paths in Section 4.2.2.

The location step descendant::article has two predicates, and thus two sets of schema paths are computed, and added to the field of predicate schema paths in the context records (see L(e[q_1]...[q_n], fp, xp) in Figure 4.8). So far we have two schema paths, and thus the predicates have two context records R3 (in p1) and R5 (in p2). From p2, two sets (see {(R7, R8)} and {(R9)} of Figure 4.2) of schema paths are computed, and added to the field of predicate schema paths in R5. The first record of the schema paths of the predicate expression [year] is

the record, the last schema node of which is the context node of [year]. The purpose of the record is setting the context node of the predicate expression. L(year, (R5'), -) is used to compute [year]. R5'=<-, {(D9, D11, D12, D13)}, child, {R4}, -, -> is derived from the record R5, and R5' logs the context node of [year] such that we compute the schema paths of the predicate from the schema path (R5'). L(not(self::node()[editor]/ancestor::bib)]), (R5'), -) is used to evaluate another predicate [not(self::node()[editor]/ancestor::bib)])]. Since D13 selected by self::node() does not have instance child nodes declaring element editor, the schema paths of [editor] is computed to ∅, and thus the evaluation of (self::node()[editor]/ancestor::bib]) is aborted after the evaluation of [editor]. Therefore, the schema paths of this part are computed to empty (see R10 of Figure 4.2). We present the method to evaluate predicates in Section 4.2.3.

We use L(parent::reference, p1, xp) and L(parent::reference, p2, xp) (where xp=/bib/descendant::article[year][not(self::node()[editor]/ancestor::bib)] to evaluate the last location step parent::reference of Q from p1 and p2 respectively. From p1, the parent record of R3 is R2. Since D14 in R2 declares elements with name bib, the node D14 does not conform to the node test reference of the current location step, i.e. NT(D14, reference)=false (see Figure 3.6), and thus the schema path p1 is computed to empty. From p2, the parent record of R5 is R4. Since D13 in R4 declares elements with name reference, NT(D9, reference)=true, and thus a new schema path record R11 is generated: R11→ <Q, {(D17, D2, D3, D9), (D13, D2, D3, D9)}, child, {R3, R5}, -, ->. Therefore, the resultant schema paths of Q are L = {(R1, R2, R3, R4, R5, R11}.

4.2 Computation of schema paths

We use the semantics technique to describe our XPath-XSchema evaluator, and define the following notations. Let z be a pointer in a schema path and d is a field of a schema path record, we write z.d to refer to the field d of the record to which the pointer z points. Let p be a schema path and |p| be the size of the schema path p, i.e. the number of pointers (or schema path records) in p, then p[k] indicates the *k*-th pointer (or the record to which the *k*-th pointer points) of the schema path p, and thus p[|p|].XP refers to the field XP of the last schema record of p. For readability, we often write that p[k] is the *k*-th schema path record of schema path p, instead of that p[k] is the *k*-th pointer of p, which points to a schema path record. Let S be a set of sequences of XSchema nodes, then

S(1) indicates an arbitrary sequence of nodes in S. We use the operator / to express the concatenation of two XPath expressions, e.g. XP1/XP2. If XP1=/, then XP1/XP2=/XP2; if XP1 is empty, then XP1/XP2=XP2; if XP2 is empty, then XP1/XP2=XP1.

4.2.1 Evaluating XPath expressions

The semantics of the XPath-XSchema evaluator is specified by a function L (see Figure 4.6). The function L: XPath x schema_path x XPath $\rightarrow$ Set(schema_path) takes two XPath expressions and a schema path as the arguments and yields a set of schema paths. The first XPath expression is one that is evaluated on a given XML Schema definition in this function, and the second XPath expression is the part XP2 of the given XPath query Q, which has been evaluated so far when the function is called. XP2 is bound to the XP field of a schema path record, and this field is needed for the detection of loop schema paths. The schema path in this function signature is one of the schema paths of the part XP2 of the given XPath query Q, which has been evaluated when calling this function. L(XPath, schema_path, XPath) is defined recursively on the structure of XPath expressions (see Figure 4.6).

- L(/e, -, -) = L(e, p, /), where p=(</, {(<schema>)}, -, -, -, - >)
- L(e1|e2, p, -) = L(e1, p, -) $\cup$ L(e2, p, -)
- L(e1/e2, p, xp) = {p2 | p2$\in$ L(e2, p1, xp/e1) $\wedge$ p1$\in$ L(e1, p, xp)}

Figure 4.6: The function L: XPath x schema_path x XPath $\rightarrow$ Set(schema_path) is defined recursively on the structure of XPath expressions

4.2.2 Evaluating axes and node-tests

For evaluating each location step of an XPath expression, our XPath-XSchema evaluator first computes the axis a and the node-test n of the location step a::n by iteratively taking the last schema node from a node sequence of the last schema path record (note that the last node of all the node sequences in a schema path record are the same) from each schema path p in the path set as

the context node (see Figure 4.7). The path set is computed from the part of the XPath query, which has been evaluated by the XPath-XSchema evaluator. For each resultant node r selected by a::n, L first computes a node sequence s based-on the data model of XML Schema. s[1] is the instance parent node of r, s[|s|]=r and other nodes in s are intermediate ones visited when searching for r of s[1]. The function L then constructs a pointer e to a new schema path record, i.e. e→ <xp, {s}, a, z, -, -> and extends p to p' by adding the pointer e at the end of the given schema path p, denoted by p'=p+e. In Figure 4.2, the new schema path record R11→ <Q, {(D17, D2, D3, D9), (D13, D2, D3, D9)}, parent, {R3, R5}, -, ->} is generated when evaluating the part parent::reference of the query Q, and is added at the end of p by L(parent::reference, p, parent::reference). If no node is selected by the current location step, the function L computes an empty set of schema paths. For example, since no node is selected by the current location step editor, the part [editor] of Q in Example 4.1 is computed to empty by L(child::editor, p, -), and this causes that the corresponding main schema paths are computed to empty (see (R10) in Figure 4.2).

In the case of recursive schemas, a loop is identified whenever the XPath-XSchema evaluator revisits an instance XSchema node N without any progress in the processing of the query. In order to avoid an infinite evaluation, we do not continue the evaluation along the node N, once a loop has been detected. We detect loops in the following way: let e=<xp, {s}, a, z, -, -> be a new schema path record generated when computing L(a::n, p, xp). If there exists a record p[k] in p such that S(1)[|S(1)|]=s[|s|] ∧ S=p[k].S ∧ p[k].XP=xp, a loop is detected and the loop path segment is lp = (e, p[k+1], …, p[|p|]). lp is added to the field of loop schema paths in the schema path record p[k], where the loop occurs (e.g. (R6) and (R5) in Figure 4.2). A loop might occur when an XPath query contains the axis descendant, ancestor, preceding or following, which are boiled down to the recursive evaluation of the axis child or parent respectively. For computing L(descendant::n, p, xp), we first compute p_i, where $p_i \in$ L(child::node(), p_{i-1}, xp) ∧ $p_{i-1} \in$ L(child::node(), p_{i-2}, xp) ∧…∧ p_1=L(child::node(), p, xp). If no loop is detected in the path p_i, i.e. $\forall k \in \{1, \ldots, |p_i|-1\}$: $p_i[k].XP \neq p_i[|p_i|].XP \vee (S_1(1)[|S_1(1)|] \neq S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S)$, then let $p_i'=p_i$ and L^r(self::n, p_i', xp) is computed in order to construct a possible new path from p_i. If a loop path segment ($p_i[|p_i|]$, $p_i[k+1]$, …, $p_i[|p_i|-1]$) is detected in the path p_i, i.e. $\exists k \in \{1, \ldots, |p_i|-1\}$: $p_i[k].XP=p_i[|p_i|].XP \wedge S_1(1)[|S_1(1)|]=S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S$, then the schema path record $p_i[k]$, from which the loop starts, is modified by integrating the new detected loop schema path, the new sequence of nodes and the new parent pointer, i.e.

$p_i[k] \rightarrow$ <$p_i[k].XP$, $p_i[k].S \cup p_i[|p_i|].S$, a, $p_i[k].z \cup p_i[|p_i|].z$, $p_i[k].lp \cup \{(\ p_i[|p_i|],\ p_i[k+1],\ \ldots,\ p_i[|p_i|-1])\}$, $p_i[k].f$>. Note that all the schema paths, which contain the pointers to the schema path record, are also aware of this modification. When a loop is detected, instead of setting $p_i'=p_i$, p_i' is set to empty, i.e. if a loop is detected in p_i, p_i will not contribute to the further computation of schema paths anymore.

- L(self::n, p, xp) = { p+<xp/self::n, S, 'self', p[|p|].z, -, -> |
  S=p[|p|].S $\wedge$ NT(S(1)[|S(1)|], n) }

- L(child::n, p, xp) = { p+<xp/n, {s}, 'child', p[|p|], -, -> |
  NT(s[|s|], n) $\wedge$ S=p[|p|].S $\wedge$ isiElement(S(1)[S(1)]) $\wedge$ (
  (s$\in$ iChild(S(1)[|S(1)|]) $\wedge$ n$\neq$text()) $\vee$
  (s$\in$ iTextChild(S(1)[|S(1)|] $\wedge$ (n=text() $\vee$ n=node())))) }

- L^r(self::n, p, xp) = { p | NT(S(1)[|S(1)|], n) $\wedge$ S=p[|p|].S }

- L(descendant::n, p, xp) = { p' | p'$\in \cup_{i=1}^{\infty} L^r$(self::n, p'_i, xp) $\wedge$ (
  ($p_i'=p_i \wedge p_i \in$ L(child::node(), p_{i-1}, xp) $\wedge$
  $\forall k \in \{1, \ldots, |p_i|-1\}$: ($p_i[k].XP \neq p_i[|p_i|].XP \vee$
  $(S_1(1)[|S_1(1)|] \neq S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S)$) $\wedge$
  $p_{i-1} \in$ L(child::node(), p_{i-2}, xp) $\wedge \ldots \wedge p_1 \in$ L(child::node(), p, xp))
  $\vee$
  ($p'_i=\perp \wedge$
  ($p_i[k] \rightarrow$<$p_i[k].XP$, $p_i[k].S \cup p_i[|p_i|].S$, $p_i[k].a$, $p_i[k].z \cup p_i[|p_i|].z$,
  $p_i[k].lp \cup \{(p_i[|p_i|], p_i[k+1], \ldots, p_i[|p_i|-1])\}$, $p_i[k].f$>) $\wedge$
  $\exists k \in \{1, \ldots, |p_i|-1\}$: ($p_i[k].XP=p_i[|p_i|].XP \wedge$
  $S_1(1)[|S_1(1)|]=S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S) \wedge$
  $p_i \in$ L(child::node(), p_{i-1}, xp) $\wedge$ $p_{i-1} \in$ L(child::node(), p_{i-2}, xp) $\wedge \ldots \wedge$
  $p_1 \in$ L(child::node(), p, xp))) }

- L(parent::n, p, xp) = { p + <xp/parent::n, S, 'parent', Z1.z, -, -> |
  S=Z1.S $\wedge$ Z1$\in$ p[|p|].z $\wedge$ NT(S(1)[|S(1)|], n) }

- L(ancestor::n, p, xp) = { p' | p'$\in \cup_{i=1}^{\infty} L^r$(self::n, p'_i, xp) $\wedge$ (
  ($p_i'=p_i \wedge p_i \in$ L(parent::node(), p_{i-1}, xp) $\wedge$
  $\forall k \in \{1, \ldots, |p_i|-1\}$: ($p_i[k].XP \neq p_i[|p_i|].XP \vee$
  $(S_1(1)[|S_1(1)|] \neq S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S)) \wedge$
  $p_{i-1} \in$ L(parent::node(), p_{i-2}, xp) $\wedge \ldots \wedge p_1 \in$ L(parent::node(), p, xp))
  $\vee$
  ($p'_i=\perp \wedge$

$(p_i[k] \rightarrow <p_i[k].XP, p_i[k].S \cup p_i[|p_i|].S, p_i[k].a, p_i[k].z \cup p_i[|p_i|].z,$
$p_i[k].lp \cup \{(p_i[|p_i|], p_i[k+1], ..., p_i[|p_i|-1])\}, p_i[k].f>) \wedge$
$\exists k \in \{1, ..., |p_i|-1\}: (p_i[k].XP = p_i[|p_i|].XP \wedge$
$S_1(1)[|S_1(1)|] = S_2(1)[|S_2(1)|] \wedge S_1 = p_i[k].S \wedge S_2 = p_i[|p_i|].S) \wedge$
$p_i \in L(parent::node(), p_{i-1}, xp) \wedge p_{i-1} \in L(parent::node(), p_{i-2}, xp) \wedge \ldots \wedge$
$p_1 \in L(parent::node(), p, xp))) \}$

- L(DoS::n, p, xp) = L(self::n, p, xp) $\cup$ L(descendant::n, p, xp)
- L(AoS::n, p, xp) = L(self::n, p, xp) $\cup$ L(ancestor::n, p, xp)
- L(FS::n, p, xp) = { p+<xp/FS::n, {s}, 'FS', p[|p|].z, -, -> |
  s $\in$ iFS(s1) $\wedge$ NT(s[|s|], n) $\wedge$ s1 $\in$ p[|p|].S }
- L(following::n, p, xp) = L(AoS::node()/FS::node()/Dos::n, p, xp)
- L(PS::n, p, xp) = { p+<xp/PS::n, {s}, 'PS', p[|p|].z, -, -> |
  s $\in$ iPS(s1) $\wedge$ NT(s[|s|], n) $\wedge$ s1 $\in$ p[|p|].S }
- L(preceding::n, p, xp) = L(AoS::node()/PS::node()/DoS::n, p, xp)
- L(attribute::n, p, xp) = { p+<xp/attribute::n, {s}, 'attribute', p[|p|], -, -> |
  s $\in$ iAttribute(S(1)[|S(1)|]) $\wedge$ NT(s[|s|], n) $\wedge$ S=p[|p|].S }

Figure 4.7: The function L: XPath x schema_path x XPath $\rightarrow$ Set(schema_path) for evaluating axes and node tests

4.2.3 Evaluating predicates

The schema paths L(q, fp, -) of a predicate q are added into the field of predicate schema paths in the record, where the last node of the field of the node sequences is the context node of the predicate, e.g. L(e[q], p, xp) = {(p'[1], p'[2], ..., p'[|p'|-1]) + <p'[|p'|].XP, p'[|p'|].S, p'[|p'|].a, p'[|p'|].z, p'[|p'|].lp, p'[|p'|].f $\cup$ L(q, fp, -)> | p' $\in$ L(e, p, xp) $\wedge$ L(q, fp, -) $\neq \emptyset$ $\wedge$ fp=(<-, p'[|p'|].S, 'self', p'[|p'|].z, -, ->)} (see Figure 4.8). fp logs the context node of the predicate such that we compute the schema paths of the predicate from fp. When L(q, fp, -) is computed to empty, the main schema paths are computed to an empty set of schema paths, i.e. L(e[q], p, xp)=$\emptyset$ if L(q, fp, -) =$\emptyset$. When [q] = [q_1 or q_2], L(q_1 or q_2, fp, -) computes a schema

path with only one record for the predicate expression q_1 or q_2, i.e. {(<'or', $L(q_1, fp, -) \cup L(q_2, fp, -)$>)} that consists of a keyword or and two sets of schema paths computed from q_1 and q_2. The schema path is added into the field of predicate schema paths of the record, where the last node in the field of the node sequences is the context node of [q_1 or q_2]. If both $L(q_1, fp, -)$ and $L(q_2, fp, -)$ are computed to empty, the schema paths of the predicate q_1 or q_2 are computed to the empty set, i.e. $L(q_1 \text{ or } q_2, fp, -) = \emptyset$ if $L(q_1, fp, -) = \emptyset \wedge L(q_2, fp, -) = \emptyset$.

- $L(e[q], p, xp) = \{ (p'[1], p'[2], \ldots, p'[|p'|-1]) +$
  $<p'[|p'|].XP, p'[|p'|].S, p'[|p'|].a, p'[|p'|].z, p'[|p'|].lp, p'[|p'|].f \cup L(q, fp, -)> \mid$
  $p' \in L(e, p, xp) \wedge L(q, fp, -) \neq \emptyset \wedge fp=(<-, p'[|p'|].S, \text{'self'}, p'[|p'|].z, -, ->)\}$
- $L(e[q_1]\ldots[q_n], p, xp) = \{ (p'[1], p'[2], \ldots, p'[|p'|-1]) +$
  $<p'[|p'|].XP, p'[|p'|].S, p'[|p'|].a, p'[|p'|].z, p'[|p'|].lp,$
  $p'[|p'|].f \cup L(q_1, fp, -) \cup \ldots \cup L(q_n, fp, -)> \mid$
  $p'=L(e, p, xp) \wedge L(q_1, fp, -) \neq \emptyset \wedge \ldots \wedge L(q_n, fp, -) \neq \emptyset \wedge$
  $fp=(<-, p'[|p'|].S, \text{'self'}, p'[|p'|].z, -, ->) \}$
- $L(q_1 \text{ and } q_2, fp, -) = \{ (<\text{'and'}, L(q_1, fp, -) \cup L(q_2, fp, -)>) \mid$
  $L(q_1, fp, -) \neq \emptyset \wedge L(q_2, fp, -) \neq \emptyset \}$
- $L(q_1 \text{ or } q_2, fp, -) = \{ (<\text{'or'}, L(q_1, fp, -) \cup L(q_2, fp, -)>) \mid$
  $L(q_1, fp, -) \neq \emptyset \vee L(q_2, fp, -) \neq \emptyset \}$
- $L(q_1 = q_2, fp, -) = \{ (<\text{'='}, L(q_1, fp, -) \cup L(q_2, fp, -)>) \mid$
  $L(q_1, fp, -) \neq \emptyset \wedge L(q_2, fp, -) \neq \emptyset \}$
- $L(\text{not}(q), fp, -) = \{ (<\text{'not'}, L(q, fp, -)>) \}$
- $L(q \text{ op } C, fp, -) = L(q[\text{self::node()} \text{ op } C], fp, -)$, where $q \neq \text{self::node()}$
- $L(\text{self::node()} \text{ op } C, fp, -) = \{ (<\text{self::node()} \text{ op } C>) \}$

Figure 4.8: The function L: XPath x schema_path x XPath → Set(schema_path) for evaluating predicates

4.2.4 Integrating data type checking

The XML Schema language [W3C Schema2 2004] defines 44 built in simple types, and allows users to define new simple types. If values of an element or an attribute in an XPath query do not conform to the value type of the element or the attribute specified in the given XML Schema definition, the XPath query selects an empty set of nodes for any XML document, which is valid according to the XML Schema definition. Therefore, integration of data type checking, when evaluating XPath queries on an XML Schema definition, can detect more unsatisfiable queries.

The data type checking is involved in the computation of the schema paths of the predicate expression self::node() op C, and thus we modify the function L(self::node() op C, p, -) of Figure 4.8 in order to integrate type-checking (see Figure 4.9). In the XPath language, the value of an element is the text node of the element, and thus e.g. two predicate expressions child::mark=1.0 and child::mark/child::text()=1.0 are semantically equal. Therefore, if the node selected by self::node() is an element node, we evaluate child::text()/self::node()=C rather than self::node()=C in order to make the node selected by self::node() be a text node. If the constant C of the predicate expression self::node()=C conforms to the value type of the node specified by self::node(), the predicate expression itself as the schema paths is added to the field of the predicate schema paths of the record, the last node of the node sequences of which is the context node of the predicate expression. If C does not conform to the type constraints, the predicate expression self::node()=C is computed to the empty set of schema paths, i.e. L(self::node()=C, p, -)=∅, and thus the corresponding main schema paths are computed to the empty set of schema paths. The auxiliary function typeChecking(type, C) validates whether or not the constant C conforms to the given type; the auxiliary function valueType(N) returns the type of values of the element or the attribute declared in the node N and the restricting facets of the values.

- L(self::node() op C, p, -) = { p1 | (
  (p1∈ L(child::text()/self::node() op C, p, -) ∧
  ¬isiText(N) ∧ ¬isiAttribute(N) ∧ N=S(1)[|S(1)|] ∧ S=p[|p|].S)
  ∨
  (p1=(<self::node() op C>) ∧ typeChecking(valueType(N), C) ∧
  N=S(1)[|S(1)|] ∧ S=p[|p|].S ∧ (

```
(valueType(N)=(T, -) ∧ (
   N=@<type=T> ∨ N=<attribute type=T…> ∧ built-in(T) )
∨
( valueType(N)=computeType(N1, facets) ∧ (
     ( N1=<simpleType name=T...> ∧ N1∈ succeeding(N) ∧
          N=<attribute type=T…> ∧ ¬built-in(T) )
     ∨
     ( N1=<simpleType…> ∧ N1∈ child(N)  ∧
          N=<attribute…> ∧ N@<type>=⊥ )) ∧
  |facets|=12 ∧ facets[1]=null ∧ … ∧ facets[12]=null )
∨
(valueType(N)=('string', -) ∧ (
    N=<complexType mixed= 'true'…> ∨
    N=<complexContent mixed= 'true'…> )
∨
( valueType(N)=computeType(N, facets) ∧ (
          N=<simpleType…> ∨ N=<simpleContent…> ) ∧
     |facets|=12 ∧ facets[1]=null ∧ … ∧ facets[12]=null )))}
```

Figure 4.9: The function L(self::node() op C, p, -) is modified for integrating data type checking

XML Schema specifies a specific data-type for values of an element if the element is of a simple type, i.e. the element consists of only a text node without attributes and sub-elements. The attributes are always of simple types. The type of values of an element of simple type and an attribute can be either the built-in simple types of the XML Schema language or user-defined simple types. New simple types are derived from existing simple types, which are called the *base* types of the derived types, by restricting the range of base types. XML Schema applies one or more *facets* to restrict the legal values of base types. Thus, a new simple type is a particular combination of a base type and the facets. Base types can be built-in or derived, and thus in order to know what a new simple type is, one must find the source of the derivation, i.e. the built-in simple type, and all the restrictions imposed by the sequence of the derivations. The function computeType(N, facets) (see Algorithm 4.1) computes the type of values of an element or an attribute, if the element or the attribute is not of a built-in simple type, where N is the instance text node of the node that

declares the element or N is the succeeding node <simpleType…> of the instance attribute node that declares the attribute.

Whenever an instance text node N is the attribute node type=T of an element declaration node, then T must be a built-in simple type. In this case, the value type of the element is T without restricting facets, i.e. valueType(N)=(T, -). Let N be the attribute node type='string' in D4=<element name='title' maxOccurs='1' type='string'/> of Figure 2.2, then valueType(N)=('string', -). When an XSchema node declares an element with a derived simple type, it has a succeeding node N=<simpleType...> as its instance text node. The derived simple type is computed by the function computeType(N, facets). For example, D19 in Figure 2.2 defines an element year without the attribute node type=T. D19 has a succeeding node D20=<simpleType>, which indicates that elements year are of a derived simple type. D20 defines a new simple type by specifying a base type and by restricting the range of values of the base type. The base type is specified in the child node D21=<restriction base='int'> of D20, and the restricting facets are described in the child nodes D22 and D23 of D21.

Since an attribute is always of simple type, the attribute node can be declared with a built-in simple type, or with a user-defined simple type, or with an anonymously new simple type. Therefore, if N=<attribute type=T…> and built-in(T), i.e. T is a built-in simple type, the value type of the attribute is the built-in simple type without restricting facets, i.e. valueType(N)= (T, -). If N=<attribute type=T…> and ¬built-in(T), then T is defined by a node N1=<simpleType name=T…> that is a succeeding node of N, and the value type of the attribute defined in N is computed by the function computeType(N1, facets). If an instance attribute node N does not contain a named type, i.e. N@<type>=⊥, the instance attribute node has an anonymous type that is defined in a child node N1=<simpleType…> of N, the value type of the attribute declared in the node N is computed by the function computeType(N1, facets).

When a schema node declares an element with only text data and attributes, it has an instance text node N=<simpleContent...>. XML Schema also specifies a specific data type for the text data of the element, and the type of values of the element is computed by computeType(N, facets). The function computeType(N, facets) is further explained later on.

Whenever an element contains elements and text nodes for its value, i.e. declared as <complexType mixed='true'…> or <complexContent mixed='true'...>, XML Schema does not impose any specific data type for values of the element. Therefore, the value is considered as character string, and there is no restrict-

ing facet either, i.e. we do not check the data type in this case. For example, D2=<complexType name='articleType' mixed='true'> in Figure 2.2 is an instance text node of D17 and D13, which declare an element article of complex type. Therefore, valueType(D2) = ('string', -) and typeChecking(valueType(D2), C) = true, and thus L(self::node()=C, p, -) = {(<self::node()=C>)}.

Algorithm 4.1 computeType(N, facets) describes how to retrieve the type of values of an attribute and an element according to the syntax for simpleTypeD and simpleContentD (see Definition 2.1 in Chapter 2). XML Schema identifies 12 restricting facets, and thus the argument facets is an array containing 12 string data. We use the name of facets specified in [W3C Schema2 2004] as the index of the array to which the value of the facet is bound. Figure 4.10 presents an example, which demonstrates Algorithm 4.1. Note that in this work we consider only the XML Schema definitions, which are syntactically and semantically correct.

Algorithm 4.1: computeType (N, facets)

```
input: node N, string[12] facets;
output: (string base, string[12] facets);

    N1∈ child(N);
    If (N1=<extension…>) {
        base=attribute(N1, 'base');
        if (built-in(base)) return (base, facets);
        else {
            N2∈ succeeding(N1), where N2=<simpleType…>;
            return computeType(N2, facets);
        }
    }
    If (N1=<restriction…>) {
        base=attribute(N1, 'base');
        if (∃N2∈ succeeding(N1): N2=<simpleType…>)
          (base, facets)=computeType(N2, facets);
        ∀N2∈ succeeding(N1) {
            if (N2=<length value=V />)  facets[length]=V;
            if (N2=<minLength value=V />) facets[minLength]=V;
            if (N2=<maxLength value=V />) facets[maxLength]=V;
            if (N2=<pattern value=V />) facets[pattern]=V;
```

```
            if (N2=<enumeration value=V />) facets[enumeration]=V;
            if (N2=<whiteSpace value=V />) facets[whiteSpace]=V;
            if (N2=<maxInclusive value=V />) facets[maxInclusive]=V;
            if (N2=<maxExclusive value=V />) facets[maxExclusive]=V;
            if (N2=<minInclusive value=V />) facets[minInclusive]=V;
            if (N2=<minExclusive value=V />) facets[minExclusive]=V;
            if (N2=<totalDigits value=V />) facets[totalDigits]=V;
            if (N2=<fractionDigits value=V />) facets[fractionDigits]=V;
        }
        return (base, facets);
    }
```

In Algorithm 4.1, node N1=<extension base=*Name*> is a child node of <simpleContent...>; node N2=<restriction base=*Name*> is a child node of <simpleType...>. Both nodes indicate the base type of the derivation, which may be either a built-in or a derived simple type. If the base type is not a built-in simple type, there is a node <simpleType name=*Name*> with the same *Name*, which defines the base type of the derived type, and which is a succeeding node of N1 or N2. Thus, a new simple type might be derived recursively from a sequence of existing simple types, until the *base* is a built-in simple type. The facets that restrict the range of values of the base type are identified by several child nodes of <restriction...>. Furthermore, the restrictions imposed by a derived type override the restrictions from its base type. If <restriction...> does not have a succeeding node <simpleType...>, the attribute base of the node <restriction...> must be a built-in simple type. This means that we find the source of derivation and all restricting facets, i.e. we compute the type of values of the element or the attribute.

4.2.5 Integrating occurrence constraints checking

XML Schema specifies some constraints that control the occurrence of elements and attributes and their values. When an element is declared with maxOccurs=0 (and minOccurs=0, because it is an error if minOccurs≠0) or a model group of the element is declared with maxOccurs=0, or when an attribute is declared with use='prohibited', the element and the attribute must not appear in any instance document. When an element or an attribute is declared to have

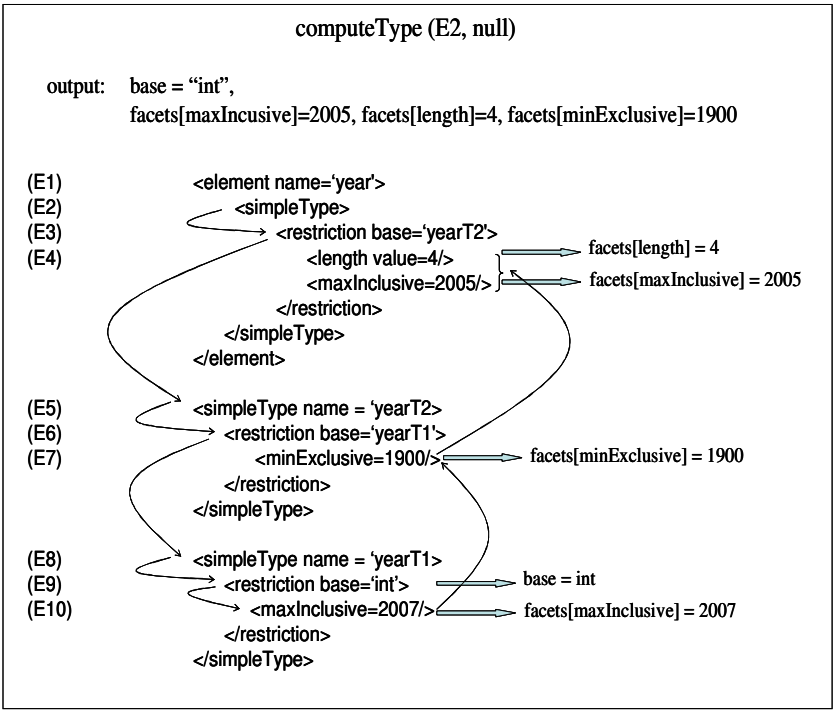

Figure 4.10: An example demonstrating Algorithm 4.1

a fixed value, e.g. fixed='100', values of the element or the attribute in all instance documents must be 100.

In order to integrate the occurrence constraints checking, we modify the data model of XML Schema, especially the functions iChild(x) and iAttribute(x) in Figure 3.1, as follows (see Figure 4.11):

- iChild(N) = { y+(N1) | (y=(N) ∧ isiRoot(N) ∧ N1∈ child(N) ∧ isiElement(N1))
  ∨
  y∈ iChild-helper(N) ∧ N1∈ succeeding(y[|y|]) ∧ isiElement(N1) ∧
  ∀i∈ {2, 3, …, |y|}: (
  ((y[i]=<group maxOccurs=D…> ∨
  y[i]=<sequence maxOccurs=D…> ∨
  y[i]=<choice maxOccurs=D…> ∨

y[i]=<all maxOccurs=D….>) ∧ D>0)
∨
(y[i]≠<group…> ∧ y[i]≠<sequence…> ∧
y[i]≠<choice…> ∧ y[i]≠<all…>)) ∧
(y[|y|]=<element ref=E maxOccurs=D…> ∧ D>0) ∨
[|y|]≠<element ref=E…>) ∧
attribute(N1, maxOccurs)>0 }

- iAttribute(N) = { y+(N1) | y∈ iChild-helper(N) ∧ N1∈ succeeding(y[|y|]) ∧
isiAttribute(N1) ∧ (
(y[|y|]=<attribute ref=A> ∧ attribute(y[|y|], 'use')≠'prohibited')
∨
(y[|y|]≠<attribute ref=A> ∧ attribute(N1, 'use')≠'prohibited'))}

Figure 4.11: Functions iChild(x) and iAttribute(x) in the XML Schema data model in Figure 3.1 is modified for integrating occurrence constraints checking

The function iChild(N) first computes a set S of node sequences using the auxiliary function iChild-helper(N). Each sequence y∈ S consists of N and the nodes visited after N but before an instance child node of N. If the succeeding nodes N1 of y[|y|] are not the instance element nodes, then no node sequence is computed from y. In the case of a succeeding node N1 of y[|y|] being an instance element node, iChild(N) returns the node sequence y+(N1), only when each model group of N1 is declared with maxOccurs>0, i.e. if u is a node in y, then u is either a node of a model group with maxOccurs>0, or is a node rather than the node of a model group. If y[|y|]=<element ref=E maxOccurs=D...>, then y[|y|] has a succeeding node N1=<element name=E...>. N1 is an instance child node of N only when D>0. Note that we do not check the attribute maxOccurs of the instance parent node y[1] of N1, because the elements defined in instance ancestor nodes of N1 have been checked for the occurrence constraints.

The constraints on fixed values are closely related with type-checking, and thus the function L(self::node() op C, p, -) is further modified as follows (see Figure 4.12):

- L(self::node() op C, p, -) = { p1 | (
(p1∈ L(child::text()/self::node() op C, p, -) ∧
¬isiText(N) ∧ ¬isiAttribute(N) ∧ N=S(1)[|S(1)|] ∧ S=p[|p|].S)
∨

```
( p1=(<self::node() op C>) ∧ C=V ∧ N=<attribute fixed=V…> ∧
  isiAttribute(N) ∧ N=S(1)[|S(1)|] ∧ S=p[|p|].S )
∨
( p1=(<self::node() op C>) ∧ C=V ∧ N1=<element fixed=V…> ∧
  N1=s[1]  ∧ s∈ p[|p|].S ∧ isiText(N) ∧ N=S(1)[|S(1)|] ∧ S=p[|p|].S )
∨
( p1=(<self::node() op C>) ∧ typeChecking(valueType(N), C) ∧
  N=S(1)[|S(1)|] ∧ S=p[|p|].S ∧ (
      ( valueType(N)=(T, -) ∧ N=@<type=T> ∧
            N1@<fixed>=⊥ ∧ N1=s[1] ∧ s∈ p[|p|].S )
      ∨
      ( valueType(N)=(T, -) ∧ N=<attribute type=T…> ∧
          built-in(T) ∧ N@<fixed>=⊥ )
      ∨
      ( valueType(N)=computeType(N1, facets) ∧ (
          ( N1=<simpleType name=T...> ∧ N1∈ succeeding(N) ∧
              N@<fixed>=⊥ ∧ N=<attribute type=T…> ∧ ¬built-in(T) )
          ∨
          ( N1=<simpleType…> ∧ N1=child(N) ∧ N@<fixed>=⊥ ∧
              N@<type>=⊥ ∧ N=<attribute…> )) ∧
        |facets|=12 ∧ facets[1]=null ∧ … ∧ facets[12]=null )
      ∨
      ( valueType(N)=('string', -) ∧ N1@<fixed>=⊥ ∧ N1=s[1] ∧
          s∈ p[|p|].S  ∧ (
              N=<complexType mixed='true'…>  ∨
              N=<complexContent mixed='true'…> ))
      ∨
      ( valueType(N)=computeType(N, facets) ∧ N1@<fixed>=⊥ ∧
          N1=s[1] ∧ s∈ p[|p|].S ∧ (
              N=<simpleType…> ∨ N=<simplexContent…> ) ∧
        |facets|=12 ∧ facets[1]=null ∧ … ∧ facets[12]=null )))))}
```

Figure 4.12: The function L(self::node() op C, p, -) is further modified for integrating fixed value checking

Let self::node()=C be a predicate, then we use L(self::node()=C, p, -) to compute the schema paths of the predicate. In L(self::node()=C, p, -), if N=<attribute…> is the node selected by self::node(), N can carry the attribute

fixed. If N contains the attribute fixed, i.e. N=<attribute fixed=V...>, the schema paths of the predicate self::node()=C is {(<self::node()=C>)} if and only if C=V; the schema paths of the predicate self::node()=C is computed to the empty set if C≠V, and thus the corresponding main paths are computed to the empty set. If N does not contain the attribute fixed, i.e. N@<fixed>=⊥, C must conform to the type of values of the attribute defined in N, in order to compute {(<self::node()=C>)} from the predicate self::node()=C; if C does not conform to the constraint of type, then L(self::node()=C, p, -)=∅. When the node selected by self::node() is an attribute node @<type=T> or a node <simpleType...> or a node <simpleContent...> or a node <complexType...> or a node <complexContent...>, these nodes do not contain the attribute fixed, which can be contained by the instance parent node of these nodes, i.e. N1=<element...>, which is the first node in the corresponding node sequence.

4.2.6 Filtering redundant schema paths

Let Q1 and Q2 be two XPath queries, and let Q1=a[b] and Q2=a[not(b)]. If a schema specifies that b must occur whenever a occurs, then the predicate [b] in Q1 is always evaluated to true and the predicate [not(b)] in Q2 is always be evaluated to false. Therefore, Q1 is equal to a semantically (denoted as Q1≡a) and Q2 is unsatisfiable with respect to this schema. Let Q3=a[@b] and Q4=a[not(@b)]. If a schema declares that the attribute b is required, then the predicate [@b] in Q3 is always evaluated to true and the predicate [not(@b)] in Q4 is always be evaluated to false. Therefore, Q3 is equal to a semantically and Q4 is unsatisfiable with respect to this schema. We call b (and @b) a redundant part. The corresponding schema paths of b (and @b) are the redundant schema paths, and can be eliminated. Therefore, eliminating redundant schema paths can not only reduce the size of an XPath query, but also discover more unsatisfiable queries. We first study the basic concepts and properties of schema paths in order to filter the redundant schema paths.

Definition 4.2 (*defining sequences*): Let x=(N1, N2, ..., N) be a sequence of schema nodes. If x∈iChild(N1) ∨ x∈iTextChild(N1) ∨ x∈iAttributeChild(N1), then x is a defining sequence of N.

Definition 4.3 (*unconditional nodes*): Let e_c be an element (or an attribute) declared by an instance XSchema node N; let the element e_p be the parent of e_c. If e_p occurs in an instance XML document, e_c must occur as a child (or an attribute) of e_p in the XML document. We call the instance XSchema node N a *unconditional* node.

Theorem 4.1: Let x be a defining sequence of an instance element node N. If $\forall i \in \{2, ..., |x|\}$: x[i]≠<choice...> ∧ (x[i]@<minOccurs>=⊥ ∨ attribute(x[i], minOccurs)≥1), then N is an unconditional instance element node.

Theorem 4.1 describes that the node N is an unconditional node, if no node in the defining sequence x of N is a node of <choice...>, and if a node in x carries the attribute minOccurs, the value of minOccurs must be greater than 0.

Theorem 4.2: Let x be a defining sequence of an instance attribute node N. If x[|x|-1]=<attribute use='required' ref=...> ∨ N=<attribute use='required' name=...>, then N is an unconditional instance attribute node.

If an instance attribute node N contains the construct use='required', i.e. N = <attribute use='required' name=...>, then N is an unconditional instance attribute node. If N is a global attribute declaration (i.e. a child node of <schema...>), N is not allowed to contain the attribute use, and the node before N in x must be an attribute reference node, i.e. x[|x|-1]=<attribute ref=...>. If x[|x|-1] contains use='required', then N is an unconditional instance attribute node.

Definition 4.4 (*redundant set of schema paths*): Let L be a set of schema paths computed from an XPath expression. If $\exists p \in L$: p is a redundant schema path, then L is a set of redundant schema paths.

Definition 4.5 (*redundant schema paths*): Let p be a schema path. p is a redundant schema path, if p=(< 'true()'>) or if $\forall e \in p$: e is a redundant schema path record.

Definition 4.6 (*redundant schema path records*): Let e be a schema path record. If the last node in the field of the node sequences is unconditional and each set of the predicate schema paths is redundant, then e is a redundant schema path record.

Let S be a set, and S' be a subset of S, we write S-S' to represent removing the subset S' from S. The following Theorem 4.3 describes the rules to eliminate redundant schema paths and filter unsatisfiable XPath queries.

Theorem 4.3: Let e be a schema path record, p be a schema path, and $\perp$ indicate an empty schema path or an empty schema path record, then

- e=<XP, S, a, z, lp, f> = <XP, S, a, z, lp, ->, if $\forall L \in f$: L is redundant.
- e=<XP, S, a, z, lp, f1$\cup$f2> = <XP, S, a, z, lp, f1>, if $\forall L \in f2$: L is redundant.
- p=(< 'and', {p1, p2}>) = p1 if p2 is redundant.
- p=(< 'and', {p1, p2}>) = p2 if p1 is redundant.
- p=(< 'or', {p1, p2}>) = (< 'true()'>), if p1 or p2 is redundant.
- p=(p[1], p[2], ..., p[k], p[k+1], ..., p[|p|]) = (p[1], p[2],.., p[k]) if each of p[k+1], ..., p[|p|] is redundant, where $p \in L \wedge L \in e.f \wedge (e=<xp, S, a, z, lp, f> \vee e=<$ 'and', f> $\vee$ e=<'or' f> $\vee$ e=<'not' f>).
- p=(< 'not' {p1}>) = $\perp$ if p1 is redundant.
- p=(< 'and', {p1, p2}>) = $\perp$ if p1=$\perp \vee$ p2=$\perp$.
- p=(< 'or', {p1, p2}>) = $\perp$ if p1=$\perp \wedge$ p2=$\perp$.
- p={p[1], p[2], ..., p[|p|]} = $\perp$ if $\exists i \in \{1, ..., |p|\}$: p[i] = $\perp$.
- e=<XP, S, a, z, lp, f> =$\perp$, If $\exists L \in f$: $\forall p \in L$: p = $\perp$.

We check whether or not a schema path is redundant by only checking the last schema path record. If the last schema path record is redundant, we eliminate it and the pre-last schema path record becomes the last one; if the last schema path record is not redundant, the schema path is not redundant. If a schema path is eliminated completely, then this schema path is redundant, and thus the corresponding set of schema paths is redundant. In this way, each schema path record is checked at most once, and thus the complexity of eliminating redundant schema paths is linear to the number of the schema path records of a schema path.

4.3 Complexity analysis

We first analyze the complexity of our approach in the worst case. Different from instance XML documents the topology of which is a tree, an XML Schema definition is a directed graph. In the directed graph, which leads to the worst-case complexity, each node has directed edges to all nodes. Therefore, we assume that in an XML Schema definition XSD in the worst case, each node in XSD is an instance node and each node is a succeeding node of all the

nodes. In an XPath query Q in the worst case, each location step in Q selects all the instance nodes in XSD.

Let a be the number of location steps in the query Q. Let N be the number of nodes and i be the number of the instance nodes in an XML Schema definition XSD, where $i \leq N$. In the worst case, from each schema path p of the result of the previous location step, firstly N nodes are visited and selected as the resultant nodes and these N nodes are directly reachable from the context node. Let the length of the schema path p be s. Therefore, N new schema path records are created and N schema paths with the length of s+1 are computed. From each of N visited nodes, N succeeding nodes are visited and selected as the resultant nodes, one of which is revisited. No new schema paths are computed from the revisited nodes, and they do not contribute to the further computation of schema paths either, but the revisited nodes indicate the occurrence of a loop. Therefore, N-1 new schema path records are created, one existing schema path record is modified by integrating the new loop schema path, and N-1 new schema paths with length of s+2 are computed. Therefore, there are $N+N*N$ nodes visited and $N+N*(N-1)$ schema paths with length from s+1 to s+2 computed so far. From each of $N*(N-1)$ nodes, N succeeding nodes are visited and selected as the resultant nodes, two of which are revisited. Therefore, N-2 new schema path records are created, two existing schema path records are modified by integrating the new loop schema path, and N-2 new schema paths with length s+3 are computed. $N+N*N+N*(N-1)*N$ nodes are visited and $N+N*(N-1)+N*(N-1)*(N-2)$ schema paths with length from s+1 to s+3 are computed so far. After a location step is evaluated, from each schema path p of the result of the previous location step, $N+N*N+N*(N-1)*N+\ldots+N*(N-1)*(N-2)*\ldots*2*N = N*\sum_{k=0}^{N-1} N!/(N-k)!$ nodes are visited and $N+N*(N-1)+N*(N-1)*(N-2)+\ldots+N*(N-1)*(N-2)*\ldots*2*1 = \sum_{k=1}^{N} N!/(N-k)!$ schema paths are computed with length from s+1 to s+N.

Let $X = N*\sum_{k=0}^{N-1} N!/(N-k)!$ and $P = \sum_{k=1}^{N} N!/(N-k)!$. In the worst case, having evaluated the first location step, X nodes are visited and P schema paths are created with length from 1 to N; having evaluated the first two location steps, $X + P*X$ nodes are visited and P^2 schema paths are created with length from 2 to N+N; having evaluated Q, $X+P*X+ P^2*X+\ldots+P^{a-1}*X = X*\sum_{j=0}^{a-1} P^j$ nodes are visited and P^a schema paths are created with length from a to $a*N$. Since $\sum_{k=1}^{N}(N!/(N-k)!) < N!*3$ and $\sum_{k=0}^{N-1}(N!/(N-k)!) < N!*2$, thus $X*\sum_{j=0}^{a-1} P^j = N*\sum_{k=0}^{N-1} N!/(N-k)!*\sum_{j=0}^{a-1} (\sum_{k=1}^{N} N!/(N-k)!)^j < N*N!*2*\sum_{j=0}^{a-1}(N!*3)^j < N*N!*2*a*(N!*3)^{a-1}$. Therefore, the XPath-XSchema evaluator visits at most $O(N*N!*a*(N!*3)^{a-1})$ nodes, and creates

at most $O((N!*3)^a)$ different schema paths, each of which contains at most $O(a*N)$ pointers, and thus $O((N!*3)^a)$ schema paths contains at most $O(a*N*(N!*3)^a)$ pointers to at most $O(N*N!*a*(N!*3)^{a-1})$ schema path records, when evaluating Q.

Therefore, the worst case complexity of our approach in terms of runtime and space is $O(a*N*(N!*3)^a)$.

The XML Schema definitions of the worst case, where each node has all the nodes as succeeding nodes and each node is an instance node, are rare. A query that selects all the nodes of an XML instance document is /descendant-or-self::node(). Other queries with multiple location steps each of which selects up to all nodes are typically not used. Therefore, it makes sense to investigate the complexity of our approach at typical cases.

According to the schema and the queries given in the XPath benchmark [Franceschet 2005], we assume that the typical cases are characterized as follows: each node in an XML Schema definition XSD has only a small number of succeeding nodes compared with the number N of nodes in XSD; for each location step of an XPath query Q, the number of nodes visited is in average less than a constant C, and thus less than C schema paths are created for each location step. Therefore, after Q is evaluated for the typical case, $a*C$ nodes are visited and C schema paths are created, the length of each of which is at most $a*N$.

Therefore, under these assumptions, the complexity of runtime and space of our approach is $O(a*N*C)$ at the typical cases. When the number of the nodes visited is in average less than N for each location step and this is quite typical based on the schema and queries given in the XPath benchmark [Franceschet 2005], the complexity of our approach in terms of runtime and space is $O(a*N*N)$ for the typical case.

We develop a prototype of our XPath-XSchema evaluator. Since the XPath-XSchema evaluator is closely related to the satisfiability test of XPath queries presented in Chapter 6, a comprehensive performance analysis of our prototype is given in Chapter 6. The experimental results show the efficiency and usability of our XPath-XSchema evaluator.

Chapter 5 XPath Rewriting

After computing the schema paths of an XPath query, we can construct an XPath query, which is equivalent to the original one, but in which redundant location steps are eliminated, wildcards are replaced with specific node-tests, and reverse axes and recursive axes are eliminated wherever possible. The rewriting approach of XPath queries includes mapping a set of schema paths to a (regular) XPath expression, and optimizing the mapped XPath expression by a set of equivalence rules.

5.1 Mapping schema paths to (regular) XPath Expressions

We develop a group of functions (see Figure 5.1), which map a set of schema paths to a standard or a regular XPath expression. An XPath expression is a regular XPath expression if the Kleene star $*$ operation is allowed in the XPath expression. Let e be an XPath expression or a part of an XPath expression, then e* indicates an arbitrary repetition of e, e.g. a/b*/c = a/(b^0 | b^1 | b^2 | ...)/c = a/(⊥ | b | b/b |...)/c, where ⊥ represents the empty expression.

The function M(L) in Figure 5.1 maps a set of schema paths L={p_1, ..., p_m} to an XPath query Q'. The function M(p) maps a schema path p=(r_1, ..., r_n) to a sub-expression e of the query Q'. The function M(r) maps a schema path record r to a pattern of the sub-expression e. The patterns are concatenated with '/' in order to form the sub-expression e=M(p)=M(r_1)+'/'+...+'/'+M(r_n), where we use '+' to denote the concatenation of strings. Disjunctions of the sub-expressions form the mapped query Q'=M(L)=M(p_1)+'|'+...+'|'+M(p_n). In order to compute a pattern from a schema path record <XP, S, a, z, lp, f>, <o, f> or <e>, we need the following functions. The function location(S, a) computes the axis and the node-test of a pattern; the function loops(lp) computes the union of loop patterns. Let us assume that B is a pattern, then we define B* as a loop pattern, in which the

Kleene star $*$ denotes an arbitrary repetition of the pattern B. As an example, if B=b, then B*=(b^0 | b | b/b | b/b/b |...), where b^0 is the empty expression $\perp$.

Let L be a set of schema paths, p be a schema path and r be a schema path record, such that L={p_1,..., p_m} and p=(r_1,..., r_n), where p$\in$L. The semantics of the mapping function M, which maps a set of schema paths to a (regular) XPath expression, is defined in Figure 5.1. Note that in the mapping functions of Figure 5.1, the two fields XP and z in a schema path record r are left out since they do not contribute to the computation of the mapping.

- M($\varnothing$) = 'false()'
- M(L) = M(p_1)+'|'+ …+'|'+M(p_m)
- M(p) = M(r_1)+M(r_2)+'/'+…+'/'+M(r_n), if N='/' $\wedge$ N=S(1)[|S(1)|] $\wedge$ S= r_1.S
- M(p) = M(r_1)+'/'+…+'/'+M(r_n), if N$\neq$'/' $\wedge$ N=S(1)[|S(1)|] $\wedge$ S= r_1.S
- M(<S, a, -, ->) = location(S, a)
- M(<S, a, -, {f_1, ..., f_n}>) = M(<S, a, -, ->)+'['+M(f_1)+']'+…+'['+M(f_n)+']'
- M (<S, a, lp, ->) = 'descendant::'+attribute(N, 'name'), where a='child' $\wedge$ N=S(1)[|S(1)|]
- M (<S, a, lp, ->) = 'ancestor::'+attribute(N, 'name'), where a='parent' $\wedge$ N=S(1)[|S(1)|]
- M^r(<S, a, lp, ->) =loops(lp)+location(S, a)
- M(<S, a, lp, {f_1, ..., f_n}>) = M(<S, a, lp, ->)+'['+M(f_1)+']'+…+'['+M(f_n)+']'
- M^r(<S, a, lp, {f_1, ..., f_n}>) = M^r(<S, a, lp, ->)+'['+M(f_1)+']'+…+'['+M(f_n)+']'
- M(<'not', {f}>) = 'not'+'('+ M(f)+')'
- M(<'or', {f1, f2}>) = M(f1)+' or '+M(f2)
- M(<'=', {f1, f2}>) = M(f1)+' = '+M(f2)
- M(<self::node() op C>) = 'self::node() op C', where op$\in${=, <, > , $\geq$, $\leq$, $\neq$}
- location (S, -) = '/', where S(1)[|S(1)|]= '/'
- location(S, a) = a+'::'+attribute(N, 'name'), where (isiElement(N) $\vee$ isiAttribute(N)) $\wedge$ N=S(1)[|S(1)|]
- location(S, a) = a+'::text()', where isiText(N) $\wedge$ N=S(1)[|S(1)|]
- location(S, a) = a+'::node()' where N= '/' $\wedge$ N=S(1)[|S(1)|]
- loops(lp) = loops({lp_1,…,lp_k}) = '('+'('+M(lp_1)+'/')*'+' | '…' | '+'('+M(lp_k)+'/')*'+')'

Figure 5.1: Functions mapping schema paths to a (regular) XPath expression

If we use the function M^r(<S, a, lp, ->), we get a regular XPath expression with loop patterns using the Kleene star $*$, which is not a standard XPath operator; if we use the function M(<S, a, lp, ->), we get a standard XPath expres-

sion, which conforms to the XPath specification [W3C XPath1.0 1999][W3C XPath2.0 2003], without loop patterns.

Proposition 5.1: Let L be a set of schema paths, let XP^r be the regular XPath expression mapped from L, and XP be the standard XPath expression mapped from L. The evaluation of XP returns the same node set as XP^r for any valid XML document.

Proof. According to the semantics of the XPath-Schema evaluator (see Figure 4.5 in Chapter), a loop occurs only when our XPath-XSchema evaluator processes the location steps, which contains the axis descendant or ancestor. Processing of all other recursive axes like following and preceding are boiled down to the two axes. All the descendant nodes (or ancestor nodes respectively) of the context node of the location step will be visited. The descendant (or ancestor respectively) nodes are logged into the corresponding schema path records whenever these nodes fulfill the constraints of the current location step and the following locations steps. The function M(<S, a, lp, ->) retrieves the nodes NS, which we divide into three different kinds of nodes: the first kind of nodes fulfills the constraints of the current and following location steps and the constraints of the loop patterns, i.e. the nodes retrieved by M^r(<S, a, lp, ->); the second kind of nodes fulfills the constraints of the current and following location steps, but does not fulfill the constraints of the loop patterns, i.e. these nodes, which are contained in the result of the mapped sub-queries of some of the other successfully detected schema paths of Q; the third kind of nodes fulfills the constraints of the current location step, but does not fulfill the constraints of following location steps, i.e. these nodes are not logged into any schema path and will be filtered when XP is evaluated on instance XML documents. According to the XPath language, the result of an XPath query does not contain any duplicates. Therefore, the total mapped XPath expression using either M or M^r returns the same node set for all possible XML documents.

Example 5.1: The schema paths in Figure 4.2 are mapped to the regular XPath expression Q^r = /bib/article/((reference/article/)*)reference/article[self::article/year][not(false())]/parent::reference, and the standard XPath expression Q' = /bib/article/descendant::reference/article[self::article/year][not(false())]/parent::reference . **Figure 5.2** describes a call graph of applying the functions M, which maps the schema paths in Figure 4.2 to the regular XPath expression Q^r, where the mapped location steps are in abbreviated syntax, e.g. bib rather than child::bib for readability and simplification.

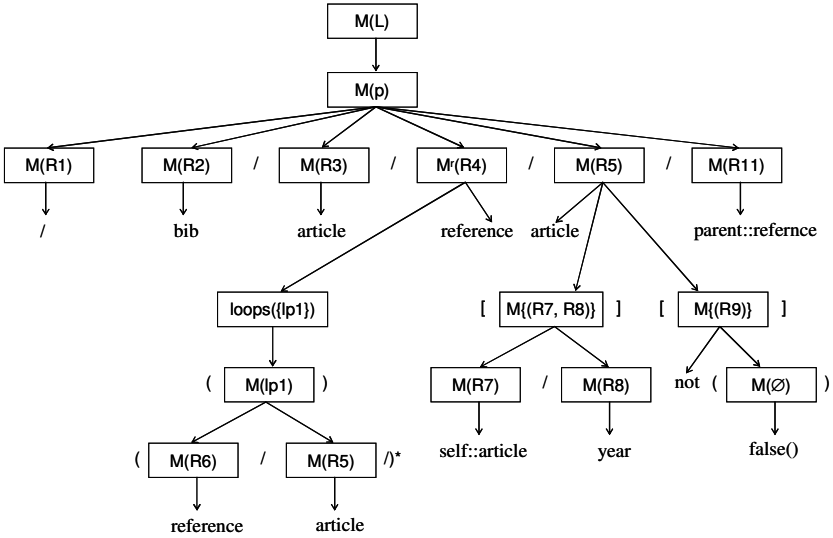

Figure 5.2: A call graph of applying the functions M to map the schema paths in Figure 4.2 to the regular XPath expression Q^r = /bib/article/((reference/article/)*)reference/article[self::article/year][not(false())]/parent::reference, where the mapped location steps are in abbreviated syntax, e.g. bib rather than child::bib for readability**.**

5.2 Optimizing mapped (regular) XPath Expressions

The mapped XPath query can be furthered optimized by eliminating redundant parts and reverse axes. For this optimization, we develop a set of rewriting rules. Different from the rewriting rules presented in [Olteanu et al. 2002], which eliminates reverse axes based-on the symmetry of the XPath axes, we eliminate reverse axes mainly based-on the symmetry of the schema paths. For example, [Olteanu et al. 2002], which offers a rule-based approach to eliminate reverse axis without considering schema information, eliminates the parent axis by generating a self axis. In comparison, our rules eliminate the parent axis without generating the self axis, as we have already considered the schema information when generating the schema paths. The reverse axes, which are re-

maining after the elimination of redundant location steps, can be eliminated using the rule-set in [Olteanu et al. 2002].

Let a be an axis, n be a node-test, e be a pattern and q be a qualifier. The rewriting rules, which eliminate reverse axes and redundant parts in the XPath expression mapped from a set of schema paths, are defined in Fig. 7.

- e/attribute::n1/parent::n2[q] ≡ e[q][attribute::n1]
- e/child::n1/parent::n2[q] ≡ e[q][child::n1]
- e1/child::n1/e2/parent::n3[q] ≡ e1[q][child::n1/e2], where e2 contains only the axes FS and PS
- e/attribute::n1[parent::n2[q]] ≡ e1[q]/attribute::n2
- e1/child::n1[parent::n2[q]] ≡ e1[q]/child::n1
- e1/child::n1/e2[parent::n3[q]] ≡ e1[q]/child::n1/e2, where e2 contains only the axes FS and PS
- e1[attribute::n1/parent::n2[q]] ≡ e1[q][attribute::n1]
- e1[child::n1/parent::n2[q]] ≡ e1[q][child::n1]
- e1[child::n1/e2/parent::n3[q]] ≡ e1[q][child::n1/e2], where e2 contains only the axes FS and PS
- e/self::n[q] ≡ e[q]
- e[q][q] ≡ e[q]
- e[q]/q ≡ e/q
- e[true()] ≡ e
- [not(false())] ≡ [true()]
- [q or true()] ≡ [true()]
- [q or false()] ≡ [q]
- [q and true()] ≡ [q]
- e*/parent::n ⊆ ancestor::n
- e*/child::n ⊆ descendant::n

Figure 5.3. Rules for optimizing the queries mapped from schema paths

Note that in Figure 5.3, e*/child::n is the pattern that is mapped by M^r[<S, a, lp, ->] and descendant::n is the pattern that is mapped by M[<S, a, lp, ->], when a='child'. As shown in Proposition 5.1, although descendant::n retrieves a superset of the node set retrieved by e*/child::n, the entire XPath query, which is rewritten from the mapped XPath expression, returns the same node set for all possible XML documents when using either descendant::n or e*/child::n.

Example 5.2: The regular XPath expression Q^r and the standard XPath expression Q' in Example 5.1, which are mapped from the schema paths in Figure 4.2 in Chapter 4, can be optimized further using the optimizing rules in Figure 5.3.

```
Qr = /bib/article/((reference/article/)*)reference/article[self::article/year]
        [not(false())]/parent::reference
```

$\Rightarrow$ is optimized to

```
Qr = /bib/article/((reference/article/)*)reference/article[year]
```

```
Q' = /bib/article/descendant::reference/article[self::article/year]
        [not(false())]/parent::reference
```

$\Rightarrow$ is optimized to

```
Q' = /bib/article/descendant::reference/article[year]
```

Chapter 6 XPath Satisfiability Tester

Our schema-based XPath satisfiability tester evaluates an XPath query on an XML Schema definition, and computes a set of *schema paths* to the possible nodes specified by the XPath query when it is evaluated by a common XPath evaluator on instance XML documents of the schema. If an XPath query does not conform to the structure, semantics, data type and occurrence constraints given in an XML Schema definition, the schema paths for the XPath query are computed to the empty set of schema paths, and thus the XPath query is unsatisfiable according to the schema.

If a non-empty set of schema paths is computed for an XPath query Q, we rewrite Q to Q' based on the schema paths of Q, which integrate the structure and semantics constraints in the XML Schema definition. Q' is equivalent to Q but contains more information than Q by substituting specific node tests for wildcards, by eliminating redundant parts, by eliminating reverse axes and by substituting non-recursive axes (e.g. *child*) for recursive axes (e.g. *descendant*) whenever possible, and thus can reveal more conflicting constraints. Our approach then checks whether or not the constraints in Q' are consistent with each other, and filters the queries with conflicting constraints.

If an XPath query is not detected as unsatisfiable, then the query may be satisfiable or may be unsatisfiable, since the satisfiability test for the XPath subset (see Definition 2.2) supported by our approach in the presence of the schemas (see Definition 2.1) supported by our approach is undecidable (see [Benedikt et al. 2005]).

Definition 6.1 (*Satisfiability of XPath queries*): An XPath query Q is satisfiable according to an XML Schema definition XSD, if there exists an XML document D that is valid according to XSD, and the evaluation of Q on D returns a non-empty result. Otherwise, Q is unsatisfiable according to XSD.

6.1 A framework of the satisfiability tester

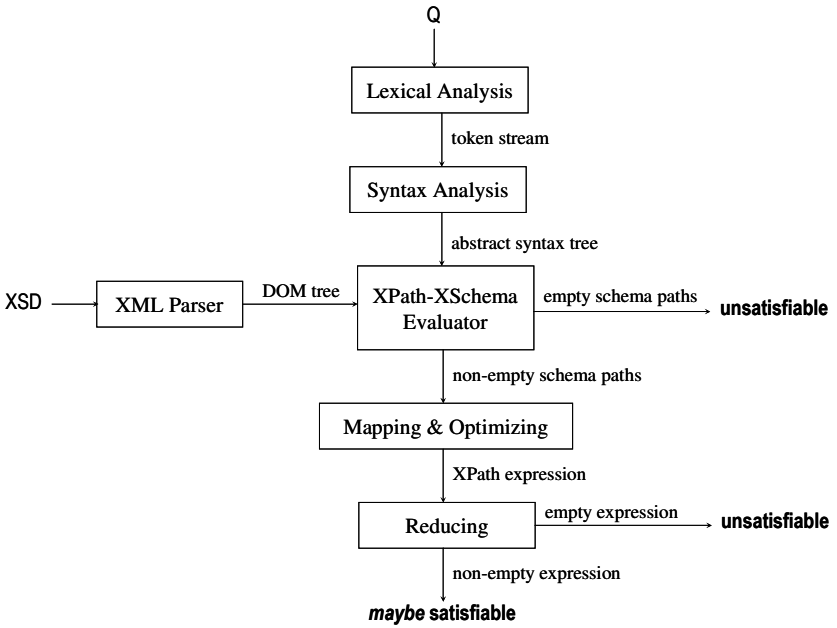

Figure 6.1: The framework of our satisfiability tester

Figure 6.1 describes the framework of our satisfiability tester. One of the inputs of the satisfiability tester is an XPath query Q. We first use standard compiler techniques to analyze the lexical structure and syntax of the XPath query. The *Lexical Analysis* transforms the given XPath expression to a stream of tokens, which is the input of the *Syntax Analysis*. Based on a given XPath grammar, the *Syntax Analysis* generates the abstract syntax tree of the XPath query Q. Another input of our satisfiability tester is an XML Schema definition XSD, which is analyzed and transformed to a DOM tree by an *XML Parser*. Our XPath-XSchema evaluator evaluates Q in the terms of its abstract syntax tree on the DOM tree of XSD based on the data model of XML Schema, and computes a set of schema paths. If the set of schema paths is the empty set, then the XPath query is unsatisfiable with respect to the schema. If the set of

schema paths is non-empty, then the schema paths are mapped to an XPath expression using the mapping functions in Figure 5.1 in Chapter 5, and the mapped XPath expression is optimized using the equivalence rules in Figure 5.3 in Chapter 5. Finally, we apply the rules of conflicting constraints given in Figure 6.22 in Section 6.3 in this chapter to reduce the mapped and optimized XPath expression. If the XPath expression is reduced to the empty expression $\perp$, then the given XPath query Q is unsatisfiable. Otherwise, Q is *maybe* satisfiable.

6.2 Filtering XPath queries not conforming to schema constraints

Proposition 6.1 (*Unsatisfiable XPath queries*): If the evaluation of an XPath query Q on a given XML Schema definition XSD by the XPath-XSchema evaluator generates the empty set of schema paths, then Q is unsatisfiable according to XSD.

Proof. The XPath-XSchema evaluator is constructed in such a way that the XPath-XSchema evaluator returns the empty set of schema paths, if the constraints given in Q and the constraints given in XSD exclude the constraints of the other, i.e. the navigational paths described by Q cannot be mapped to the corresponding navigational paths in XSD, or the values of attributes or elements given in Q do not conform to the type of values of elements or attributes specified in XSD, or the attributes and elements prohibited by XSD are specified by Q to appear in instance XML documents. Therefore, there does not exist a valid XML document according to XSD, where the application of Q returns a non-empty result. □

If an XPath query is computed to a non-empty set of schema paths by our XPath-XSchema evaluator on an XML Schema definition, the XPath query is only *maybe* satisfiable, since the satisfiability test of XPath queries formulated in the supported subset of XPath is undecidable [Benedikt et al. 2005] and our satisfiability tester is incomplete. Our XPath-XSchema evaluator checks whether or not each location step in an XPath query Q conforms to the constraints given in the XML Schema definition, but our XPath-XSchema evaluator does not check whether or not two or more location steps in Q contradict

each other. We present the approach to filtering XPath queries with contradictory constraints from the queries themselves in Section 6.3.

6.2.1 Performance analysis

We have implemented a prototype of our approach in order to demonstrate the optimization potential for avoiding the evaluation of unsatisfiable XPath queries. The performance analysis focuses on the detection of unsatisfiable XPath queries by our approach and the evaluation of these unsatisfiable queries by common XPath evaluators. We also study the overhead of evaluating satisfiable XPath queries by our approach, where we compare the time of evaluating the satisfiable queries by our approach with the time of evaluating unsatisfiable queries by our approach and with the time of evaluating these satisfiable queries by common XPath evaluators, in order to prove the usability of our approach.

The test system and used XML data sets are presented in Section 1.4 in Chapter 1.

6.2.1.1 XPath queries

We design three groups of unsatisfiable queries and two groups of satisfiable queries. The first group of unsatisfiable queries Q1-Q11 (see Table 6.1) is modified from some of the XPathMark benchmark queries (see [Franceschet 2005]) to contain erroneous semantics and structures according to the schema benchmark.xsd (see the corresponding appendix); the second group of unsatisfiable queries Q12-Q26 (see Table 6.3) does not conform to value-types or occurrence constraints specified in benchmark.xsd. We correct the errors of semantics and structures in the first group of unsatisfiable queries Q1-Q11 and get a group of satisfiable queries Q1'-Q11' (see Table 6.2); we modify the second group of unsatisfiable queries Q12-Q26 and obtain another group of satisfiable queries Q12'-Q26', which conform to the value-types and occurrence constraints. The third group of unsatisfiable queries Q27-Q34 (see Table 6.5) is designed to conform to constraints given in benchmark.dtd, but the not functions in these queries contain redundant parts. Although these queries are computed to a non-

empty set of schema paths, the set of schema paths are reduced to empty after eliminating redundant schema paths (see Theorem 4.3 in Chapter 4). Furthermore, the queries in these groups are also designed to contain as many constructs of the XPath language as possible in order to test how the different constructs of the XPath language influence the processing performance. We present the average results of ten evaluations of these queries.

Table 6.1: Queries with incorrect semantics or structures according to benchmark.xsd

Unsatisfiable queries		Reasons for unsatisfiability
Q1	/site/closed_auctions/closed_auction/ annotation/ description/parlist/text()	parlist has no text node.
Q2	/site/regions/∗/item[parent::america]	item has no parent america.
Q3	/site/open_auctions/open_auction [bidder//title]	bidder has no descendant title.
Q4	/site/people/person[age or gender]	person has neither child age nor child gender.
Q5	//person[age or gender]	person has neither child age nor child gender.
Q6	//keyword[italic][bold]	keyword has no child italic.
Q7	/descendant-or-self::persons	persons does not exist.
Q8	//open_auction[bidder//title]	bidder has no descendant title.
Q9	//∗/person[age or gender]	person has neither child age nor child gender.
Q10	//keyword/ancestor-or-self::mail[@title]	mail has no attribute title.
Q11	//keyword/ancestor::listitem/type	Listitem has no child type.

Table 6.2: Queries with correct semantics and structures according to benchmark.xsd

Satisfiable queries	
Q1'	/site/closed_auctions/closed_auction/annotation/descrition/parlist
Q2'	/site/regions/∗/item[parent::namerica]
Q3'	/site/open_auctions/open_auction[bidder]
Q4'	/site/people/person
Q5'	//person

Q6'	//keyword[bold]
Q7'	/descendant-or-self::person
Q8'	//open_auction[bidder]
Q9'	//*/person
Q10'	//keyword/ancestor-or-self::mail
Q11'	//keyword/ancestor::listitem

Table 6.3: Queries not conforming to data-types or occurrence constraints in benchmark.xsd

Unsatisfiable queries		Reasons for unsatisfiability
Q12	/site/people/person/race	race is a prohibited element, i.e. maxOccurs = 0.
Q13	//person/race	race is a prohibited element
Q14	/site[@owner= 'A']	owner is a prohibited attribute, i.e. use = 'prohibited'.
Q15	//site[@owner='A']	owner is a prohibited attribute
Q16	/site/people/person/watches /watch/@expression	expression is a prohibited attribute.
Q17	//watch/@expression	expression is a prohibited attribute.
Q18	//*/@expression	expression is a prohibited attribute.
Q19	/site/people/person[creditcard= '1234 4567 890a 1234']	creditcard is of pattern \d{4}\s*\d{4}\s*\d{4}\s*\d{4}.
Q20	//creditcard[self::node()= '1234 7890 1234']	creditcard is of pattern \d{4}\s*\d{4}\s*\d{4}\s*\d{4}.
Q21	//*[creditcard= '1234 456 7890 1234']	creditcard is of pattern \d{4}\s*\d{4}\s*\d{4}\s*\d{4}.
Q22	//happiness[self::node()=11]	happiness has maxInclusive = 10.
Q23	/site/people/person/profile [gender='M']	gender has enumeration male, female.
Q24	//gender[self::node()='f']	gender has enumeration male, female.
Q25	/site/catgraph/edge [self::node()='s']	edge has no value.
Q26	//edge[self::node=123.45]	edge has no value.

Table 6.4: Queries conforming to data-types and occurrence constraints in benchmark.xsd

Satisfiable queries	
Q12'	/site/people/person
Q13'	//person
Q14'	/site
Q15'	//site
Q16'	/site/people/person/watches/watch
Q17'	//watch
Q18'	//∗
Q19'	/site/people/person[creditcard= '1234 4567 8900 1234']
Q20'	//creditcard[self::node()='1234 7890 1234 7890']
Q21'	//∗[creditcard='1234 4567 7890 1234']
Q22'	//happiness[self::node()= 9]
Q23'	/site/people/person/profile[gender='male']
Q24'	//gender[self::node()='female']
Q25'	/site/catgraph/edge
Q26'	//edge

Table 6.5: Queries with not functions computed to false according to benchmark.xsd

Unsatisfiable queries		Reasons for unsatisfiability
Q27	/site/people/person/watches/watch [not(@open_auction)]	(@open_auction) is redundant
Q28	/∗/people/person[not(name)]	(name) is redundant
Q29	/site/∗/∗/annotation [not(author/@person)]	(author/@person) is redundant
Q30	/site[not(categories/category/description)]	(categories/category/description) is redundant
Q31	//watch[not(@open_auction)]	(@open_auction) is redundant
Q32	//person[not(name)]	(name) is redundant
Q33	/site[not(descendant::description)]	(descendant::description) is redundant
Q34	//profile[(age)][not(business)]	(business) is redundant

6.2.1.2 Filtering queries with incorrect semantics and structures

Figure 6.2 presents the time of evaluating the queries Q1-Q11 on benchmark.xsd by our XPath-XSchema evaluator, when returning the empty set of schema paths. Our evaluator can evaluate XPath queries Q1-Q4 without recursive axes very fast, less than 0.003 seconds; evaluating queries with recursive location steps, i.e. Q5-Q11, is time-consuming because all descendant nodes of the context node are visited when evaluating recursive location steps. Furthermore, the number of schema paths computed from each location step also significantly impacts the processing performance of our evaluator. For example, the queries Q5-Q8 have a comparable processing time while each of the queries Q5-Q7 contains only one recursive location step but the query Q8 contains two recursive location steps. Only one schema path is computed when evaluating the part //open_auction[bidder of Q8, and the location step bidder has less descendant nodes. The queries Q9-Q11 have a comparable evaluation time while Q9 contains only one recursive location step but each of Q10 and Q11 contains two recursive location steps. The recursive location step //∗ in Q9 selects all the instance element and text nodes, and thus a large number of schema paths are generated when evaluating the location step.

Figure 6.3 and Figure 6.4 present the time of evaluating these queries using the Saxon and the Qizx evaluator respectively when the empty result is returned. Figure 6.5 and Figure 6.6 present the speed-up factors achieved by our approach over the Saxon evaluator when evaluating Q1-Q11 and Q5-Q11 respectively. Figure 6.7 and Figure 6.8 present the speed-up factors achieved by our approach over the Qizx evaluator when evaluating Q1-Q11 and Q5-Q11 respectively. The experimental results show that our approach can check the satisfiability of XPath queries effectively. Our approach is about 1400 times (and 280 times respectively) faster on the average when evaluating the queries without recursive axis, and 35 times (and 8 times respectively) faster on the average when evaluating the queries with recursive axes at 12 Megabytes in comparison with the evaluation of the unsatisfiable queries when using the Saxon evaluator (and the Qizx evaluator respectively).

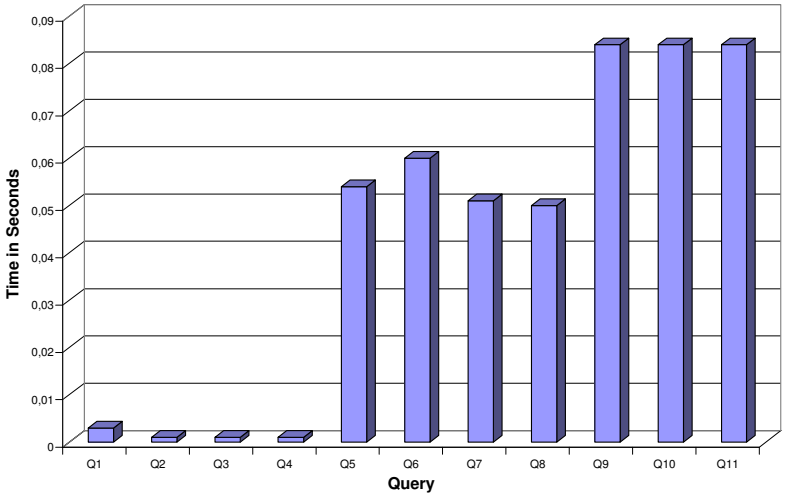

Figure 6.2: Filtering Q1-Q11 by our approach

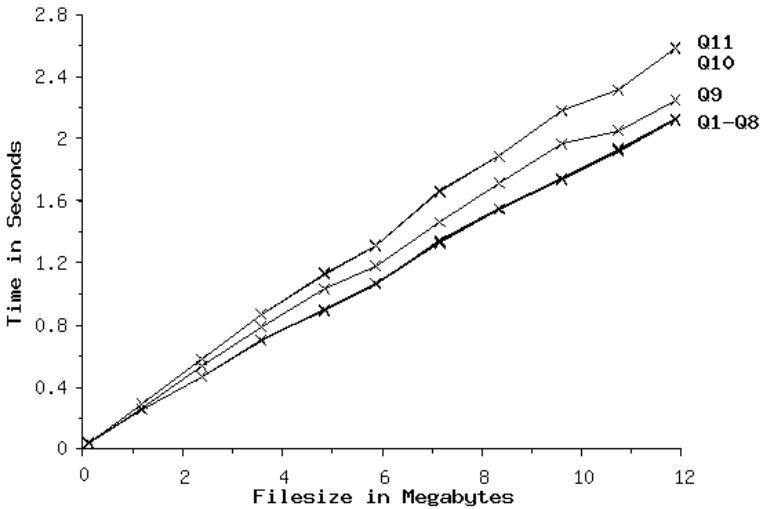

Figure 6.3: Evaluating Q1-Q11 using the Saxon evaluator

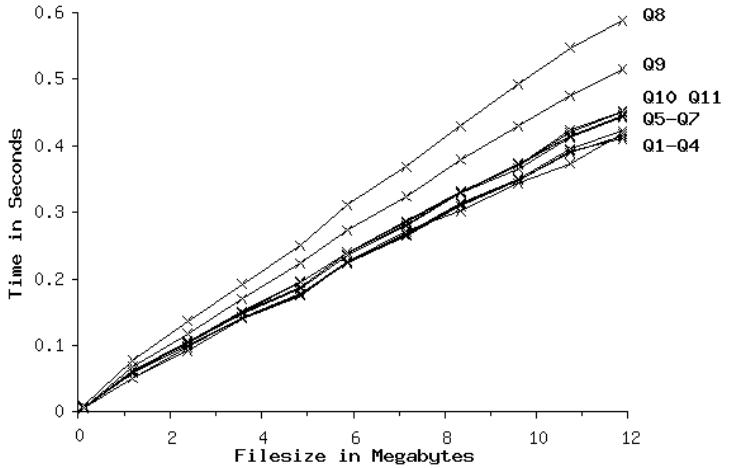

Figure 6.4: Evaluating Q1-Q11 using Qizx evaluator

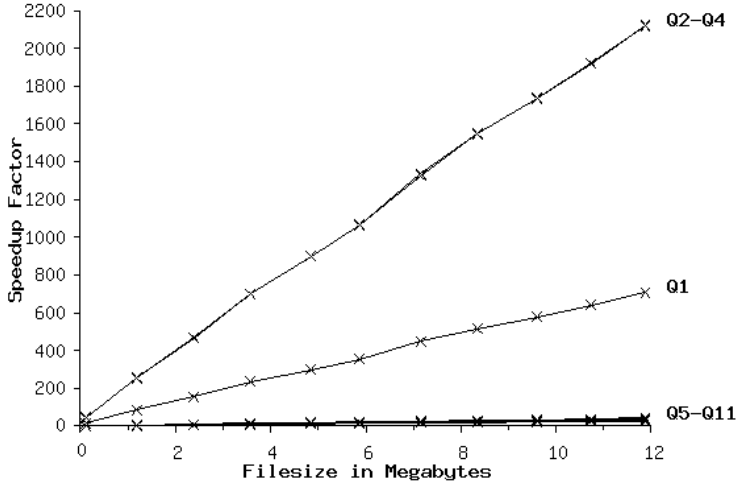

Figure 6.5: Speedup by our approach over Saxon when evaluating Q1-Q11

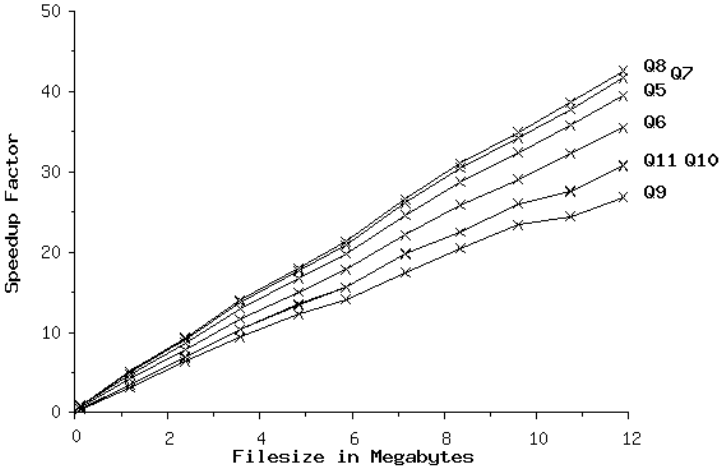

Figure 6.6: Speedup by our approach over Saxon when evaluating Q5-Q11

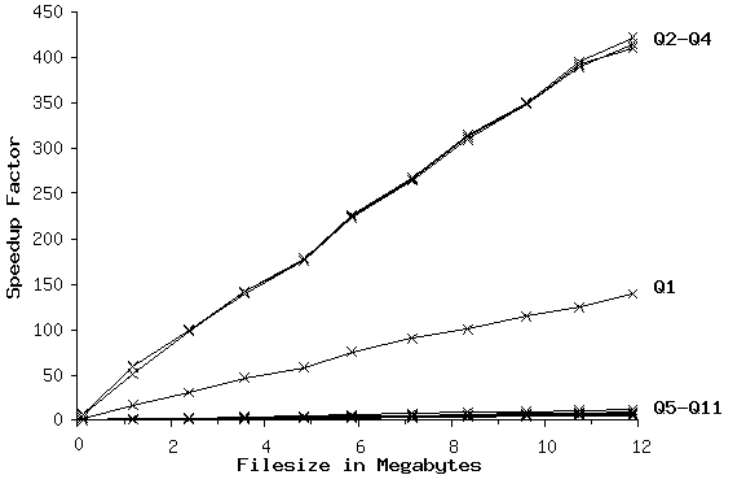

Figure 6.7: Speedup by our approach over Qizx when evaluating Q1-Q11

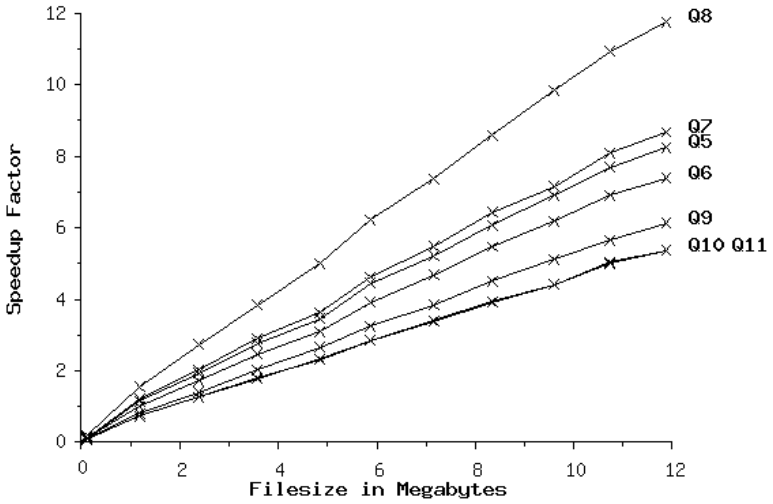

Figure 6.8: Speedup by our approach over Qizx when evaluating Q5-Q11

6.2.1.3 Filtering queries not conforming to data-type or occurrence constraints

Figure 6.9 presents the time of evaluating the XPath queries Q12-Q26 on benchmark.xsd by our XPath-XSchema evaluator, when it returns the empty set of schema paths. Figure 6.9 shows similar results for the influence of different XPath constructs on the processing performance. Figure 6.10 and Figure 6.11 present the time of evaluating these queries using the Saxon and the Qizx evaluator respectively, when the empty result is returned. Figure 6.12 and Figure 6.13 present the speed-up factors achieved by our approach over the Saxon evaluator when evaluating Q12-Q26 and the queries with recursive location steps among these queries respectively. Figure 6.14 and Figure 6.15 present the speed-up factors achieved by our approach over the Qizx evaluator when evaluating Q12-Q26 and the ones with recursive location steps of these queries respectively. Likewise, the experimental results show that our ap-

proach can check the satisfiability of XPath queries effectively. Our approach is about 1400 times (and 284 times respectively) faster on the average when evaluating the queries without recursive axis, and 34 times (and 10 times respectively) faster on the average when evaluating the queries with recursive axes than Saxon (and Qizx respectively) at 12 Megabytes in comparison with the evaluation of the unsatisfiable queries.

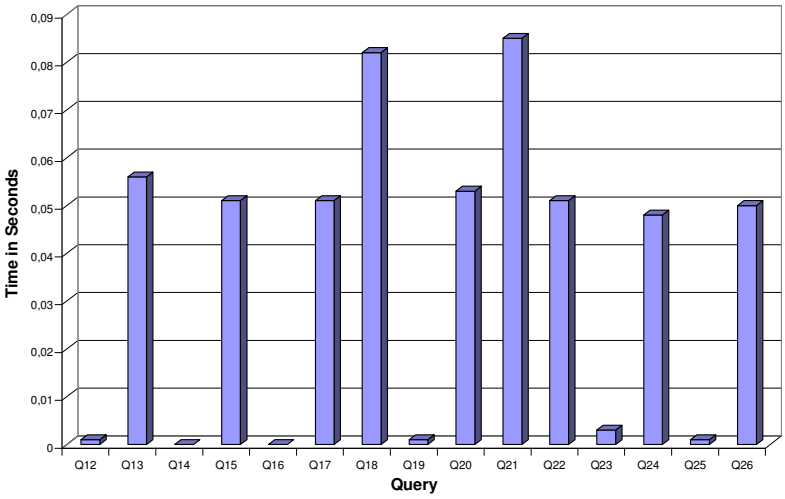

Figure 6.9: Filtering Q12-Q26 by our approach

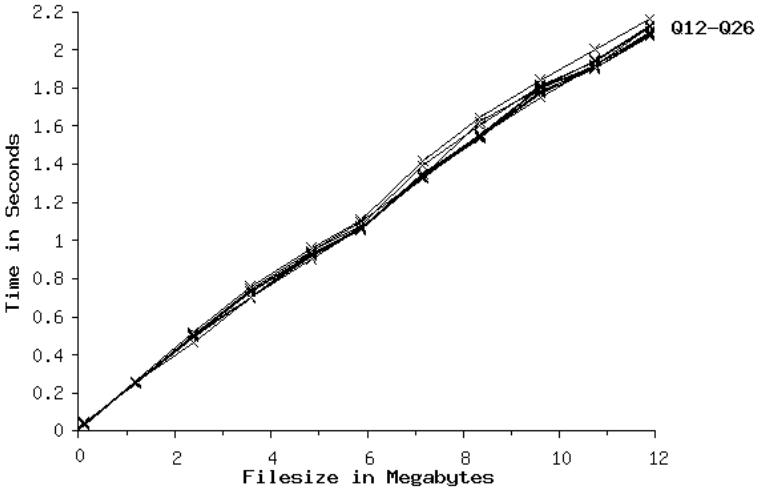

Figure 6.10: Evaluating Q12-Q26 using the Saxon evaluator

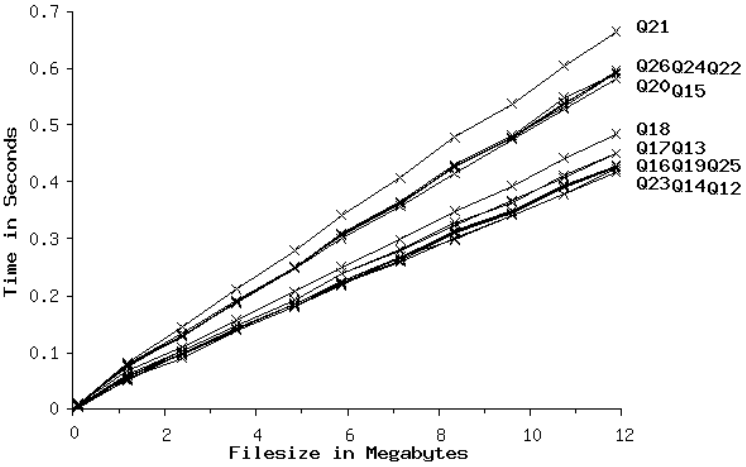

Figure 6.11: Evaluating Q12-Q26 using the Qizx evaluator

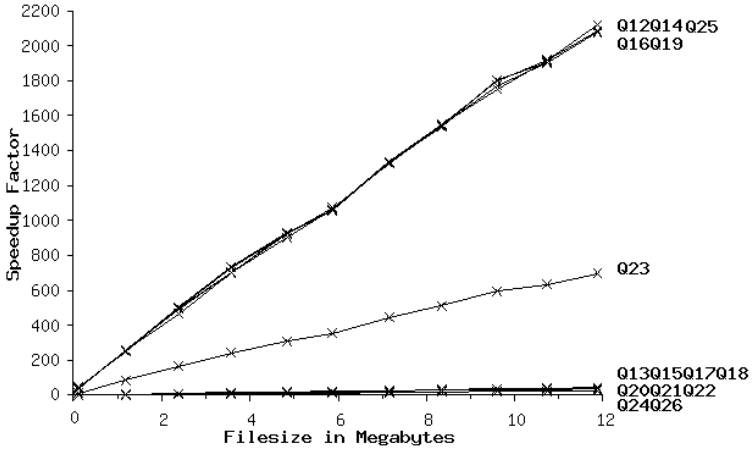

Figure 6.12: Speedup by our approach over Saxon when evaluating Q12-Q26

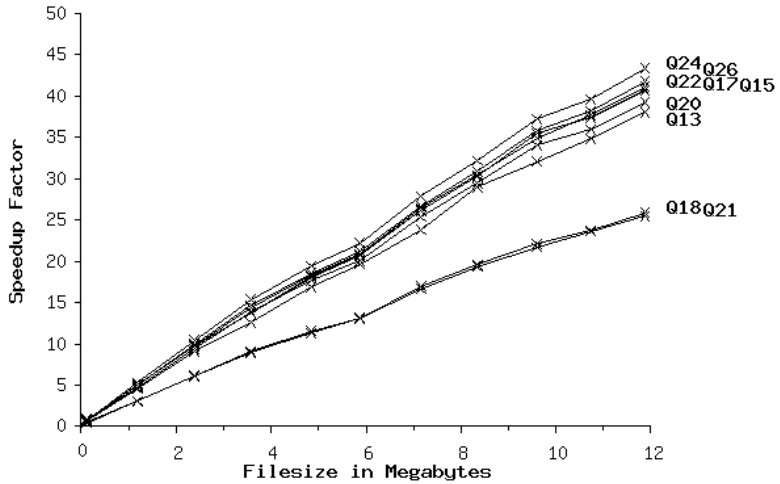

Figure 6.13: Speedup by our approach over Saxon when evaluating the queries with recursive location steps of Q12-Q26

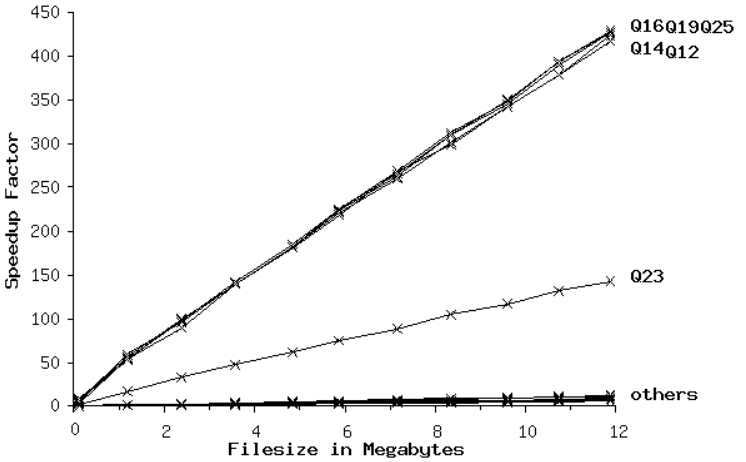

Figure 6.14: Speedup by our approach over Qizx when evaluating Q12-Q26

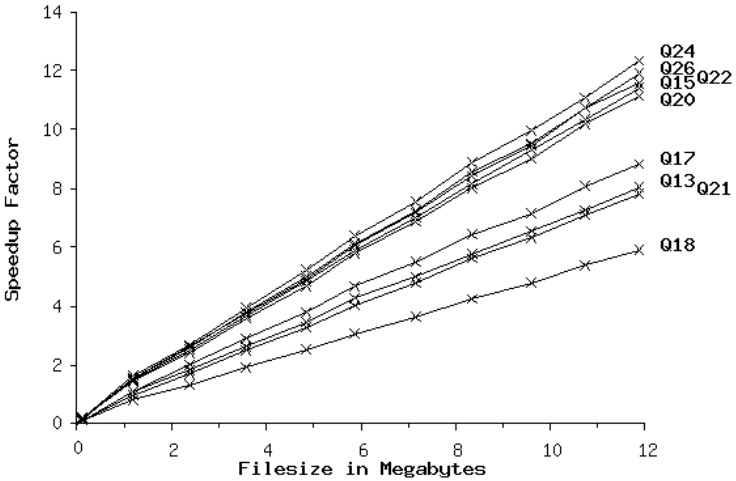

Figure 6.15: Speedup by our approach over Qizx when evaluating the queries with recursive location steps of Q12-Q26

6.2.1.4 Filtering queries with redundant schema paths

Figure 6.16 presents the time of filtering queries Q27-Q34 by our approach. The time consists of two parts: the first part is the time of computing schema paths; the second part is the time of reducing the schema paths to empty by eliminating redundant schema paths. The times used to check redundant schema paths are so little that it typically cannot be captured, i.e. are below 1 millisecond. Figure 6.16 once again shows similar results for the influence of different XPath constructs on the processing performance. Therefore, it is easy to see that our approach can filter the unsatisfiable queries, from which non-empty schema paths are computed, more efficiently and can achieve similar speedup factors as shown above.

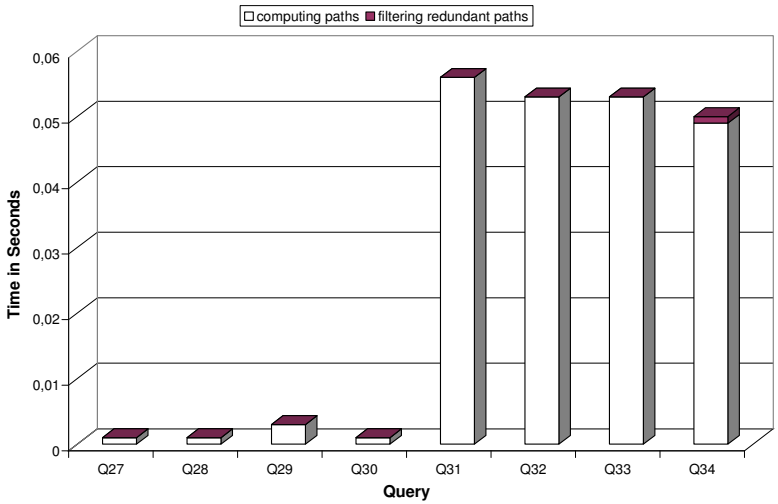

Figure 6.16: Filtering Q27-Q34 by our approach

6.2.1.5 Measuring the overhead of evaluating satisfiable queries

Figure 6.17 presents the time of evaluating the satisfiable XPath queries Q1'-Q11' on benchmark.xsd by our XPath-XSchema evaluator, when it returns an

non-empty set of schema paths, and the time of evaluating the unsatisfiable XPath queries Q1-Q11 by our evaluator for ease of comparison. Figure 6.18 presents the time of evaluating the satisfiable XPath queries Q12'-Q26' on benchmark.xsd by our XPath-XSchema evaluator, when it returns a non-empty set of schema paths, and the time of evaluating the unsatisfiable XPath queries Q12-Q26 by our evaluator. The two figures show that the overhead of evaluating satisfiable XPath queries is very close to the overhead of evaluating unsatisfiable XPath queries. Furthermore, the results also show that the overhead of evaluating satisfiable queries is generally less than the overhead of the corresponding unsatisfiable queries. Compared with the unsatisfiable queries, the corresponding satisfiable queries contain less location steps since the part leading to unsatisfiability in the unsatisfiable queries is removed. Figure 6.19 and Figure 6.20 present the ratio of the time used by our approach over the time used by Saxon and Qizx respectively for the evaluation of Q1'-Q11'. The results show that the ratio of the time of evaluating Q1'-Q4' without recursive location steps by our approach over the time used by Saxon is on average 0.06% (and over the time used by Qizx is 0.3% respectively); that the ratio of the time of evaluating Q5'-Q11' with recursive location steps by our approach over the time used by Saxon is on average 2.7% (and over the time used by Qizx is 12% respectively), when the size of data is 12 Megabytes.

However, when the size of XML documents is very small (<100 Kilobytes), the overhead of evaluating satisfiable XPath queries by our approach is quite high compared to the time used by the common XPath evaluators. When the size of XML data is 100 Kilobytes, the ratio of the time of evaluating the XPath queries Q1'-Q4' without recursive axes by our approach over Saxon (and Qizx) is on average 2% (and 18% respectively); the ratio of the time of evaluating XPath queries Q5'-Q11' with recursive axes by our approach over the time by Saxon (and by Qizx respectively) is on average 140% (and 700% respectively); In the worst case, the ratio of the time of evaluating Q1'-Q11' by our approach over the time used by Saxon is 30% when the size of XML data is 1 Megabytes, 7% when the size of data is 4 Megabytes, and 4% when the size of data is 6 Megabytes. In the worst case, the ratio of the time of evaluating Q1'-Q11' by our approach over the time used by Qizx is 100% when the size of XML data is 1 Megabytes, 33% when the size of data is 4 Megabytes, 22% when the size of data is 6.2 Megabytes. Although the ratio of the time of evaluating satisfiable XPath queries by our approach over common XPath evaluators is high for very small instance XML documents, the absolute time

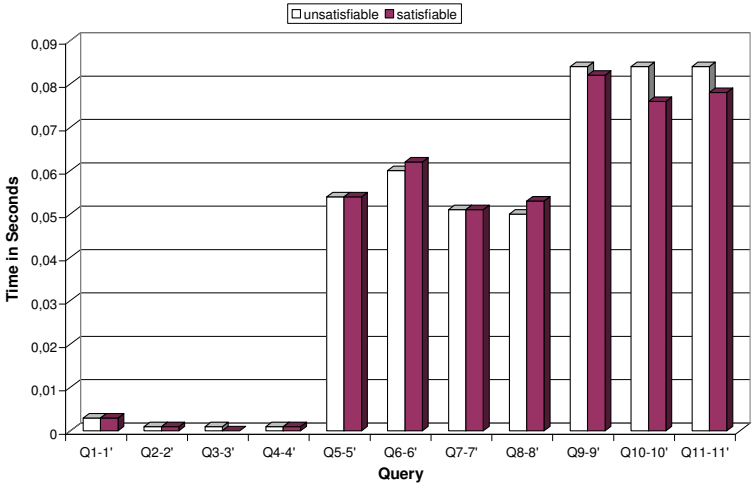

Figure 6.17: Evaluating satisfiable queries Q1'-Q11' and unsatisfiable queries Q1-Q11 by our evaluator

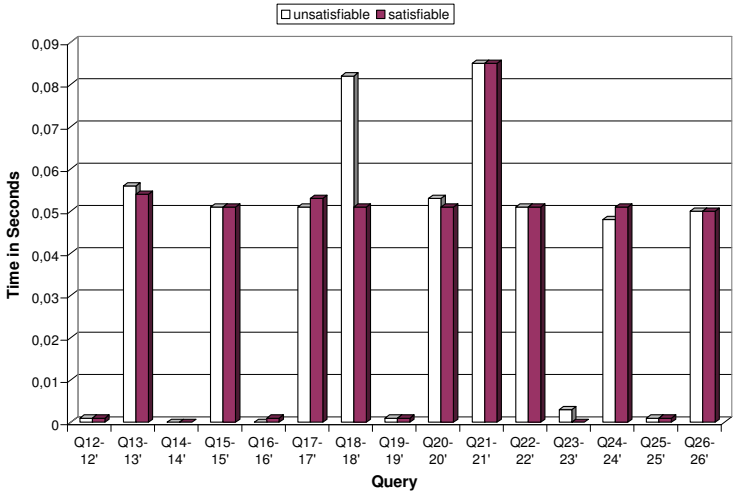

Figure 6.18: Evaluating satisfiable queries Q12'-Q26' and unsatisfiable queries Q12-Q26 by our evaluator

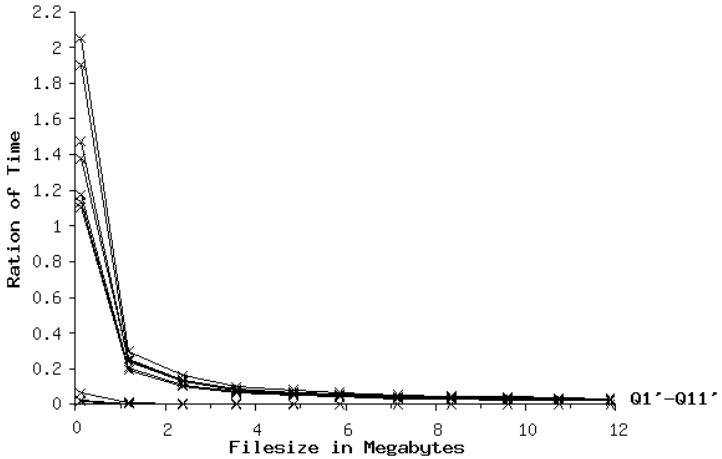

Figure 6.19: Ratio of time by our approach over Saxon when evaluating Q1'-Q11'

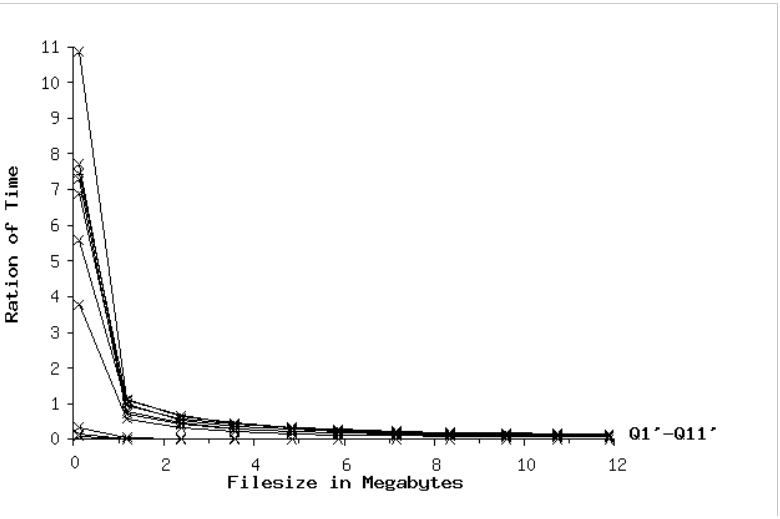

Figure 6.20: Ratio of time by our approach over Qizx when evaluating Q1'-Q11'

used by our approach is very small, i.e. 0.08 seconds in the worst case when evaluating Q1'-Q11'.

6.3 Filtering XPath queries with conflicting constraints

If an XPath query is computed to a non-empty set of schema paths by our XPath-XSchema evaluator on an XML Schema definition, the XPath query is only *maybe* satisfiable, since the satisfiability test of XPath queries formulated in the supported subset of XPath is undecidable [Benedikt et al. 2005] and our evaluator does not check whether or not two or more location steps in Q contradict each other. For example, if Q = a[not(b)]/∗ is computed to a non-empty set of schema paths on a given schema, then Q is not detected as unsatisfiable. However, if the non-empty set of schema paths is mapped to the XPath expression a[not(b)]/b, Q is unsatisfiable in that not(b) is contrary to b. We say that such queries contain *hidden* conflicting constraints. In this section, we present the approach to filtering queries with conflicting constraints.

If an XPath query is computed to a non-empty set of schema paths, we rewrite the query by mapping the schema paths to an XPath expression using the approach presented in Chapter 5. Then we apply the rules in Figure 6.22 to the XPath expression rewritten from the schema paths of the query to reduce the XPath expression. If an query is reduced to the empty expression ⊥, the query contains conflicting constraints and is unstatisfiable. The rewritten queries can make hidden conflicting constraints visible by excluding redundant parts and wildcards, and by eliminating reverse axes and recursive axes. Therefore, although the rule set in Figure 6.22 can be directly applied to given queries, the application of the rules to the rewritten queries can filter more unsatisfiable queries.

Example 6.1: Figure 6.21 demonstrates the process of filtering an XPath query //europe/∗[parent::∗[not(item)]] with hidden conflicting constraints according to the schema benchmark.xsd (see the corresponding appendix).

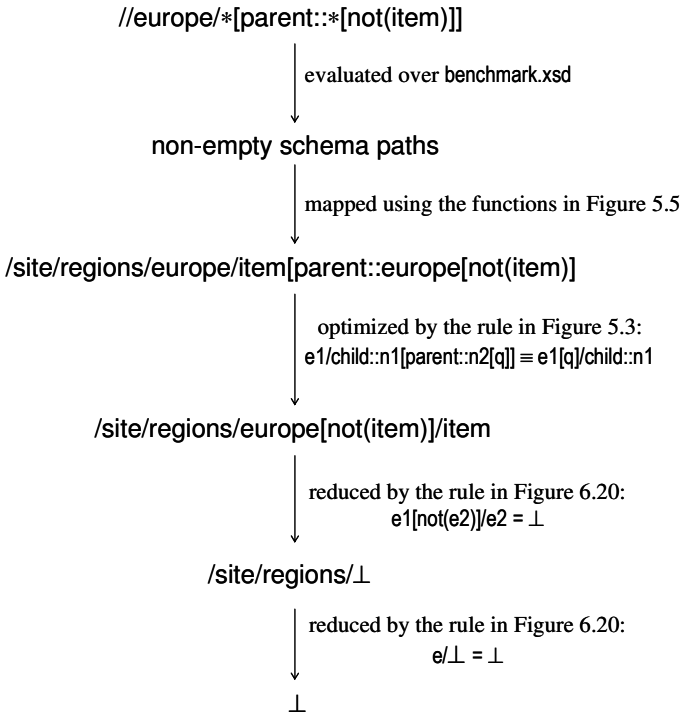

Figure 6.21: Example of filtering XPath queries with hidden conflicting constraints

Let e (e1, e2… respectively) be an XPath expression. If a sub-expression of an XPath query is reduced to the empty expression ⊥, the XPath query is reduced to ⊥.

- `⊥ | ⊥ = ⊥`
- `e/⊥ = ⊥`
- `⊥/e = ⊥`
- `e[⊥] = ⊥`
- `⊥[e] = ⊥`
- `⊥ and e = ⊥`
- `⊥ or ⊥ = ⊥`

- e1[not(e2)]/e2=⊥
- e1[not(e2)][e2]=⊥
- e1[not(e2)][e2/e3]=⊥
- e[@t=c1][@t=c2]=⊥ if c1≠c2
- e[@t<c1][@t=c2]=⊥ if c1≤c2
- e[@t<c1][@t>c2]=⊥ if c1≤c2
- e[@t<c1][@t≥c2]=⊥ if c1≤c2
- e[@t≤c1][@t=c2]=⊥ if c1<c2
- e[@t≤c1][@t>c2]=⊥ if c1≤c2
- e[@t≤c1][@t≥c2]=⊥ if c1<c2

Figure 6.22: Rules for filtering queries with conflicting constraints

6.3.1 Performance analysis

Section 6.1.1 presents a comprehensive performance analysis on detecting the unsatisfiable XPath queries that do not conform to the constraints in an XML Schema definition, i.e. the schema paths of the queries are computed to the empty set, and experimental results show that our approach can achieve a speedup up to several orders of magnitude over common XPath evaluators when detecting such unsatisfiable XPath queries. The performance analysis in this section focuses on the unsatisfiable XPath queries, which conform to the constraints imposed by an XML Schema definition, but contain hidden conflicting constraints. Our approach first computes the schema paths of the queries by evaluating the queries on a given XML Schema definition (see Chapter 4), then rewrites these queries based on the schema paths (see Chapter 5) in order to make hidden conflicting constraints visible, and finally applies the rules in Figure 6.22 on the rewritten queries to filter the queries with conflicting constraints. We study the detection of the unsatisfiable XPath queries by our approach and the evaluation of these unsatisfiable queries by common XPath evaluators.

The test system and used XML data sets are presented in Section 1.4 in Chapter 1.

6.3.1.1 XPath queries

We design a group of queries Q35-Q49 (see Table 6.6), which conform to the semantics, structure, data-type and occurrence constraints given in benchmark.xsd (see the corresponding appendix), but contain hidden conflicting constraints. Thus, the schema paths of these queries are computed to a non-empty set. Queries Q35'-Q49' in Table 6.6 are the rewriting of queries Q35-Q49 based on their schema paths. The rewritten queries disclose the hidden conflicting constraints. Furthermore, the queries Q35-Q49 are also designed to contain as many constructs of the XPath language as possible in order to test how the different constructs of the XPath language influence the processing performance. We present the average results of ten evaluations of these queries.

Table 6.6: Queries Q35-Q49 and rewritten queries Q35'-Q49' according to benchmark.xsd

Original and rewritten Queries	
Q35	/site/catgraph[not(edge)]/∗
Q35'	/site/catgraph[not(edge)]/edge
Q36	/site/catgraph[not(edge)]/self::node()/∗
Q36'	/site/catgraph[not(edge)]/edge
Q37	/site/regions/europe[(@area or ∗/name) and not(item)]
Q37'	/site/regions/europe[item/name][not(item)]
Q38	/site/regions/europe/∗[parent::∗[not(item)]]
Q38'	/site/regions/europe[not(item)]/item
Q39	//europe/∗[parent::∗[not(item)]]
Q39'	/site/regions/europe[not(item)]/item
Q40	/site/closed_auctions/closed_auction/buyer[@∗][not(@person)]
Q40'	/site/closed_auctions/closed_auction/buyer[@person][not(@person)]
Q41	/site/closed_auctions/closed_auction/buyer[@∗]/self::∗[not(@person)]
Q41'	/site/closed_auctions/closed_auction/buyer[@person][not(@person)]
Q42	//buyer[@∗][not(@person)]
Q42'	/site/closed_auctions/closed_auction/buyer[@person][not(@person)]
Q43	/site/people/person/profile[@∗>50][@income<10]
Q43'	/site/people/person/profile[@income>50][@income<10]
Q44	/site/people/person/profile[@∗>50]/interest/parent::∗[@income<10]
Q44'	/site/people/person/profile[@income>50][@income<10][interest]

Q45	/site/people/person/profile[@∗>50][@∗<99][@income<10]
Q45'	/site/people/person/profile[@income>50][@income<99][@income<10]
Q46	/site/people/person/profile[@∗>50][@∗<99][@∗>30][@income<10]
Q46'	/site/people/person/profile[@income>50][@income<99] [@income>30][@income<10]
Q47	/site/people/person/profile[@∗>50][@∗<99][@∗>30][@∗>40] [@income<10]
Q47'	/site/people/person/profile[@income>50][@income<99] [@income>30][@income>40][@income<10]
Q48	//profile[@∗>50][@income<10]
Q48'	/site/people/person/profile[@income>50][@income<10]
Q49	//profile[@∗>50][@∗<99][@∗>30][@income<10]
Q49'	/site/people/person/profile[@income>50][@income<99] [@income>30][@income<10]

6.3.1.2 Filtering queries with conflicting constraints

Figure 6.23 presents the time of filtering the unsatisfiable queries Q35-Q49 by our approach, consisting of three times: the time of computation of schema paths, i.e. evaluating Q35-Q49 on benchmark.xsd; the time of rewriting of Q35-Q49 based on the schema paths, i.e. mapping schema paths to an XPath expression and optimizing the XPath expression by the rules in Figure 5.3; the time of filtering XPath expressions with conflicting constraints by the rules in Figure 6.22. The overhead of filtering these unsatisfiable queries is mainly evaluating them on schema. Among 15 queries, Q39, Q42, Q48 and Q48 are queries with recursive axes, which we call recursive queries; others do not contain recursive axes, which we call non-recursive queries. Non-recursive queries can be evaluated very fast and is on average 7.2 faster than the recursive queries. The overhead of rewriting and rule application is quite low, and the time of rewriting and rule application for some queries even cannot be captured, i.e. are less than 1 millisecond. The time of rewriting and rule application is 32.6% of the time of computing the schema paths for the non-recursive queries, the time ratio is 2.6% for the recursive queries, and the time ratio is 11% for all the queries.

Figure 6.24 and Figure 6.25 present the speedup achieved by our approach over Saxon and Qizx when the evaluation of Q45-Q49 returns the empty result.

The results show that our approach can detect unsatisfiable queries efficiently. At a data size of 6 Megabytes, our approach is 199 times (and 39.6 times) faster on average when evaluating the non-recursive queries, and 35.6 times (and 10 times) faster on average when evaluating the recursive queries, than Saxon (and Qizx). At a data size of 12 Megabytes, our approach is 392 times (and 80 times) faster on average when evaluating the non-recursive queries, and 69.5 times (and 20 times) faster on average when evaluating the recursive queries, than Saxon (and Qizx).

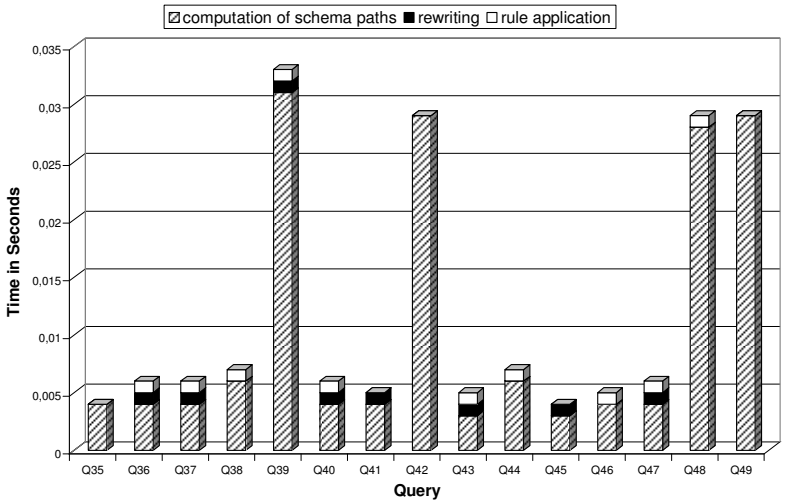

Figure 6.23: Filtering queries Q35-Q49 by our approach

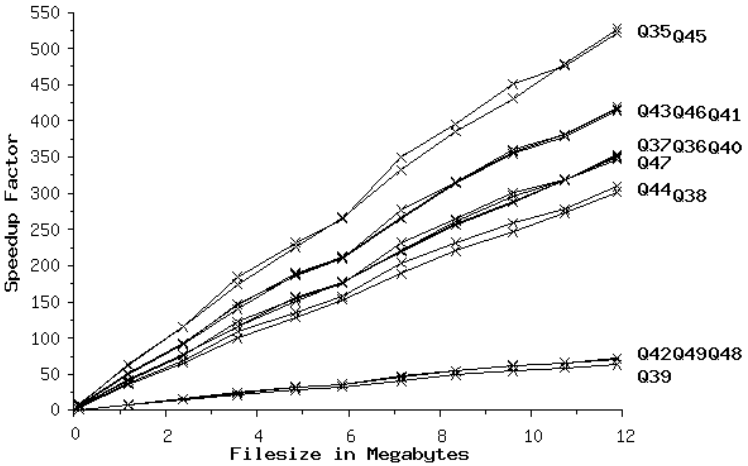

Figure 6.24: Speedup by our approach over Saxon when evaluating Q35-Q49

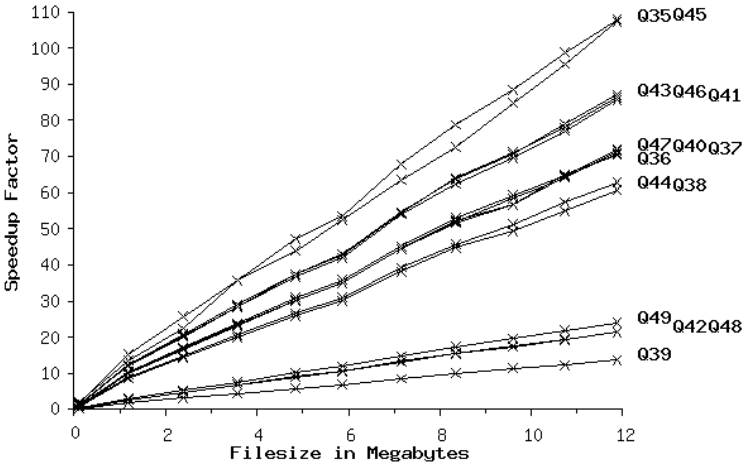

Figure 6.25: Speedup by our approach over Qizx when evaluating Q35-Q49

Chapter 7 XPath Containment under XML Schema Definitions

The schema paths of an XPath query are a re-representation of the query by integrating the constraints in the XML Schema definition. Therefore, we can study the containment of XPath queries under the constraints of schema in terms of the schema paths of XPath queries.

XPath containment in the presence of schemas has a high complexity. For example, the containment of XPath under schema constraints, even for XPath queries with only *child* axis and *predicates* (denoted as $XPath^{\{/,\ [\ ]\}}$) is coNP-complete [Neven and Schwentick 2003][Wood 2001]. The reason for the high complexity comes from the fact that inferring even some simple constraints from some schemas seems to be intractable [Wood 2001][Wood 2003]. Our approach can infer the constraints that impact the containment test from those schemas, where each element is declared at most once in a content model. However, our approach cannot infer some constraints from those schemas, where some elements are declared more than once in a content model. Inferring some simple constraints from such schemas are turned out to be intractable [Wood 2001].

Furthermore, our approach can support the reverse axes and the axes depending on the document order. The complexity and decidability of XPath containment, when these axes are allowed, are still unknown. Therefore, we present a fast but incomplete containment tester for XPath in the presence of XML Schema definitions. Given two queries Q1 and Q2, our tester returns that Q1 contains Q2, or Q1 may not contain Q2.

In order to check if the query Q1 contains the query Q2, we first use the rules in [Olteanu et al. 2002] to eliminate the reverse axes in these queries, then we evaluate them on the given XML Schema definition and get two sets of schema paths. Afterwards, we normalize the schema paths, and study XPath containment in terms of the normalized schema paths of XPath queries.

7.1 Problem studied

Let Q be an XPath query. We denote the result of evaluating Q over an XML document t and context node $k \in t$ by Q(t, k); we write |Q| for the length of Q and Q[i] for the *i*-th location step in Q.

Definition 7.1 (Containment of XPath queries): Let Q and Q' be two XPath queries. For any XML document t and any context node $k \in t$, if Q'(t, k) is a subset of Q(t, k), i.e. $Q(t, k) \supseteq Q'(t, k)$, then Q contains Q', denoted by $Q' \supseteq Q$. Otherwise, Q does not contain Q', denoted by $Q \not\supseteq Q'$.

According to the semantics of XPath, we have the following Theorem 7.1.

Theorem 7.1: Let Q and Q' be two XPath queries, and $Q'=Q'_1/Q'_2/Q'_3/.../Q'_{|Q|}$. $Q \supseteq Q'$, if $Q[1](t, k) \supseteq Q'_1(t, k) \wedge Q[2](t, k) \supseteq Q'_2(t, k) \wedge ... \wedge Q[|Q|](t, k) \supseteq Q'_{|Q|}(t, k)$.

According to Theorem 7.1, we get the conclusion that $Q \supseteq Q'$ if there is a corresponding part Q'_i in Q' for the location step Q[i] in Q such that $Q[i] \supseteq Q'_i$. Therefore, we have the following Proposition 7.1.

Proposition 7.1: Let Q and Q' be XPath expressions, and $Q'=Q'_1/Q'_2/Q'_3/.../Q'_{|Q|}$. $Q \supseteq Q'$, if $Q[1] \supseteq Q'_1 \wedge Q[2] \supseteq Q'_2 \wedge ... \wedge Q[|Q|] \supseteq Q'_{|Q|}$.

Example 1.1: Figure 7.1 gives several examples of containment of queries, and indicates the corresponding containment part Q'_i in Q' for the location step Q[i] in Q marked by the arrow from Q[i] to the last location step of Q'_i. Note that Q'_i represents one or more location steps of Q'. In this figure, XPath queries are represented as a graph, where vertices indicate the node test, the vertical single lines indicate the axis *child*, the horizontal single line indicate the *following-sibling* (FS) axis, double lines indicate the axis *descendant*, and dotted lines indicate predicates.

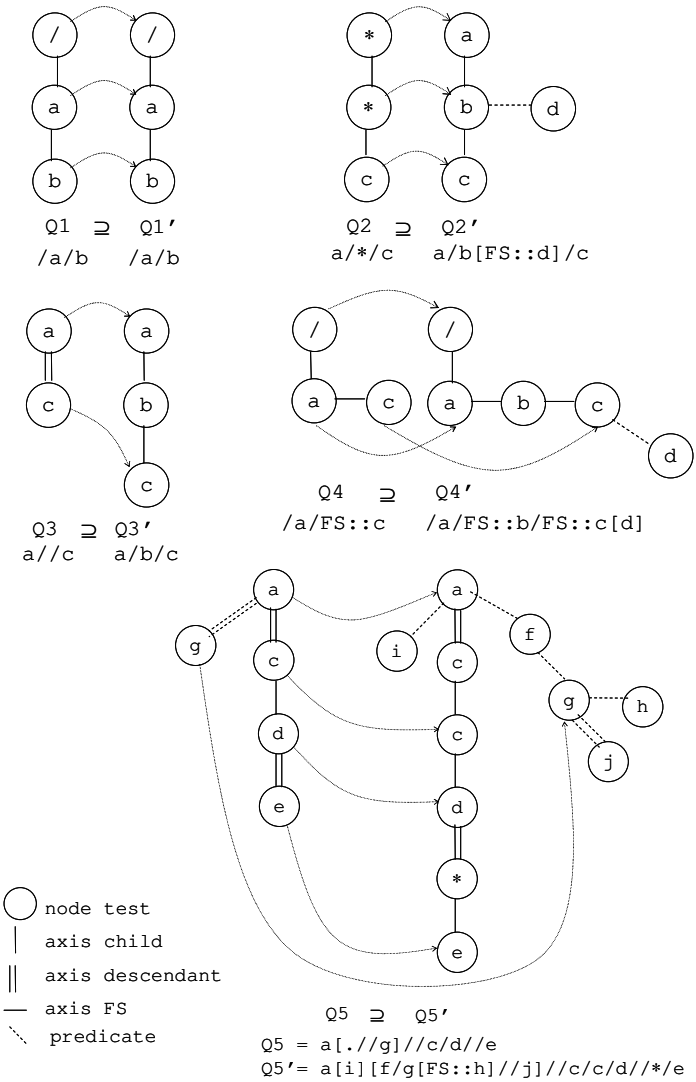

Figure 7.1: : Queries Qx and Qx', where Qx ⊇ Qx'. The corresponding containment part Qx'_i in Qx' for each location step Qx[i] in Qx is marked by an arrow from Qx[i] to the last location step of Qx'_i.

Therefore, the basic step of the approach to checking if an XPath query Q contains another XPath query Q' is to find the corresponding containment part of the Q' for each location step in Q. Furthermore, while a query Q may not contain a query Q' in general, Q may contain Q' when more constraints are given. For example, the query Q=/site/categories/category does not contain the query Q'=/site/categories/∗. But Q contains Q' with respect to the schema benchmark.xsd (see the corresponding appendix). The schema paths (see Chapter 4) of an XPath query are a re-representation of the XPath query by integrating constraints in the schema. Although schema paths can be easily mapped to an XPath expression, the mapped XPath expression cannot preserve all the constraints integrated in the schema paths. Therefore, we study the containment of XPath queries in terms of the schema paths of queries, i.e. we study how to find a corresponding schema path segment of the second schema path for each schema path record in the first schema path such that there is a certain containment relation between them.

7.2 Normalization of schema paths

Normalization of schema paths includes eliminating redundant schema paths, shifting predicate schema paths backwards and factoring out disjunction inside a schema path in order to test XPath containment with respect to schemas, which we present in Section 7.3.

7.2.1 Filtering redundant schema paths of predicates

Filtering redundant schema paths of predicates is already presented in Section 4.1.1.4. We here want to highlight the role of filtering redundant schema paths in XPath containment. As well as eliminating unnecessary parts of a query and filtering more unsatisfiable queries, the elimination of redundant schema paths can also help to detect more containment cases. For example, let Q1=b[c/d] and Q2=b, then Q1 $\not\supseteq$ Q2 in general. However, if a constraint specifies whenever b occurs c must occur and whenever c occurs d must occur, then the part [c/d] in Q1 is redundant and can be eliminated. After eliminating the redundant part,

Q1 is equal to b semantically, indicated as Q1≡b, and thus it is easy to see Q1⊇Q2 under this constraint. Therefore, filtering redundant schema paths is also important for the containment test.

7.2.2 Shifting schema paths of predicates backwards.

If Q1=a/b[c] and Q2=a[b/c]/b, then Q1 !⊇ Q2. However, if a schema imposes that b occurs at most once, then Q2≡a/b[c], and thus it is easy to see that Q1⊇Q2 under the schema. Furthermore, a location step with *self* axis also needs to be eliminated in order to simplify the containment test. For example, let Q3=a/b/self::b[c] and Q4=a/b[c]. While Q3≡Q4, deciding Q1⊇Q4 is more easier than deciding Q1⊇Q3. Therefore, we need to shift the schema paths of predicates backwards and eliminate self axes wherever possible.

In order to shift schema paths and eliminate the schema path records with self axis, we need to factor out the disjunction relation in a schema path. There are two kinds of disjunctions in a schema path. When a predicate is [q1 or q2], the two sets of schema paths computed from q1 and q2 respectively have the disjunction relation, and the schema paths in a schema path set of a predicate q has the disjunction relation.

In order to factor out disjunctions, we reshape schema path records by using the newly introduced binary operators · and | the operands of which are schema paths. These operators · and | are defined by the following rules:

- <XP, S, a, z, lp, {L1, L2, …, Ln}> := <XP, S, a, z, lp, (L1·L2· … ·Ln)>
- L={(< ‘or’, {L1, L2}>)} := (L1 | L2)
- L={(< ‘and’, {L1, L2}>)} := (L1 · L2)
- L={p1, p2, …, pn} := (p1 | p2 | … | pn),

We use the transformation rule, e.g. (a | b)·c = (a·c) | (b·c), to factor out the disjunction operator, and get a new schema path record, i.e.

<XP, S, a, z, lp, $(p_{1,1}\cdot \ldots \cdot p_{1,n_1}) \mid (p_{2,1}\cdot \ldots \cdot p_{2,n_2}) \mid \ldots \mid (p_{m,1}\cdot \ldots \cdot p_{m,n_m})$>

Then we further translate | outside a schema path record, and get

<XP, S, a, z, lp, $(p_{1,1}\cdot \ldots \cdot p_{1,n_1})$> | <XP, S, a, z, lp, $(p_{2,1}\cdot \ldots \cdot p_{2,n_2})$> | …| <XP, S, a, z, lp, $(p_{1,m}\cdot \ldots \cdot p_{m,n_m})$>

After each schema path record is transformed to this form, we get a schema path with the form e.g. p=(e1, (e2|e3)). We once again use the transformation rule (a | b)·c = (a·c) | (b·c) to factor out the disjunction inside a schema path, e.g. L={p}={(e1, (e2|e3))} is transformed to L={p1, p2}={(e1, e2), (e1, e3)}, after factoring out the disjunction inside p.

After factoring out inner disjunction relations, we get a new set of schema paths, where each schema path record consists of such record as e=<XP, S, a, z, lp, {{(p1)}, {(p2)},..., {(pm)}}>. In this record, each set in the sets of schema paths of predicates contains only one schema path. This record can be mapped to a location step with several predicates, each of which is an XPath expression without disjunction, e.g. a[b][c/d]...[x/y/z]. We call such schema paths *disjunction-free* schema paths. For readability and simplification of representation, we now represent a schema path record of disjunction-free schema paths as e=<XP, S, a, z, lp, L>=<XP, S, a, z, lp, {p1, p2,..., pm}>.

When a schema path record e has e.a='self', then the schema path record is computed from a location step with self axis. e.S and e.z are the same as the S field and z field in the record before it, and e.lp=∅, according to the semantics of the XPath-XSchema evaluator. Therefore, we have the following two theorems about eliminating the schema path records with a='self'.

Theorem 7.2: Let p be a disjunction-free schema path. Let p[i]=<XP_i, S_i, a_i, z_i, lp_i, L_i> and p[i+1]=<XP_{i+1}, S_{i+1}, a_{i+1}, z_{i+1}, lp_{i+1}, L_{i+1}>. If a_{i+1} = 'self', then p = (p[1], p[2], ..., p[i-1], p[i], p[i+2], p[i+3], ..., p[|p|]) and p[i]=<XP_i, S_i, a_i, Z_i, lp_i, $L_i \cup L_{i+1}$>.

Theorem 7.3: Let p be a disjunction-free schema path. Let p[i]=<XP_i, S_i, a_i, z_i, lp_i, L_i>. If $\exists q \in L_i$: q[1].a='self', then p[i]=<XP_i, S_i, a_i, z_i, lp_i, $(L_i - \{q\}) \cup q[1].L \cup \{(q[2], ..., q[|q|])\}$.

Theorem 7.2 integrates a schema path record e with e.a='self' into the schema path record before it. It is comparable to the elimination of location steps with self axis, e.g. Q1 = a[b]/self::a[c] = a[b][c]. Figure 7.2 demonstrates the use of Theorem 7.2 with Q1. Theorem 7.3 integrates a schema path record e with e.a='self' in a schema path of a predicate into the schema path record that is qualified by the schema path of the predicate. It is comparable to the elimination of location steps with self axis in a predicate, e.g. Q2 = a[b][self::a[c]/d] = a[b][c][d]. Figure 7.3 demonstrates the use of Theorem 7.3 with Q2.

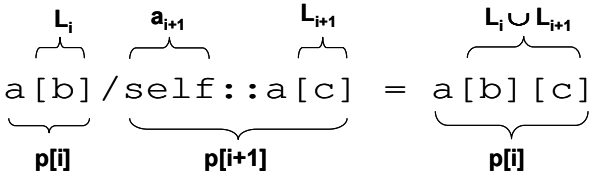

Figure 7.2: Theorem 7.2 is comparable to the elimination of location steps with *self* axis in an XPath query

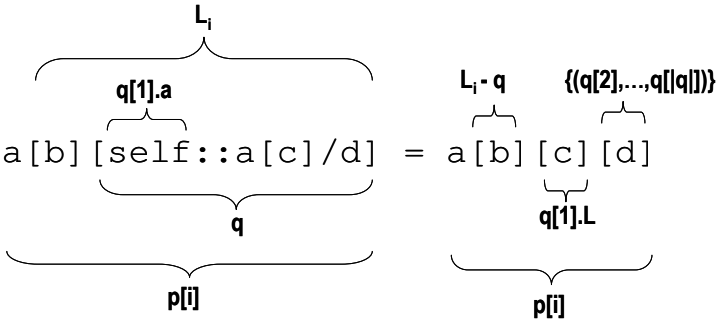

Figure 7.3: Theorem 7.3 is comparable to the elimination of location steps with *self* axis in predicates of an XPath query

In order to shift schema paths of predicates backwards, we need the concept of the *defining sequence* that is given in Definition 4.2 in Chapter 4, and some new concepts given in the following definitions.

Definition 7.2 (*nodes of declaring an element that can occur at most once)*: Let x be a defining sequence of an instance schema node N. If the element declared by N can occur *at most once* as a child of the element declared by x[1], then N is a schema node of declaring an element that can occur at most once.

Definition 7.3 (*nodes of declaring an element that can occur more than once)*: Let x be a defining sequence of an instance schema node N. If the element declared by N can occur *more than once* as a child of the element declared by x[1], then N is a schema node of declaring an element that can occur more than once.

Theorem 7.4: Let x be a defining sequence of an instance schema node N of type *iElement*. If ∀i∈{2, ..., |x|}: (x[i]@<maxOccurs>=⊥ ∨ attribute(x[i], maxOc-

curs)=1), then N is a node of declaring an element that can occur at most once, denoted by maxOccurs(element(N))=1. If $\exists i \in \{2, ..., |x|\}$: attribute(x[i], maxOccurs)>1, then N is a node of declaring an element that can occur more than once, denoted by maxOccurs(element(N))>1.

In Theorem 7.4, if N declares that an element can occur at most one time, then each node M (except the first one) in x does not carry the attribute maxOccurs, i.e. M@<maxOccures>=$\perp$ or carries the attribute maxOccurs with value 1, i.e. attribute(M, maxOccurs)=1; if N declares that an element can occur more than one time, then at least one node M (except the first one) in x carries the attribute maxOccurs with value greater than 1, i.e. attribute(M, maxOccurs)>1.

Theorem 7.5 below describes how to shift predicate schema paths backwards.

Theorem 7.5: Let p be a disjunction-free schema path. Let $p[i]=<XP_i, S_i, a_i, Z_i, lp_i, L_i>$ and $p[i+1]=<XP_{i+1}, S_{i+1}, a_{i+1}, Z_{i+1}, lp_{i+1}, L_{i+1}>$. If $\exists q \in L_i$: $S=S_{i+1} \wedge S=q[1].S \wedge$ maxOccurs(element(S(1)[|S(1)|]))=1, then $L_i = L_i - \{q\}$ and $L_{i+1} = L_{i+1} \cup q[1].L \cup \{(q[2], ..., q[|q|])\}$.

Theorem 7.5 is comparable to shift a predicate in an XPath expression backwards, e.g. Q = a[b[c]/d][e]/b[f] = a[e]/b[c][d][f] if b can occur at most one time. Figure 7.4 presents the application of Theorem 7.5 to Q.

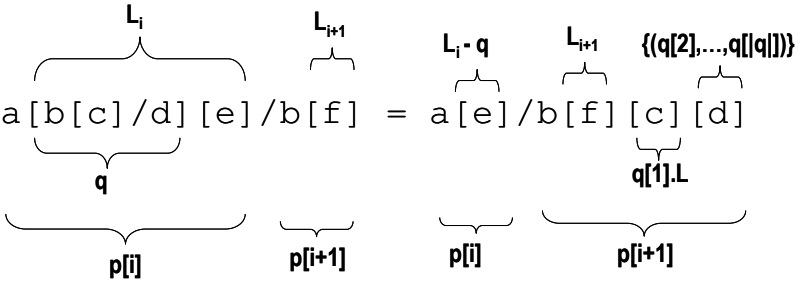

Figure 7.4: Theorem 7.5 is comparable to shift a predicate backwards in an XPath query a[b[c]/d][e]/b[f], where b can occur at most once

Complexity Analysis: In order to eliminate self axis and shift schema paths backwards, each schema path record is checked at most once. Therefore, eliminating self axis and shifting predicate schema paths backwards are linear

to the number of records of a schema path. However, factoring out disjunction is exponential in the number of records of a schema path.

7.2.3 Combining schema paths of predicates

Let Q1=a[b[c][d]] and Q2=a[b/c][b/d], then Q1 !⊇ Q2. However, if a schema defines that b can occur at most once, D2 can be rewritten to a[b[c][d]], and thus Q1⊇Q2 with respect to the schema. Therefore, we need to integrate predicates whenever possible in order to test the containment of queries.

Theorem 7.6: Let L be the disjunction-free paths of predicates; p, q ∈L and p≠q. If p[1].S=q[1].S ∧ maxOccurs(element(S(1)[|S(1)|]))=1, then p[1].L = p[1].L ∪ q[1].L ∪ {(q[2], q[3],..., q[|q|])} and L=L - {q}.

Theorem 7.6 is comparable to the combination of predicates in an XPath expression, e.g. Q = a[b[c]/d][b[e]/f] = a[b[c][e][f]/d] if b can occur at most once. Figure 7.5 demonstrates the application of Theorem 7.6 to Q.

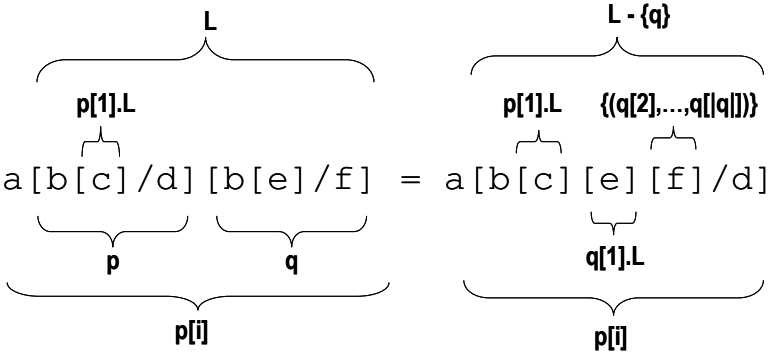

Figure 7.5: Theorem 7.6 is comparable to the combination of predicates in an XPath query a[b[c]/d][b[e]/f], where b can occur at most once

Complexity Analysis: In order to combine the schema paths of predicates, each schema path record is checked at most once. Therefore, the combination of schema paths of predicates is linear to the number of records of a schema path.

Now, we have a set of normalized disjunction-free schema paths, which are formally defined as follows.

Definition 4 (*normalized disjunction-free schema paths*): a normalized disjunction–free schema path is a sequence of schema path records e=<XP, S, a, z, lp, L> or e=<self::node() op C>, where L is a set of normalized disjunction-free schema paths, and e.a≠'self' ∧ ∀q∈L: q is not redundant ∧ q[|q|] is not redundant ∧ ∀q1, q2∈L: q1≠q2 ∧ (q1[1].S≠q2[1].S ∨ (q1[1].S=q2[1].S ∧ maxOccurs(element(S(1)[|S(1)|]))> 1 ∧ S=q1[1].S)).

7.3 Containment of schema paths

The schema paths of an XPath query are a re-representation of the query by integrating the constraints given in an XML Schema definition, and thus we can study the containment of XPath queries under the constraints of the schema in terms of the schema paths of XPath queries. Note that from now on all the schema paths mentioned are the normalized disjunction-free schema paths.

7.3.1 Re-presentation of schema paths.

Since some contents of a schema path record do not contribute the containment test and the current representation of schema paths is not efficient for discussion of the containment test either, we re-represent the normalized schema paths using a notation similar to the XPath language according to the following rules.

set of schema paths	L := L[1] | L[2] | … | L[|L|]
schema path	p := p[1]/p[2]/…/p[|p|]
schema paths of predicates	[L] = [L[1]·L[2]·… ·L[|L|]] = [L[1]] [L[2]] … [L[|L|]]
schema path record	<XP, S, z, a, -, -> := ^{a}N
	<XP, S, z, a, lp, -> := ^{a}N*
	<XP, S, z, a, -, L> := ^{a}N[L]

$$<XP, S, z, a, lp, L> := {}^{a}N^{*}[L],$$
$$\text{where } N = S(1)[|S(1)|].$$

The two fields XP and z in a schema path record do not contribute to the containment test, and are left out. N is the most important schema node for the containment test of schema paths, so we present only this node here rather than the node sequences for simplicity. However, when we say that two schema nodes are equal, we also imply that the defining sequences of two nodes are equal. The Kleene star $*$ indicates a schema path record with non-empty loop schema paths, and this means that N is both the initial and end node of the loop paths. Note that we do not present the content of loop schema paths here, as the specific content of loop schema paths is not important in checking containment of schema paths, which will been seen later. Let e be a schema path record, we also write e.N for N in e; e.a for a in e; e.L for L in e; and e.lp for lp in e. Furthermore, when a='child', a can be left out, e.g. ${}^{child}N=N$.

According to the semantics of XPath, we have the following theorems on the schema paths p, p1, p2.

Theorem 7.7: p1/p2 = p1[p2]/p2.

Theorem 7.8:

$$[p] = [p[1]/p[2]/\ldots/p[|p|]]$$
$$= [p[1][p[2]/\ldots/p[|p|]]$$
$$= [(<XP, S, z, a, lp, L \cup \{(p[2],\ldots, p[|p|])\}>)],$$
$$\text{where } p[1]= <XP, S, z, a, lp, L>.$$

7.3.2 Properties of schema paths

We now look at the basic properties of schema paths related with the containment of schema paths.

Let Q1=a[b] and Q2=a[c], then Q1!$\supseteq$Q2. If a constraint specifies that b must occur when c occurs, then Q1$\supseteq$Q2 with respect to the constraint. In this case, we say b is implied by c. Such occurrence implications can be inferred from the schema paths.

Definition 7.5 (*occurrence implications*): Let M and N be two instance nodes of an XML Schema definition. If the node declared by M occurs in an XML document, then the node declared by N must occur in the XML document. We say that M implies N, denoted as M=>N.

We identify the following occurrence implications among instance nodes in an XML Schema definition.

Theorem 7.9: Let M and N be two instance nodes of an XML Schema definition, and let the defining sequences of M and N be x and y respectively. M=>N, if

- N = M, (which we call the implication of *equivalence*).
- N is the instance parent of M, i.e. x[1]=N, (which we call the implication of *child-parent*),
- N is an instance child of M, and N is an *unconditional* node (see Definition 4.3 in Chapter 4), (which we call the implication of *parent-child*)
- N is an instance sibling of M, and $\forall i \in \{1, ..., k\}$: x[k]=y[k] $\wedge$ x[k+1]$\neq$y[k+1] $\wedge$ x[k]$\neq$<choice...> $\wedge$ $\forall j \in \{k+1, ..., |y|\}$: (y[j]@<minOccurs>=$\perp$ $\vee$ attribute(y[j], minOccurs)$\geq$1) $\wedge$ x[j]$\neq$<choice...>), (which we call the implication of *sibling*).

It is easy to see if a node N is implied by its parent, then N must be implied by any sibling of it, but the reverse is not true. Since the reverse axes do not exist in schema paths, the implication of *child-parent* is not important. Therefore, when studying the predicate containment of schema paths, the implications of *equivalence* and *sibling* are the most important occurrence implications.

Theorem 7.10: A schema path record with the loop schema paths, i.e. N* (or N*[L]) can be transformed to a union of an infinite set of schema paths without loop schema paths, the first schema path record of which is N and the last schema path record of which is N (or N[L]), and between two Ns is a certain combination of the loop schema paths.

Theorem 7.11: A schema path with loop schema paths can be transformed to a union of an infinite set of disjunction-free schema paths without loop schema paths.

Definition 7.6 (*size of schema paths*): The number of the schema path records in a schema path p is called the size of p, denoted by ||p||.

Definition 7.7: (*Depth of a schema path record and of a schema path*): If a record e of a schema path p is the *i*-th record related to the context of p, then depth of e according to p is i, denoted by |e|. If no other records in p have greater depth than e, then |e| is also the depth of p, denoted by |p|.

Let p=a/b[c[d]/e]. ||p||=5. |a|=1, |b|=2 and |p|=2 related to the context of p. |c|=1, |d|=2, |e|=2 and |c[d]/e|=2 related to the context of the predicate c[d]/e. The concept of the depths is needed for analyzing the run-time complexity of the containment test.

7.3.3 Containment test

The schema paths consist of several features: child axis (/), following-sibling axis (FS), attribute axis (A), the loop (*) and predicates [ ]. We indicate the schema paths with these features as schema path$^{\{/,\ FS,\ A,\ *,\ [\ ]\}}$, and the schema paths consisting of a subset S of these features as schema path^{S}, e.g. schema path$^{\{/,\ *,\ [\ ]\}}$.

Unless stated explicitly, we typically use C and D for two sets of schema paths, c and d for two schema paths, N and M for two schema nodes, and F and H for two sets of predicate schema paths.

We describe basic concepts and properties of containment of schema paths in Section 7.3.3.1. Afterwards, we study the containment test for schema path$^{\{/,\ *,\ [\ ]\}}$ in Section 7.3.3.2, for XPath$^{\{/,\ FS,\ *,\ [\ ]\}}$in Section 7.3.3.3. In Section 7.3.3.4, we discuss the containment test when the attribute axis and comparison predicates are allowed.

7.3.3.1 Concepts and properties

We assume that a schema path can be evaluated over XML documents in the same way as an XPath expression. Therefore, we denote by C(t, k) the result of applying a schema path C over an XML document t and a context node k∈t.

Definition 7.8 (***containment of schema paths***): Let C and D be two schema paths. For any XML document t and any context node $k \in t$, if D(t, k) is a subset of C(t, k), i.e. $C(t, k) \supseteq D(t, k)$, then C contains D, denoted by $C \supseteq D$. Otherwise C does not contain D, denoted by $C \not\supseteq D$.

Definition 7.9 (***predicate containment of schema paths***): Let C and D be two schema paths. We call C predicate-contains D, denoted by $[C] \supseteq [D]$, if for any XML document t and any context node $k \in t$, that [D] is evaluated to true implicates that [C] is evaluated to true. Otherwise, C does not predicate-contain D, denoted by $[C] \not\supseteq [D]$.

Theorem 7.12 (*predicate containment of schema paths*): Let C and D be two schema paths. $[C] \supseteq [D]$, iff for any XML document t and any context node $k \in t$, D(t, k) is evaluated to a non-empty set of nodes implicates that C(t, k) is evaluated to a non-empty set of nodes, i.e. $D(t, k) \neq \varnothing \Rightarrow C(t, k) \neq \varnothing$.

For example, let C=a/b and D=a[b][c]. The application of C selects child nodes b of nodes a; the application of D selects nodes a that must have child nodes b and c. This means if D selects a non-empty set of nodes, then C must also select a non-empty set of nodes. Therefore, $[C] \supseteq [D]$. Let C=a and D=b. If b implies a, then $[C] \supseteq [D]$.

Theorem 7.13: Let C and D be two sets of schema paths. $C \supseteq D$, iff $\forall d \in D: \exists c \in C: c \supseteq d$.

Theorem 7.14: If $C1 \supseteq D1$, $C2 \supseteq D2$, and C2 and D2 are two sets of schema paths from two relative XPath expressions, then $C1/C2 \supseteq D1/D2$.

Proposition 7.2: If $C \supseteq D$, then $[C] \supseteq [D]$.

Proof. If $C \supseteq D$, then the application of D returns a non-empty result implies that the application of C returns a non-empty result according to Definition 7.8. Therefore, $[C] \supseteq [D]$ according to Theorem 7.12.

Proposition 7.3: c=N[F] and d=M[H]. $c \supseteq d$, iff $N=M \wedge [F] \supseteq [H]$.

Proof. If N=M, then N and M select the same node set, i.e. for an arbitrary XML document t and an arbitrary context node $k \in t$, N(t,k)=M(t,k). If $[F] \supseteq [H]$, then the

nodes selected by [H] are also selected by [F], and thus N[F](t,k)⊇M[H](t,k). Therefore, if (N=M ∧ [F]⊇[H]) then c⊇d.

If N ≠ M then N and M select different nodes; if [F]!⊇[H], then the nodes selected by [H] may not be selected by [F]. In the two cases, some nodes selected by d may not be selected by c, and thus c!⊇d. □

Proposition 7.4: Let c, d1 and d2 be schema paths. [c] ⊇ ([d1] | [d2]), iff [c]⊇[d1] ∧ [c]⊇[d2].

Proof. If [c]⊇[d1] and [c]⊇[d2], then that either of [d1] and [d2] is true implies [c]=true. ([d1] | [d2])=true indicates that [d1]=true or [d2]=true. Therefore, if [c]⊇[d1] and [c]⊇[d2], then [c]⊇([d1] | [d2]).

Assume, [c]⊇([d1] | [d2]), but [c]!⊇[d1]. We always can construct an XML document such that [d1]=true, [d2]=false and [c]=false. In this case, ([d1] | [d2])=true but [c]=false. Therefore, [c]!⊇([d1] | [d2]), which is contrary to the assumption. □

Proposition 7.5: Let c, d1 and d2 be schema paths. [c]⊇[d1][d2], iff [c]⊇[d1] ∨ [c]⊇[d2].

Proof. [c]⊇[d1] or [c]⊇[d2] implies that we can get the conclusion of [c]=true, only when [d1]=true and [d2]=true. [d1][d2]=true indicates that [d1]=true and [d2]=true. Therefore, if [c]⊇[d1] or [c]⊇[d2], then [c]⊇[d1][d2].

Assume [c]⊇[d1][d2], but [c]!⊇[d1] and [c]!⊇[d2]. We always can construct an XML document, where [d1]=true and [d2]=true, but [c]=false. Thus, [d1][d2]=true, but [c]=false. Therefore, [c]!⊇[d1][d2], which is contrary to the assumption. □

7.3.3.2 Schema path$^{\{/,\ *,\ [\ ]\}}$

We first look at the simple cases of containment, e.g. N⊇N[F]. Proposition 7.6 describes what is the corresponding schema path d for the schema path c=N or c=N[F] such that c⊇d.

Proposition 7.6: Let c and d be two schema paths in schema path$^{\{/,\ *,\ [\ ]\}}$.

(1) Let c=N. c⊇d, iff (d=M ∨ d=M[H]) ∧ N=M.
(2) Let c=N[F]. c⊇d, iff d=M[H] ∧ N=M ∧ [F]⊇[H].

Proof. Let c=N. M[H] selects a subset of M. Since N=M, M[H] selects a subset of N, and thus c⊇d. c selects only child nodes of the context nodes. If |d|>1, d selects the child or descendant nodes of the context nodes, and thus c and d may

select different nodes. Therefore, $c \not\supseteq d$ if $|d|>1$. If $d.lp \neq \varnothing$, e.g. $d=M^*$, then d can be transformed to a union of an infinite set of schema paths with the length from 1 to infinite according to Theorem 7.10, and thus $c \not\supseteq d$.

Assume $c \supseteq d$. If $N \neq M$, then N and M select different nodes; if $|d|>1$ or $d.lp \neq \varnothing$, then c and d select different nodes either, and thus $c \not\supseteq d$, which is contrary to the assumption.

The second part (2) can be analogously proved. □

Let $Q1=a$ and $Q2=a[b]/c/d$. If Q2 returns a non-empty result, then the nodes a must exist, and thus Q1 must return a non-empty result. Therefore, $[Q1] \supseteq [Q2]$. Proposition 7.7 describes how to test the predicate-containment of two schema paths.

Proposition 7.7: Let c and d be two schema paths in schema path$^{\{/,\ ^*,\ [\ ]\}}$.

(1) Let $c=N$. $[c] \supseteq [d]$, iff $d[1].N \Rightarrow N$.

(2) Let $c=N[F]$. $[c] \supseteq [d]$, if $d[1].N=N \wedge [F] \supseteq [d[1].L][d[2]/\ldots/d[|d|]] \wedge d[1].lp=\varnothing$.

Proof. Let $c=N$. If d returns a non-empty set of nodes, then $d[1].N$ must select a non-empty result. Since $d[1].N \Rightarrow N$, c also returns a non-empty set of nodes according to Definition 7.5, and thus $[c] \supseteq [d]$. If N is not implied by $d[1].N$, then we always can construct an XML document such that $d[1].N$ exists but N does not exist. Therefore, $[c] \not\supseteq [d]$.

Let $c=N[F]$. Let $d2=d[2]/\ldots/d[|d|]$ and $[d]=[d[1][d2]]=[M[H][d2]]$, where $M=d[1].N$ and $H=d[1].L$. Since $[F] \supseteq [H][d2]$, then $M[F] \supseteq M[H][d2]$ according to Proposition 7.3, and thus $[M[F]] \supseteq [M[H][d2]]$ according to Proposition 7.2. Since $M=N$, then $[N[F]] \supseteq [M[H][d2]]$. □

If $M \Rightarrow N$ but $M \neq N$, the children of M and N are not sibling nodes. Therefore, there are no occurrence implications between the children of M and N, and we cannot infer whether or not $[F] \supseteq [H][d2]$. If $d[1].lp \neq \varnothing$, then $[d]=[M^*[H][d2]]$, and thus $[d]=[M[H][d2]] \mid \ldots \mid [M/\ldots/M[H][d2]]$ according to Theorem 7.10. $[c] \supseteq [M^*[H][d2]]$ iff $[c]$ contains each sub schema path, i.e. $[c] \supseteq [M[H][d2]] \wedge \ldots \wedge [c] \supseteq [M/\ldots/M[H][d2]]$ according to Proposition 7.4. If $N=M$ and $[F] \supseteq [H][d2]$, then $[c] \supseteq [M[H][d2]]$. For other schema paths with two or more M nodes, between the first and last M is a certain combination of the loop schema paths. Let $di=M/M_2\ldots/M[H][d2]$, and we do not know how to decide whether or not $[F] \supseteq [M_2\ldots/M[H][d2]]$. Therefore, we only present a necessary condition for the predicate containment in the second part of this proposition.

Now, we study how to check c⊇d when the schema path c contains non-empty loop schema paths.

Proposition 7.8: Let c and d be two schema paths in schema path$^{\{/,\ *,\ [\ ]\}}$.

(1) Let c=N*. c⊇d, iff d[1].N=N ∧ d[|d|].N=N.

(2) Let c=N*[F]. c⊇d, iff d[1].N=N ∧ d[|d|].N=N ∧ [F]⊇[d[|d|].L].

Proof. According to Theorem 7.10, c can be transformed to a union of an infinite set of schema paths without loop schema paths, the first schema path record of which is N and the last schema path record of which is N (or N[L]). Between two Ns is a certain combination of the loop schema paths. We transform d to a union of disjunction-free schema paths without loop schema paths according to Theorem 7.11. For each disjunction-free schema path d' of d, if there is a disjunction schema path c' of c such that c'⊇d', then c⊇d according to Theorem 7.13. Therefore, the key point is to prove that each disjunction-free schema path of d is a certain combination of the loop schema paths attached to N. That an instance schema node N is a head of some loop schema paths means that some instance descendant nodes of N are N itself. The last N is a child of the node d[|d|-1].N that is a child of d[|d|-2].N that is a child of d[|d|-3].N, and so on. If d[|d|-k].N=N, then (d[|d|-k], d[|d|-k+1], ..., d[|d|-1], d[|d|]) is a loop schema path. This means the path between Ns is one of the loop schema paths, and the path between the first N and last N is a certain combination among these loop schema paths. If N occurs only once, this means that visiting of nodes does not proceed further than the instance descendant nodes with N as a child. Therefore, iff d[1].N=N and d[|d|].N=N, each disjunction-free schema path of d is a certain combination of the loop schema paths attached to N. □

Proposition 7.9: Let c and d be two schema paths in schema path$^{\{/,\ *,\ [\ ]\}}$.

(1) Let c=N*. [c]⊇[d], iff d[1].N=>N.

(2) Let c=N*[F] and d=M[H]. [c]⊇[d], iff N=M and [F]⊇[H], or N=M and there is a record $M_i[H_i]$ in H, such that N=M_i and [F]⊇[H_i].

Proof. Let c=N*. c can be transformed to a union of disjunction-free schema paths without loop schema paths according to Theorem 7.10. If a disjunction-free schema path of c selects a non-empty node set, then [c] is true. If [d]=true, then d[1].N selects a non-empty result. Since d[1].N=>N, then N selects a non-

empty result. Since N is a disjunction-free schema path of c, [c]⊒[d]. If d[1].N does not imply N, then we cannot infer [c]=true from [d]=true.

Let c=N*[F]. If there is a node M_i in [H] somewhere such that N=M_i, we can rewrite [d]=[M[H]]=[M[H']/M_1[H_1]/.../M_i[H_i]]. If N=M ∧ N=Mi ∧ [F]⊒[H_i], then N*[F]⊒M[H']/M_1[H_1]/.../M_i[H_i] as proved in Proposition 7.8. Therefore, c⊒d, and thus [c]⊒[d]. For example, c=a*[b] and d=a[a/b]. We can rewrite [d]=[a/a[b]], and thus a*[b] ⊒ a/a[b] according to Proposition 7.8, and [c] ⊒ [d] according to Proposition 7.2. The proof of the rest is trivial, and thus is left out. □

Proposition 7.10: Let c and d be two schema paths in schema_path$^{\{/,\ *,\ [\ ]\}}$.

(1) Let c∈ schema path$^{\{/,\ *\}}$. c⊒d, if

$$c[1]\supseteq d_1 \wedge c[2]\supseteq d_2 \wedge \ldots \wedge c[|c|]\supseteq d_{|c|} \wedge d=d_1/d_2/d_3/\ldots/d_{|c|}.$$

(2) Let c∈ schema path$^{\{/,\ *,\ []\}}$. c⊒d, if

$$c[1]\supseteq d_1[d_2/d_3/\ldots/d_{|c|}] \wedge$$
$$c[2]\supseteq d_2[d_3/d_4/\ldots/d_{|c|}] \wedge$$
$$\ldots.$$
$$c[|c|]\supseteq d_{|c|} \wedge$$
$$d=d_1/d_2/d_3/\ldots/d_{|c|}. \square$$

Proof. Let c∈ schema path$^{\{/,\ *,\ []\}}$. According to Theorem 7.7, d = $d_1/d_2/\ldots/d_{|d|}$ = d_1**[d_2/.../$d_{|d|}$]**/d_2**[d_3/d_4/.../$d_{|d|}$]**/.../$d_{|d|}$. By recursively applying Theorem 7.14, we conclude c⊒d. □

Complexity Analysis. ∀k∈{1, ..., |c|}, if c[k] has no loop schema paths, we use Proposition 7.6 to find the corresponding part d_k from d such that c[k]⊒d_k[d_{k+1}/.../$d_{|c|}$]; if c[k] has loop schema paths, we use Proposition 7.8 to find the corresponding part d_k from d such that c[k]⊒d_k[d_{k+1}/.../$d_{|c|}$].

When c[1].lp=∅, c[1].N is only checked with d[1].N; When c[1].lp≠∅, c[1].N is checked at most once with each of d[1].N, d[2].N, …, d[|d|].N. We assume c[1]⊒d[1]/d[2]/.../d[m], where m≥1. When c[2].lp=∅, c[2].N is only checked with d[m+1].N; When c[2].lp≠∅, c[2].N is checked at most once with each of d[m+1].N, d[m+2].N, …, d[|d|].N. Thus, c[1].N is checked at most |d| times, c[2].N is checked at most |d|-1 times, …, and the c[|c|].N is checked at most |d|-|c|+1 times. Note that we write |c| for the depth of the schema path c (see Definition 7.7) and ||c|| for the number of schema path records in c (see Definition 7.6). Therefore, the run-time complexity of Proposition 7.6 is O(|c|) ≤ O(||c||), and of Proposition 7.8 is $O(|d|+|d|-1+\ldots+|d|-|c|+1) = O(\sum_{k=|d|}^{|d|-|c|+1} k) \le O(|d|*|c|) \le O(||c||*||d||)$.

We use Proposition 7.7 and Proposition 7.9 to check the containment of predicate schema paths. Each N in [c] is compared at most one time with each N in [d], which has the same depth; every N* in [c] is compared at most one time with each N in [d], which has the same or greater depth. Let m_k denote the number of the records in [c] with depth *k*. Let n_k denote the number of the records in [d] with depth *k*. Thus, the run-time complexity of Proposition 7.7 is $O(\sum_{k=1}^{|c|} m_k * n_k) \leq O(||c|| * ||d||)$, and of Proposition 7.9 is $O(\sum_{k=1}^{|c|} (m_k * \sum_{t=k}^{|d|} n_t)) \leq O(||c|| * ||d||)$.

Let mi_k be the number of the records with depth *k* in the predicates [Fi] of the *i*-th main schema path record in c; let |Fi| be the depth of [Fi]. Let ni_k be the number of the records with depth *k* in the predicates [Hi] of the *i*-th main schema path record in d; let Di be the part of d after the *i*-th record and [Di] become a predicate of the *i*-th location step of d. Let Di_k be the number of the records in [Di] with depth *k*. Therefore, the runtime-complexity of the predicate containment of each main schema path record is $O(\sum_{k=1}^{|Fi|} mi_k * (\sum_{t=k}^{|Hi|} ni_t + \sum_{t=k}^{|Di|} Di_k)) \leq O(||Fi|| * ||Hi+Di||)$. In Proposition 7.10, each main schema path record in c is checked at most one time with each main record in d, which has the same or greater depth. Therefore, the run-time complexity of Proposition 7.10 is $O(\sum_{k=|d|}^{|d|-|c|+1} k + \sum_{i=1}^{|c|} \sum_{k=1}^{|Fi|} mi_k * (\sum_{t=k}^{|Hi|} ni_t + \sum_{t=k}^{|Di|} Di_t)) \leq O(||c|| * ||d||)$. □

7.3.3.3 Schema path$^{\{/,\ FS,\ *,\ [\ ]\}}$

Let Q1 and Q2 be two XPath queries. Let Q1=w and Q2=u/FS::v/FS::w. Q1 selects all children w of the context node. Q2 selects the children w of the context node, which are following siblings of the nodes v and v in turn are following siblings of the children u of the context node. Therefore, Q2 selects a subset of Q1, and thus Q1⊇Q2. The following Proposition 7.11 describes how to decide containment of such two queries in terms of the corresponding schema paths.

Proposition 7.11: Let c and d be two schema paths in schema path$^{\{/,\ FS,\ *,\ [\ ]\}}$.

(1) Let c=^{child}N. c⊇d, iff

$$d[1].a=\text{'child'} \wedge d[1].lp=\varnothing \wedge \forall i \in \{2, 3, \ldots, |d|\}:d[i].axis=\text{'FS'} \wedge N=d[|d|].N.$$

(2) Let c=^{child}N[F]. c⊇d, iff

$$d[1].a=\text{'child'} \wedge d[1].lp=\varnothing \wedge \forall i \in \{2, 3, \ldots, |d|\}:d[i].axis=\text{'FS'} \wedge N=d[|d|].N \wedge [F] \supseteq [d[|d|].L].$$

Proof. If |d|=1, Proposition 7.11 is reduced to Proposition 7.6. In the case of |d|>1, c selects all the children N of the context node, and d selects the children N of the context node, which must fulfill the constraints specified in d[2], ..., d[|d|], i.e. which are the following-sibling nodes of some children of the context node. Therefore, d selects a subset of the result of c, and thus c⊇d.

If d[1].a≠'child', then the nodes selected by d are not descendant of the context node; if ∃i∈{2, 3, ..., |d|}: d[i].axis≠'FS', then d may select the children of children of the context node, and thus c!⊇d. □

Let Q1=FS::v and Q2=FS::u/FS::v. Q1 selects all the following sibling v of the context node. Q2 selects the following sibling v of the context node, which are following siblings of u and u is following sibling of the context. Therefore, Q2 selects a subset of Q1, and thus Q1⊇Q2.

Proposition 7.12: Let c and d be two schema paths in schema path$^{\{/,\ FS,\ *,\ [\ ]\}}$.

(1) Let c=^{FS}N. c⊇d, iff ∀i∈{1, ..., |d|}: d[i].axis='FS' ∧ N=d[|d|].N.

(2) Let c=^{FS}N[F]. c⊇d, iff ∀i∈{1, ..., |d|}: d[i].axis='FS' ∧ N=d[|d|].N ∧ [F]⊇[d[|d|].L].

Proof. c selects all the following-sibling N of the context node, and d selects the following-sibling N of the context node, which are the following-sibling nodes of some following-siblings of the context node. Therefore, d selects a subset of the result of c, and thus c⊇d. If ∃i∈{1, ..., |d|}: d[i].axis≠'FS', then d may select certain descendant nodes of the context node. Therefore, c and d may select different nodes, and thus c!⊇d. □

Proposition 7.13 studies the predicate-containment of schema paths when the axis following-sibling (FS) is introduced.

Proposition 7.13: Let c and d be two schema paths in schema path$^{\{/,\ FS,\ *,\ [\ ]\}}$, and d=^{child}M[H].

(1) Let c=^{child}N. [c]⊇[d], iff M=>N , or

d can be rewritten to d'=M[H1]/^{FS}M1[H2]/.../^{FS}M'[H'] such that c⊇d'.

(2) Let c=^{child}N[F]. [c]⊇[d], iff M=N and [F]⊇[H], or

d can be rewritten to d'=M[H1]/^{FS}M2[H2]/.../^{FS}M'[H'] such that N⊇d' and [F]⊇[H'].

We use the following example to explain Proposition 7.13. Let Q1=a and Q2=b[c][FS::d/FS::a[b]]. We can rewrite Q2 to Q2'= b[c]/FS::d/FS::a[b] and thus Q1⊇Q2' according to Proposition 7.11. Since [Q2]≡[Q2'], if Q2 returns a non-

empty result, then Q2' return a non-empty result, and thus Q1 return a non-empty result. Therefore, $[Q1] \supseteq [Q2]$. The proof of Proposition 7.13 is analogous to the proofs of Proposition 7.12 and Proposition 7.7.

When the loop schema paths are present, we need the following lemma in order to decide the containment of schema paths in schema path$^{\{/,\ FS,\ *,\ [\ ]\}}$.

Lemma 7.1: Let c and d be two schema paths in schema path$^{\{/,\ FS,\ *,\ [\ ]\}}$, c=N and $c \supseteq d$. Furthermore, let M be an XML schema node.

(1) Let d[i].N be a following-sibling of M, and M is a following-sibling of d[i-1].N. If we create a new schema path record $d[i']=^{FS}M$ or $d[i']=^{FS}M[H]$, and a schema path d'=d[1]/d[2]/…/d[i-1]/d[i']/d[i]/d[i+1]/…/d[|d|], then $c \supseteq d'$.

(2) Let M be a child of the context node and d[1].N is a following-sibling of M. If we create a new schema path record d[1']=M or d[1']=M[H] and set d[1].a='FS', and a schema path d''=d[1']/d[1]/d[2]/…/d[|d|], then $c \supseteq d''$. □

Proof. Since c=N and $c \supseteq d$, then d[1].a='child' and $\forall i \in \{2, \ldots, |d|\}$: d[i].axis = 'FS' and N=d[|d|].N according to Proposition 7.11. From the construction of d', d'[1].a='child' and $\forall i \in \{2, \ldots, |d'|\}$: d'[i].axis='FS' and N=d[|d'|].N. Therefore, $c \supseteq d'$ according to Proposition 7.11. Likewise, d''[1].a='child' and $\forall i \in \{2, \ldots, |d''|\}$: d''[i].axis='FS' and N=d[|d'|'].N. Therefore, $c \supseteq d''$ according to Proposition 7.11. □

Proposition 7.14: Let c and d be two schema paths in schema path$^{\{/,\ FS,\ *,\ [\ ]\}}$.

(1) Let c=N*. $c \supseteq d$, if $d[1].N=N \wedge d[1].a='child' \wedge d[|d|].N=N$.

(2) Let c=N*[F]. $c \supseteq d$, if $d[1].N=N \wedge d[1].a='child' \wedge d[|d|].N=N \wedge [F] \supseteq [d[|d|].L]$.

Proof. That an instance XML schema node N is a head of some loop schema paths means that some instance descendant nodes of N are N itself. Since c=N*, d[1].N=N means that d[1].N is the head of some loop schema paths. Furthermore, d[1].a='child' means that the path d consists of some descendant nodes of the context node. When N occurs once again, N can be visited as a child or a following sibling of the node d[|d|-1].N before it, and d[|d|-1].N can be visited as a child or a following sibling of the node d[|d|-2].N before it, and so on. Finally, d[2] can be visited as a child or following sibling of d[1].N.

When N is a child of d[|d|-1].N, d[|d|-1].N and N are two nodes occurring in a loop schema path in this order. When N is a following sibling of d[|d|-1].N, d[|d|-1].N is not in a loop schema path. However, if d[|d|-1].N is a child of d[|d|-2].N, then (d[|d|-2].N, N) is a part of a loop schema path, or if each of d[|d|-1].N, d[|d|-

2].N,…, d[|d|-k].N is a following sibling of the node before it, and d[|d|-k].N is a child of d[|d|-k-1].N, then (d[|d|-k-1].N, N) is a part of a loop schema path. Therefore, when the following sibling axis is allowed, between two Ns is not a direct combination of the loop schema paths, rather than a variant of a certain combination of the loop schema paths. According to Proposition 7.8 and Lemma 7.1, this proposition holds. □

When the following-sibling axis is introduced, Proposition 7.10 still holds, and the complexity of containment also keeps unchanged.

7.3.3.4 Attribute axes and comparison predicates

Now, we study the XPath containment when the attribute axis is allowed.

Proposition 7.15: Let c and d be two schema paths in schema path$^{\{/,\ FS,\ *,\ [\ ],\ A\}}$.

(1) Let c=^{attribute}N. c$\supseteq$d, iff d= ^{attribute}N $\vee$ d=^{attribute}N[H].

(2) Let c=^{attribute}N[F]. c$\supseteq$d, iff d=^{attribute}N[H] $\wedge$ [F]$\supseteq$[H] .

Proposition 7.16: Let c and d be two schema paths in schema path$^{\{/,\ FS,\ *,\ [\ ],\ A\}}$.

(1) Let c=^{attribute}N. [c]$\supseteq$[d], iff d= ^{attribute}N $\vee$ d=^{attribute}N[H].

(2) Let c=^{child}N[F]. [c]$\supseteq$[d], iff d=^{attribute}N[H] $\wedge$ [F]$\supseteq$[H].

The correctness of Proposition 7.15 and of Proposition 7.16 is easy to see, and thus their proofs are left out.

A location step with the attribute axis can appear only as the last location step in an XPath expression or as the last location step in any predicate in the absence of reverse axes. For example, Q1=a/b/@c, and Q2=a[b[@c=1]]/d. Mapped to the corresponding schema paths, the schema path record with a='attribute' should be the last schema path record with or without a predicate schema path like (<self::node()=C>), in the main schema path and any predicate schema paths. Therefore, the theorems and propositions above still hold in the presence of the attribute axis. The introduction of the attribute axis does not increase the complexity of containment.

A special schema path of predicates consists of an expression, e.g. (<self::node()=100>). Let c=(<self::node()>1>) and d=(<self::node()>2>), then

[d]=true always means [c]=true. Proposition 7.17 studies the predicate containment of such two schema paths. The correctness of this proposition is easy to see, and thus the proof is left out.

Proposition 7.17: Let V1 and V2 be a number or a string; o1 and o2 $\in$ {=, >, ≥, <, ≤}. Let c and d be schema paths, c=(self::node() o1 V1). [c]$\supseteq$[d], iff

d=(self::node() o2 V2) $\wedge$ (

(o1='=' $\wedge$ o2='=' $\wedge$ V1=V2) $\vee$
(o1='>' o2='>' $\wedge$ V2≥V1) $\vee$ (o1='>' o2='≥' $\wedge$ V2>V1) $\vee$ (o1='>' o2='=' $\wedge$ V2>V1) $\vee$
(o1='≥' o2='>' $\wedge$ V2≥V1) $\vee$ (o1='≥' o2='≥' $\wedge$ V2≥V1) $\vee$ (o1='≥' o2='=' $\wedge$ V2≥V1) $\vee$
(o1='<' o2='<' $\wedge$ V2≤V1) $\vee$ (o1='<' o2='≤' $\wedge$ V2<V1) $\vee$ (o1='<' o2='=' $\wedge$ V1<V2) $\vee$
(o1='≤' o2='<' $\wedge$ V2≤V1) $\vee$ (o1='≤' o2='≤' $\wedge$ V2≤V1) $\vee$ (o1='≤' o2='=' $\wedge$ V2≤V1)).

Complexity Analysis. It is easy to see when all the features are allowed in a schema path, Proposition 7.10 and the complexity analysis still hold. Let ||c|| and ||d|| be the number of records of the schema paths c and d respectively. The complexity of deciding c$\supseteq$d is O(||c||*||d||).

7.4 Performance analysis

We develop a prototype of our approach to the containment of XPath queries in terms of containment of schema paths of queries. The performance study focuses on the efficiency of our approach. We measure the time cost of evaluating XPath queries on an XML Schema definition, normalizing schema paths and checking containment of schema paths. The test system and used XML data sets are described in Section 1.4 in Chapter 1.

7.4.1 XPath queries

We design 12 pairs of queries Q1-Q1', …, Q12-Q12'. The first one of each of first 11 pair of queries contains the second one. The query Q4 contains Q4' with and without respect to constraints; other containment holds only in the constraints of the XML Schema definition benchmark.xsd (see the corresponding

appendix). Table 7.1 presents the used queries Q1-Q12 and Q1'-Q12' and the result of whether or not Qx contains Qx', where x∈{1, ..., 15}. Table 7.1 also gives the semantics counterparts of some queries in order to see the containment results more easily.

Table 7.1: Queries Q1-Q12 and Q1'-Q12' and the result of containment

Queries		Qx⊇Qx'
Q1	/site/categories/category	√
Q1'	/site/categories/∗ ≡ /site/categories/category	
Q2	/site/regions/∗/item	√
Q2'	/site/regions[europe]/asia/item ≡ /site/regions/asia/item	
Q3	/site/regions[asia]/∗/item ≡ /site/regions/∗/item	√
Q3'	/∗/regions[∗]/namerica/item ≡ /site/regions/namerica/item	
Q4	/site/open_auctions/open_auction[bidder[personref/@person='person0']/following-sibling::bidder[personref/@person='person1']]	√
Q4'	/site/open_auctions/open_auction[bidder[personref/@person='person0']/following-sibling::bidder/following-sibling::bidder[personref/@person='person1']]	
Q5	/site/people/person[profile[interest][age]]/name	√
Q5'	//person[profile/age][profile/interest]/name ≡ /site/people/person[profile[interest][age]]/name	
Q6	/site/people/person[((address and phone) or profile[age or (gender or sex)]) or (homepage or (address or phone))] ≡ /site/people/person[((address and phone) or profile[age or gender] or homepage or address or phone]	√
Q6'	/site/people/person[(address and phone and homepage) or profile[age and gender]]	
Q7	/site//increase[. > 20][. <80] ≡ /site/open_auctions/open_auction/bidder/increase[text()>20][text()<80]	√

Q7'	//bidder[date][time]/increase[text()<60][self::node()>40] ≡ /site/open_auctions/open_auction/bidder/increase [text()<60][text()>40]	
Q8	//person/profile[business][education] ≡ /site/people/person/profile[education]	√
Q8'	//people/person[profile/education]/profile ≡ /site/people/person/profile[education]	
Q9	/site/people/self::∗/person/address ≡ /site/people/person/address	√
Q9'	/∗/categories/following-sibling::catgraph/following-sibling::people/person/address ≡ /site/categories/following-sibling::catgraph /following-sibling::people/person/address	
Q10	//listitem	√
Q10'	//listitem//listitem//listitem	
Q11	//listitem//keyword	√
Q11'	/descendant-or-self::listitem/descendant-or-self::keyword ≡ //listitem//keyword	
Q12	//listitem//listitem//listitem	×
Q12'	//listitem	

7.4.2 Containment test

We test the containment of the queries in Table 7.1 and get the same containment result for each pair of queries as given in Table 7.1.

Figure 7.6 presents the time of containment test, including the time of computing the schema paths of the second query of each pair of queries in Table 7.1, i.e. evaluating the query over the XML Schema definition benchmark.xsd, the time of normalizing the schema paths, and the time of checking containment of schema paths of the two queries. Figure 7.6 shows that the main cost of the containment test is computing the schema paths of a query, and the main cost of computing schema paths is the evaluation of recursive location steps. Evaluating the query Q10' with three recursive location steps is about 2 times slower than evaluating the queries Q5', Q7', Q8' and Q12' with one recursive lo-

cation step. Evaluating the query Q11' with two recursive location steps is about 1.5 times slower than evaluating the queries Q5', Q7', Q8' and Q12'. The queries Q1'-Q4', Q6' and Q9' without recursive location steps can be evaluated very fast.

The cost of normalizing schema paths and checking containment is mainly dependent on the number of schema paths. For example, since only one schema path is computed from the queries Q1'-Q9', the time of normalizing schema paths and checking containment is not significant. 9 schema paths are computed from Q10' and up to 90 schema paths are computed from Q11', where we observe some cost when normalizing schema paths and checking containment.

In order to get the conclusion that a query Q1 contains another Q2, we have to show that each schema path of Q2 is contained by a schema path of Q1. In order to conclude that Q1 does not contain another Q2, we only need to check that there is a schema path of Q2, which is not contained by any schema path of Q1. Therefore, we can get the conclusion of Q1!⊇Q2 usually faster than get the conclusion of Q1⊇Q2. Unless the schema path of Q2, which is not contained by any schema path of Q1, is checked as the last one. For example, the query Q12 does not contain Q12', and the time of checking containment is not significant. Overall, the absolute cost for containment test is quite little, about 0.1 seconds in the worst case in all experiments.

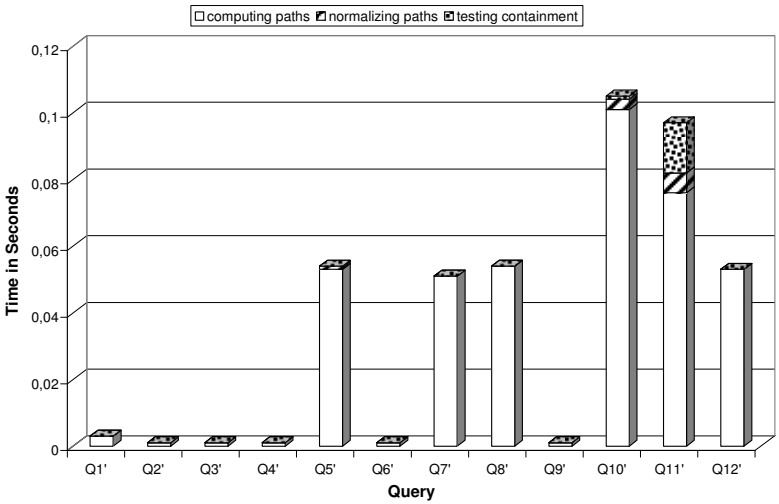

Figure 7.6: Time to check containment of the queries in Table 7.1

Chapter 8 Related Work

A large amount of work deals with the optimization of XPath queries in order to speed up XML querying.

8.1 XPath evaluation

Since all major XPath engines take time exponential in the size of the input queries when evaluating XPath queries [Gottlob et al. 2002], there has been work on physical optimization of XPath expressions. [Gottlob et al. 2003a] studies theoretically the evaluation complexity of XPath queries and obtains the combined complexity of XPath evaluation to be P-hard in terms of the data complexity and the query complexity of XPath 1.0. [Gottlob et al. 2002] develops bottom-up processing of XPath expressions, which runs in polynomial time in terms of the data and the query size. [Gottlob et al. 2003b] improves the time and space efficiency of the approach suggested in [Gottlob et al. 2002]. [Gottlob et al. 2003c] presents the efficient algorithms for processing queries in XPath 1.0, including the bottom-up evaluation of XPath and the top-down evaluation of XPath.

Furthermore, indexing techniques (see e.g. [Rao and Moon 2004] and [Wang et al. 2003]) and structural join algorithms (see e.g. [Bruno et al. 2002] and [Jiang et al. 2003]) are also developed to speed up XPath processing.

8.2 XPath rewriting

A number of research efforts are dedicated on rewriting of XPath expressions to optimize XPath queries and thus speed up XPath evaluation.

[Kwong and Gertz 2002] suggests an algorithm for rewriting and satisfiability test of XPath expressions in the presence of DTDs. Different from [Kwong and Gertz 2002], which enumerates all possible paths from a DTD, we directly generate the paths for a given XPath query by evaluating the XPath query on the XML Schema definition. Furthermore, we support recursive schemas that are not considered by [Kwong and Gertz 2002]. We consider all XPath axes, but the axes that depend on the document order are not supported by [Kwong and Gertz 2002].

[Olteanu et al. 2002], [Benedikt et al. 2003] and [Chan et al. 2004] study logical rewriting and optimization of XPath based on the properties of XPath expressions: [Olteanu et al. 2002] eliminates reverse axes for efficient evaluation on streaming data, [Benedikt et al. 2003] identifies useful rewriting rules and [Chan et al. 2004] minimizes wildcard steps to speed up XPath evaluation. [Chan et al. 2004] suggests an approach to minimizing wildcards in the absence of schemas. In comparison to [Chan et al. 2004], we support to eliminate wildcards completely in XPath queries. [Olteanu et al. 2002] eliminates reverse axes in XPath expressions according to the axis symmetry of XPath, while we eliminate reverse axes based on the symmetry of schema paths as well as of XPath axes. Thus, we can eliminate reverse axes without adding additional location steps. [Fan et al. 2005] develops an algorithm to rewrite XPath queries to regular XPath queries on recursive DTDs, but only forward axes are considered and the reverse axes and the axes depending on the document order are not allowed. Our approach can rewrite XPath queries to regular and standard XPath queries in the case of recursive schemas, and supports all XPath axes. Furthermore, similar to [Kwong and Gertz 2002], [Fan et al. 2005] enumerates all the paths from a DTD, but we construct only the paths from an XML Schema definition for a given XPath query.

Several research efforts focus on the minimization of XPath expressions (see e.g. [Amer-Yahia et al. 2001], [Ramanan 2002] and [Wood 2001]) by eliminating redundant steps since the size of XPath expressions significantly impacts the processing of queries. The study on the minimization of XPath closely relates to the issues of equivalence and containment with respect to two XPath queries (see e.g. [Miklau and Suciu 2004] and [Wood 200]). [Amer-Yahia et al. 2001], [Ramanan 2002] and [Wood 2001] reduce redundant parts of tree pattern queries by the equivalence and containment analysis of two subpatterns. [Groppe 2005] and [Groppe et al. 2006a] reformulates XPath expres-

sions according to XSLT stylesheets in order to reduce the amount of data transmitted and transformed.

8.3 XPath satisfiability

Many research efforts are dedicated to the satisfiability problem of XPath queries.

[Benedikt et al. 2005] theoretically studies the complexity problem of XPath satisfiability in the presence of DTDs, and shows that the complexity of XPath satisfiability depends on the considered subsets of XPath queries and DTDs. Among other things, [Benedikt et al. 2005] shows that the complexity of XPath satisfiability ranges from PTIME to NP-complete without negation operation in XPath expressions; ranges from PSPACE-complete to undecidable in the case of existence of negation operators and the recursive schemas. We present a practical algorithm for testing the satisfiability of XPath queries with negation operation and with respect to recursive schemas.

[Hidders 2003] investigates the problem of XPath satisfiability in the absence of schemas. Without respect to schemas, an XPath query is unsatisfiable if the query does not conform to the XML data model. [Hidders 2003] gives the complexity results for different fragments of XPath. [Groppe et al. 2008] and [Groppe et al. 2006] thoroughly summarize the structure constraints of XPath imposed by the XML data model and the constraints from XPath expressions themselves to filter unsatisfiable XPath queries without respect to schemas.

[Lakshmanan et al. 2004] examines the satisfiability test of tree pattern queries (i.e. reverse axes are not considered) with respect to non-recursive schemas, and shows that the satisfiability problem of tree pattern queries is NP-complete under the non-recursive schemas. [Lakshmanan et al. 2004] does not deal with the negation operation in XPath.

[Kwong and Gertz 2002] suggests an algorithm to test the satisfiability of XPath queries, but allows only non-recursive DTDs and does not support the XPath axes related with document order, e.g. axes following-sibling, preceding-sibling, following and preceding. We support recursive schemas and all XPath axes. Our approach can filter the XPath queries that do not conform to the constraints of value types as well as the constraints of structures imposed in an

XML Schema definition, and the XPath queries with conflicting constraints. Only the XPath queries, which do not conform to the constraints of structures given in a DTD, are detected as unsatisfiable by [Kwong and Gertz 2002].

[Kempa 2003] presents an approach to checking if XPath expressions conform to the structural constraints in a schema. In [Kempa 2003], XML Schema definitions are formalized and represented as regular hedge expressions [Brüggemann-Klein et al. 2001]. The hedge expressions cover only the structural constraints of a schema. Some constraints, e.g. restrictions on built-in simple types, in the schema are not preserved after the formalization. Therefore, [Kempa 2003] cannot check the value types of elements and attributes, nor it can check fixed value constraints and occurrence constraints. Furthermore, [Kempa 2003] can not rewrite and optimize XPath expressions.

[Geneves et al. 2007] is a most recent work on XPath satisifiability under constraints. [Geneves et al. 2007] translates schemas to regular tree languages (see [Hosoya et al. 2000]) and provides an algorithm to test XPath satisfiability under regular tree type. Similar to [Kempa 2003], [Geneves et al. 2007] cannot check the type of values of elements and attributes, nor it can check fixed value constraints and occurrence constraints. It cannot rewrite and optimize XPath expressions either.

[Groppe 2005] discusses how to reduce the containment and intersection test of XPath expressions to the satisfiability test, which extends the applications of the satisfiability test.

8.4 XPath containment

Many research efforts deal with the containment problem of XPath queries in the absence of constraints.

[Miklau and Suciu 2004] shows that containment for XPath queries with the child axes /, the descendant axes //, predicates [] and the wildcards ∗ is coNP-complete, and provides an algorithm of checking if an XPath query C contains another query D, with runtime complexity of $O(|C|*|D|*(w+1)^d)$ where w is the length of the maximum continuous ∗ location steps in C and d is the number of the descendant location steps in D. The complexity of XPath containment is shown to be PTIME if any of ∗, [] and // is absence in the considered XPath fragment: without ∗, [Amer-Yahia et al. 2001] describes a PTIME containment

algorithm based on tree patterns; without predicates, containment is shown to be in PTIME by [Milo and Suciu 1999]. In the absence of descendant axes, [Wood 2001] shows that XPath containment is in PTIME.

In 1981 [Yannakakis 1981] has proposed a PTIME containment algorithm for acyclic conjunctive queries. Containment of relational conjunctive queries is known to be NP-complete in [Chandra and Merlin 1977]. [Florescu et al. 1998] shows that containment for conjunctive queries with regular path expressions over semi-structured data is decidable.

The following contributions focus on XPath containment under integrity constraints and DTDs.

[Neven and Schwentick 2003] uses automata-theory techniques to show that the XPath containment is EXPTIME-complete under DTDs, and containment is PSPACE-complete if variables are included. [Wood 2003] proves the decidability of containment for various XPath fragments in the presence of DTDs and a certain class of integrity constraints. [Deutsch and Tannen 2001] studies the problem of containment of XPath expressions under integrity constraints using the technique of first-order translation, and shows that the complexity of containment is NP-hard for XPath expressions with equality testing and binding of variables, and is $\prod^{P}_{2}$ if disjunction is additionally considered. [Schwentick 2004] presents the complexity results for XPath containment for various fragments of XPath with and without respect to DTDs, and describes the techniques used to obtain upper bounds of the complexity results. [Cate and Lutz 2007] studies the complexity of XPath containment, when path intersection, path equality, path complementation, for-loops and transitive closure are allowed. [Cate and Lutz 2007] shows that the complexity of containment in this XPath fragment ranges from EXPTIME (with path equality) and 2-EXPTIME (with path intersection) to non-elementary (with path complementation or for-loops), and adding transitive closure does not increase the complexity.

Chapter 9 Conclusions

In this thesis, we propose a data model for the XML Schema language, which identifies the navigation paths of XPath queries on an XML Schema definition. We develop an XPath-XSchema evaluator, which evaluates XPath queries on an XML Schema definition based on the data model of XML Schema. The XPath-XSchema evaluator returns a set of schema paths, which integrate the constraints imposed by the schema. We develop a satisfiability tester and a containment tester of XPath based on the schema paths. We also suggest an approach to rewriting and optimizing XPath expressions in the presence of XML Schema definitions.

Our satisfiability tester can filter the XPath queries, which do not conform to the constraints of structures, semantics, data types or occurrences imposed by a schema, and the XPath queries, which contain visible and invisible conflicting constraints. When an XPath query does not conform to the constraints in a given schema, our evaluator computes the empty set of schema paths, and thus the XPath query is unsatisfiable. If a non-empty set of schema paths is computed for an XPath query, we rewrite the query from the schema paths, and apply the rules of conflicting constraints to the rewritten queries to further filter the queries with conflicting constraints.

The experimental results of the prototype of our satisfiability tester show that application of our approach can significantly optimize the evaluation of XPath queries by filtering unsatisfiable XPath queries. A speed-up factor up to several orders of magnitude is possible. Furthermore, our approach has a low overhead (<0.08 seconds), and does not significantly increase the total processing time of satisfiable queries when XML documents are not very small (not less than 100 Kilobytes).

Our containment tester checks the containment of XPath queries in terms of the normalized schema paths of the queries. The most constraints, which impact the containment test, can be easily inferred using the normalized schema paths. We prove the correctness of our approach to XPath containment, and

analyze the complexity of the approach. Our experimental results show that our approach has a low overhead (less than 0.1 seconds).

9.1 Future work

The work described in this thesis can be extended and carried on further to

- support more features of XPath 2.0 and the XML Schema language to
  - detect more unsatisfiable XPath queries, which are not tested as unsatisfiable in this work, and
  - find more containment cases, which cannot be detected in this work,

  by extending the XPath-XSchema evaluator and the set of rules of conflicting constraints.
- investigate the performance benefits in those application scenarios, which use the containment test, e.g. caching of XPath query results.
- investigate more applications of our data model of XML Schema, e.g. our data model can be applied to the validation of XML documents.
- investigate more applications of the schema paths of queries, e.g. the intersection of XPath expressions in the presence of schemas.
- transfer our results for XPath to XQuery and XSLT by
  - using an XQuery query or XSLT stylesheet instead of the XPath query, or
  - investigating application scenarios, which use the satisfiability test or containment test applied to the *output schemas* (see [Groppe and Groppe 2006d] and [Groppe et al. 2007a]) of an XQuery query or an XSLT stylesheet, e.g. caching the results of an XPath query applied to the results of XQuery queries or XSLT stylesheets.

Benchmark.xsd

In this section, we present the XML Schema definition benchmark.xsd, which we use for the performance analysis. This schema is manually adapted according to the DTD benchmark.dtd of the XPathMark benchmark (see [Franceschet 2005]) and the instance documents generated using the data generator of [Franceschet 2005] in order to integrate as many constructs of XML Schema as possible and specify more specific data types for values of elements and attributes, which are only declared as #PCDATA in benchmark.dtd.

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

 <xs:annotation>
  -- This schema was manually adapted according to --
  -- benchmark.dtd and the instance documents in order to  --
  -- integrate as many constructs of XML Schema as possible and  --
  -- specify more specific data types for values of elements and attributes, --
  -- which are only declared as #PCDATA in benchmark.dtd --
 </xs:annotation>

 <xs:element name='site' type='siteType'/>

 <xs:complexType name='siteType'>
   <xs:sequence>
     <xs:element name='regions' type='regionsType'/>
     <xs:element name='categories' type='categoriesType'/>
     <xs:element name='catgraph' type='catgraphType'/>
     <xs:element name='people' type='peopleType'/>
     <xs:element name='open_auctions' type='open_auctionsType'/>
     <xs:element name='closed_auctions' type='closed_auctionsType'/>
   </xs:sequence>
   <xs:attribute name='owner' type='xs:string' use='prohibited'/>
 </xs:complexType>

 <xs:complexType name='regionsType'>
    <xs:sequence>
      <xs:element name='africa' type='regionType'/>
      <xs:element name='asia' type='regionType'/>
```

```
    <xs:element name='australia' type='regionType'/>
    <xs:element name='europe' type='regionType'/>
    <xs:element name='namerica' type='regionType'/>
    <xs:element name='samerica' type='regionType'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='categoriesType'>
  <xs:sequence>
    <xs:element name='category' maxOccurs='unbounded'>
      <xs:complexType>
            <xs:sequence>
              <xs:element name='name' type='xs:string'/>
              <xs:element name='description' type='descriptionType'/>
            </xs:sequence>
            <xs:attribute name='id' use='required' type='xs:ID'/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='catgraphType'>
  <xs:sequence>
    <xs:element name='edge' type='edgeType' minOccurs='0'
              maxOccurs='unbounded'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='peopleType'>
   <xs:sequence>
     <xs:element name='person' type='personType' minOccurs='0'
            maxOccurs='unbounded'/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name='open_auctionsType'>
   <xs:sequence>
     <xs:element name='open_auction' type='open_auctionType'
            minOccurs='0' maxOccurs='unbounded'/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name='closed_auctionsType'>
```

```
  <xs:sequence>
    <xs:element name='closed_auction' type='closed_auctionType'
            minOccurs='0' maxOccurs='unbounded'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='regionType'>
  <xs:sequence>
    <xs:element name='item' type='itemType' minOccurs='0'
           maxOccurs='unbounded'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='edgeType'>
 <xs:attribute name='from' use='required' type='xs:IDREF'/>
 <xs:attribute name='to' use='required' type='xs:IDREF'/>
</xs:complexType>

<xs:complexType name='personType'>
  <xs:complexContent>
    <xs:extension base='personType0'>
      <xs:sequence>
        <xs:element name='profile' type='profileType' minOccurs='0'/>
        <xs:element name='watches' type='watchesType' minOccurs='0'/>
        <xs:element name='race' type='xs:string' minOccurs='0' maxOccurs='0'/>
      </xs:sequence>
   </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='personType0'>
  <xs:sequence>
    <xs:element name='name' type='xs:string'/>
    <xs:element name='emailaddress' type='xs:string'/>
    <xs:element name='phone' type='xs:string' minOccurs='0'/>
    <xs:element name='address' type='addressType' minOccurs='0'/>
    <xs:element name='homepage' type='xs:string' minOccurs='0'/>
    <xs:element name='creditcard' type='creditcardType' minOccurs='0'/>
  </xs:sequence>
 <xs:attribute name='id' use='required' type='xs:ID'/>
</xs:complexType>

<xs:complexType name='open_auctionType'>
```

```
  <xs:sequence>
    <xs:element name='initial' type='xs:float'/>
    <xs:element name='reserve' type='reserveType' minOccurs='0'/>
    <xs:element name='bidder' type='bidderType' minOccurs='0'
              maxOccurs='unbounded'/>
    <xs:element name='current' type='xs:float'/>
    <xs:element name='privacy' type='privacyType' minOccurs='0'/>
    <xs:element name='itemref' type='itemrefType'/>
    <xs:element name='seller' type='sellerType'/>
    <xs:element name='annotation' type='annotationType'/>
    <xs:element name='quantity' type='quantityType'/>
    <xs:element name='type' type='typeType'/>
    <xs:element name='interval' type='intervalType'/>
  </xs:sequence>
  <xs:attribute name='id' use='required' type='xs:ID'/>
</xs:complexType>

<xs:complexType name='closed_auctionType'>
  <xs:sequence>
    <xs:element name='seller' type='sellerType'/>
    <xs:element name='buyer' type='buyerType'/>
    <xs:element name='itemref' type='itemrefType'/>
    <xs:element name='price' type='xs:float'/>
    <xs:element name='date' type='dateType'/>
    <xs:element name='quantity' type='quantityType'/>
    <xs:element name='type' type='typeType'/>
    <xs:element name='annotation' type='annotationType' minOccurs='0'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='itemType'>
  <xs:sequence>
    <xs:element name='location' type='xs:string'/>
    <xs:element name='quantity' type='quantityType'/>
    <xs:element name='name' type='xs:string'/>
    <xs:element name='payment' type='xs:string'/>
    <xs:element name='description' type='descriptionType'/>
    <xs:element name='shipping' type='xs:string'/>
    <xs:element name='incategory' type='incategoryType'
          maxOccurs='unbounded'/>
    <xs:element name='mailbox' type='mailboxType'/>
  </xs:sequence>
 <xs:attribute name='id' use='required' type='xs:ID'/>
```

```
 <xs:attribute name='featured'/>
</xs:complexType>

<xs:complexType name='descriptionType'>
  <xs:choice>
    <xs:element name='text' type='textType'/>
    <xs:element name='parlist' type='parlistType'/>
  </xs:choice>
</xs:complexType>

<xs:complexType type='addressType'>
  <xs:sequence>
    <xs:element name='street' type='xs:string'/>
    <xs:element name='city' type='xs:string'/>
    <xs:element name='country' type='xs:string'/>
    <xs:element name='province' type='xs:string' minOccurs='0'/>
    <xs:element name='zipcode' type='xs:string'/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name='creditcardType'>
  <xs:restriction base='xs:string'>
    <xs:pattern value='\d{4}\s*\d{4}\s*\d{4}\s*\d{4}'/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name='profileType'>
  <xs:sequence>
    <xs:element name='interest' type='interestType' minOccurs='0'
              maxOccurs='unbounded'/>
    <xs:element name='education' type='educationType' minOccurs='0'/>
    <xs:element name='gender' type='genderType' minOccurs='0'/>
    <xs:element name='business' type='businessType'/>
    <xs:element name='age' type='ageType' minOccurs='0'/>
  </xs:sequence>
 <xs:attribute name='income' type='xs:flocat'/>
</xs:complexType>

<xs:complexType name='watchesType'>
  <xs:sequence>
    <xs:element name='watch' minOccurs='0' maxOccurs='unbounded'>
      <xs:complexType>
        <xs:complexContent>
```

```
          <!-- emptyp content model -->
          <xs:restriction base='xs:anyType'>
            <xs:attribute name='open_auction' use='required' type='xs:IDREF'/>
            <xs:attribute name='expression' use='prohibited' type='xs:string'/>
          </xs:restriction>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name='reserveType' mixed='true'>
 </xs:complexType>

 <xs:complexType name='bidderType'>
  <xs:sequence>
    <xs:element name='date' type='dateType'/>
    <xs:element name='time' type='xs:time'/>
    <xs:element name='personref' type='personrefType'/>
    <xs:element name='increase' type='xs:float'/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name='privacyType'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='Yes'/>
    <xs:enumeration value='No'/>
  </xs:restriction>
</xs:simpleType>

 <xs:complexType name='itemrefType'>
  <xs:attribute name='item' use='required' type='xs:IDREF'/>
 </xs:complexType>

 <xs:complexType name='sellerType'>
   <!-- empty content model -->
   <xs:complexContent>
     <xs:restriction base='xs:anyType'>
       <xs:attribute name='person' use='required' type='xs:IDREF'/>
     </xs:restriction>
   </xs:complexContent>
 </xs:complexType>
```

```
<xs:complexType name='annotationType'>
  <xs:sequence>
    <xs:element name='author'>
      <!-- anonomously complex type definition -->
      <xs:complexType>
        <!-- empty content model -->
        <xs:complexContent>
          <xs:restriction base='xs:anyType'>
            <xs:attribute name='person' use='required' type='xs:IDREF'/>
          </xs:restriction>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
    <xs:element name='description' type='descriptionType' minOccurs='0'/>
    <xs:element name='happiness'>
      <xs:simpleType>
        <xs:restriction base='happinessType1'/>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name='quantityType'>
  <xs:restriction base='xs:int'>
    <xs:minInclusive value='0'/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name='typeType'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='Regular'/>
    <xs:enumeration value='Featured'/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name='intervalType'>
  <xs:sequence>
    <xs:element name='start' type='dateType'/>
    <xs:element name='end' type='dateType'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='buyerType'>
```

```
 <xs:attribute name='person' use='required' type='xs:IDREF'/>
</xs:complexType>

<xs:complexType name='incategoryType'>
 <xs:attribute name='category' use='required' type='xs:IDREF'/>
</xs:complexType>

<xs:complexType name='mailboxType'>
  <xs:sequence>
    <xs:element name='mail' type='mailType' minOccurs='0'
          maxOccurs='unbounded'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='textType' mixed='true'>
  <xs:choice minOccurs='0' maxOccurs='unbounded'>
   <xs:element name='bold' type='textType'/>
   <xs:element name='keyword' type='textType'/>
   <xs:element name='emph' type='textType'/>
  </xs:choice>
</xs:complexType>

<xs:complexType name='parlistType'>
  <xs:sequence minOccurs='0' maxOccurs='unbounded'>
    <xs:element name='listitem' type='listitemType'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='interestType'>
 <xs:attribute name='category' use='required' type='xs:IDREF'/>
</xs:complexType>

<xs:complexType name='educationType' mixed='true'>
</xs:complexType>

<xs:simpleType name='genderType'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='male'/>
    <xs:enumeration value='female'/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name='businessType'>
```

```
  <xs:restriction base='xs:string'>
    <xs:enumeration value='Yes'/>
    <xs:enumeration vaule='No'/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name='ageType'>
  <xs:restriction base='ageType1'>
    <xs:maxInclusive value='99'/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name='ageType1'>
    <xs:restriction base='xs:int'>
      <xs:minInclusive value='18'/>
    </xs:restriction>
</xs:simpleType>

<xs:complexType name='personrefType'>
 <xs:attribute name='person' use='required' type='xs:IDREF'/>
</xs:complexType>

<xs:complexType name='authorType'>
 <xs:attribute name='person' use='required' type='xs:IDREF'/>
</xs:complexType>

<xs:simpleType name='happinessType1'>
  <xs:restriction base='happinessType2'>
      <xs:maxInclusive value='10'/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name='happinessType2'>
  <xs:restriction base='xs:int'>
    <xs:maxInclusive value='100'/>
    <xs:minInclusive value='0'/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name='mailType'>
  <xs:sequence>
    <xs:element name='from' type='xs:string'/>
    <xs:element name='to' type='xs:string'/>
```

```
      <xs:element name='date' type='dateType'/>
      <xs:element name='text' type='textType'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='listitemType'>
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='text' type='textType'/>
      <xs:element name='parlist' type='parlistType'/>
    </xs:choice>
  </xs:complexType>

  <xs:simpleType name='dateType'>
    <xs:restriction base='xs:string'>
      <xs:pattern value="\d{2}/\d{2}/\d{4}"/>
    </xs:restriction>
  </xs:simpleType>

<!-- never used elements

  <xs:element name='amount'>
    <xs:complexType mixed='true'>
    </xs:complexType>
   </xs:element>

   <xs:element name='income' type='xs:float'/>

 <xs:element name='status'>
  <xs:complexType mixed='true'>
  </xs:complexType>
 </xs:element>

-->

</xs:schema>
```

References

[Amer-Yahia et al. 2001] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. “Minimization of Tree Pattern Queries”, in ACM SIGMOD 2001, pp 497-508. Santa Barbara, California, USA, 2001.

[Balmin et al. 2004] A. Balmin, F. Özcan, K. Beyer, and R.J. Chchrane. “A Framework for Using Materialized XPath Views in XML Query Processing”, in VLDB 2004, Volume 30, pp. 60 – 71. Toronto, Canada.

[Bancilhon et al. 1986] F. Bancilhon, D. Maier, Y. Sagiv and J. D. Ullman. “Magic Sets and Other Strange Ways to Implement Logic Programs”, in PODS 1986, pp. 1-15. Cambridge, Massachusetts, USA, 1986.

[Behrend 2003] A. Behrend. “Soft Stratification for Magic Set Based Query Evaluation in Deductive Databases”. In PODS 2003, pp. 102-110. San Diego, California, USA, 2003.

[Benedikt et al. 2005] M. Benedikt, W. Fan, F. Geerts. “XPath Satisfiability in the presence of DTDs”, in PODS, pp. 25-36. Baltimore, Maryland, USA, 2005.

[Benedikt et al. 2003] M. Benedikt, W. Fan and G. M. Kuper. “Structural properties of XPath fragments”, in ICDT 2003. LNCS 2572, pp. 79-95.

[Borland 2001] Borland. “XML Application Developer’s Guide, JBuilder, version 5”, Borland Software Corporation, Scotts Valley, CA, 2001.

[Brown et al. 2001] A. Brown, M. Fuchs, J. Robie, P. Wadler. “MSL: A model for W3C XML Schema”, in Proceedings of International WWW Conference, Hong-Kong, 2001.

[Bruno et al. 2002] N. Bruno, N. Koudas, and D. Srivastava. “Holistic twig joins: optimal XML pattern matching”, in SIGMOD 2002.

[Brüggemann-Klein et al. 2001] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. “Regular Tree and RegularHedge Languages over Unranked Alphabets: Version 1”, Technical Report HKUSTTCSC-2001-05, Theoretical Computer Science Center, Hong Kong University of Science & Technology, April 3 2001.

[Cate and Lutz 2007] Balder ten Cate, Carsten Lutz. “The complexity of query containment in expressive fragments of XPath 2.0”, in PODS 2007: 73-82.

[Chan et al. 2004] C. Y. Chan, W. Fan, Y. Zeng. “Taming XPath Queries by Minimizing Wildcard Steps”, in VLDB 2004.

[Chandra and Merlin 1977] A. K. Chandra, and P. M. Merlin. “Optimal implementation of conjunctive queries in relational databases”, in STOC 1977.

[Chaudhuri 1998] S. Chaudhuri. “An Overview of Query Optimization in Relational Systems”, in PODS 1998. Seattle, Washington, USA, 1998.

[Choi 2002] B. Choi. “What are real DTDs like?”, in WebDB 2002. pages 43-48.

[CWI 2003] CWI. XMark – An XML benchmark project. http://www.xml-benchmark.org/. 2003.

[Deutsch and Tannen 2001] A. Deutsch, and V. Tannen. “Containment and Integrity Constraints for XPath Fragments”, in KRDB 2001.

[Exolab 2001] ExoLab Group. “Castor”, ExoLab Group, http://castor.exolab.org/, 2001.

[Fan et al. 2004] W. Fan, C. Chan, M. Garofalakis. “Secure XML querying with security views”, in SIGMOD 2004.

[Fan et al. 2005] W. Fan, J. X. Yu, H. Lu, J. Lu and Y. Zeng. “Query Translation from XPath to SQL in the Presence of Recursive DTDs”, in VLDB 2005.

[Florescu et al. 1998] A. Florescu, A.Y. Levy, and D. Suciu. “Query containment for conjunctive queries with regular expressions”, in PODS 1998

[Franc 2004] X. Franc. “Qizx/open version 0.4p1”, http://www.xfra.net/qizxopen/, 2004.

[Franceschet 2005] M. Franceschet. “XPathMark – An XPath benchmark for XMark”, Research report PP-2005-04, University of Amsterdam, 2005.

[Furfaro and Masciari 2003] F. Furfaro and E. Masciari. “On the minimization of XPath queries”, in VLDB 2003.

[Geneves et al. 2007] Pierre Geneves, Nabil Layaida and Alan Schmitt. “Efficient static analysis of XML paths and types”, in PLDI 2007: 342-351

[Gottlob et al. 2002] G. Gottlob, C. Koch and R. Pichler. “Efficient Algorithms for Processing XPath Queries”, in VLDB 2002.

[Gottlob et al. 2003a] G. Gottlob, C. Koch and R. Pichler. “The complexity of XPath query evalution”, in Proceedings of 22nd Symposium on Principles of Database Systems (PODS), San Diego, 2003.

[Gottlob et al. 2003b] G. Gottlob, C. Koch and R. Pichler. “XPath query evaluation: Improving time and space efficiency”, in ICDE, Bangalore, 2003.

[Gottlob et al. 2003c] Georg Gottlob, Christoph Koch and Reinhard Pichler. “XPath processing in a nutshell”, in ACM SIGMOD Record, 32(1):12-19, 2003.

[Groppe 2005] S. Groppe. “XML Query Reformulation for XPath, XSLT and XQuery”, Sierke-Verlag, Göttingen, Germany, 2005. ISBN 3-933893-24-0.

[Groppe and Groppe 2006a] J. Groppe and S. Groppe, “Filtering unsatisfiabile XPath queries”, In ICEIS 2006.

[Groppe and Groppe 2006b] J. Groppe, S. Groppe, “A prototype of a schema-based XPath satisfiability tester”, In DEXA 2006.

[Groppe and Groppe 2006c] J. Groppe and S. Groppe, “Satisfiability-test, rewriting and refinement of users’ XPath queries according to XML Schema definitions”, in ADBIS 2006.

[Groppe and Groppe 2006d] S. Groppe and J. Groppe. “Determining the Output Schema of an XSLT Stylesheet”, In ADBIS 2006, Thessaloniki, Hellas, September 2006.

[Groppe and Groppe 2008] J. Groppe and S. Groppe, “Filtering unsatisfiable XPath queries”, Data & Knowledge Engineering Journal, special issue with selected papers from the 8th International Conference on Enterprise Information Systems (ICEIS’ 2006), Volume 64, issue 1, pp.134-169, 2008. http://dx.doi.org/10.1016/j.datak.2007.06.018

[Groppe and Linnemann 2008] Jinghua Groppe, Volker Linnemann: Discovering Veiled Unsatisfiable XPath Queries. In ICEIS 2008, June 12 - 16, 2008, INSTICC, Barcelona, Spain.

[Groppe et al. 2008] S. Groppe, J. Groppe and S. Böttcher. “XPath Query Simplification with regard to the Elimination of Intersect and Except Operators”, in Data & Knowledge Engineering Journal, special issue with selected papers from the 3rd XML Data and Schema Management Workshop (XSDM 2006) in conjunction with IEEE ICDE 2006. Volume 65(2), pp. 198-222, 2008. http://dx.doi.org/10.1016/j.datak.2007.09.002.

[Groppe et al. 2007a] S. Groppe, J. Groppe and V. Linnemann. “How to Determine Output Schemas of XQuery Queries”, BNCOD Workshop on

Web Information Management (BNCODwebim'07), Glasgow, Great Britain, 2007.

[Groppe et al. 2006] S. Groppe, S. Böttcher and J. Groppe. "XPath Query Simplification with regard to the Elimination of Intersect and Except Operators", in XSDM 2006 in association with ICDE 2006.

[Groppe et al. 2006a] S. Groppe, S. Böttcher, G. Birkenheuer, and A. Höing. "Reformulating XPath Queries and XSLT Queries on XSLT Views", Journal Data & Knowledge Engineering (DKE), 2006.

[Hammerschmidt et al. 2005] B. C. Hammerschmidt, M. Kempa and V. Linnermann. "The index update problem for XML data in XDBMS", in ICEIS 2005.

[Hidders 2003] J. Hidders. "Satisfiability of XPath Expressions", in DBPL 2003. LNCS 2921, pp. 21–36.

[Hosoya et al. 2005a] Haruo Hosoya, Alain Frisch and Giuseppe Castagna. "Parametric polymorphism for XML", in The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 50-62, 2005.

[Hosoya et al. 2000] H. Hosoya, J. Vouillon, and B. C. Pierce. "Regular expression types for XML". In Proceedings of the International Conference on Functional Programming (ICFP), pages 11–22, Sep., 2000.

[Ioannidis 1996] Y. E. Ioannidis. "Query optimization", in ACM Computing Surveys, Vol. 28, No. 1, 1996.

[ISO Schematron 2006] ISO/IEC 19757. "Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation – Schematron". March 2006. http://www.schematron.com/.

[Jarke and Koch 1984] M. Jarke, and J. Koch. "Query Optimization in Database Systems", in ACM Computing Surveys, Vol. 16, No. 2, 1984.

[Jiang et al. 2003] H. Jiang, W. Wang, H. Lu and J. X. Yu. "Holistic twig joins on indexed XML documents", in VLDB 2003.

[Kay 2004] M. H. Kay. "Saxon - The XSLT and XQuery Processor". http://saxon.sourceforge.net. 2004.

[Kempa 2003] Martin Kempa. "Programmierung von XML-basierten Anwendungen unter Berücksichtigung der Sprachbeschreibung", PhD thesis, Institut für Informationssysteme, Universität zu Lübeck, 2003. Aka Verlag, Berlin, (in German).

[Kempa and Linnemann 2003a] Martin Kempa and Volker Linnemann. "The XOBE Project", in Proceedings of the First Hangzhou-Lübeck Conference

on Software Engineering (HL-SE'03), 1. - 2. November 2003, Hangzhou, P.R. China, S. 80-87.

[Kempa and Linnemann 2003b] Martin Kempa and Volker Linnemann. “Type Checking in XOBE”, in G. Weikum, H. Schöning, E. Rahm (Eds.): Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung, 26. - 28. Februar 2003, Leipzig, S. 227-246.

[Kwong and Gertz 2002] A. Kwong and M. Gertz. “Schema-based optimization of XPath expressions”. Techn. Report University of California, 2002.

[Lakshmanan et al. 2004] L. Lakshmanan, G. Ramesh, H. Wang and Z. Zhao. “On Testing Satisfiability of Tree Pattern Queries”, in VLDB 2004.

[Lee and Chu 2000] D. Lee and Wesley W. Chu. “Comparative Analysis of six XML schema lanuages”, in ACM SIGMOD Record archive Volume 29, Issue 3. September 2000.

[Levy at el. 1995] A. V. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. “Answering queries using views”, in PODS, San José, California, USA, 1995.

[Martens and Neven 2004] W. Martens and F. Neven. “Fronties of tractability for typechecking simple XML transformations”, in VLDB 2004.

[Microsoft 2001] Microsoft Corporation. “.NET Framework Developer’s Guide”. http://msdn.microsoft.com/library/default.asp. 2001.

[Miklau and Suciu 2004] G. Miklau, and D. Suciu. “Containment and Equivalence for a fragment of XPath”, in Journal of the ACM, Vol. 51, No. 1, January 2004, pp. 2-45.

[Milo and Suciu 1999] T. Milo, and d. Suciu. “Index Structures for Path Expressions”, in ICDT 1999: 277-295.

[Moeller 2002] Anders Moeller. “Document structure description 2.0”. http://www.brics.dk/DSD/dsd2.html. December 2002.

[Neven and Schwentick 2003] F. Neven and F. Schwentick. “XPath containment in the presence of disjunction, DTDs, and variables”, in Proceedings of 19th International Conference on Data Engineering (ICDE), 2003.

[OASIS 2001] Organization for the Advancement of Structured Information Standards (OASIS). “RELAX NG Specification Committee Specification”, 3 December 2001.

[Olteanu et al. 2002] D. Olteanu, H. Meuss, T. Furche and F. Bry. “XPath: looking forward”, XML-Based Data Management (XMLDM), EDBT Workshops 2002.

[Oracle 2001] Oracle Corporation. “Oracle9i, Application Developer’s Guide – XML”, Release 1 (9.0.1), Redwood City, CA 94065, USA, June 2001, shelley Higgins, Part Number A88894-01.

[Ramanan 2002] P. Ramanan. “Efficient algorithms for minimizing tree pattern queries”, in SIGMOD 2002.

[Rao and Moon 2004] P. Rao and B. Moon. “PRIX: indexing and querying XML using Prufer sequences”, in ICDE 2004.

[Shields und Meijer, 2001] Mark Shields and Erik Meijer. “Type-indexed rows”, in the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 261-275, London, Jan 2001.

[Schuhart 2006] Henrike Schuhart. “Design and Implementation of a Database Programming Language for XML-based Applications”. Dissertation zur Dr. rer.nat., Institut für Informationssysteme, Technisch-Naturwissenschaftliche Fakultät, Universität zu Lübeck, Juni 2006. DISDBIS 97, Akademische Verlagsgesellschaft Aka GmbH, Berlin 2007.

[Schuhart et al. 2006] Henrike Schuhart, Beda Christoph Hammerschmidt and Volker Linneman. “Integrating Statically Typechecked XML Data Technologies Into Pure Java Architectures”, in Proceedings of the ISCA 15th International Conference on Software Engineering and Data Engineering (SEDE-2006), Los Angeles, California USA, July 6-8, 2006, S. 217-222.

[Schwentick 2004] T. Schwentick. “XPath query containment”, in SIGMOD Record 33 (1): 101-109. 2004.

[Sun 2001] Sun Microsystems, Inc. “Java 2 Platform, Standard Edition, v 1.3.1, API Specification”. http://java.sun.com/j2se/1.3/docs/api/index.html. December 2001.

[Trier 2007] University of Trier, Computer Science Bibliographie. http://dblp.uni-trier.de/. Accessed 17 July 2007.

[UniProtKB 2008a] The UniProtKB/SwissProt group. SwissProt Knowledgebase in XML format. ftp://ftp.expasy.org/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.xml.gz. Accessed in 25 January 2008.

[UniProtKB 2008b] The UniProtKB/TrEMBL group. Trembl Knowledgebase in XML format. ftp://ftp.expasy.org/databases/uniprot/current_release/knowledgebase/complete/uniprot_trembl.xml.gz. Accessed in 25 January 2008.

[UniRef 2008a] UniRef database. UniRef50 in XML format. ftp://ftp.expasy.org/databases/uniprot/uniref/uniref50/uniref50.xml.gz. Accessed in 25 January 2008.

[UniRef 2008b] UniRef database. UniRef90 in XML format. ftp://ftp.expasy.org/databases/uniprot/uniref/uniref90/uniref90.xml.gz. Accessed in 25 January 2008.

[UniRef 2008c] UniRef database. UniRef100 in XML format. ftp://ftp.expasy.org/databases/uniprot/uniref/uniref100/uniref100.xml.gz. Accessed in 25 January 2008.

[Vlist 2001] Eric van der Vlist. "Comparing XML Schema Languages". Published on XML.com, http://www.xml.com/pub/a/2001/12/12/schemacompare.html. Accesed 26 November 2007.

[Vlist 2003] Eric van der Vlist. "Examplotorn". http://examplotron.org/. February 2003.

[Wadler 2000] P. Wadler. "Two semantics for XPath". Internal Technical Note of the W3C XSL Working Group, 2000. http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf.

[Wadler 1999] P. Wadler. "A formal semantics of patterns in XSLT". In Proceedings of Markup Technologies, Philadelphia, PA, USA, 1999.

[Wallace und Runciman, 1999] Malcolm Wallace and Colin Runciman. "Haskell and XML: Generic combinators or type-based translation?", in Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP 1999), volume 34-9 of ACM Sigplan Notices, pages 148-159, N.Y., September 27-29 1999. ACM Press.

[Wang et al. 2003] H. Wang, S. Park, W. Fan and P. S. Yu. "ViST: a dynamic index method for querying XML data by tree structures", in SIGMOD 2003.

[Washington 2008] University of Washington, XML data repository. http://www.cs.washington.edu/research/xmldatasets/www/repository.html. Accessed 25 January 2008.

[Wikipedia 2007] Wikipedia, "the freeencyclopedia: XML schema language comparison". http://en.wikipedia.org/wiki/XML_Schema_Language_Comparison. Accessed 26 November 2007.

[Wood 2003] P. T. Wood. "Containment for XPath fragments under DTD constraints", in ICDT, Siena, Italy, 2003.

[Wood 2001] P. T. Wood. "Minimising simple XPath expressions", in 4th International Workshop on Web and Databases (WebDB), 2001.

[Wood 200] P. T. Wood. “On the equivalence of XML patterns”, in LNCS 1861, pages 1152-1166. Springer, 2000.

[W3C HTML4.01 1999] World Wide Web Consortium (W3C). “HTML 4.01 Specification”. W3C Recommendation 24 December 1999.

[W3C XHTML1.0 2002] World Wide Web Consortium (W3C). “XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)”. A Reformulation of HTML 4 in XML 1.0. W3C Recommendation 26 January 2000, revised 1 August 2002.

[W3C Schema0 2004] W3C. “XML Schema part 0: Primer second edition”, W3C Recommendation, www.w3.org/TR/xmlschema-0, October 2004.

[W3C Schema1 2004] W3C. “XML Schema part 1: Structures second edition”, W3C Recommendation, www.w3.org/TR/xmlschema-1, 2004.

[W3C Schema2 2004] W3C. “XML Schema part 2: Datatypes second edition”, W3C Recommendation, www.w3.org/TR/xmlschema-2, 2004.

[W3C XForms1.0 2007] W3C. “XForms 1.0 (Third Edition)”. W3C Recommendation 29 October 2007. http://www.w3.org/TR/xforms/

[W3C XLink1.0 2001] W3C. “XML Linking Language (XLink) Version 1.0”. W3C Recommendation 27 June 2001, http://www.w3.org/TR/xlink/.

[W3C XML1.0 2004] W3C. “Extensible Markup Language (XML) 1.0 (Third Edition)”. W3C Recommendation, www.w3.org/TR/REC-xml, 2004.

[W3C XML-data 1998] W3C. “XML-data”. W3C Note 05 Jan 1998. http://www.w3.org/TR/1998/NOTE-XML-data/.

[W3C XPath1.0 1999] W3C. “XPath version 1.0”. W3C Recommendation. www.w3.org/TR/xpath/, 1999.

[W3C XPath2.0 2003] W3C. “XPath Version 2.0”. W3C Working Draft. www.w3.org/TR/xpath20/. 2003.

[W3C XPointer1.0, 2001] W3C. “XML Pointer Language (XPointer) Version 1.0”. W3C Last Call Working Draft 8 January 2001. http://www.w3.org/TR/WD-xptr.

[W3C XQuery1.0, 2004] W3C. “XQuery 1.0: An XML Query Language”. W3C Working Draft, 2004.

[W3C XQuery1.0-XPath2.0, 2004] W3C. “XQuery 1.0 and XPath 2.0 Formal Semantics”. W3C Working Draft, http://www.w3c.org/TR/xquery-semantics/, 2004.

[W3C XSLT1.0, 1999] W3C. “XSL Transformations (XSLT) Version 1.0”. W3C Recommendation, http://www.w3.org/TR/1999/REC-xslt-19991116, 1999.

[W3C XSLT2.0, 2003] W3C. “XSL Transformations (XSLT) Version 2.0”, W3C Working Draft. http://www.w3.org/TR/2003/WD-xslt20-20031112/, 2003.

[Xu and Özsoyoglu 2005] W. Xu and Z.M. Özsoyoglu. “Rewriting XPath queries using materialized views”, Trondheim, Norway. In VLDB 2005.

[Yannakakis 1981] M. Yannakakis. “Algorithms for acyclic database schemes” in VLDB 1981.

Publications

My recent publications contain:

Journal papers

- Jinghua Groppe, Sven Groppe: Filtering Unsatisfiable XPath Queries. In the international journal: *Data & Knowledge Engineering Journal (DKE)*, special issue with selected papers from *the 8th International Conference on Enterprise Information Systems (ICEIS 2006)*, Volume 64, issue 1, p. 134 to p. 169, 2008. http://dx.doi.org/10.1016/j.datak.2007.06.018.

- Sven Groppe, Jinghua Groppe, Stefan Böttcher: Simplifying XPath Queries for Optimization with regard to the Elimination of Intersect and Except Operators. In the international journal: *Data & Knowledge Engineering Journal (DKE)*, special issue with selected papers *from the 3rd XML Data and Schema Management Workshop (XSDM 2006)* in conjunction with *IEEE ICDE 2006*. Volume 65, issue 2, p. 198 to p. 222, 2008. http://dx.doi.org/10.1016/j.datak.2007.09.002.

- Sven Groppe, Jinghua Groppe: Output Schemas of XSLT Stylesheets and their Applications. Accepted for publication in the international journal: *Information Sciences Journal*, 2008. http://dx.doi.org/10.1016/j.ins.2008.06.024.

- Sven Groppe, Jinghua Groppe, Stefan Boettcher, Thomas Wycisk, Le Gruenwald: Optimizing the Execution of XSLT Stylesheets for Querying Transformed XML Data. Accepted for publication in *the international journal: Knowledge and Information Systems (KAIS)*, 2008. http://dx.doi.org/10.1007/s10115-008-0144-4.

Conference/Workshop papers

- Jinghua Groppe, Volker Linnemann: Discovering Veiled Unsatisfiable XPath Queries. In Proceedings of the *10th International Conference on Enterprise Information Systems (ICEIS 2008)*, June 12 - 16, 2008, INSTICC, Barcelona, Spain.

- Sven Groppe, Jinghua Groppe, Volker Linnemann, Dirk Kukulenz, Nils Hoeller, Christoph Reinke: Embedding SPARQL into XQuery / XSLT. In *23rd ACM Symposium on Applied Computing (ACM SAC 2008)*, Fortaleza, Ceara, Brasilien, 2008.

- Matthias Droop, Markus Flarer, Jinghua Groppe, Sven Groppe, Volker Linnemann, Jakob Pinggera, Florian Santner, Michael Schier, Felix Schoepf, hannes Staffler, Stefan Zugal: Embedding XPATH Queries into SPARQL Queries. In Proceedings *of the 10th International Conference on Enterprise Information Systems (ICEIS 2008)*, June 12 - 16, 2008, INSTICC, Barcelona, Spain, Copyright: INSTICC

- Sven Groppe, Jinghua Groppe, Dirk Kukulenz, Volker Linnemann: A SPARQL Engine for Streaming RDF Data. In *3rd International Conference on Signal-Image Technology & Internet-Based Systems (SITIS 2007)*, Shanghai, China, 2007. This paper received an honorable mention at the SITIS'07 Conference.

- Matthias Droop, Markus Flarer, Jinghua Groppe, Sven Groppe, Volker Linnemann, Jakob Pinggera, Florian Santner, Michael Schier, Felix Schöpf, Hannes Staffler and Stefan Zugal: Translating XPath Queries into SPARQL Queries. In *On the Move (OTM 2007) Federated Conferences and Workshops (DOA, ODBASE, CoopIS, GADA, IS), 6th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007)*, Vilamoura, Algarve, Portugal, 2007.

- Sven Groppe, Jinghua Groppe and Volker Linnemann: How to Determine Output Schemas of XQuery Queries. In *BNCOD Workshop on Web Information Management (BNCODwebim '07)*, Glasgow, Great Britain, 2007.

- Sven Groppe, Jinghua Groppe, Volker Linnemann: Using an Index of Precomputed Joins in order to Speed Up SPARQL Processing. In *9th International Conference on Enterprise Information Systems (ICEIS 2007)*, Funchal, Portugal, 2007.

- Jinghua Groppe, Sven Groppe: Satisfiability-Test, Rewriting and Refining of Users' XPath Queries according to XML Schema Definitions. In *10th East-European Conference on Advances in Databases and Information Systems (ADBIS 2006)*, Thessaloniki, Hellas, September 2006.

- Sven Groppe, Jinghua Groppe: Determining the Output Schema of an XSLT Stylesheet. In *10th East-European Conference on Advances in Databases and Information Systems (ADBIS 2006)*, Thessaloniki, Hellas, September 2006.

- Jinghua Groppe, Sven Groppe: A Prototype of A Schema-based Satisfiability Tester. In *17th International Conference on Database and Expert Systems Applications (DEXA 2006),* Krakow, Poland, September 2006.

- Jinghua Groppe, Sven Groppe: Filtering Unsatisfiable XPath Queries. In *8th International Conference on Enterprise Information Systems (ICEIS 2006)*, Paphos-Cyprus, May 2006.

- Sven Groppe, Stefan Boettcher, Jinghua Groppe: XPath Query Simplification with regard to the Elimination of Intersect and Except Operators. In *3rd International Workshop on XML Schema and Data Management (XSDM'06)*. (In conjunction with *IEEE ICDE 2006*) Atlanta, USA, April 2006.

- Sven Groppe, Jinghua Groppe, Stefan Böttcher and Marc-André Vollstedt: Shifting Predicates to Inner Sub-Expressions for XQuery Optimization. In *the International Conference on Signal-Image Technology & Internet-Based Systems (SITIS 2006)*, Hammamet, Tunisia, 2006.

- Matthias Becker, Sven Groppe, Jinghua Groppe, EasyFormFill: A Mapping-Based Application to Automatically Fill HTML-Forms. In *Semantics 2006*, Vienna, Austria, 2006.

- Ralf Bettentrupp, Sven Groppe, Jinghua Groppe, Stefan Boettcher and Le Gruenwald: A prototype for translating XSLT into Xquery. In *8th International Conference on Enterprise Information Systems (ICEIS 2006)*, Paphos-Cyprus, May 2006.

- Jinghua Groppe, Sven Groppe: Automated Profile Mapping. In *the IASTED International Conference on Databases and Applications (DBA 2006)*, Innsbruck, Austria, 2006.

- Jinghua Groppe, Wolfgang Mueller: Profile Management Technology for Smart Customizations in Private Home Applications. In *the 6th International workshop on Database and Expert Systems Applications*, Copenhagen, Denmark, August 2005.

- Wolfgang Mueller, Jinghua Wang: Javacard-enabled Smart Cards for Collaborative Engineering Environments. In *the Workshop on Challenges in Collaborative Engineering*, Poznan, Poland, April 2003.

Index

O

P

Q

R

S

U

V

X