XPath Query Satisfiability and Containment under DTD Constraints

A Dissertation Submitted to

Birkbeck, University of London

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

by

Manizheh Montazerian

Department of Computer Science

Birkbeck College

Malet Street, Bloomsbury, London WC1E 7HX

JANUARY 16, 2014

I certify that this thesis, and the research to which it refers, are the product of my own work, and that any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Signed:

To my loving sons Sepehr, Sina and Parsa

and

my dear husband Rasoul

Abstract

In this thesis, we consider the XML query language XPath, along with XML documents whose integrity constraints are presented in the form of document type definitions (DTDs). In particular, we study the problems of XPath satisfiability and XPath containment in the presence of DTDs. The motivation for studying XPath is that it is the main language for navigating in and extracting information from XML documents. The motivation for studying DTDs, as opposed to some other newer XML schema formalism, is that DTDs are well-known, stable, and well-tested. Furthermore, they are still widely-used in various domains.

The major contributions of the thesis can be classified into those on satisfiability and those on containment of XPath queries under DTDs. With respect to the satisfiability problem, we show that the XPath satisfiability problem for the fragment $XP^{\{/,[]\}}$ is NP-hard in general. In order to study whether this worst-case behaviour arises often in practice, we investigate real-world DTDs and discover that the majority of them satisfy a property we called the *covering* property. We then show that XPath satisfiability for the fragment $XP^{\{/,[],*,//,\cup\}}$ is in PTIME under covering DTDs. We also show that it is decidable in PTIME for duplicate-free DTDs (a property introduced in [91]), which also occur often in practice.

Despite the positive results for the satisfiability problem, we prove that XPath containment under covering DTDs for $XP^{\{/,[]\}}$ is still coNP-hard. However, we define a class of DTDs, called *well-behaved* DTDs, under which containment for $XP^{\{/,[]\}}$ is tractable

provided that certain constraints inferred from the DTD are given. These constraints are modified forms of previously defined sibling constraints and functional constraints. Finally, we show that, given a set of such constraints, containment of queries in $XP^{\{/,[]\}}$ under a special case of well-behaved DTDs, called *well-formed DTDs*, is tractable. Well-formed DTDs also arise frequently in practice.

Acknowledgements

I would like to thank my supervisor, Dr. Peter Wood, who has greatly helped me during my Ph.D. studies. His support has been far beyond the usual limit of the support expected from a supervisor. I can claim that he has been even more caring than me about my studies. I extremely appreciate his help and support, and I wish him a life full of goodness and success.

I thank my children Sepehr, Sina and Parsa who tolerated difficulties and hardship stemming from my study. Finally, I would like to thank Rasoul for his kindness, love and support, and everyone who either directly or indirectly helped me doing my Ph.D.

Manizheh Montazerian

Table of contents

List of Figures

List of Tables

Table 1: List of symbols

Σ	alphabet
Σ^a	alphabet appearing in R^a
R^a	content model of element a
$\emptyset$	emptyset
ε	empty string
$\delta_B R$	derivation of R with respect to bag B
$[w]$	bag of symbols appearing in string w
$\{w\}$	set of symbols appearing in string w
$\lambda(v)$	label of node v
$root(t)$	root node of tree t
$[R]$	unordered regular expression
$\overline{R}$	covering regular expression
$\cup$	max union
$\uplus$	additive union
$\|b\|_A$	multiplicity of symbol b in bag A
$v(X)$	node v with bag of children X
$\mathbb{C}$	set of constraints
$\mathbb{C}^+$	closure of $\mathbb{C}$
X^+	BSC bag closure of X

Chapter 1

Introduction

The eXtensible Markup Language (XML) is one of the major data storage formats used within both databases and the World Wide Web [15]. It is a recommendation issued by the World Wide Web Consortium (W3C), an organization aimed at promoting the infrastructure of the Web. The explosive growth of dynamic applications, such as e-commerce and e-learning, in recent years has remarkably increased the number of organizations that use XML to exchange data. Over the past years, XML also became popular for general-purpose platform-independent data exchange [14, 47, 79].

An XML document is modelled as a rooted tree called an XML tree. The labels in an XML tree are called XML tags, which belong to an infinite alphabet. Every node in the tree corresponds to an element; the root of the tree corresponds to the root element of the document. Figure 1.1 displays an instance of an XML tree and its corresponding document.

Two standards associated with XML technology are *XML schemas* and the *XPath language*, which are introduced next.

In contrast to relational databases, XML documents are self-describing and do not necessarily require any schema or type system. However, there are usually logical relations, or constraints, among XML tags. Such constraints constitute a so-called XML schema,

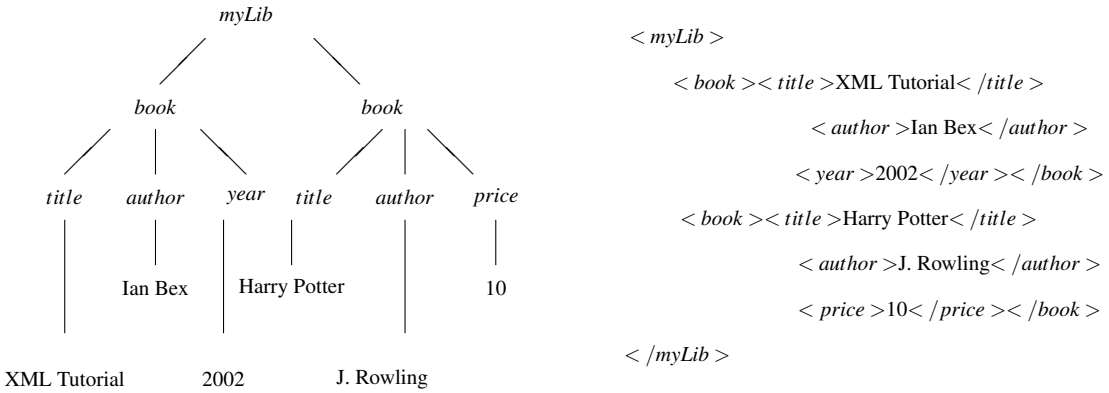
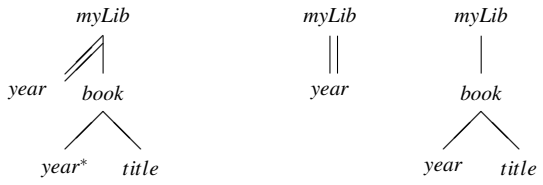


Figure 1.1: An instance of an XML tree and its corresponding document.

which could be represented in many ways. In other words, XML documents may be constrained by a schema. Exploiting the schema associated with an XML document provides the possibility for more efficient query processing. Several schema formalisms have emerged, among which are those recommended by the W3C, which include Document Type Definitions (DTDs) [26] and the XML Schema Definition Language [15], and more formal language-based schema systems, such as RELAX NG [30] and Schematron [49]. Schema languages are either *grammar-based* or *rule-based* languages. In the former, the language is created based on a context-free grammar and according to top-down production rules in a specified formalism. DTD, XML Schema, and Relax-NG belong to this group. In rule-based languages, the rules that XML document must satisfy are specified. Schematron belongs to this category. The specification provided by a schema language is either open, meaning all that is not forbidden is allowed, or closed, meaning all that is not allowed is forbidden. In Chapter 2, some of the well-known schema languages are explained in more detail.

XPath [9] is a W3C standard for navigating a document tree and selecting a set of nodes for processing. XPath expressions are a core component of XML transformers like XSL(T) [8, 28] and other query languages such as XQuery [13]. An expression of XPath can be interpreted as a tree pattern query. A tree pattern query is a tree where nodes are element names and the hierarchical relationships between nodes are speci-



q q_1 q_2

Figure 1.2: Tree patterns for q, q_1 and q_2.

fied by child ('/') and descendant ('//') edges. For example, let q be the expression `myLib[.//year]/book[title]/year` presented as the tree pattern shown in Figure 1.2, where double-lines represent descendant edges and single-lines show child-edges. The node marked with an asterisk "$*$" is a *result node* which corresponds to the output of the query. Each XPath expression belongs to an XPath fragment indicated by listing the allowed operators. For instance, q belongs to $\mathrm{XP}^{\{/,[],//\}}$ which is an XPath fragment where only descendant, child and filter ('[]') operators are allowed. In Chapter 2, XPath fragments are explained in more detail.

XPath is the main XML selection language. Indeed, XPath is used as a sublanguage in other XML languages to specify the documents in which users are interested. If an XPath expression matches a document or a part of a document, that document or part is returned to the user. Although, for every expression e and every document d, the test whether e matches d is not efficient, there is a partial order on expressions which may simplify the test. More clearly, for some expressions p and q, it might hold that whenever a document matches p, it also matches q. This leads to the problem of *XPath query containment*. More formally, the problem of query containment is, for two XPath expressions p and q, to decide whether every document matching p also matches q (denoted by $p \subseteq q$). If the answer to this decision problem is positive, then given that a document matches p, there is no need to check for whether it matches q. Correspondingly, if we know that q does not match a document then p will not match it either.

Algorithms for XPath query containment are important in several contexts. XPath expressions are used as basic patterns in other XML languges like XQuery (to bind vari-

ables), XSLT (to match expressions), XLink and XPointer (to reference elements). In every such context an instance of the containment problem is present (for more detail, see Chapter 2). In addition, as XPath is used to define keys in the XML Schema Definition Language, understanding the key inference problem requires an understanding of the problem of XPath containment. For example, if an expression p represents a key for a given schema, then every expression p_1 such that $p_1 \subseteq p$ is also a key.

Further, the problem of XPath query containment, together with the closely related problem of *XPath query equivalence* (two-way containment), may be viewed as a core problem of *query optimisation*. For example, consider the query q shown in Figure 1.2. The figure shows the graphical tree patterns of q and its two sub-patterns, q_1 and q_2. It is easy to verify that $q_2 \subseteq q_1$. This implies that every document that matches q_2 also matches q_1. Therefore, the sub-pattern q_1 of q is redundant and can be eliminated. This means that q may be reduced to q_2.

In the above containment analysis, no schema was considered. Suppose now that there is a schema associated with the documents being queried. The existence of the schema affects the containment problem in that, while a query p may not contain a query q in general, it may do so when only documents valid with respect to the schema are considered. That is because not all of the documents matching q preserve (the constraints implied by) the schema, and those which do so will certainly match p as well. In this thesis, we study containment in the presence of DTD constraints (DTD-containment for short). Given two XPath queries p and q and a DTD D, deciding whether p is contained in q under D, which is called D-containment and denoted by $p \subseteq_D q$), is the problem of deciding whether, for every XML document $d \in SAT(D)$, the output of p on d is contained in the output of q on d. Similar to [91], we use $SAT(D)$ to denote the set of all document trees satisfying D.

Example 1.1 *Consider the following DTD rule:*

```
<!ELEMENT book (title, author+, year?, price?)>
```

According to this DTD rule, in every document which satisfies the DTD rule, each `book` *element should have one* `title` *child. This means, for example, that* `myLib/book/price` $\subseteq_D$ `myLib/book[title]/price`.

A related problem studied in this thesis is the *XPath satisfiability problem*, which is to decide, given a query q, whether or not the evaluation of q returns a non-empty result for some input document. It can be shown that the XPath satisfiability problem is (the negation of) a special case of the containment problem, where the containing XPath expression returns an empty set on any document tree (for more detail, see Chapter 5) [7]. The XPath satisfiability problem can be used in query optimisation to avoid the submission, and consequently redundant computation, of *unsatisfiable queries*. Thus, applying the satisfiability test before executing a query can save processing time and query costs.

1.1 Contributions

XPath D-containment and XPath D-satisfiability are two computationally hard problems, unless $P = NP$. Solving these problems requires analysis of DTD content models. Each content model specifies, for each label, a set of sequences of its children by a regular expression. In this thesis, our main goal is to find certain cases for which D-containment and D-satisfiability become tractable. Our approach is to analyse the problem and determine which (undesirable) features of regular expressions cause the high (apparently exponential) complexity in processing. Then, we define a reasonably limited form of DTDs, with emphasis on characteristics of real-world DTDs, to avoid the undesirable features.

One undesirable feature which increases the complexity is the existence of the disjunction operator in the regular expressions. The complexity of deciding XPath D-satisfiability is reduced when DTDs are disjunction-free. However, the disjunction-freeness requirement is very restrictive from the practical point of view. We define the covering property of DTDs which is less restrictive than the disjunction-free property but can still reduce the

complexity of deciding *D*-satisfiability. The tractability results of disjunction-free DTDs are inherited by covering DTDs. Another undesirable feature is duplicate labels in the DTD content models. Previously, duplicate-freeness was proposed to prevent duplicate labels, but this feature is also very restrictive. Our solution is to define a superset of duplicate-free DTDs that semantically limits each query such that each node either has at most one child with some label or can have any number of children with that label. We call such DTDs *well-behaved DTDs*, formally defined in Chapter 6.

The following are the main contributions of this thesis:

1. We have defined an operation on a regular expression *R*, yielding a new regular expression called the *derivative* of R. The notion of the derivative of a regular expression had been previously introduced [17], but in that case was concerned with *sequences* of symbols. We define the derivative of a regular expression with respect to a *bag* of symbols, which is tailored to the needs of XPath containment and satisfiability. We have devised an algorithm to compute the derivative of a regular expression with respect to a bag, but have proved that computing the derivative of a regular expression with respect to a bag of symbols is NP-hard in general. These results are provided in Chapter 4.

2. We have proposed a method based on the derivatives of regular expressions to reduce the problem of extracting sibling constraints to the problem of extracting child constraints from DTDs. We have presented a PTIME algorithm to extract child constraints from DTDs. Another method is also presented to extract functional constraints from DTDs using derivatives of regular expressions. These results are presented in Chapter 4.

3. We introduce, in Chapter 4, two types of constraints called *Bag Sibling Constraints (BSCs)* and *Bag Functional Constraints (BFCs)*. Later, in Chapter 6, we define a new DTD property, called *well-behaved* and prove that BSCs and BFCs are nec-

essary and sufficient to capture *D*-containment of queries in $XP^{\{/,[]\}}$ under *well-behaved* DTDs. Finally, we show that, given a set of BSCs and BFCs, *D*-containment of queries in $XP^{\{/,[]\}}$ under a special case of well-behaved DTDs is tractable.

4. We have examined real-world DTDs, using Google search and DTD benchmarks available on the web, and discovered a new property, called the *covering* property, which most of them satisfy. We also observed that the minority of the examined real DTDs which did not possess the covering property were duplicate-free. The notion of a duplicate-free DTD was introduced in [91]. It was redefined and referred to as SOREs (Single Occurrence Regular Expressions) in [10], where it was shown that SOREs capture by far the majority of regular expressions occurring in practical DTDs. Our investigation of real-world DTDs, which led to the discovery of the prevalent property of covering, revealed the fact that, for many real-world cases, the XPath satisfiability problem can be solved in PTIME. These results are provided in Chapter 5.

5. The XPath satisfiability problem for the fragment $XP^{\{/,[]\}}$, denoted by SAT($XP^{\{/,[]\}}$), is NP-hard in general. This result follows from a result in [91]. In Chapter 5, we show that it is decidable in PTIME for duplicate-free DTDs, although results from [7] imply that it remains NP-hard for the fragments $XP^{\{/,[],*\}}$ and $XP^{\{/,[],//\}}$. More significantly in this chapter, we show that XPath satisfiability for the fragment $XP^{\{/,[],*,//,\cup\}}$, i.e. SAT($XP^{\{/,[],*,//,\cup\}}$), is in PTIME for covering DTDs. As another result, we have proved that, by combining the methods for covering and duplicate-free DTDs, SAT($XP^{\{/,[]\}}$) can be decided in PTIME if each rule has at least one of the covering or duplicate-free properties.

6. We define, in Chapter 6, a class of regular expressions called *well-formed* for which the computation of derivatives is tractable, and show that this class arises commonly in XML DTDs. This class is a sub-class of well-behaved expressions and a super-

class of duplicate-free expressions. Wood in [92] showed that *D*-containment under duplicate-free DTDs for $XP^{\{/,[]\}}$ is in PTIME. We show that it is also in PTIME under well-formed DTDs. We also use the results in [7] to show that the problem is in coNP-hard for the fragments $XP^{\{/,//\}}$, $XP^{\{/,[],*\}}$, and $XP^{\{/,[],\cup\}}$ even if the queries are duplicate-free.

1.2 Structure of the thesis

This thesis is organised as follows. Chapter 2 presents an introduction to XML and its related concepts including XPath, XML query languages, and XML schemas.

In Chapter 3, we survey previous work on XPath containment and XPath satisfiability. This provides an overview of previous results on XPath containment, including proposed techniques and their complexities, and the results on XPath satisfiability.

In Chapter 4, we introduce the derivative of a regular expression with respect to a *bag* of symbols. This chapter also covers the various types of constraints inferred from DTDs which have been previously defined in the literature. Two new types of DTD constraints, together with their properties, are defined, and some algorithms to extract constraints from DTDs based on the derivatives of regular expressions are proposed.

In Chapter 5, we concentrate on the satisfiability problem under two special classes of DTDs, covering and duplicate-free DTDs, for a variety of XPath sub-fragments of $XP^{\{/,[],*,//,\cup\}}$.

Chapter 6 deals with introducing a new DTD property, called *well-behaved* and two types of constraints, BSCs and BFCs, which are necessary and sufficient to capture *D*-containment of queries in $XP^{\{/,[]\}}$under *well-behaved DTDs*. We also show that *D*-containment of queries in $XP^{\{/,[]\}}$under a special case of well-behaved DTDs, called *well-formed*, is tractable provided that the set of constraints is given.

The last chapter of this thesis, Chapter 7, summarises the thesis and introduces some possible avenues for future work.

Chapter 2

Overview of XML, Query Languages, and Schemas

This chapter gives an introduction to XML and some of its dependent concepts, namely XPath, XML query languages, and XML schemas.

2.1 XML

XML [15], which stands for eXtensible Markup Language, is a framework for defining and using markup languages. The primary purpose of introducing markup languages is to facilitate the sharing of structured data across different information systems, particularly via the Internet. They provide facilities to describe data formats, data types, data linking, data transfer, and data processing [2, 70]. Markup languages are used for creating units of information called XML documents. An XML document consists of textual data and markup. The markup indicates the syntactical structure of the document.

XML documents are organised into elements. An element starts from a start tag and ends at an end tag, including the tags themselves. Any attributes are found in the start tag. An element's content (which could be just text) lies between the tags. Each document has a root element that is unique and the ancestor of all the other elements. As an example,

Figure 1.1 shows an XML document.

XML usually uses an XML schema to describe the underlying data. XML without a schema is designed to be self-descriptive. Essentially, schemas help XML developers to describe the structure and data within their XML documents. Validating an XML document is the process of verifying whether it conforms to the set of structural and content rules expressed in its associated schema. For example, if a document contains an undefined element, then it is not valid.

A number of different schema languages for XML documents have been proposed in the past. In Section 2.4, we give an overview of DTD, XML-Schema, Relax-NG, EDTD, and Schematron, which are among the best-known schema languages.

2.2 Syntax of XPath

XPath [9, 27] is a simple query language, which allows for querying and navigation of XML documents. XPath expressions are usually embedded in other high-level XML related technologies such as XPointer [32], XQuery [13], and XSL Transformations [28]. XPath queries are formulated using references to various XML structural elements, such as elements and attributes. They also provide basic facilities for manipulation of strings, numbers, and Booleans.

Below we describe only the subset of XPath studied in this thesis. The primary syntactic construct in XPath is the *expression*. One important kind of expression is a *location path step*. Every location path step can be expressed using a straightforward syntax. A location path step l is an expression of the form $axis :: nodeTest\ [exp]*$, which is evaluated with respect to a *context node*. In the step l, *axis* refers to one of the XPath axes. XPath has 13 axes among which we only consider the four most frequently used ones, namely the *self*, *child*, *descendant* and *descendant-or-self* axes. Each axis specifies the relationship between the context node and the nodes to be selected next. Further, *nodeTest* is one of $node()$, a tag name, or a wildcard "$*$". The syntax "[]" denotes a *filter* which filters the

results based on the criteria enclosed within "[]"; $[exp]*$ is a (possibly empty) sequence of filters in which *exp* is an expression whose syntax is given by the following rule:

$$exp ::= l \mid l/exp \mid exp \cup exp \mid /exp.$$

A *location path* selects a set of nodes relative to the context node.[1] There are two kinds of location path: *relative* and *absolute* location paths [27]. A relative location path consists of a sequence of one or more location steps separated by "/". An initial sequence of steps selects a set of nodes relative to a context node. Then, each node in the selected set is used as a context node for the subsequent step. The sets of nodes which are identified by that step are unioned together. Finally, the result is the set of nodes identified by the composition of the steps. For example, `child::book/child::author` selects the `author` element children of the `book` element children of the context node, or, in other words, the `author` element grandchildren that have `book` parents.

An absolute location path consists of "/" optionally followed by a relative location path. A "/" by itself selects the root node of the document as the context node. If it is followed by a relative location path, then the location path selects the set of nodes that would be selected by the relative location path, relative to the root node of the document.

As mentioned before, XML data can be modelled as a forest of ordered trees, where each node corresponds to an element and the edges represent element-subelement relationships. XPath queries, which are based on the structural composition of an XML document, can be modelled as *unordered tree patterns* [25, 6]. A tree pattern p is specified by a set of nodes labeled with symbols in $\Sigma \cup \{*\}$, a set of edges which are either descendant-edges shown as double lines or child-edges shown as single lines, and a k-tuple of nodes called *result nodes* denoted by $result(p)$. The integer k is called the arity of p.

Example 2.1 *Consider the following XPath expression*

[1] In this thesis, we use the same semantics as [84] and fix the root node as the context node.

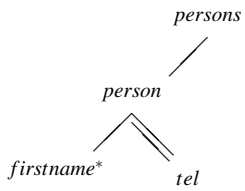


Figure 2.1: The tree pattern corresponding to the XPath expression in Example 2.1.

```
/persons/person[.//tel]/firstname
```

The tree pattern representation of this expression is shown in Figure 2.1. The arity is one (i.e. $k = 1$), and the double line connectors represent the descendant axis, whereas the single line connectors represent the child axis. The node labeled with `firstname` *is the result node (marked with an asterisk "$*$"), and the path from the root to the result node is called the* distinguished path. *A distinguished path is only well-defined for a tree pattern with a single result node.*

The following notation is adopted from [92] and [65]. Let p be a tree pattern, with result node x, and t be a tree in T_Σ. The result of evaluating p on t, denoted by $p(t)$, is defined as:

$$p(t) = \{(h(x)) \mid h \text{ is a homomorphism from the nodes of } p \text{ to the nodes of } t\}$$

where h is defined as follows:

- $h(root(p)) = root(t)$
- h preserves labels: for each node v in p, $h(v)$ has the same label as v unless v carries a wildcard
- h preserves child edges: for each pair of nodes u, v in p, if (u,v) is a child edge in p then $(h(u), h(v))$ is a child edge in t.
- h preserves descendant edges: for each pair of nodes u, v in p, if (u,v) is a descendant edge in p then $h(u)$ is an ancestor of $h(v)$ in t.

A pattern p is called a *Boolean pattern* if its arity is zero. For a Boolean pattern p and tree t, $p(t)$ is true if there is a homomorphism from p to t; it is false otherwise.

It has been shown that, for the problem of containment, it is sufficient to limit the patterns to Boolean tree patterns [65]. In particular, every k-ary tree pattern can be translated to a Boolean tree pattern such that for any k-ary patterns p, q and their corresponding translations $p\prime$, $q\prime$, $p \subseteq q$ iff $p\prime \subseteq q\prime$. Similar to [92], we consider XPath queries as *tree pattern queries* and in the rest of the thesis, we consider only Boolean tree patterns.

A *containment mapping* between two tree patterns is similar to homomorphism from a query pattern to a tree [92].

In this thesis, we study XPath expressions that use various subsets of the following operators: "/", "//", "[]", "$*$", and "$\cup$", which we refer to as child, descendant, filter, wildcard, and union, respectively. Each XPath fragment is indicated by listing the allowed operators, as proposed in [65]. For instance, $\mathrm{XP}^{\{/,[],//\}}$denotes the XPath fragment where only child, descendant, and filter are allowed.

2.3 XQuery and other XML languages

In this section, we very briefly introduce three other XML languages recommended by W3C: XQuery, XSLT, and XPointer.

2.3.1 XQuery

XQuery is a query language for XML, defined by W3C [13]. XQuery is a strongly typed functional language, which supports the common processing, transformation, and querying tasks of XML applications. Informally speaking, the relation between XQuery and XML is similar to that between SQL and relational databases. XQuery provides the means to extract and manipulate data from XML documents or any data source that can be viewed as XML, such as relational databases or office documents [72].

XPath expressions are basic patterns used in XQuery for navigation and extracting fragments of XML documents. Optimising XPath expressions hence eliminates unnecessary operations such as tree navigation and reduces the cost of the evaluation process. One way to check whether an XPath query q_1 can be optimised to a simpler XPath query q_2 is to check whether q_1 contains q_2 and q_2 contains q_1. For this reason, some optimisation algorithms are based on the containment problem [91]. The containment problem for XPath is studied in this thesis.

2.3.2 Extensible Stylesheet Language Transformations (XSLT)

The main purpose of using XSLT is to transform an XML document either into another XML document or to another type of document, such as HTML or XHTML, that is recognised by a browser [28]. This is normally performed by transforming each XML element into an (X)HTML element. In simple words, XSLT transforms an XML source-tree into an XML result-tree.

XPath containment is a key component for the static analysis of XSLT and for improving its performance. As already explained, XPath is a fundamental part of XSLT which allows for definition of matching expressions. In practice, complex XPath expressions are difficult to interpret and are, therefore, error prone. One way to verify the consistency (satisfiability) of an expression p is to check that it is not contained in an XPath expression which returns the empty set on any document tree ($p \subseteq \emptyset$). XPath containment algorithms can also be helpful to optimise the XPath expressions. For example, let $p_1 \cup p_2$ be an XPath expression such that $p_1 \subseteq p_2$. Then, evaluation of p_1 is redundant since all nodes have already been selected in the evaluation of p_2.

2.3.3 XML Pointer Language (XPointer)

XPointer is an XML language which defines several schemes, one of which is the addressing scheme for referencing parts of an XML document. Providing this capability, XPointer

is used by other applications to identify parts of XML documents or their locations. It can be used to reference any element or subsets of elements within a document.

Example 2.2 *Consider the XML document shown in Figure 1.1:*

- `xpointer(//book)` *addresses all* `book` *elements in the XML document.*

XPath is a fundamental part of XPointer which allows for the referencing of parts of the documents. Once again, optimising such XPath expressions reduces unnecessary operations, such as tree navigation, and speeds up evaluation of expressions.

2.4 Schema languages for XML

In this section, we introduce five representative XML schema language proposals: DTD [26], XML-Schema [36], RELAX-NG [30], EDTD [73], and Schematron [49].

2.4.1 Document Type Definition (DTD)

DTDs are the most commonly used schemas, whose main benefit is their simplicity. Nevertheless, there exist serious problems in defining types and in referencing mechanisms [35, 48]. In particular, DTDs have few basic types, and they lack modularity. Another drawback is the lack of type safety of references. Also, the content of a node depends only on the label of that node and not on its context.

Definition 2.1 *A DTD over a finite alphabet* Σ *is a tuple* (D, S_0, Σ) *where* $S_0 \in \Sigma$ *is the start symbol, and D is a mapping from* Σ *to a set of regular expressions over* Σ*. Let* $a \in \Sigma$*,* R^a *be the regular expression which is associated with a by D,* Σ^a *be the alphabet of symbols in* R^a*, and* $L(R^a)$ *be the language denoted by* R^a*. Then we say that* R^a *is the content model for a and write* $a \rightarrow R^a$ *(which we also call a* production rule*).*

Definition 2.2 *Given a DTD* (D, S_0, Σ)*,* G_D *is the* dependency graph *of D whenever it contains an edge from node labeled a to a node labeled b if and only if* $b \in (\Sigma^a)$*. A DTD* (D, S_0, Σ) *is called* recursive *if* G_D *has a cycle.*

From now on, we refer to a DTD simply by D rather than (D, S_0, Σ) and assume that Σ is the set of symbols appearing in D. In examples, we will usually drop the arrow symbol from rules, and will often use the DTD syntax for regular expressions, namely, ",", for concatenation, "|" for alternation (disjunction), "$*$" for reflexive transitive closure, "$+$" for transitive closure and "?" for optional.

A tree t satisfies a DTD D if and only if $\lambda(root(t)) = S_0$, where $\lambda(root(t))$ is the label of $root(t)$, and for each node u in t with n children $u_1, \ldots, u_n : (\lambda(u_1)) \ldots (\lambda(u_n)) \in L(R^{\lambda(u)})$. The set of all trees that satisfy D is denoted by $SAT(D)$.

DTDs contain two kinds of definition components, element definitions and attribute definitions. In this thesis, we only consider the element definitions. Element definitions have the form:

```
<!ELEMENT s ( content-model-s )>
```

where `s` is the name of the element to be defined, and `content-model-s` is a regular expression over element names, or `#PCDATA`, or `#EMPTY`. Regular expressions may employ the well-known operators: iteration ($*$ and $+$), alternative choice (|), optional (?) and sequence (,), as well as parentheses for grouping. `#EMPTY` denotes a regular expression accepting only the empty sequence.

The content models are required to be *deterministic* [42]. Intuitively, a regular expression is deterministic if, when processing the input from left to right, it can always be determined which symbol in the expression matches the next input symbol without lookahead. The following shows an example of a non-deterministic content model:

```
<!ELEMENT Contact ((ZipCod, Tel?) | ZipCod)>
```

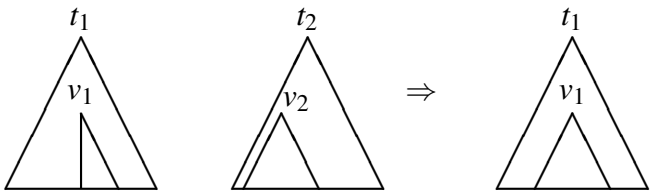


Figure 2.2: Label-guarded subtree exchange. The labels of v_1 and v_2 are the same.

```
<!ELEMENT ZipCod (#PCDATA)>
<!ELEMENT Tel     (#PCDATA)>
```

For a document instance such as `<Contact><ZipCod>12345</Zipcod></Contact>`, it is not clear (without lookahead) whether the content of the `contact` element is as an instance of `(ZipCod, Tel?)` or of `(ZipCod)`. This means the content model can match identical XML element sequences in more than one way. Such a content model is called *ambiguous*.

Among all well-known schema languages, DTDs are the most widespread because of their simplicity. The DTD language is compact and highly readable. However, there is limited support for defining the type of the contained data. DTDs are primarily structural in nature. They do not have the ability to specify that an element contains, for example, an integral number, a real number, or a date. Furthermore, the classes of XML documents definable by DTDs are restricted to local classes [69]. In fact, only languages which have the property of *label-guarded subtree exchange* can be expressed by DTDs [73]. More precisely, the set of trees T_Σ is definable by a DTD if and only if for each pair of trees (t_1, t_2) in T_Σ, and pair of nodes v_1 in t_1 and v_2 in t_2 with the same label, the trees obtained by exchanging the subtrees rooted at v_1 and v_2 are also in the set T_Σ. This property is called label-guarded subtree exchange and is illustrated in Figure 2.2.

Example 2.3 *Consider the following DTD rules:*

```
<!ELEMENT Book      (Title, Author+, Publisher?)>
```

```
<!ELEMENT Publisher (Name, Address)>
<!ELEMENT Author    (Name, Birthday)>
<!ELEMENT Name      ((PublishedBy)|(FirstName,LastName))>
```

Suppose that the desirable constraint is: "a `PublishedBy` *element can only occur as the grandchild of a* `Publisher` *element while* `FirstName` *and* `LastName` *elements can only occur as the grandchildren of an* `Author` *element". No DTD can express this constraint, because it contradicts the label-guarded subtree exchange property. In other words, the content of a node depends only on its label.*

2.4.2 XML Schema Definition Language

A large number of schema languages have been proposed to overcome the limitations of DTDs. In [1], 15 different schema languages for XML are listed besides DTDs. One of these languages, the XML Schema Definition Language (XSDL) [12, 82], is considered as the only schema language for XML documents recommended by the W3C. XSDL offers facilities for describing the structure and constraining the contents of XML documents, including those which exploit the XML namespace facility [82].

XSDL is more powerful than DTDs. The first and most evident improvement is the switch to an XML-based syntax, which provides a high degree of flexibility and automatic processability. Another major contribution of XSDL is the *Post Schema Validation Infoset (PSVI)*, i.e. the additional information that the validation adds to the nodes of the XML document so that downstream applications can use of it for their own purposes. The most important advantages of PSVI are certainly the type information and the set of legal values that nodes can have [82].

Although XSDL has many complicated mechanisms, it is not believed to be very expressive from the viewpoint of formal languages [63, 69]. It has been shown that XSDL is closer in expressiveness to DTDs than to tree automata [11], because the XSDL specification enforces an extra constraint called the *Element Declaration Consistent (EDC)*

constraint. An EDC constraint is imposed to facilitate validation tasks. It prohibits the occurrence of two different types being associated with the same element name in the same content model. This remarkably simplifies the process of validation. In fact, for an XSDL admitting EDC, there exists a very efficient one-pass algorithm to validate a document against the schema [63].

2.4.3 Relax-NG

RELAX-NG [30] is a schema language for XML which is based on two preceding languages, TREX [29], designed by James Clark, and RELAX [62], designed by Murata Makoto. The central concept of RELAX-NG is patterns, which extend the concept of content model. In particular, a pattern in RELAX-NG is an expression over elements, text nodes and attributes, whereas, in DTDs, a content model is an expression over elements (and, very limitedly, text). External definitions of datatypes can be used to restrict the set of values of text nodes and attributes. The most common datatype library is the one defined by XSDL in [12].

Relax-NG supports namespaces, modularity, extensibility and obtains a higher expressive power by extending DTDs with a typing mechanism which allows one to define types, possibly recursively, in terms of other types. Although Relax-NG has only two built-in data types (string and token), it allows for the definition of many more.

In contrast to DTDs and XSDL, Relax-NG imposes no restriction on elements with unordered content and allows for ambiguous definitions, i.e., elements with the same name and different content models, in the same context [54]. The symmetric treatment of elements, attributes and text nodes and the introduction of ambiguous definitions allows Relax-NG to specify a number of co-constraints on XML documents, such as mutual exclusion and inter-dependencies between elements and attributes. To illustrate this feature, consider the following example:

Compared to XSDL, Relax-NG has slightly poorer expressiveness in certain aspects.

For example, there is no specification to define a precise number or range of repetitions of patterns under Relax-NG; under XSDL, there is. Relax-NG has only two built-in data types, while XSDL provides many more. In practice, however, most Relax-NG processors support the XSDL set of data types. Finally, another limitation of Relax-NG is its inability to define default values for elements and attributes.

2.4.4 Extended Document Type Definition (EDTD)

The expressive power of DTDs can be extended by adding types. Recall that in DTDs, the type of an element is its name. The Extended DTDs (EDTDs) are defined as follows [73]:

Definition 2.3 *An extended DTD is a tuple* $E = (\Sigma, \Delta, d, S_d, \mu)$*, where* Δ *is a finite set of types,* μ *is a mapping from* Δ *to* Σ*, and* (Δ, d, S_d) *is a DTD; i.e.,* d *is a function that maps symbols in* Δ *to regular expressions over* Δ*, and* S_d *is the start symbol.*

Example 2.4 *Consider the following EDTD rules:*

```
<!ELEMENT market     (oldCar | newCar)*>
<!ELEMENT oldCar     (code, year?, price)>
<!ELEMENT newCar     (code, model, price)>
```

Where $\mu(\texttt{oldCar}) = \mu(\texttt{newCar}) = \texttt{Car}$*, i.e.* `oldCar` *and* `newCar` *are types that are associated to* `Car` *elements. All other types are associated with the element of the same name; for instance, type* `market` *corresponds to a* `market` *element.*

In EDTDs, each type is assigned to a unique element name and the start symbol has only one possible type. The types are selected from a finite set of types [73]. Formally, a tree t satisfies an EDTD E if there exists an assignment of types to all nodes of t such that the typed tree satisfies E.

From a structural perspective, EDTDs express exactly the well-known regular tree languages. In particular, EDTDs are as expressive as unranked tree automata [16]. It

should be noted that the formal underpinnings of the schema language Relax-NG are also based upon regular tree languages.

2.4.5 Schematron

Schematron [49] is a rule-based validation language, defined as an alternative to existing grammar-based approaches. Schematron and Relax-NG are parts of ISO standards for a *Document Schema Definition Language* [83]. A Schematron document defines a sequence of `<rules>`, logically grouped in `<pattern>` elements. Each rule has a context attribute, which is an XPath pattern determining the elements in the instance document to which the rule applies. Within a rule, a sequence of `<report>` and `<assert>` elements is specified, each having a test attribute. Such a test attribute is an XPath expression which is evaluated to a Boolean value for each node in the context.

The content of both `<report>` and `<assert>` is a declarative sentence in natural language. The `<report>` and `<assert>` elements are effectively the inverse of each other. That is, in the former the content is output when the test of the `<report>` succeeds, whereas, in the latter it is output when the test fails. Thus, the `<report>` element is used to tag negative assertions about the instance document, while the `<assert>` element is used to tag positive ones. Therefore, the output of the Schematron validation process is a list of assertions.

An advantage of Schematron is its suitability of being embedded within other schema languages [35, 75]. Another advantage is its expressive power with respect to the other schema languages already introduced in this section. However, its way of specifying basic structure of a document, i.e. which elements can go where, results in a schema which is more complicated than necessary. This is usually overcome by combining Schematron with Relax-NG or XSDL, which allows Schematron rules to specify additional constraints on the structure defined by XSDL or Relax-NG and hence avoid verbose schemas.

2.4.6 Comparisons between the schema languages

In [54] six schema languages including DTD, XML Schema and Schematron have been compared in terms of their expressive power. These schemas have been categorised into three classes from the least expressive, class 1, to the most expressive, class 3. DTD belongs to the first class which has the least support for schema structure and does not support schema data types and constraints. On the other hand, XML Schema and Schematron are categorised to be in the third class which corresponds to the greatest expressive power. XML Schema fully supports features for schema data types and structures, and Schematron enjoys a very flexible pattern language that allows for describing the detailed semantics of the schema.

Murata et. al. [69] compare six schema languages from the viewpoint of formal language theory. They define four subclasses of regular tree grammars, namely local, single-type, restrained-competition, and regular grammar. They show that the most expressive class is the regular class. However, it may provide more than one interpretation of a document. RELAX and TREX (the two predecessors of Relax-NG) belong to this class. The results indicate that DTD is placed in the local class which is the least expressive class. Finally, XML-Schema is in the single-type class which is more expressive than the local class.

In [85], the strengths and weaknesses of four schema languages, namely, DTDs, XSDL, Relax-NG, and Schematron, have been compared along five major dimensions:

1. Content models and datatypes: expressive power of the rules with respect to defining constraints on structures and data

2. Modularity: power and flexibility of defining and using independent modules

3. Namespaces: degree of support for namespaces

4. Linking: expressiveness with respect to defining relations between attributes and elements of a document

5. Co-constraints: possibility for expressing constraints on attributes and elements based on the presence or values of other attributes and elements

It was reported, in [85], that most of the above-mentioned features are supported by Schematron, especially the co-constraints feature, which is not supported by XSDL and DTDs at all, and which is supported by Relax-NG in a rather limited way. With respect to user-defined types, XSDL was reported to be the best provider for built-in datatypes, whereas Schematron has a limited number of data types and cannot specify default values.

All the above-mentioned works regarding the comparison of XML schema languages indicate that DTDs do not provide as much expressiveness as the other XML schema languages. This may lead to an apparent inferiority of DTDs compared to the other schema languages. However, DTDs are still the most prevalent schema used in real-world applications. The main point to note here is how much of the extra expressiveness provided by the other schemas is used in practice. Research on this is carried out in [63], where the authors examine several hundred real-world schemas. They report that 85 percent of the schemas they observed are equivalent to DTDs. This indicates that DTDs are still among the best XML schema choices and that many would prefer to use DTDs rather than other more complicated schema languages.

It is important to note that the choice of a suitable schema language is application-dependant. In fact, no schema language could be claimed to be the best, providing all the necessary features for all XML documents. For example, DTDs are good with respect to character entities, XSDL has a rich set of predefined data types and an advanced derivation mechanism, Schematron is the best provider for XPath-based rule constraint checking, and Relax-NG provides support for simple and straightforward syntax and the ability to define regular tree languages.

Chapter 3

Previous Work

The answer to a given XPath query is built by matching the tree pattern representing the query against a document. The efficiency of the matching operation depends on the size of the query, so it is important to have queries of minimum size. To achieve this goal, queries should be rewritten to avoid redundant conditions. As explained in Chapter 1, the problem of minimising and optimising XPath queries is related to the problem of XPath containment. Another problem related to XPath containment is the *satisfiability* of an XPath expression, which can be seen as a special case of XPath containment (for more detail, see Chapter 5).

In this Chapter, we review previous studies on XPath containment and XPath satisfiability. Section 3.1 gives an overview of previous results on XPath containment including techniques and the complexity results. In Section 3.2, we discuss the known results on XPath satisfiability.

3.1 Previous results on XPath Containment

Algorithms for query containment are studied in different contexts. They have been applied to find redundant conjuncts in relational conjunctive queries in the context of query optimisation [3, 21, 77], to specify when queries are independent of updates to

the database [57], to maintain integrity constraints [44], to reformulate queries using views [22, 55], and as a tool in data integration [39, 40, 56].

None of the above-mentioned works study the containment problem for XPath queries. In this section, we review techniques that have been used for deciding containment for XPath queries as well as the complexity results for this problem obtained from the literature.

3.1.1 Techniques for Solving Query Containment

The containment problem has been studied both in the presence and in the absence of constraints. In these studies, there are several complexity results most of which have matching upper and lower bounds. To obtain the upper bounds for various fragments of XPath, several techniques have been introduced [78]. These techniques are based on canonical models, containment mapping, tree automata, and the chase procedure. All these techniques use the fact that, for XPath queries p and q, $p \not\subseteq q$ if and only if there is a tree t on which $p(t) \neq \emptyset$ but $q(t) = \emptyset$.

1. **Canonical models**

   Canonical models were introduced in [65] to check containment between Boolean patterns in $\mathrm{XP}^{\{/,[],*,//\}}$. To decide $p \subseteq q$, this technique tries to prune the search space based on the fact that every counter-example, if any, matches p. The containment of boolean patterns reduces to the implication of them, i.e. $p \subseteq q$ if and only if $\forall t \in T_\Sigma$: $p(t) \Rightarrow q(t)$, where p and q are two boolean patterns as defined in Chapter 2. It has also been shown that the containment of boolean patterns can be restated in terms of *models* [65]. A model of a boolean pattern p is defined as a tree $t \in T_\Sigma$ on which p evaluates to *true*: $Mod(p) = \{t \in T_\Sigma \mid p(t) \text{ is } true\}$. Now, $p \subseteq q$ if and only if $Mod(p) \subseteq Mod(q)$ [65].

   To solve the containment problem, it suffices to search for a tree t on which $p(t)$

is true and $q(t)$ is false. Clearly, it suffices to search for t in $Mod(p)$. However, $Mod(p)$ may be an infinite set. To further restrict this set, Miklau et al. [65] introduced *canonical models*. Using this method, the containment of $XP^{\{/,[],*,//\}}$ expressions is shown to be coNP-complete [65].

2. **Containment mapping**

   In this method [5, 65], $p \subseteq q$ if there exists a containment mapping from q to p (see the definitions in Chapter 2).

   For example, let p and q be the two queries whose tree patterns are shown in Figures 3.1(a) and 3.1(b), respectively. Obviously $p \not\subseteq q$, as there is only one target in p for the b-node of q but no possible targets for its c and d children.

   Although the proposed containment algorithms based on containment mapping are very efficient, the existence of a containment mapping is not always a necessary condition for containment, such as when $*$ and $//$ are allowed in the expressions [65].

   Finally, it has also been shown that for fragments in $XP^{\{/,[],*,//\}}$, the containment mapping technique is essentially a special case of the canonical model technique, in which only one tree has to be tested [78].

3. **Tree automata**

   The previous two methods have been used to solve the containment problem in the absence of constraints, whereas, the method of tree automata, introduced in [71], is applied to solve the problem in the presence of constraints. Given queries p and q and a DTD D, it has been shown that $p \subseteq q$ if and only if $\forall t \in SAT(D) : t \models p \Rightarrow t \models q$, where $t \models p$ means there exists a homomorphism from p to t [65]. The automata method computes the set C of all counter-examples to $p \subseteq q$, i.e. C contains all trees t such that $t \models p$ and $t \not\models q$. For containment, the problem is then to check whether C is empty. Although C might be infinite, it can often be represented by a tree automaton. Let A_p and $A_{\overline{q}}$ be, respectively, the automata which accept a

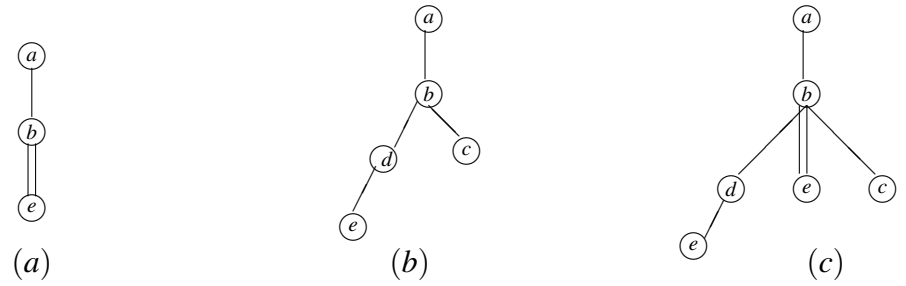


Figure 3.1: (a) tree pattern of p, (b) tree pattern of q, and (c) tree pattern of $Chase_D(p)$.

tree t if $t \models p$ and $t \not\models q$. Neven et al [71] has shown that, for an expression p in $XP^{\{/,//,*,[],\cup\}}$, an exponential size automaton A_p can be constructed such that A_p accepts a tree t if and only if $t \models p$. Therefore, this approach gives an EXPTIME algorithm for containment of $XP^{\{/,//,*,[],\cup\}}$ in the presence of DTDs, which has been shown in [71] to be optimal.

4. **Chase procedure**

   In the relational case the homomorphism technique can be extended by the chase procedure [61] to check query containment in the presence of integrity constraints. This approach can also be used for deciding XPath containment in the presence of constraints [5, 92]. As an example, Figure 3.1(a) and Figure 3.1(b) show two tree pattern queries for expressions p and q with no homomorphism from q to p. Now, consider, for instance, the following DTD rules:

```
<!ELEMENT b (d,c,(x|y))>
<!ELEMENT d (e,(x|y))>
```

   The rules imply that each b element has a d-child as well as a c-child, and each d element has an e-child. Applying the chase procedure with these constraints to the query p will add a d-child and a c-child to the b-node and an e-child to the d-node. The resulting query is called the chase of p by D and is denoted by $Chase_D(p)$.

Now, there is an obvious homomorphism from q (Figure 3.1(b)) to $Chase_D(p)$ (Figure 3.1(c)).

3.1.2 Complexity of Deciding Query Containment

XPath supports a wide variety of operators whose presence or absence affects the complexity of the containment problem. This has led to the study of various XPath fragments that include only certain operators. In practice, many applications do not need the power of the full XPath language; they use only a fragment of XPath. For example, XML Schema specifies integrity constraints with an XPath fragment that does not support the parent or ancestor axes.

One of the most frequently used XPath fragments is the fragment that consists of the following operators: $/$, $//$, $[\,]$, $*$, $|$, which denote, respectively, child, descendant, filter, wildcard, and disjunction. As in [65], we denote an XPath fragment by listing the allowed operators. For instance, $\mathrm{XP}^{\{/,[],//\}}$denotes the XPath fragment where only child, descendant and filter are allowed.

The containment problem can be categorised into two separate problems:

- Containment in the absence of constraints
- Containment in the presence of constraints such as a DTD

For the former problem, the inputs are two queries, say p and q, and we would like to see if the answer of p always includes the answer of q. Whereas, in the latter problem, in addition to the input queries, there is another input which is a set of constraints on the documents (trees) being queried. Therefore, while query p may not contain query q in general, it may be the case that, in the presence of the constraints, p does contain q when both are evaluated on documents valid for the constraints. More discussion on this issue is presented later in Chapter 6. In the rest of this section, we review previous results on XPath containment in both cases.

Containment in the absence of constraints

In this section, we review the previous results on XPath containment in the absence of constraints. Table 3.1 summarises the results.

One of the main challenges with respect to the containment problem is to achieve algorithms with low, ideally PTIME, complexity and there has been much research in the literature in this regard [5, 74, 92]. Most of the PTIME algorithms already proposed for the containment problem, e.g. for $XP^{\{/,[],//\}}$and $XP^{\{/,*,[]\}}$, are based on the use of the homomorphism method. Given a fragment of XPath, if the existence of a homomorphism is a necessary and sufficient condition for containment, then there exists a PTIME algorithm. However, in some fragments, for instance when $*$ and $//$ are allowed in tree patterns, the existence of a homomorphism is no longer a necessary condition [65].

Most of the early results on query containment concern relational conjunctive queries and their extensions [21, 38]. In particular, it has been proved that containment for conjunctive queries is NP-complete [21]. However, it is in PTIME for acyclic conjunctive queries [93]. Several works have considered the extension of containment algorithms for queries involving order [44, 51, 57], and queries over complex objects [58].

Florescu et al. [38] consider the problem of query containment for a query language over semi-structured data in which the queries are allowed to include regular path expressions over the attributes and express queries about the schema, roughly a combination of conjuctive queries with Regular Path Queries(RPQ). An RPQ ρ asks for all pairs of objects that are connected by a path conforming to a regular expression R. They show that containment of queries expressed in their query language is decidable(PSPACE-hard).

Calvanese et al. [20] study Conjunctive Two-way Regular Path Queries (C2RPQs). Two-way Regular Path Queries (2RPQs) include the inverse operator, which enables us to navigate edges backwards, and C2RPQs is the conjunction of 2RPQs, which enables us to perform joins and projections over 2RPQs. Using two-way automata, a PSPACE upper bound is shown for this class of queries (for unions of C2RPQs).

Table 3.1: The complexity of containment for different XPath fragments.

Fragment	**Complexity**	**Reference**
$XP^{\{/,[],//\}}$	PTIME	[5]
$XP^{\{/,//\}}$	PTIME	[18]
$XP^{\{/,*,[]\}}$	PTIME	[93, 91, 65]
$XP^{\{/,//,[],evars\}}$	NP-hard	[34]
$XP^{\{/,[],*,//\}}$	CoNP-Complete	[65]
$XP^{\{/,//,*,[],\cup\}}$	CoNP-Complete	[71]
$XP^{\{*,//\}}$	PSPACE-Complete	[80]
$XP^{\{/,//,[],\cup,evars\}}$	Π_2^p-hard	[34]
$XP^{\{/,//,[],*,evars\}}$	Π_2^p-hard	[34]
$XP^{\{/,//,[],\cup,evars,\neq\}}$	Π_2^p-hard	[71]
$XP^{\{/,//,[],\cup,*,evars,\neq\}}$	Undecidable	[71]

Deutsch and Tannen [34] define XBind queries and show that XPath containment is a particular case of (single-atom) XBind containment. When the fragment allows for navigation to descendant-or-self/ancestor-or-self, arbitrary equalities, disjunction/alternation, and inequalities deciding containment is Π_2^p-complete, whereas it is NP-complete when the fragment disallows path alternation, filter, disjunction, ancestor-or-self navigation, non-equalities and equality on node identities.

For XPath expressions, Miklau and Suciu [65] show that containment of $XP^{\{/,[],*,//\}}$ is coNP-complete, while for any combination of two of the constructs $*$, $//$, and $[]$ the containment problem is in PTIME. In the absence of descendant edges, for $XP^{\{/,*,[]\}}$, whose queries can be viewed as conjunctive queries over a tree structure, a PTIME containment algorithm follows from classic results on acyclic conjunctive queries [93]. This bound for $XP^{\{/,*,[]\}}$ is also noted in [91]. For $XP^{\{/,[],//\}}$, without label wildcards, containment is shown to be in PTIME and every tree pattern query has a unique equivalent minimal query [5]. Queries in $XP^{\{/,//,*\}}$ are a special case of regular expressions on strings, for which there is a PSPACE-complete containment algorithm in general [80]. For the fragment of regular string expressions in $XP^{\{/,//,*\}}$, a linear time containment algorithm is announced in [66]. A PTIME algorithm for queries in $XP^{\{/,//\}}$ is provided in [18].

As another result in [65], the authors describe a sound and complete EXPTIME algo-

rithm for containment of two expressions in $XP^{\{/,[],*,//\}}$. They also claim that the algorithm runs in PTIME whenever the number of $//$'s in a contained pattern is bounded by a constant. Even when the number of filters or wildcards is restricted, the containment of $XP^{\{/,[],*,//\}}$ is coNP-complete. Also for queries in $XP^{\{/,[],//,*\}}$, when every node has at most one child, motivated by [66], the authors of [65] describe a sound and complete algorithm which always runs in PTIME, however, it may return false negatives in some cases, where the containing query has filters ($[\,]$). Their method shows that containment can be decided in PTIME by combining adjacent $//$'s and $*$'s in patterns into single units and then searching for a homomorphism.

In [37], the authors analyse the complexity of the minimisation problem and show that it is the same as the complexity of the containment problem. They identify a subclass of $XP^{\{/,[],*,//\}}$ which can be minimised in PTIME. They show that the queries in this fragment have the subpattern property, which means a minimum size tree pattern equivalent to p can be found among the subpatterns of p. Obviously, this property does not hold in the presence of constraints.

Wood in [91] studies the minimisation of XPath queries in $XP^{\{/,[]\}}$ which corresponds to a class of conjunctive queries. He shows that containment for $XP^{\{/,[]\}}$ is decidable in PTIME. Moreover, in [89], he shows that the Datalog fragment needed for $XP^{\{/,[],*,//\}}$ has a decidable containment problem. The technique is based on the chase procedure introduced in Section 3.1.1.

In other research, Neven and Schwentick [71] have concentrated on XPath expressions with child and descendant axes that can only navigate downwards in an XML tree, and do not use the order between siblings. They show that, in principle, adding disjunction to the fragment studied by Miklau and Suciu [65], $XP^{\{/,[],*,//\}}$, does not make the containment problem any harder when the input alphabet is infinite. However, when the set of allowed element names (labels) in XML documents is restricted and is given as a part of the input, they prove that the containment problem turns to PSPACE-complete. The reason for this

Table 3.2: The complexity of containment for expressions with disjunction.

/	//	[]	|	$*$	**Complexity**
+	+	+		+	coNP-complete
+	+	+	+	+	coNP-complete
+			+		coNP-complete
	+		+		coNP-complete
+	+	+	+	+	PSPACE-complete (given alphabet)
+	+		+		PSPACE-complete (given alphabet)

complexity jump is that when the alphabet is finite, disjunction allows the expression of negation. The results on fragments with disjunction are shown in Table 3.2 [71].

The containment of XPath expressions with variables has been studied in [34]. The XPath semantics allows variables to be used in XPath expressions: an expression matches a document if there exists a suitable assignment for the variables. They show that containment of $\mathrm{XP}^{\{/,//,[],*,evars\}}$ and $\mathrm{XP}^{\{/,//,[],\cup,evars\}}$ is Π_2^P-hard, where *evars* denote variables with existential semantics. This result has been extended in [71] by showing that containment of $\mathrm{XP}^{\{/,//,[],\cup,evars,\neq\}}$, that is, inequality tests on variables and attribute values are allowed, remains in Π_2^P. Surprisingly, the further addition of $*$ to this fragment makes the containment problem undecidable.

Geerts and Fan [41] provide the lower bounds for the containment problem for XPath fragments with sibling axes and establish the complexity of the problem in the absence of DTDs. They show that the problem is coNP-hard for $XP^{\{[],\leftarrow,\cup\}}$ and $XP^{\{[],\rightarrow,\cup\}}$, is PSPACE-hard for $XP^{\{[],\leftarrow,\neg\}}$ and $XP^{\{[],\rightarrow,\neg\}}$, finally, it is undecidable for $XP^{\{[],\rightarrow,\leftarrow,\rightarrow*,\uparrow,\cup,=,\neg\}}$, a fragment with filters, immediate right/left sibling, right sibling, parent, union, data-value joins, and negation.

All the above-mentioned works are restricted to the case where no constraints exist. In this thesis, on the contrary, the containment problem is investigated in the presence of constraints. In particular, DTD constraints are considered for a number of XPath fragments.

Containment in the presence of constraints

All the research reviewed in the previous section involve the containment problem in the absence of constraints. However, as already mentioned, it may happen that a query p does not contain a query q in general but does contain it under some constraints on the underlying database. Therefore, this presents the containment problem under constraints as another challenge, which has been the subject of much research over recent years [5, 33, 74, 88].

The advantages offered by imposing constraints on XML data are numerous. Constraints capture the semantics of data objects in an XML document. As a result, it facilitates automatic validation of the document structure.

The query containment problem for relational conjunctive queries in the presence of integrity constraints, in particular functional and inclusion dependencies, was first studied in [50]. Calvanese et al. [19] consider the problem of conjunctive query (with regular expressions) containment in the presence of a special class of inclusion dependencies and establish a number of decidability/undecidability results. In [94], the integrity constraints are extended to implication constraints and referential constraints, which are generalised forms of functional constraints and inclusion dependencies, respectively. In order to handle incomplete information in the database, Wei and Lausen [87] suggest that disjunctions need to be expressed as integrity constraints. They introduce disjunctive referential integrity constraints and give a sound and complete algorithm, Π_2^P-complete for checking the containment of conjunctive queries under disjunctive referential and implication constraints. The technique for handling disjunctive referential constraints is related to the well-known minimal model semantics for disjunctive logic programming [59].

Deutsch et al. [33] consider XPath containment in the presence of DTDs and Simple XPath Integrity Constraints (SXICs). Although SXICs can express many constraints implied by DTDs, DTDs and SXICs are in general incomparable. SXICs cannot express the order of sibling elements in DTD and the feature that an element admits only subelements

Table 3.3: The complexity of containment in the presence of DTDs [71].

DTD	/	//	[]	∪	∗	**Complexity**
+	+	+				in P
+	+		+			coNP-complete
+		+	+			coNP-hard
+	+	+	+	+	+	EXPTIME-complete
+	+	+		+		EXPTIME-complete

of given tags.

The authors obtain that containment of $XP^{\{/,//,[],\cup,evars\}}$ under bounded SXICs is decidable. Bounded SXICs are a subclass of SXICs which allow the same generality as all SXICs in the left-hand-side of the implication, but restrict the form of the right-hand-side of the implication. They claim that the problem is in EXPTIME in the size of constraints, while the complexity drops to NP when both expressions and constraints are disjunction-free. They also show that the problem is undecidable (1) under unbounded SXICs or (2) in the presence of bounded SXICs and DTDs. When only DTDs are present, they give a PSPACE lower bound and leave the exact complexity as an open question.

A general result establishing a strong upper bound for a large fragment of XPath is due to Marx [64]. He shows that the containment problem for navigational XPath, allowing navigation along all axes, even relative to a specialized form of DTD(similar to XML Schema), is in EXPTIME. He proves that containment testing for $XP^{\{/,//,*,[],\cup\}}$ under DTD constraints is in EXPTIME. This result has also been reported in [71].

Another result in [71] shows that containment under DTD constraints for $XP^{\{/,//,\cup\}}$ and for $XP^{\{/,[],*,//\}}$ are hard for EXPTIME. In addition, the authors study the complexity of more restrictive fragments in the presence of DTDs. It turns out that DTD-containment of $XP^{\{/,//\}}$ is in PTIME. On the other hand, DTD-containment of $XP^{\{/,[]\}}$ and $XP^{\{/,[],//\}}$ are coNP-complete and coNP-hard, respectively. The results about the containment problem in the presence of DTDs are summarised in Table 3.3 [71].

When sibling axes are included, Geerts and Fan [41] establish the complexity of the containment problem for various XPath fragments in the presence of various restricted

forms of DTD. For non-recursive, disjunction-free and fixed DTDs, they show that the problem is undecidable for $XP^{\{[],\rightarrow,\leftarrow,\rightarrow*,\uparrow,\cup,=,\neg\}}$, a fragment with filters, immediate right/left sibling, right sibling, parent, union, data-value joins, and negation. On the same setting, it is coNP-hard for $XP^{\{[],\rightarrow\}}$ and $XP^{\{[],\leftarrow\}}$ and is EXPTIME-hard for $XP^{\{[],\rightarrow,\uparrow,\neg\}}$. Finally, the problem is PSPACE-hard for $XP^{\{[],\rightarrow,\neg\}}$ and $XP^{\{[],\leftarrow,\neg\}}$ under non-recursive and star-free DTDs.

In the presence of DTD constraints, in the most relevant study to ours, Wood [92] proves that the containment for $\text{XP}^{\{/,[]\}}$ is coNP-hard. He defines two types of constraints, called *Sibling Constraints* (SCs) and *Functional Constraints* (FCs), which are implied by DTDs. He obtains that SCs capture containment for queries in $\text{XP}^{\{/,[]\}}$ that are *duplicate-free*. A query p in $\text{XP}^{\{/,[]\}}$ is duplicate-free if each node in the tree pattern corresponding to p has children with distinct labels. However, when *DTDs* are *duplicate-free*, both constraints are necessary and sufficient to decide containment in PTIME.

The notion of a duplicate-free DTD was introduced in [91] and also used in [92]. A regular expression R is *duplicate-free* if each symbol occurs exactly once in R. A DTD D is called *duplicate-free* if and only if each content model in D is duplicate-free. For instance, $(b?,(a|c)+)$ is a duplicate-free regular expression while $a,(b|a)$ is not. In recent work [10], duplicate-free regular expressions are redefined and referred to as SOREs (Single Occurrence Regular Expressions). That paper also shows that SOREs capture the far majority of the regular expressions occurring in practical DTDs.

More recently, Lakshmanan et al. [53] have shown that containment for queries in $\text{XP}^{\{/,[],//\}}$ under disjunction-free and recursion-free (simplified) DTDs can be reduced to containment of tree patterns. They have used a chase algorithm with a set of constraints including intermediate node, parent-child and cousin constraints.

In [24], the authors investigate the query minimisation problem for $\text{XP}^{\{/,[],//\}}$ under FBST-constraints which includes child, descendant, subtype, parent, ancestor and sibling constraints. They design an efficient algorithm based on containment mappings to min-

imise a given XPath expression both to obtain a unique minimal query and to enumerate all possible minimal queries. Their proposed method is similar to [74]. However, their constraints are not sufficient even for $XP^{\{/,[]\}}$ when the underlying DTD is not duplicate-free. For example, assume queries $p = a[b[d][f][g]]$ and $q = a[b[c][f][g]][b[d][e][g]][b[d][f][h]]$ and the following DTD rules:

```
<!ELEMENT a (b,b)>
<!ELEMENT b (c,f,g)?,(d,e,g)?,(d,f,h)?>
```

There is no containment mapping from p to q because no b-node in q has all children of the b-node of p. However, p contains q in all trees in $SAT(D)$, because whenever there is a homomorphism from q to a tree t in $SAT(D)$, at least two of the b-nodes of q must map to the same b-node in t and the union of any two sets of children of b-nodes in q includes $\{d, f, g\}$, so there will be a homomorphism from p to t.

Finally, Wang and Yu [86] have defined a new set of constraints to deal with disjunction-free but recursive DTDs. They have proposed two chase algorithms, one of which may not terminate in some cases (even in $XP^{\{/,//\}}$) and the other of which can result in a set of chased queries whose size is exponential in the number of descendant-edges in the query. Montazerian and Wood [67] have introduced a rewriting algorithm for $XP^{\{/,//\}}$ and produced a complete procedure for deciding containment using the chase and the set of previously-defined constraints. They considered recursive but non-disjunctive DTDs.

Because the D-containment problem is NP-hard in general, one of the main purposes of this thesis is to determine specific XPath fragments and types of DTD for which the problem becomes tractable. In particular, we determine characteristics, especially in real-world DTDs, which reduce the complexity of D-containment.

3.2 Previous results on XPath Satisfiability

Particularly when an XPath query is to be evaluated over documents known to be valid with respect to a schema, it is possible that the query might be *unsatisfiable*, that is, the query always returns an empty result, no matter what document (valid with respect to the DTD) is queried. For instance, irrespective of the underlying DTD, the query `self::a/self::b` is always unsatisfiable as it looks for a node labeled with both *a* and *b*.

Example 3.1 *As another example, consider the query $a[b]/c$ and the following DTD rule:*

```
<!ELEMENT a (b|c)>
```

The query is unsatisfiable, because the rule implies that every a-node can have either a b-node or a c-node as a child, but not both.

Relatively little work has been done on detecting whether a given XPath query is satisfiable [7, 41, 45, 52]. However, it is potentially important to detect unsatisfiable XPath queries and optimise queries to remove expressions that will always return an empty result set. Indeed, Lakshmanan et al. show that checking satisfiability as a first step in query processing often yields substantial savings in overall query processing time [52].

Satisfiability analysis can be related to the containment problem. In fact, the satisfiability problem is reducible to the complement of the containment problem [7]. More specifically, Given a DTD D, for any XPath fragment XP and query $p \in XP$, p is D-satisfiable if and only if $p \not\subseteq \emptyset_D$ under D, where $\emptyset_D$ is a special query that returns the empty set on any XML tree of $SAT(D)$. While there has also been much work on containment analysis, as reviewed in the previous section, previous results on the containment cannot answer the questions of satisfiability analysis. Indeed, as already indicated by [7], the lower bounds for containment are often much higher than its satisfiability counterpart.

Table 3.4: The complexity of satisfiability in the absence of DTDs [45].

$\Uparrow$	[]	$\cup$	$\cap$	$\neg$	**Complexity**
✓					PTIME
	✓				PTIME
			✓		NP-complete
✓	✓				NP-complete
	✓	✓			NP-complete
✓	✓	✓	✓		NP-complete
				✓	NP-hard

3.2.1 Satisfiability in the absence of constraints

Lakshmanan et al. [52] study the satisfiability problem for a tree pattern formalism similar to XPath in the absence of DTDs. The formalism expresses tree-shaped queries with a node identity operator which can compare data values. In particular, they reduce satisfiability reasoning to making inferences about relationships between nodes and/or their contents or attribute values. They identify conditions under which the problem can be solved in PTIME.

Hidders classifies the complexity of satisfiability for various XPath fragments as either PTIME or NP-hard [45]. He shows that deciding satisfiability for filters, sibling axes, backward axes($\uparrow *,\uparrow$), and the root test is in PTIME. The main contribution of [45] is to show that testing satisfiability for each of $XP^{\{\Uparrow,[]\}}$, $XP^{\{\cap\}}$, $XP^{\{[],\cup\}}$, and $XP^{\{\Uparrow,[],\cup,\cap\}}$ is NP-complete. Moreover, he proves that when filtering, forward($\downarrow *,\downarrow$) and backward axes, and order are present, satisfiability can be tested in PTIME. To prove this, he uses a *tree description graph*. The procedure he applies is slightly similar to the chase procedure used in [52]. He also shows that when all the axes and root are present, but none of the set operations or filtering is allowed, satisfiability can again be tested in PTIME. Hidders points out that the satisfiability problem is NP-complete when forward axes and intersection operators are present. Finally, he obtains that satisfiability is NP-hard in the presence of a complement operator. The results for different XPath fragments are summarised in Table 3.4 [45].

In the absence of DTDs, Geerts and Fan [41] show that the satifibility for queries in $XP^{\{[],\cup,\rightarrow\}}$ and $XP^{\{[],\cup,\leftarrow\}}$ is NP-hard. The problem is PSPACE-hard for $XP^{\{[],\neg,\rightarrow\}}$ and $XP^{\{[],\neg,\leftarrow\}}$ and is undecidable for $XP^{\{[],\rightarrow,\leftarrow,\rightarrow*,\uparrow,\cup,=,\neg\}}$.

All the above-mentioned works on the satisfiability problem are restricted to the case where no constraints exist, whereas, in this thesis, the satisfiability problem is investigated in the presence of DTD constraints.

3.2.2 Satisfiability in the presence of constraints

Lakshmanan et al [52] study the satisfiability problem for a tree pattern formalism which is similar to XPath. They investigate the problem in the presence of non-recursive disjunction-free DTDs. Based on the proposed formalism, they introduce a constraint graph which consists of a structural part to capture structural constraints and a value-based part for value-based constraints. They propose some inference rules to close the graph with respect to all constraints implied by the given constraints and show that the rules are complete when the query contains no wildcards. Moreover, they show that testing satisfiability is in PTIME for the tree patterns containing child, descendant, filter and NIC, where *NIC* stands for *Node Identity Constraint* which is a structural constraint. These results do not overlap with our results as they only consider a limited form of DTDs.

Groppe et al. [43] focus on the satisfiability problem for XPath, including all XPath axes and negation in predicates, and propose a schema-based satisfiability tester which checks whether or not an XPath query conforms to the constraints in a given schema. The proposed tester evaluates XPath queries on XML Schema definitions. They represent an XML Schema definition by a directed graph and prove that in the worst case, i.e. each node in the graph has edges to all nodes, the complexity of their approach is $O(a \times N \times (N! \times 3)^a)$, where a is the number of location steps in the query Q and N is the number of nodes in the given XML Schema. They consider real-world schemas where each node in the given schema, S, has only a small number of succeeding nodes compared with the number of

nodes in S. In addition, they assume that in a given query Q, the average number of visited nodes in each location step is less than a constant C. They prove that the complexity of their approach under these assumptions is $O(a \times N \times C)$. Finally, they implement experimental results to show the correctness and the performance of their tester. They show that the overhead of checking satisfiable XPath queries by their approach is very low. Since the problem has been shown to be undecidable [7], their approach is sound but incomplete.

Inspired by our proposed covering property published in [68], the authors in [46] introduce a subclass of DTDs, called DCDTDs, whose content models are a subclass of covering DTDs and a superclass of disjunction-free DTDs. In DCDTDs (Disjunction Capsulated DTDs), every disjunction operators in the content models are capsulated by wildcards. They show that the satisfiability problem for XPath fragments with forward(child and descendant), backward(parent) and sibling axes along with union and filter operators, under DCDTDs is tractable.

The most relevant work to our study is [7], in which the authors study a variety of widely-used XPath fragments and show the impact of different XPath operators on the satisfiability problem in the presence of DTDs. More specifically, they study the problem for XPath fragments with and without upward axes (parent and ancestor axes), negation, recursion (XPath with descendant and ancestor axes), and data-value joins (XPath with comparisons of data values). They show that the presence of filters makes the satisfiability analysis harder. They prove that satisfiability is in PTIME for $\mathrm{XP}^{\{/,//,*,\cup\}}$ under any DTDs and for $\mathrm{XP}^{\{/,//,*,[],\cup\}}$ under disjunction-free DTDs. They identify the tractable cases and the factors which lead to NP-completeness. They prove that, with negation, the complexity alters from PSPACE to EXPTIME, while it changes from NEXPTIME to undecidable in the presence of both data values and negation. Their results also indicate that XPath satisfiability in the presence of DTDs for the fragment with all of the features of [45], i.e. $\mathrm{XP}^{\{/,[],\cap,\cup,\neg\}}$, is in EXPTIME.

The results in [7] are extended in [41] by adding sibling axes to the fragments stud-

ied in [7] in the presence and absence of DTDs and under various restricted DTDs. They indicate that, in the presence of DTDs, for XPath fragments with child, filter, sibling and upward axes satisfiability is NP-hard. Moreover, they give PTIME bounds in the absence of filters and in the presence of sibling and upward axes. They also establish lower and upper bounds for satisfiability analysis in various settings, which range from NLOGSPACE to undecidable. They prove that in the absence of XPath filters ([]), the presence of sibling axes does not impact the complixity. However, in the presence of filters, sibling axes make the problem even harder. For instance, XPath satisfiability for $XP^{\{/,[]\}}$ under fixed or under disjunction-free DTDs is in PTIME, but the complexity rises to NP-hard in the presence of sibling axes. They amend the result reported in [64] which places an EXPTIME bound on the complexity of satisfiability for an XPath fragment with all axes, union, filters and negation in the presence of DTDs. More specifically, they show that the complexity for the fragment without descendant and ancestor axes is still in EXPTIME.

3.3 Summary

In summary, extensive research has focused on analysing the complexity of D-containment and D-satisfiability problems, which are shown to be intractable in general, unless $P = NP$. Consequently, a number of works attempt to reduce the complexity of these problems for certain cases. In particular, it is shown the under duplicate-free DTDs, D-containment of $XP^{\{/,[]\}}$ is tractable. It is also shown that under disjunction-free DTDs, D-satisfiability of $XP^{\{/,[],*,//,\cup\}}$ is tractable [7].

However, research in this direction is still required due to the (co)NP-hardness of the problems. The existing results are limited both with respect to the number of cases for which the problems become tractable and with respect to the restrictive nature of them. For example, the disjunction-freeness is rather restrictive and is not often preserved in practice. Therefore, the main goal of this thesis is to provide further cases for which the D-satisfiability and D-containment problems become tractable. Furthermore, in devising

such cases, we closely observed real-word DTDs in order to avoid proposing cases that are too restrictive.

In this thesis, we concentrate on the satisfiability problem under two special classes of DTDs for a variety of XPath fragments with child axes (/), descendant axes (//), filters ([ ]), unions($\cup$) and wildcards (*). We establish various lower and upper bounds for the problem in these settings. The two classes of DTDs we consider are *duplicate-free* DTDs and *covering* DTDs. The covering property is less restrictive than the disjunction-free property but can still be easily analyzed. In Chapter 5, we show that under covering DTDs the satisfiability problem for $\text{XP}^{\{/,[],*,//,\cup\}}$ is in PTIME. Ishihara et al. extend this result and show that adding sibling-axes to this fragment makes the problem NP-complete [46]. We also show that testing satisfiability for $\text{XP}^{\{/,[]\}}$ is decidable in PTIME for duplicate-free DTDs. Motivated by our results presented in [68], Suzuki et al. [81] show that the problem is also in PTIME for XPath fragments consisting of child, parent, and sibling axes. When only a constant number of descendant axes are considered, the problem is still in PTIME. Without any restrictions, however, the problem becomes NP-complete, even for fragments without sibling axes.

Chapter 4

Constraints Inferred from DTDs

The containment and equivalence problems of various fragments of XPath queries under DTDs have been studied in the literature [34, 5, 91, 74, 92, 71, 71, 31] which was surveyed in the previous chapter. Some of these proposals are based on the use of constraints derived from DTDs, as opposed to the use of the DTDs themselves. Constraints are used in order to transform the DTD-containment/DTD-equivalence problem to one of simple containment/equivalence [74, 5, 31, 91, 92]. There are, however, two issues with such an approach: (i) whether the derived constraints completely capture the restrictions imposed by the DTDs and (ii) how to extract such constraints from the DTDs.

Regarding the first issue, Wood [91, 92] has defined two constraints called *Sibling Constraints (SCs)* and *Functional Constraints (FCs)* and has shown that SCs are necessary and sufficient to decide containment under DTDs, when restricted to *duplicate-free* queries in the XPath fragment $XP^{\{/,[]\}}$. He also proves that SCs and FCs are necessary and sufficient to decide containment under DTDs in PTIME for $XP^{\{/,[]\}}$, as long as the DTD is *duplicate-free* (defined in Chapter 3). However, these constraints are not powerful enough to capture DTD-containment for $XP^{\{/,[]\}}$ when neither the query nor the underlying DTD is duplicate-free. In this chapter, we introduce two new types of constraints called *Bag Sibling Constraints (BSCs)* and *Bag Functional Constraints (BFCs)* which not only gener-

alise, respectively, SCs and FCs, but also overcome their limitation to sets, as opposed to bags, of labels. In particular, we show in Chapter 6 that BSCs and BFCs are necessary and sufficient to decide containment under *well-behaved* DTDs for $XP^{\{/,[]\}}$. Duplicate-free DTDs are a subset of well-behaved DTDs (the property of well-behavedness is defined in Chapter 6). In the general case, however, i.e. when there is no restriction on the DTD, the full power of the DTD is necessary to decide containment for $XP^{\{/,[]\}}$ [92].

In this chapter, we also define derivatives of regular expressions, with respect to a *bag* of symbols. Derivatives of regular expressions have been previously defined [17], but with respect to *strings* of symbols. We use regular expression derivatives to determine whether a given query q is satisfiable with respect to a given DTD D. Based on this idea, we define the DERIVATIVE NON-EMPTINESS problem and prove that it is NP-complete. In Chapter 6, we define a new class of regular expressions for which solving DERIVATIVE NON-EMPTINESS is in PTIME.

The rest of this chapter is organised as follows. Section 4.1 provides the basic notation and definitions used in the rest of the chapter. In Section 4.2, our variation of derivatives of regular expressions is defined. Section 4.3 introduces the DERIVATIVE NON-EMPTINESS problem. Section 4.4 describes the concept of DTDs and some types of DTD constraint previously defined in the literature. New types of DTD constraints, together with their properties, are defined in Section 4.5. Finally, Section 4.6 concludes the chapter.

4.1 Basic Definitions

The following notation is adapted from [4] and [23]. A *bag* is a set of *annotated elements*; the annotation of an element, also called its *multiplicity*, is a positive integer. If bag B contains the element a, we write $a \in B$. The multiplicity of element a in bag B is denoted by $|a|_B$. If $a \notin B$, then $|a|_B = 0$. When enumerating the elements in a bag we use superscripts to indicate the multiplicity of elements. Thus a bag containing 2 copies of a and 3 copies of b is written as $\{a^2, b^3\}$.

We use $\subseteq$ to denote (bag) *containment*, defined as follows: $A \subseteq B$ if $\forall x \in A$, $|x|_A \leq |x|_B$. We use $\cup$ to denote (bag) *union* (also called *maximal union*), defined as follows: $x \in A \cup B$ if $x \in A$ or $x \in B$ and $|x|_{A \cup B} = max(|x|_A, |x|_B)$. On the other hand, *additive union*, denoted by $\uplus$, is defined as follows: $x \in A \uplus B$ if $x \in A$ or $x \in B$ and $|x|_{A \uplus B} = |x|_A + |x|_B$. Finally, given two bags A and B, $A - B$ is the bag of symbols resulting from removing, for each $b \in A$, $|b|_B$ occurrences of b from $A(|b|_A = 0$ if $|b|_A \leq |b|_B)$. Note that for maximal union $A - (B \cup C)$ is not necessarily equal to $(A - B) - C$, but, for additive union, $A - (B \uplus C) = (A - B) - C = (A - C) - B$.

Given a string v and a bag of symbols B, we denote by $v - B$ the set of strings resulting from removing, for each $b \in B$, $|b|_B$ occurrences of b from v. For example $babba - \{a, b^2\} = \{ab, ba\}$. For string v, we denote the bag of symbols appearing in v by $[v]$. Hence $[babba] - \{a, b^2\} = \{a, b\}$.

If w is a string of symbols over alphabet Σ, then we use the notations $[w]$ and $\{w\}$ to denote, respectively, the bag and the set of symbols appearing in w.

4.2 Bag derivatives of regular expressions

Brzozowski [17] defined derivatives of regular expressions with respect to *strings*. In this section, we define derivatives with respect to *bags* of symbols. Throughout, we use Σ as an alphabet of symbols.

Definition 4.1 *Let $B \subseteq \Sigma$ be a (non-empty) bag of symbols, and $L \subseteq \Sigma^*$ be a language. The* derivative *of L with respect to B, denoted $\delta_B L$, is the language*

$$\{u \mid v \in L, B \subseteq [v], u \in v - B\}. \tag{4.1}$$

Recall that, according to the notation introduced in Section 4.1, $[v]$ is a bag of symbols, while $v - B$ is a set of strings.

We next define the derivative of a regular expression. Recall that a regular expression over a set Σ is defined recursively as follows [76]:

- $\emptyset$ is a regular expression.
- ε is a regular expression.
- The symbol a is a regular expression where $a \in \Sigma$.
- (P,Q), $(P|Q)$ and P^* are regular expressions, where P and Q are regular expressions.

In this chapter, by PQ, we mean the concatenation P,Q of regular expressions P and Q.

Definition 4.2 *Let $b,c \in \Sigma$ $(b \neq c)$, and R be a regular expression over Σ. The derivative $\delta_{\{b\}}R$ of R with respect to the singleton bag $\{b\}$ is an expression recursively defined as follows:*

$$\begin{aligned} \delta_{\{b\}}b &= \varepsilon \\ \delta_{\{b\}}c &= \delta_{\{b\}}\varepsilon = \delta_{\{b\}}\emptyset = \emptyset \end{aligned}$$

If P and Q are regular expressions over Σ, then

$$\begin{aligned} \delta_{\{b\}}(P|Q) &= (\delta_{\{b\}}P)|(\delta_{\{b\}}Q) \\ \delta_{\{b\}}(PQ) &= \delta_{\{b\}}PQ|P\delta_{\{b\}}Q \\ \delta_{\{b\}}(P^*) &= P^*\delta_{\{b\}}PP^* \end{aligned}$$

For $b \in \Sigma$ and regular expression R over Σ, let us denote by $\delta^n_{\{b\}}R$ the n'th derivative of R with respect to $\{b\}$, that is,

$$\delta^n_{\{b\}}R = \overbrace{\delta_{\{b\}}(\delta_{\{b\}}\cdots(\delta_{\{b\}}}^{n \text{ times}}R)).$$

Definition 4.3 *Let R be a regular expression over Σ, and $B = \{b_1^{n_1}, \ldots, b_m^{n_m}\}$ be a bag of symbols over Σ. A regular expression denoting the derivative of R with respect to B, denoted $\delta_B R$, is defined as follows:*

$$\delta_B R \;=\; \delta^{n_1}_{\{b_1\}}(\delta^{n_2}_{\{b_2\}} \cdots (\delta^{n_m}_{\{b_m\}} R)) \tag{4.2}$$

Next, we show that the language denoted by the (bag) derivative of a regular expression R is equal to the derivative of the language denoted by R.

Theorem 4.1 *Let R be a regular expression and B a bag of symbols over Σ. Then $\delta_B L(R) = L(\delta_B R)$.*

Proof. We first consider the case of $B = \{b\}$, $b \in \Sigma$. The proof proceeds by induction, based on Definition 4.2.

Base case: There are four cases to consider: (1) when $R = b$, (2) when $R = c$ $(c \neq b)$, and (3) when $R = \varepsilon$, and (4) when $R = \emptyset$. The proofs for cases (3) and (4) are similar to that for case (2) and so are omitted.

(1) When $R = b$ and $\delta_b R = \varepsilon$:

$$\begin{aligned}
\delta_{\{b\}} L(R) &= \delta_{\{b\}} L(b) \\
&= \{u \mid v \in \{b\}, \{b\} \subseteq [v], u \in v - \{b\}\} \\
&= \{u \mid \{b\} \subseteq [b], u \in b - \{b\}\} \\
&= \{\varepsilon\} \;=\; L(\varepsilon) \;=\; L(\delta_{\{b\}} R)
\end{aligned}$$

(2) When $R = c$ and $\delta_{\{b\}} R = \emptyset$:

$$\begin{aligned}
\delta_{\{b\}} L(R) &= \delta_{\{b\}} L(c) \\
&= \{u \mid v \in \{c\}, \{b\} \subseteq [v], u \in v - \{b\}\} \\
&= \{u \mid \{b\} \subseteq [c], u \in c - \{b\}\}
\end{aligned}$$

$$= \{u \mid \{b\} \subseteq [c], u \in \emptyset\} = \emptyset = L(\delta_{\{b\}}R)$$

Induction step: There are three cases to consider: (1) when $R = P|Q$, (2) when $R = PQ$, and (3) when $R = P^*$.

(1) When $R = P|Q$ and $\delta_{\{b\}}R = (\delta_{\{b\}}P)|(\delta_{\{b\}}Q)$:

$$\begin{aligned}
\delta_{\{b\}}L(R) &= \delta_{\{b\}}L(P|Q) \\
&= \{u \mid v \in L(P|Q), \{b\} \subseteq [v], u \in v - \{b\}\} \\
&= \{u \mid (v \in L(P)) \vee (v \in L(Q)), \{b\} \subseteq [v], u \in v - \{b\}\} \\
&= \{u \mid v \in L(P), \{b\} \subseteq [v], u \in v - \{b\}\} \cup \{u \mid v \in L(Q), \{b\} \subseteq [v], u \in v - \{b\})\} \\
&= \delta_{\{b\}}L(P) \cup \delta_{\{b\}}L(Q)
\end{aligned}$$

By the inductive hypothesis:

$$L(\delta_{\{b\}}P) \cup L(\delta_{\{b\}}Q) = L(\delta_{\{b\}}P|\delta_{\{b\}}Q) = L(\delta_{\{b\}}R)$$

(2) When $R = PQ$ and $\delta_{\{b\}}(PQ) = \delta_{\{b\}}PQ|P\delta_{\{b\}}Q$

$$\begin{aligned}
\delta_{\{b\}}L(R) &= \delta_{\{b\}}L(PQ) \\
&= \{u \mid v \in L(PQ), \{b\} \subseteq [v], u \in v - \{b\}\} \\
&= \{u \mid v_1 \in L(P), v_2 \in L(Q), \{b\} \subseteq [v_1v_2], u \in v_1v_2 - \{b\}\} \\
&= \{u \mid v_1 \in L(P), v_2 \in L(Q), (\{b\} \subseteq [v_1], u_1 \in v_1 - \{b\}, u = u_1v_2) \vee \\
&\quad (\{b\} \subseteq [v_2], u_2 \in v_2 - \{b\}, u = v_1u_2)\} \\
&= \{u_1v_2 \mid v_1 \in L(P), \{b\} \subseteq [v_1], u_1 \in v_1 - \{b\}, v_2 \in L(Q)\} \cup \\
&\quad \{v_1u_2 \mid v_1 \in L(P), \{b\} \subseteq [v_2], u_2 \in v_2 - \{b\}, v_2 \in L(Q)\} \\
&= \{u_1v_2 \mid u_1 \in \delta_{\{b\}}L(P), v_2 \in L(Q)\} \cup \\
&\quad \{v_1u_2 \mid v_1 \in L(P), u_2 \in \delta_{\{b\}}L(Q)\}
\end{aligned}$$

By the inductive hypothesis:

$$\begin{aligned} &= \{u_1v_2 \mid u_1 \in L(\delta_{\{b\}}P), v_2 \in L(Q)\} \cup \{v_1u_2 \mid v_1 \in L(P), u_2 \in L(\delta_{\{b\}}Q)\} \\ &= L(\delta_{\{b\}}PQ) \cup L(P\delta_{\{b\}}Q) \;=\; L(\delta_{\{b\}}PQ \mid P\delta_{\{b\}}Q) \;=\; L(\delta_{\{b\}}R) \end{aligned}$$

(3) When $R = P^*$ and $\delta_{\{b\}}(P^*) = P^*\delta_{\{b\}}PP^*$

$$\begin{aligned} \delta_{\{b\}}L(R) &= \delta_{\{b\}}L(P^*) \\ &= \{u \mid v \in L(P^*), \{b\} \subseteq [v], u \in v - \{b\}\} \\ &= \{u \mid v \in L(\varepsilon \mid P^+), \{b\} \subseteq [v], u \in v - \{b\}\} \\ &= \{u \mid v \in L(\varepsilon) \cup L(P^+), \{b\} \subseteq [v], u \in v - \{b\}\} \\ &= \{u \mid v \in L(P^+), \{b\} \subseteq [v], u \in v - \{b\}\} \end{aligned}$$

Since $P^+ = P^*PP^*$:

$$\begin{aligned} &= \{u \mid v = v_1v_2v_3, v_1 \in L(P^*), v_2 \in L(P), v_3 \in L(P^*), \{b\} \subseteq [v_2], u \in v_2 - \{b\}\} \\ &= \{v_1uv_3 \mid v_1 \in L(P^*), v_2 \in L(P), v_3 \in L(P^*), \{b\} \subseteq [v_2], u \in v_2 - \{b\}\} \\ &= \{v_1uv_3 \mid v_1 \in L(P^*), u \in \delta_{\{b\}}L(P), v_3 \in L(P^*)\} \end{aligned}$$

By the inductive hypothesis:

$$\begin{aligned} &= \{v_1uv_3 \mid v_1 \in L(P^*), u \in L(\delta_{\{b\}}P), v_3 \in L(P^*)\} \\ &= L(P^*\delta_{\{b\}}PP^*) \;=\; L(\delta_{\{b\}}R) \end{aligned}$$

Based on Definition 4.3, we now consider the case when B is not necessarily a singleton. For simplicity we assume $B = \{b_1, b_2, \ldots, b_n\}$, $n \geq 1$, where for each b_i and b_j $(i \neq j)$, b_i may or may not be equal to b_j. Therefore, we have $\delta_{\{b_1,b_2,\ldots,b_n\}}R = \delta_{b_1}(\delta_{b_2}\ldots(\delta_{b_n}R))$. We

proceed by induction on $|B|$.

Base case: For $|B| = 1$, $B = \{b_1\}$ and $\delta_{\{b_1\}}L(R) = L(\delta_{\{b_1\}}R)$ as already shown.

Induction step: We assume the result holds for $|B| = k$, $k \geq 1$, i.e. $\delta_{\{b_1,b_2,\ldots,b_k\}}L(R) = L(\delta_{b_1}(\delta_{b_2}(\ldots(\delta_{b_k}R))$, and prove that it also holds for $|B| = k+1$. Let us assume $B_1 = \{b_1,\ldots,b_k\}$ and $B_2 = \{b_1,\ldots,b_k,b_{k+1}\}$. Then we have

$$\begin{aligned}\delta_{\{b_1,b_2,\ldots,b_k,b_{k+1}\}}L(R) &= \{u \mid v \in L(R), B_2 \subseteq [v], u \in v - B_2\} \\ &= \{u \mid v \in L(R), B_1 \uplus \{b_{k+1}\} \subseteq [v], u \in v - (B_1 \uplus b_{k+1})\}\end{aligned}$$

Recall from Section 4.1 that $A - (B \uplus C) = (A - B) - C = (A - C) - B$, hence:

$$= \{u \mid v \in L(R), B_1 \uplus \{b_{k+1}\} \subseteq [v], u \in (v - \{b_{k+1}\}) - B_1\}$$

Let $V = \{u \mid v \in L(R), \{b_{k+1}\} \subseteq [v], u \in v - \{b_{k+1}\}\}$. Then

$$= \{u \mid v \in V, B_1 \subseteq [v], u \in v - B_1\}$$

Based on the proof of the base case, we have $\{u \mid v \in L(R), \{b_{k+1}\} \subseteq [v], u \in v - \{b_{k+1}\}\} = L(\delta_{b_{k+1}}(R)) = \delta_{b_{k+1}}L(R)$:

$$\begin{aligned}&= \{u \mid v \in \delta_{b_{k+1}}L(R), u \in v - B_1\} \\ &= \{u \mid v \in L(\delta_{b_{k+1}}R), u \in v - B_1\} \\ &= \delta_{B_1}(L(\delta_{b_{k+1}}R))\end{aligned}$$

And by the inductive hypothesis:

$$= L(\delta_{B_1}(\delta_{b_{k+1}}R)) = L(\delta_{b_1}(\delta_{b_2}(\ldots(\delta_{b_{k+1}}R))$$

■

4.3 The DERIVATIVE NON-EMPTINESS problem

In the previous section, we defined derivatives of regular expressions with respect to a *bag* of symbols. This is because we want to use derivatives in the static analysis of XPath queries posed on XML documents that are valid with respect to some Document Type Definition (DTD) D. More specifically, we want to determine, when querying documents valid with respect to D, whether a given node v in an XPath query q can have a set of child nodes whose labels form the bag B, irrespective of the order of the child nodes. This turns out to be equivalent to determining whether the derivative of the content model (regular expression) for v in D with respect to B is non-empty. We call this the DERIVATIVE NON-EMPTINESS problem.

We define the DERIVATIVE NON-EMPTINESS problem as follows: given regular expression R over finite alphabet Σ and (non-empty) bag $B \subseteq \Sigma$, is $L(\delta_B R)$ non-empty?

Next we show that DERIVATIVE NON-EMPTINESS is NP-complete. Then, in Chapter 6, we introduce a class of regular expressions for which DERIVATIVE NON-EMPTINESS can be decided in polynomial time.

In [91], the problem of STRING COVER was introduced in the context of determining containment of XPath queries. Given an alphabet Σ, let w be a string over Σ and B be a non-empty subset of Σ. We say that *w covers B* if w contains each symbol in B. Given a regular expression R over Σ and (non-empty) set $B \subseteq \Sigma$, STRING COVER asks if there is a string $w \in L(R)$ which covers B. STRING COVER was shown to be NP-hard in [91].

Theorem 4.2 DERIVATIVE NON-EMPTINESS *is NP-complete.*

Proof. We reduce STRING COVER to DERIVATIVE NON-EMPTINESS. Assume we are given R and (non-empty) set $B \subseteq \Sigma$. We show $\delta_B R$ is non-empty if and only if there is

a string $v \in L(R)$ which covers B.

(If) Assume that string $v \in L(R)$ covers B. Hence $B \subseteq [v]$ and therefore $\{u \mid v \in L(R), B \subseteq [v], u \in v - B\} \neq \emptyset$. Thus, $\delta_B R$ is non-empty.

(Only if) Assume that $\delta_B R$ is non-empty. Then, based on Equation (4.1), there is a $v \in L(R)$ such that $B \subseteq [v]$. This implies that v covers B.

We now show that the problem belongs to NP. Consider a certificate C for a "yes" instance to be a string $w \in L(R)$ such that $B \subseteq [w]$. We can easily check in polynomial-time in the sizes of C, B and R that the certificate is valid (e.g., by constructing an non-deterministic finite automaton(NFA) M for R to check whether $w \in L(M)$, and using a simple set-containment test). ■

We state the following straightforward result:

Proposition 4.1 DERIVATIVE NON-EMPTINESS *is in PTIME if the set B is a singleton.*

4.4 Main DTD Constraints in the Literature

In this section, we review several types of constraints previously introduced in the literature.

4.4.1 Constraints from Recursive DTDs

Child of First Node Constraints A DTD D implies the Child of First Node (CFN) Constraint $a \xrightarrow{/x} b$, where $a, b, x \in \Sigma$, if in every tree t in $SAT(D)$, on every path from an a-node to a b-node, the node immediately following the a-node must be an x-node [86].

Parent of Last Node Constraints A DTD D implies the Parent of Last Node (PLN) Constraint $a \xrightarrow{x/} b$, where $a, b, x \in \Sigma$, if in every tree t in $SAT(D)$, on every path from an a-node to a b-node, the node immediately preceding the b-node must be an x-node [86].

Essential Edge Constraints A DTD D implies the Essential Edge (EE) Constraint $a \overset{x/y}{\rightarrow} b$, where $a, b, x, y \in \Sigma$, if in every tree t in $SAT(D)$, every path from an a-node to a b-node, must contain an edge from an x-node to a y-node [86].

Example 4.1 *Consider the following recursive DTD rules:*

```
<!ELEMENT Books          (Book*)>
<!ELEMENT Book           (Publisher?,author+)>
<!ELEMENT Publisher      (Name, Book*)>
```

The DTD implies Books $\overset{/Book}{\rightarrow}$ *author, Books* $\overset{Book/}{\rightarrow}$ *author, and Books* $\overset{Book/Publisher}{\rightarrow}$ *Name (among others).*

4.4.2 Child Constraints

Given a DTD D, let a and c be symbols in Σ and R^a be the regular expression which is associated with a by D. Then D implies the Child Constraint $a \overset{R^a}{\rightarrow} c$ if in every tree t in $SAT(D)$, each a-node must have (at least) a c-child [5, 88]. When the regular expression R^a is clear from the context, it may simply be dropped and $a \rightarrow c$ be used.

The child constraints implied by the production rules in Example 4.1 are *Publisher* $\rightarrow$ *Name* and *Book* $\rightarrow$ *author*.

In what follows we show that there exists a relationship between the concept of derivatives and child constraints. Similar to [92], we define an *unordered regular expression* as follows:

Definition 4.4 *Let R be a regular expression over Σ. An unordered regular expression, $[R]$, is the set $\{[w] \mid w \in L(R)\}$.*

Theorem 4.3 *Let a and c be symbols in Σ and R^a be the content model for a. The child constraint $a \rightarrow c$ holds if and only if $[R^a] = [c\delta_c R^a]$.*

Proof. Trivially holds. ■

A child constraint $a \rightarrow c$ can be derived in PTIME [90]. The method is based on deleting transitions corresponding to c in an NFA M for R^a and then checking if $L(M)$ is empty.

4.4.3 Parent Constraints

Given a DTD D, let a and c be symbols in Σ and R^a be the regular expression which is associated with a by D. Then D implies the Parent Constraint $c \overset{R^a}{\leftarrow} a$ if, in every tree t in $SAT(D)$, each node labeled c must have a node labeled a as a parent [88]. When the regular expression R^a is clear from the context, it may simply be dropped and $c \leftarrow a$ be used.

4.4.4 Descendant Constraints

A DTD D implies a descendant Constraint $a \Rightarrow c$, where $a, c \in \Sigma$, if, in every tree t in $SAT(D)$, every node labeled a must have a descendant node labeled with c [88]. The child constraint $a \rightarrow c$ implies the descendant constraint $a \Rightarrow c$.

Descendant constraints are closed with respect to transitivity. In other words, if $a \Rightarrow c$ and $c \Rightarrow b$, then $a \Rightarrow b$.

4.4.5 Ancestor Constraints

A DTD D implies the Ancestor Constraint $a \Leftarrow c$, where $a, c \in \Sigma$, if in every tree t in $SAT(D)$, every node labeled c must always have an ancestor node labeled a [88]. The Parent constraint $c \leftarrow a$ implies the ancestor constraint $c \Leftarrow a$.

4.4.6 Cousin Constraints

A DTD D implies the Cousin Constraint (CC) $a : b \Rightarrow c$, where $a, b, c \in \Sigma$, means that in every tree t in $SAT(D)$, every node labeled a that has a descendant labeled b must also have a descendant labeled c [53].

4.4.7 Intermediate Node Constraints

A DTD D implies the Intermediate node Constraint (IC) $a \xrightarrow{c} b$, where $a, b, c \in \Sigma$, if in every tree t in $SAT(D)$, every path from an a-node to a b-node includes a c-node [53].

4.4.8 Parent-Child Constraints

A DTD D implies the Parent-child Constraint (PC) $a \Downarrow^1 b$, where $a, b \in \Sigma$, if for every t in $SAT(D)$, whenever a b-node is a descendant of an a-node, it is necessarily a child [53].

4.4.9 Sibling Constraints

A DTD D implies the SC $a : B \Downarrow c$, where $a, c \in \Sigma$ and $B \subseteq \Sigma$, if in every tree t in $SAT(D)$, each node labeled a that has children labeled with each $b \in B$, also has a child node labeled c [91].

A child constraint $a \rightarrow c$ is a special case of the SC $a : B \Downarrow c$, when $B = \emptyset$. The SCs shown in the previous example are not child constraints.

Given a DTD D, it has been shown that SCs implied by D capture D-containment for queries in $XP^{\{/,[]\}}$ that are duplicate-free. A query q in $XP^{\{/,[]\}}$ is duplicate-free if each node in the tree pattern corresponding to q has children with distinct labels. However, when DTDs are duplicate-free both SCs and FCs are necessary and sufficient to decide containment in PTIME [92]. Moreover, it has been proved that deciding whether a sibling constraint is implied by the content model of an element in a DTD is coNP-hard [91] in general, but it is in PTIME whenever the content model is duplicate-free.

In the following, we use the derivatives of regular expressions to reduce the problem of determining a sibling constraint $a : B \Downarrow c$ to the problem of determining a child constraint for the non-trivial case of $c \notin B$.

Theorem 4.4 *Let a and c be symbols in $\Sigma, B \subseteq \Sigma, c \notin B$, and R^a be the content model for a. The sibling constraint $a : B \Downarrow c$ holds if and only if the child constraint $a \overset{\delta_B R^a}{\longrightarrow} c$ holds.*

Proof. Trivially holds. ■

4.4.10 Family Constraints

A DTD D implies the Family Constraint $a[\$_1 b] \Downarrow [\$_2 c]$, where each of $\$_1$ and $\$_2$ is either / or //, if, whenever an a-node has a b-node as a child (if $\$_1$ is /) or descendant (if $\$_1$ is //), then it also has a c-node as a child (if $\$_2$ is /) or descendant (if $\$_2$ is //), respectively [67]. As special cases, if every a-node must have a c-node as a child (or descendant), this can be shown by $a \Downarrow [/c]$ (or $a \Downarrow [//c]$). When both $\$_1$ and $\$_2$ are / (respectively //), then a family constraint corresponds to an SC (respectively CC).

Example 4.2 *Consider the following DTD:*

```
<!ELEMENT a   (b | (c, (d|e)))>
<!ELEMENT b   (c)>
<!ELEMENT d   (f)>
<!ELEMENT e   (f)>
```

If an a-node has a c-child then it must have an f-descendant, and if an a-node has an f-descendant then it must have a c-child. So the family constraints $a[/c] \Downarrow [//f]$ and $a[//f] \Downarrow [/c]$ hold. Note that an a-node can have a c-descendant without having an f-descendant, so a CC will not capture the first constraint.

4.4.11 Functional Constraints

A DTD D implies the functional constraint (FC) $a \downarrow b$, where $a, b \in \Sigma$, if for every t in $SAT(D)$, no node labeled a can have two distinct children labeled b [91]. Deriving the set of FCs implied by a DTD can be done in PTIME [92].

In [92] deriving FCs was stated to be in PTIME. However, no formal proof was presented for this purpose. In what follows we use the concept of derivatives to show this.

Theorem 4.5 *Let a and c be symbols in Σ^a and R^a be the content model for a. The functional constraint $a \downarrow c$ holds if and only if $\delta_{c^2} R^a = \emptyset$.*

Proof. Trivially holds. ■

In the next section, we generalise the definition of SC and FC into two new types of constraints, respectively, called *Bag Sibling Constraint (BSC)* and *Bag Functional Constraint (BSC)*. We use these constraints in Chapter 6, where we show that they are sufficient and necessary to capture containment of $\text{XP}^{\{/,[]\}}$ under well-behaved DTDs.

4.5 Bag Sibling and Functional Constraints

In this section, we introduce two types of constraints which capture D-containment of queries in $\text{XP}^{\{/,[]\}}$ under well-behaved DTDs. These constraints are generalised forms of the well-known FCs and SCs and are called, respectively, *bag functional constraints (BFCs)* and *bag sibling constraints (BSCs)*. We first define the notion of an *infinity bag schema*.

Definition 4.5 *An* infinity bag schema *B is a bag specification in which the annotation on each element is either 1 or > 1. By $x^1 \in B$ we mean that the multiplicity of x in B is 1, while by $x^{>1} \in B$ we mean the multiplicity of x in B can be any value greater than 1. By $x \in B$, we mean either $x^1 \in B$ or $x^{>1} \in B$.*

Note that an infinity bag schema is not a bag but a specification of a set of bags. More specifically, an infinity bag schema B specifies the set of bags B_i such that for each member $b^1 \in B$, $|b|_{B_i} = 1$, for each $b^{>1} \in B$, $|b|_{B_i} > 1$, and B_i includes no symbol other than those appearing in B.

Definition 4.6 *Let A and B be two infinity bag schemas. We say $A \subseteq B$, if $\forall x \in A$ we have:*

- *if $x^{>1} \in A$, then $x^{>1} \in B$*
- *if $x^1 \in A$, then $x^1 \in B$ or $x^{>1} \in B$*

Now let B be an ordinary bag. Then $A \subseteq B$ if $\forall x \in A$:

- *if $x^{>1} \in A$, then $x^k \in B$, for some $k > 1$*
- *if $x^1 \in A$ then $x^k \in B$, for some $k \geq 1$*

Definition 4.7 *Let A and B be two infinity bag schemas, then $A \cup B = \{x^i \mid x \in A \text{ or } x \in B\}$ where :*

$$i = \begin{cases} > 1 & \text{if } x^{>1} \in A \text{ or } x^{>1} \in B \\ 1 & \text{otherwise} \end{cases}$$

and also we define $A \uplus B = \{x^i \mid x \in A \text{ or } x \in B\}$ such that:

$$i = \begin{cases} > 1 & \text{if } (x^1 \in A \text{ and } x^1 \in B) \text{ or } x^{>1} \in A \text{ or } x^{>1} \in B \\ 1 & \text{otherwise} \end{cases}$$

Definition 4.8 *A* bag sibling constraint *(BSC) is of the form $a : B \Downarrow c$, where $a, c \in \Sigma$, and B is a possibly an infinity bag schema defined on Σ. A DTD D implies the BSC $a : B \Downarrow c$ if in every tree t in $SAT(D)$, every node labeled 'a' that has a bag of children A such that $B \subseteq A$ also has a child node labeled c.*

Definition 4.9 *A* bag functional constraint *(BFC) is of the form $a : B \downarrow c$, where $a, c \in \Sigma$ and B is a possibly infinity bag schema defined on Σ. A DTD D implies the BFC $a : B \downarrow c$*

if in every tree t in $SAT(D)$, every node labeled 'a' that has a bag of children A such that $B \subseteq A$, can have at most one child node labeled c.

Example 4.3 *Consider the following DTD rule:*

$$a \quad \rightarrow \quad (b*, d) \,|\, (b, d*)$$

While this DTD rule does not imply any FCs or SCs, it does imply the BSCs $a : \{b^{>1}\} \Downarrow d$ and $a : \{d^{>1}\} \Downarrow b$, and the BFCs $a : \{b^{>1}\} \downarrow d$ and $a : \{d^{>1}\} \downarrow b$.

Note that for a bag B used in (the left hand side of) a BFC (respectively BSC), it does not matter whether the degree of a label in B is 2 or any greater integer. This property facilitates determination/application of BFCs and BSCs. In other words, it is only important whether the occurrence of a label in B is 0, 1, or > 1.

4.5.1 Properties and axioms for BSC and BFC

In this section, we introduce sound and complete axioms for BSCs and BFCs. The motivation for defining these axioms is reducing redundancy in set of DTD constraints. The running times of query optimization, containment and minimisation algorithms depend on the size of the set of constraints used as input. A smaller set of constraints guarantees faster execution of some algorithms, for example, the chase procedure.

BSC axiomatization

Let X be an infinity bag schema and $c \in \Sigma$. We use $X \Downarrow c$ and $X \downarrow c$, respectively, as opposed to $a : X \Downarrow c$ and $a : X \downarrow c$ to denote a BSC and a BFC, when the a-node is known from the context. We also use $\mathbb{C}$ to denote a set of BSCs and/or BFCs. We now present axioms satisfied by BSCs, which are analogous to the standard axioms defined for functional dependencies (FDs) in relational databases [60].

Definition 4.10 *Let v be a node with a bag of children B'. By saying that $v(X)$ holds, or is true, we mean $X \subseteq B'$. Also, saying that $v(c)$ holds or is true, means that the node v has a c-child.*

Let $X, Y = \{y_1, \ldots, y_m\}$, and Z be infinity bag schemas in Σ, $c, b \in \Sigma$ and $\mathbb{C}$ denote a set of BSCs. Let $\cup$, $\uplus$ and $\subseteq$ be the maximal union, additive union and subset operators defined on infinity bag schemas. For any BSC in $\mathbb{C}$, the following axioms hold:

1. $X \Downarrow c$ holds for each $c \in X$
2. $X \Downarrow c$ and $X \subseteq Y \Rightarrow Y \Downarrow c$
3. $X \Downarrow y_1, \ldots, X \Downarrow y_m$ and $Y \cup Z \Downarrow c \Rightarrow X \cup Z \Downarrow c$

Note that axiom 3 will become the pure transitivity axiom when $Z = \emptyset$. However, its general case, allowing for augmentation by Z, is proposed here for showing completeness of the set of axioms 1 to 3, as shown later in this section. Other rules which may be derived from the axioms include:

- $X \Downarrow c$ and $\{c\} \Downarrow b \Rightarrow X \Downarrow b$
- $X \Downarrow c$ and $\{c\} \cup Y \Downarrow b \Rightarrow X \cup Y \Downarrow b$
- $X \cup Y \Downarrow c \Rightarrow X \uplus Y \Downarrow c$.

Theorem 4.6 *Axioms 1 to 3 are sound.*

Proof. For axiom 1, we assume that $v(X)$ holds for some node v. Now if $c \in X$ and $v(X)$ holds, then obviously $v(c)$ holds. To prove axiom 2, we assume that $v(Y)$ holds for some node v. Now if $X \subseteq Y$ and $v(Y)$ holds, then $v(X)$ is true. Given that $X \Downarrow c$ holds, then $v(c)$ must hold. To prove axiom 3, assume that $X \Downarrow y_1, \ldots, X \Downarrow y_m$ and $Y \cup Z \Downarrow c$ as well as $v(X \cup Z)$ hold. Since $v(X \cup Z)$ holds, obviously $v(X)$ and $v(Z)$ must hold. Then $v(y_i)$ holds($1 \leq i \leq m$) which means $v(Y)$ is true. Since $v(Y \cup Z)$ is true, $v(c)$ is true because of

the assumption $Y \cup Z \Downarrow c$. ■

Definition 4.11 *Let $\mathbb{C}$ be a set of BSCs. We denote by $\mathbb{C}^+$ the* closure *of $\mathbb{C}$, i.e, the smallest set containing $\mathbb{C}$ such that axioms 1 to 3 cannot be applied to the set to yield a constraint not in the set.*

Definition 4.12 *Let X be an infinity bag schema $X \subseteq X^+$ and let $c \in \Sigma$. We say that X^+ is the* BSC bag closure *of X if for any constraint $X \Downarrow c$ in $\mathbb{C}^+$, $c \in X^+$.*

Definition 4.13 *A set $\mathbb{C}$ of BSCs implies a constraint $X \Downarrow c$, written $\mathbb{C} \models X \Downarrow c$, if every document tree d which satisfies all the constraints in $\mathbb{C}$ also satisfies the constraint $X \Downarrow c$.*

Theorem 4.7 *Axioms 1 to 3 are complete, i.e., given a set $\mathbb{C}$ of BSC constraints, if $\mathbb{C} \models X \Downarrow c$ then $X \Downarrow c \in \mathbb{C}^+$.*

Proof. The following proof has been adapted from [60] for the corresponding theorem on FDs. We will show that if $X \Downarrow c$ is not in $\mathbb{C}^+$, then $\mathbb{C} \models X \Downarrow c$ will not hold either. To that end, assume that the BSC constraint $X \Downarrow c$ is not in $\mathbb{C}^+$. We will show there exists a document tree containing a node v that satisfies $\mathbb{C}$ but not $X \Downarrow c$.

Let v have children labeled with each symbol in X only and not having a c child. Given that $X \Downarrow c$ is not in $\mathbb{C}^+$, we claim that the document tree t containing the node v satisfies $\mathbb{C}$ but not $X \Downarrow c$. The only way the node v does not conform to some constraints in $\mathbb{C}$ is that the constraints in $\mathbb{C}$ do not allow v to have the children labeled with the symbols in X unless it also has a child labeled with c, which is possible only if (1) c is a member of X, (2) there is a constraint $X_1 \Downarrow c$ in $\mathbb{C}$, $X_1 \subseteq X$, or (3) there is a chain of constraints that imply v has children labeled with each label in a set Y and there is a constraint $Y \cup Z \Downarrow c$ (where Z may be empty in the special case) in $\mathbb{C}$ as well. We now go through each of these cases and show that none of them is applicable.

In case (1), if c was a member of X, then axiom 1 would apply and put the constraint $X \Downarrow c$ in $\mathbb{C}^+$, which is a contradiction. Similarly, in case (2), axiom 2 would add the constraint $X \Downarrow c$ to $\mathbb{C}^+$, a contradiction. Now consider case (3). In this case there is a constraint $Y \cup Z \Downarrow c$ in $\mathbb{C}$, $Y = \{y_1, \ldots, y_m\}$ and some constraints in $\mathbb{C}$ imply the constraints $X \Downarrow y_k$ (i.e. $\mathbb{C} \models X \Downarrow y_k$), $k = 1, \ldots, m$. There are now two sub cases: (a) all the constraints $X \Downarrow y_k$, $k = 1, \ldots, m$ are in $\mathbb{C}^+$, or (b) at least one of them, say $X \Downarrow y_j$, is not a member of $\mathbb{C}^+$. In the former case, the iterative application of axiom 3 to the constraints $X \Downarrow y_k$, $k = 1, \ldots, m$ will yield the constraint $X \Downarrow c$ and we will have $X \Downarrow c \in \mathbb{C}^+$. On the other hand, the latter case, case (b), implies that there is also another constraint $X \Downarrow y_j$, in addition to the original constraint $X \Downarrow c$, that is implied by $\mathbb{C}$ but is not in $\mathbb{C}^+$. To summarise our reasoning so far, we assumed that the constraint $X \Downarrow c$ is implied by $\mathbb{C}$ but does not belong to $\mathbb{C}^+$ and we have concluded that there must be another constraint $X \Downarrow y_j$ that is so, i.e. is implied by $\mathbb{C}$ but does not belong to $\mathbb{C}^+$. This will give rise to a contradiction because repeating the same reasoning with the new constraint $X \Downarrow y_j$ will in turn imply there will be yet another constraint that is implied by $\mathbb{C}$ but not in $\mathbb{C}^+$ and these constraints only differ in their right hand symbol, but the number of distinct symbols is finite. ■

BFC axiomatization

The only inference axiom for BFCs is

$$X \downarrow c \text{ and } X \subseteq Y \subseteq \Sigma \Rightarrow Y \downarrow c$$

Other rules which may be derived from the inference axiom are:

1. $X \downarrow c \Rightarrow X \cup Y \downarrow c$
2. $X \cup Y \downarrow c \Rightarrow X \uplus Y \downarrow c$

To prove rule 1, assume that $X \downarrow c$ and also $v(X \cup Y)$. Obviously $v(X)$ holds and also $X \cup Y \downarrow c$ because of the assumption $X \downarrow c$. The correctness of rule 2 follows from the inference axiom and the fact that $X \cup Y \subseteq X \uplus Y$.

Theorem 4.8 *The inference axiom for BFCs is sound and complete.*

Proof. To prove that the axiom is sound, assume that $X \downarrow c$ holds and $X \subseteq Y \subseteq \Sigma$. Now if $v(Y)$ holds, $v(X)$ also holds because $X \subseteq Y$. Then $v(c)$ is true because of $X \downarrow c$.

To prove completeness, we will show that if $X \downarrow c$ is not in $\mathbb{C}^+$, then $\mathbb{C} \models X \downarrow c$ will not hold either. To that end, assume that the BFC constraint $X \downarrow c$ is not in $\mathbb{C}^+$. We will show there exists a document tree containing a node v that satisfies $\mathbb{C}$ but not $X \downarrow c$.

Let v have children labeled with each symbol in X and it has more than one c child. Given that $X \downarrow c$ is not in $\mathbb{C}^+$, we claim that the document tree t containing the node v satisfies $\mathbb{C}$ but not $X \downarrow c$. The only way the node v does not conform to some constraints in $\mathbb{C}$ is that the constraints in $\mathbb{C}$ do not allow v to have the children labeled with the symbols in X unless it also has only a single c-child. This is possible only if there is a constraint $Y \downarrow c$ in $\mathbb{C}$, $X \subseteq Y$. In this case the axiom would add the constraint $X \downarrow c$ to $\mathbb{C}^+$, a contradiction. ■

Note that the axioms that hold for BSCs are not true for BFCs, for instance $a : X \downarrow c$ and $a : \{c\} \downarrow d \not\models a : X \downarrow d$. As an example, consider the following DTD rule:

```
<!ELEMENT a        ((y,t)|(x,t)|t*|(x,y)*)>
```

The constraints $a : \{y\} \downarrow t$ and $a : \{t\} \downarrow x$ hold but $a : \{y\} \downarrow x$ does not.

4.6 Conclusion

In this chapter we first defined a variant of the concept of derivatives of regular expressions customised for DTDs. We proved that even to decide whether or not the derivative of a DTD rule is empty is NP-hard. We then showed the relationship between the concept of

derivatives of regular expressions and existing DTD constraints. We also introduced two types of DTD constraints along with their inference axioms.

The main motivation for studying DTD constraints is their importance as a useful tool to decide the containment problem in the presence of DTDs. In fact, the presence of a DTD D can give rise to *implicit nodes* in an XPath query. An implicit node is one that does not explicitly exist in the query but has to exist in every document satisfying the query and the DTD. This implies that two syntactically different queries might have the same semantics. Moreover, there may be two nodes in one query pattern mapped to the same document node in all trees in $SAT(D)$. For example, consider query $a[b/c][b/d][c/e][c/f]$ and the following DTD rules:

```
<!ELEMENT a       ((b*,c,d)|(b,c*))>
<!ELEMENT b       (c|d)>
<!ELEMENT c       (e?, f?)>
```

According to the DTD, every node labeled a which has more than one b-child, must also have only one c-child and only one d-child. This implies the BSCs $a : \{b^{>1}\} \Downarrow c$ and $a : \{b^{>1}\} \Downarrow d$ and the BFCs $a : \{b^{>1}\} \downarrow c$ and $a : \{b^{>1}\} \downarrow d$. So the query is equivalent to $a[b/c][b/d][c[e][f]][d]$, where the two copies of c have been merged and the implicit node d has been added.

To find an implicit node typically we want to check whether or not a given constraint is in $\mathbb{C}^+$. One way to answer this question is to check whether the constraint can be derived from $\mathbb{C}$ by using the axioms, which may take a long time. The other way is to use the concept of BSC bag closure which is similar to the concept of attribute closure in relational databases. Using attribute closure to check if a given FD can be derived from a given set of FDs is a well-known method in relational databases. More explicitly, to check if a given constraint, $X \Downarrow c$, can be derived from a set of constraints, i.e. deciding whether $\mathbb{C} \models X \Downarrow c$, the BSC bag closure X^+ can first be determined and then checked for whether it contains c.

Chapter 5

XPath Satisfiability under DTDs

In this chapter, we concentrate on the satisfiability problem under two special classes of DTDs, for a variety of XPath fragments with child axis ('/'), descendant axis ('//'), qualifier ('[]'), wildcard ('$*$') and union ('$\cup$'). The two classes of DTDs we consider are *duplicate-free* DTDs and *covering* DTDs. Most of the results presented in this chapter were published previously in [68]. Informally, a duplicate-free DTD is one in which no regular expression uses the same symbol more than once. A covering DTD, on the other hand, is one in which each regular expression R is such that the language $L(R)$ contains a string in which all the symbols used in R appear. Formal definitions of these properties are provided in Section 5.1.

We show that the classes of covering and duplicate-free DTDs comprise most real-world DTDs, i.e. those used in real-world applications. We identify a number of XPath fragments for which the complexity of the satisfiability problem reduces to PTIME when duplicate-free or covering DTDs are used. For example, the fact that satisfiability for the fragment $\mathrm{XP}^{\{/,[]\}}$ is NP-hard in general follows from a result in [91]. Here we show that it becomes decidable in PTIME for duplicate-free DTDs, although results from [7] imply that it remains NP-hard for the fragments $\mathrm{XP}^{\{/,[],*\}}$ and $\mathrm{XP}^{\{/,[],//\}}$. More significantly, for covering DTDs we show that satisfiability for the fragment $\mathrm{XP}^{\{/,[],*,//,\cup\}}$ is in PTIME.

The next section contains the definitions of the various forms of DTDs and fragments of XPath. Section 5.2 provides the results of our investigation into the relative frequency of covering and duplicate-free real-world DTDs. Section 5.3 presents our complexity results for a number of XPath fragments under duplicate-free and covering DTDs. Finally, Section 5.4 concludes the chapter.

5.1 Notation and Background Material

In this section, we define various subclasses of DTDs, the XPath fragments studied in this chapter, and the notion of XPath satisfiability.

Recall from Chapter 4 that we will use the DTD syntax for regular expressions, namely, ',' for concatenation, '|' for alternation (disjunction), '$*$' for reflexive transitive closure, '$+$' for transitive closure and '?' for optional.

XPath supports a wide variety of operators whose presence or absence affects the complexity of the satisfiability problem. This has led to the study of various XPath fragments that include only certain operators. For example, in this chapter we study the fragment with child axis ('$/$'), descendant axis ('$//$'), qualifier ('$[]$'), wildcard ('$*$') and union ('$\cup$'). Larger fragments allow operators such as negation, additional axes such as parent, ancestor and sibling, as well as comparisons involving data values or node identities [41].

Example 5.1 *The XMark benchmark project*[1] *is based on an online auction application. A fragment of the XMark DTD is given below:*

```
site          (regions, categories, catgraph, people, open_auctions,
               closed_auctions)
categories    (category+)
category      (name, description)
description   (text | parlist)
```

[1] `http://monetdb.cwi.nl/xml/`

```
open_auctions (open_auction*)
open_auction  (initial, reserve?, bidder*, current, privacy?, itemref,
               seller, annotation, quantity, type, interval)
```

with `site` *being the document (top-level) element. The XPath query*

```
/site/open_auctions/open_auction[bidder][reserve]/seller
```

selects `seller` *nodes that are children of* `open_auction` *nodes that have both a* `bidder` *and* `reserve` *child. It is easy to see that this query is satisfiable on documents valid with respect to the above DTD fragment.*

In the full DTD, a `description` *can occur as a descendant of more than one element, so one might write*

```
/site//description[text][parlist]
```

to retrieve all `description` *nodes that have both* `text` *and* `parlist` *children. However, this query is unsatisfiable with respect to the DTD, because a* `description` *can have only one of* `text` *or* `parlist` *as a child, not both.*

Definition 5.1 *Let R be a regular expression and Σ be the set of symbols appearing in R. We say that R* covers *Σ, or simply that R is* covering*, if there is a string in $L(R)$ that contains every symbol in Σ. A DTD D is called* covering *if and only if each content model in D is covering.*

Note that a number of common content models used in DTDs are covering. For example, the content models one gets from the naive representation of relational data as XML are covering, as are the so-called mixed content models found in "document-oriented" XML. Some examples of covering and non-covering content models are given in Example 5.2 below.

The notion of a duplicate-free DTD was introduced in [91, 92]. Although the definition was given in Chapter 3, we repeat it here for convenience. Let R be a regular expression

and Σ be the set of symbols appearing in R. R is *duplicate-free* if each symbol in Σ occurs exactly once in R. A DTD D is called *duplicate-free* if and only if each content model in D is duplicate-free.

Example 5.2 *All of the DTD rules for the XMark DTD fragment shown in Example 5.1 are covering, except the following*

```
description   (text | parlist)
```

since the language denoted by `(text | parlist)` *does not contain a sequence that includes both element names. In addition, all of the rules in the XMark fragment are duplicate-free. The following is an example of a rule with duplicates, taken from the XML Schema DTD*[2] *after replacing entity references and ignoring namespace prefixes*

```
schema ((include | import | redefine | annotation)*,
        ((simpleType | complexType | element | attribute
          | attributeGroup | group | notation), (annotation)*)* )
```

where the element name `annotation` *is repeated.*

Note that the definition of duplicate-free is syntactic. In other words, we can have two regular expressions which denote the same language such that one expression is duplicate-free while the other is not. For example, $a?,b$ and $(a,b)|b$ denote the same language, but only the former expression is duplicate-free.

A number of other subclasses of DTDs have been defined in order to study the complexity of problems such as XPath satisfiability. For example, [7] considers disjunction-free DTDs, while [11] considers simple regular expressions defined as follows.

Definition 5.2 *A* base symbol *is a regular expression a, $a?$ or a^* where $a \in \Sigma$; a* factor *is of the form e, e^* or $e?$ where e is a disjunction of base symbols. A* simple regular expression *is ε, $\emptyset$ or a sequence of factors.*

[2] `http://www.w3.org/2001/XMLSchema.dtd`

Clearly, a simple regular expression need not be duplicate-free nor covering. On the other hand, $a|(b,c)$ is duplicate-free but not simple, and $(a,b)^*$ is covering but not simple. We conclude that the 3 subclasses of DTDs are pairwise incomparable.

Definition 5.3 *The syntax of XPath expressions used in this chapter is given by the following grammar:*

$$\begin{aligned} q &\rightarrow \text{`}/\text{'}\ p \\ p &\rightarrow p\ \text{`}/\text{'}\ p \mid p\ \text{`}//\text{'}\ p \mid p\ \text{`}\cup\text{'}\ p \mid p\ \text{`}[\text{'}\ p\ \text{`}]\text{'} \mid \text{`}*\text{'} \mid n \mid \text{`}.\text{'} \end{aligned}$$

where q is the start symbol, n is an element name and '.' refers to the context node.

As mentioned earlier, fragments of XPath are denoted by indicating which operators are supported. So the above fragment is denoted by $\text{XP}^{\{/,[],*,//,\cup\}}$, since child axis ('/'), descendant axis ('//'), qualifiers ('[]'), wildcard ('$*$') and union ('$\cup$') are all permitted.

Papers such as [7] use an alternative syntax where $\downarrow$ denotes use of the child axis without specifying an element name. So $\downarrow$ in their syntax corresponds to $*$ in ours, with $\downarrow/a$ corresponding to a. One consequence of this is that '$*$' is implicitly permitted in all the XPath fragments considered by [7] that include the child axis, whereas we distinguish explicitly whether or not $*$ is included.

Definition 5.4 *The following notation is adapted from [7]. Given a DTD D and a query p, p is D-*satisfiable *if there is an XML tree $t \in SAT(D)$ such that the answer of p on t is not empty, denoted by $t \models p$. Given a DTD D, we denote the fact that an XML tree* satisfies *(or is* valid *with respect to) D by $t \models D$. Given a DTD D and a query p, an XML tree t* satisfies *p and D, denoted by $t \models (p,D)$, iff $t \models p$ and $t \models D$. For an XPath fragment X, the* XPath satisfiability problem *$SAT(X)$ is, given a DTD D and a query p in X, is there an XML tree t such that $t \models (p,D)$?*

5.2 Real-World DTDs

In this section, we report on our investigation of "real-world" DTDs, i.e. those frequently used in real applications. For the purpose of this chapter, we were concerned with two features of such DTDs, namely whether or not they were duplicate-free or covering. We will see that most of the real-world DTDs we studied have at least one of these two properties.

In order to examine the frequency of covering and duplicate-free DTD rules in real-world applications, we obtained 100 real-world DTDs, 86 using the Google search engine and 14 from the XML Data Repository[3]. The DTD names and a brief description of their application domains are given in Table A.1 in Appendix A.

Table 5.1 shows the classification of 20 DTDs with respect to the covering and duplicate-free properties. The first and the second columns of Table 5.1 show, respectively, the DTD names, and the number of rules in each DTD. The last four columns show, respectively, the number of rules that are (i) covering and duplicate-free, (ii) covering with duplicates, (iii) non-covering but duplicate-free, and (iv) non-covering with duplicates. The complete results (100 DTDs) are shown in Table B.1 in Appendix B.

A quick glance at Table 5.1 reveals that the majority of the rules (91.3%) in these applications possess both the covering and duplicate-free properties. Most of the rest (7.9%) have exactly one of these properties, and less than 1.0% of the rules are neither covering nor duplicate-free.

It should be pointed out that the 3 rules from the Music ML DTD[4] that contain duplicates are as follows:

```
musicrow     ((entrysegment, segment+) | (entrysegment, segment+, text))
entrysegment ((entrypart) | (entrypart, entrypart))
segment      ((subsegment) | (subsegment, subsegment))
```

[3] http://www.cs.washington.edu/research/xmldatasets

[4] http://xml.coverpages.org/musicML-DTD.txt

Table 5.1: The classification of DTD rules

DTD Name	**Number of Rules**	**Non-covering**		**Covering**	
		Dup-free	Dup	Dup-free	Dup
CDisc-11	85	0	0	84	1
Docbooks	360	18	7	314	21
Docutils	89	3	0	86	0
Ecoknowmics	224	1	0	221	2
FOT	83	0	0	83	0
FGDC-1.00	340	5	2	315	18
Geophysical ML	444	0	1	414	29
kpresenter-1.3	86	0	0	86	0
kword-1.3	79	0	0	76	3
News ML	116	0	0	112	4
Oagis	617	161	18	422	16
Resume	106	0	0	97	9
Sun-domain-1.20	109	0	0	108	1
TieXLite	143	26	6	109	2
web-facesconfig-1-1	80	0	0	76	4
web-app-2-3	77	0	0	74	3
XHTML1-Frameset	91	1	0	88	2
XHTML1-Strict	77	1	0	74	2
XHTML1-Transitional	89	1	0	86	2
XMark	77	1	0	75	1
Other	2162	19	10	2052	81
Total	5534	236	44	5053	201
Percentage	100%	4.3	0.8	91.3	3.6%

These rules are not even "unambiguous" as required by the XML specification. They can, however, easily be rewritten to be unambiguous as follows

```
musicrow     (entrysegment, segment+, text?)
entrysegment (entrypart, entrypart?)
segment      (subsegment, subsegment?)
```

resulting in the first rule becoming duplicate-free as well.

Because of our assumption that even a single rule that is non-covering (or contains duplicates) results in the DTD being classified as non-covering (or not duplicate-free), we need to determine which DTDs, as opposed to which rules, are covering (duplicate-free). Table 5.2 classifies the examined 100 DTDs into the four categories of covering and

duplicate-free (top left), covering with duplicates (top right), non-covering but duplicate-free (bottom left) and non-covering with duplicates (bottom right). As shown in Table 5.2, the largest number (47%) of DTDs possess both properties, with only 17% possessing neither property. These experiments show that most real-word DTDs should be covering or

Table 5.2: The number of DTDs (out of 100) in each of the four categories

	Duplicate-free	Duplicates
Covering	47	8
Non-covering	28	17

duplicate-free. In fact, 55% of the DTDs examined in the experiments were covering, and about 62% of the remaining (non-covering) DTDs (i.e. 28% of the whole) were duplicate-free. That is, 83% of the examined DTDs possessed at least one of the properties of being covering or duplicate-free.

5.3 XPath Satisfiability under Real-World DTDs

In this section, we concentrate on the satisfiability problem of XPath queries under "real-world" DTDs, i.e. those which are either duplicate-free or covering. We will see that although the satisfiability problem is NP-complete or worse for many XPath fragments under general DTDs, it is in PTIME for certain XPath fragments when the underlying DTDs have the duplicate-free or covering property.

5.3.1 XPath Satisfiability under Duplicate-free DTDs

Recall that a DTD is duplicate-free if each element name appears at most once in each content model. The fact that duplicate-free DTDs are easier to analyze was previously noted in [92], where it is shown that deciding containment under a DTD even for $XP^{\{/,[]\}}$ is coNP-complete, but that it reduces to PTIME when the DTD is duplicate-free. Below we show that the analysis of XPath satisfiability under duplicate-free DTDs is also simpler for certain fragments.

Before doing so, we state the following straightforward result.

Proposition 5.1 *Given a DTD D, deciding whether D is duplicate-free can be done in PTIME.*

Benedickt *et al.* show that, in general, SAT($XP^{\{/,[],*\}}$) and SAT($XP^{\{/,[],//\}}$) are both NP-hard [7]. In fact, a result in [91] implies that SAT($XP^{\{/,[]\}}$) is also NP-hard. However, we have the following result for duplicate-free DTDs.

Theorem 5.1 *Under duplicate-free DTDs, SAT($XP^{\{/,[]\}}$) is in PTIME.*

Proof. To prove the theorem, we first present two lemmas:

Lemma 5.1 *Let R be a duplicate-free regular expression and C be a nonempty subset of symbols appearing in R. Then there is a string w_c in $L(R)$ which covers all the symbols in C if and only if every subexpression $(R_1|R_2)$ of R, where both R_1 and R_2 contain a symbol in C, appears as a subexpression of $(R_3)^*$ or $(R_3)^+$ for some R_3.*

Proof. It is important to note firstly that R is duplicate-free and that R is assumed to use every symbol in C. Therefore, each symbol in C appears exactly once in R. Moreover, R_1 and R_2 should each contain a *different* symbol from C for the result to hold.

Assume R contains a subexpression $(R_1|R_2)$ to which no closure operator applies, such that R_1 contains $c_1 \in C$ and R_2 contains $c_2 \in C$, where $c_1 \neq c_2$. Then each string in $L(R)$ containing c_1 has to exclude c_2 and each string in $L(R)$ containing c_2 has to exclude c_1, which means that no string in $L(R)$ covers C.

Conversely, assume there is no string in $L(R)$ which covers C. Since all the symbols in C appear in R, there must be a pair of distinct symbols c_1 and c_2 in C such that there are strings w_1 and w_2 in $L(R)$ such that c_1 (but not c_2) appears in w_1 and c_2 (but not c_1) appears in w_2, but no string in $L(R)$ contains both c_1 and c_2. Hence there must be a subexpression $(R_1|R_2)$ in R such that c_1 appears in R_1 (or R_2) and c_2 appears in R_2 (respectively R_1). Furthermore, the expression $(R_1|R_2)$ cannot be subject to a closure operator; otherwise there

would be a string in $L(R)$ containing both c_1 and c_2. ■

Lemma 5.2 *Let p be a two-level XPath query in the fragment $XP^{\{/,[]\}}$ such that the root v_{root} has $n \geq 0$ leaf children. Then the satisfiability of p under a duplicate-free DTD D can be decided in PTIME.*

Proof. Let R_{root} be the regular expression representing the content model in the DTD D for the root node v_{root} of p. In the case that more than one child of v_{root} has the same label, say b, either the symbol b is subject to some closure operator in R_{root} (i.e. "$*$" or "$+$" applies to b or to a term in which b is contained) or it is not. In the former case, v_{root} can have a b-child (while D-consistent) if, and only if, it can have many of them, and in the latter case all such b-children must map to the same b-node in any document tree which satisfies D. Therefore, we only need to check whether $L(R_{root})$ contains a word w_c which includes all of the labels in C (with an arbitrary ordering), where C is the set, as opposed to the bag, of labels of the children of v_{root}.

For each symbol b in C, if there is no symbol b in R_{root}, which can be checked in PTIME, then the answer (to whether $L(R_{root})$ contains w_c) is false. Otherwise, Lemma 5.1 applies and we only need to check whether R_{root} contains some expression $(R_1|R_2)$ to which no closure operator applies and such that both R_1 and R_2 contain some different symbol in C. The number of "|" operators in R_{root} is O($|R_{root}|$) and to obtain each expression $(R_1|R_2)$ in R_{root} requires O($|R_{root}|$) time. For each expression $(R_1|R_2)$ in R_{root}, to check whether R_i, $i = 1,2$, covers some symbol in C requires $O(|R_i| \times |C|)$ time. ■

We now prove the theorem:

Let p be a given XPath query in the fragment $XP^{\{/,[]\}}$. For each internal node v in p, if v has more than one child with the same label, say b, then there are two possibilities: (i) the symbol b is subject to some closure operator in the regular expression, say R_v,

corresponding to the content model of v in the DTD (i.e. "$*$" or "$+$" applies to b or to a term in which b is contained), or (ii) the symbol b is not subject to a closure operator, hence all such b-children must map to the same b-node in any document tree which satisfies the DTD D (because D is duplicate-free). The latter case was previously called a *functional constraint* in [92] and was shown to be detectable in PTIME. Let us denote by $merge(p)$ the query tree resulting from the merging of nodes satisfying case (ii). It is clear that p is satisfiable iff $merge(p)$ is satisfiable.

In case (i), there is no restriction on the number of b-children of $\lambda(v)$ (where $\lambda(v)$ denotes the label of v) in any document tree which satisfies D, hence such b-children need not be merged. Let $subMerge(v)$ denote the two-level subtree of p rooted at v. This implies that $subMerge(v)$ has all the children of v as leaves.

Based on the above terminology and the definition of satisfiability, p is satisfiable if and only if the subtree $subMerge(v)$ is satisfiable for each internal node v in p. Using Lemma 5.2, to decide whether such a subtree is satisfiable is in PTIME. On the other hand, there are altogether m subtrees, where m is the number of internal nodes, that is $m = O(|p|)$. Therefore, to decide whether p is satisfiable is in PTIME. ■

Even if a DTD as a whole is not duplicate-free, the above positive results can be used. For example, given a DTD D and a query p in $\text{XP}^{\{/,[]\}}$, if every internal node in the tree representing p is labeled by an element name whose content model is duplicate-free, then the satisfiability of p can be determined in PTIME. This gives us the following.

Corollary 5.1 *Given a query p in $\text{XP}^{\{/,[]\}}$ and a DTD D, SAT($\text{XP}^{\{/,[]\}}$) is in PTIME if each rule in D in which a symbol from p appears is duplicate-free.*

We now consider the satisfiability of some other fragments of XPath under duplicate-free DTDs. The fact that $\text{SAT}(\text{XP}^{\{/,//,*\}})$ is in PTIME under duplicate-free DTDs follows trivially from a result in [7] showing that this fragment including union is in PTIME in general. However, we have the following negative results.

Theorem 5.2 *Under duplicate-free DTDs, the following problems are NP-hard:*

1. *SAT(XP$^{\{/,[],*\}}$)*
2. *SAT(XP$^{\{/,[],//\}}$)*
3. *SAT(XP$^{\{/,[],\cup\}}$)*

Proof. (1) We use the same method as used in [7], where the authors show that SAT(XP$^{\{/,[],*\}}$) is NP-hard by reduction from the 3SAT problem.

Given a Boolean formula $\phi = C_1 \wedge \cdots \wedge C_n$ over variables $x_1, \ldots, x_m$, the DTD D is defined as follows:

$$\begin{aligned} S &\rightarrow x_1, \ldots, x_m \\ X_i &\rightarrow T_i | F_i, \text{for } i \in [1,m] \\ T_j &\rightarrow C_{j_1}, \ldots, C_{j_k} \text{ /* all clauses } C_{j_i} \text{ in which } x_j \text{ appears */} \\ F_j &\rightarrow C_{j_1}, \ldots, C_{j_k} \text{/* all clauses } C_{j_i} \text{ in which } \bar{x}_j \text{ appears */} \end{aligned}$$

Furthermore, the query $XP(\phi) = /S[*/*/C_1]\cdots[*/*/C_n]$ is constructed. In fact, there exists a one-to-one correspondence between variable assignments that satisfies ϕ and XML trees in $SAT(D)$ [7]. This means ϕ is satisfiable iff $(XP(\phi), D)$ is satisfiable. As the DTD rules are duplicate-free, we can deduce that SAT(XP$^{\{/,[],*\}}$) under duplicate-free DTDs is NP-hard.

(2) The proof is the same as the proof of (1), except that $XP(\phi)$ is defined as $XP(\phi) = /S[.//C_1]\cdots[.//C_n]$.

(3) Using the same approach as (1), Benedikt *et al.* show in [7] that SAT(XP$^{\{/,[],\cup\}}$) is NP-hard. The DTD rules used in the proof are the following:

$$\begin{aligned} S &\rightarrow X \\ X &\rightarrow (X?), (T \mid F) \end{aligned}$$

and $XP(\phi) = /S[XP(C_1)]\cdots[XP(C_n)]$, where $XP(C_i)$ is defined as follows:

- For each variable x_i in ϕ, $XP(x_i) = X^i/T$ and $XP(\bar{x_i}) = X^i/F$, where X^i is the chain $X/\cdots/X$ of length i.
- For each clause C_j, $XP(C_j)$ is C_j in which each x_i is replaced by $XP(x_i)$ and each $\bar{x_i}$ is replaced by $XP(\bar{x_i})$.

As the DTD rules are duplicate-free, it can be deduced that SAT($XP^{\{/,[],\cup\}}$) under duplicate-free DTDs is NP-hard. ■

5.3.2 XPath Satisfiability under Covering DTDs

Recall that the majority of DTDs studied in Section 5.2 were classified as covering. In this section we prove that SAT($XP^{\{/,[],*,//,\cup\}}$) is in PTIME under covering DTDs. We should first point out the following fact which follows directly from a result in [91].

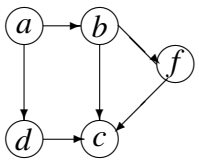
Proposition 5.2 *Given a DTD D, deciding whether D is covering is NP-complete.*

However, since we expect query processors to have to deal with relatively few, known DTDs while answering large numbers of XPath queries, the cost of detecting the covering property will be a one-off cost for each DTD.

The following theorem provides a positive result about D-satisfiability of queries in $XP^{\{/,[],*,//,\cup\}}$under covering DTDs. This result is important because it shows that the problem, which is in general NP-complete, becomes tractable when the DTD is covering for a significant XPath fragment. In addition, most real-world DTDs possess the covering property.

Theorem 5.3 *Under covering DTDs, SAT($XP^{\{/,[],*,//,\cup\}}$) is in PTIME.*

$$a \to (b|d)*$$
$$a \to f, c?$$
$$a \to c$$
$$a \to c?$$



G

Figure 5.1: A covering DTD and its digraph G.

We prove this using the same idea as Benedikt *et al.* used in [7], where they show that SAT($\text{XP}^{\{/,[],*,//,\cup\}}$) under disjunction-free DTDs is in PTIME. Clearly, a disjunction-free DTD is a special case of a covering DTD. Moreover, in [7], "$*$" is implicitly permitted in all the XPath fragments which include the child axis, whereas we distinguish explicitly whether or not $*$ is included.

Let p be a query in $\text{XP}^{\{/,[],*,//,\cup\}}$ and D be a covering DTD. We first construct a DTD digraph $G(V,E)$, with the set V of element names in D. G is rooted at $S_0 \in V$, where S_0 is the start symbol of D. For simplicity and without loss of generality, we assume that neither D nor p is empty. Let a and b be two distinct symbols in V. There is an edge in E from the vertex a to the vertex b if and only if b appears in the content model of a in D. As an example, see Figure 5.1. Note that because the DTD D is covering, the existence of such an edge in G means that an edge from an a-node to a b-node in p is also permissible irrespective of other children of the a-node in p. That is, even if p is D-satisfiable, any a-node in p can have a b-child no matter if the b-child has siblings and, if so, what labels they have. The converse also holds, that is, if a child-edge exists in a D-satisfiable query from an a-node to a b-node, then there exists an edge from the a-node to the b-node in G. Consequently, if a descendant-edge exists in a D-satisfiable query from an a-node to a b-node then there exists a path from the a-node to the b-node in G. We also use the following definitions:

Definition 5.5 *Let a and b be two nodes representing element names in G, then:*

$$hasPath(a,b) = \begin{cases} true & \text{if there is a path of length} \geq 1 \text{ from } a \text{ to } b \text{ in } G \\ false & otherwise \end{cases}$$

We consider the given XPath query p as a tree pattern query where the label of each node is either a symbol in Σ or "$*$". We define for each node u a type $u.type$ which is either $\vee$ or $\wedge$. The default node type is $\wedge$, and we use $\vee$ to model the union operator in queries in $XP^{\{/,[],*,//,\cup\}}$. More specifically, when a node u has several children $v_1,\ldots,v_n$, having $u.type = \vee$ means that it is sufficient to have any of the constraints imposed by the subtree $sub(v_i)$, $i = 1,\ldots,n$ satisfied, whereas the type $\wedge$ means that all these constraints must be satisfied. The following example, though simple, illustrates the difference between these types.

Example 5.3 *Consider two XPath queries $p = a[b//d \cup c//f]$ and $q = a[b//d][c//f]$. The tree pattern shown in Figure 5.2 represents p if $u.type = \vee$, while it represents q if $u.type = \wedge$, where u is the node labeled with a.*

Definition 5.6 *Let u and v be two nodes in p. We use $Child(u,v)$ to show there is a child-edge from u to v and $Desc(u,v)$ to show there is a descendant-edge from u to v.*

We now provide a recursive definition for the concept of $match(u)$, where u is a node in a query p rooted at r. This concept is adapted from the concept of reachability used in [7], so $match(r) = \emptyset$ if and only if p is unsatisfiable.

Definition 5.7 *We define $match(u)$ recursively as follows:*

- *If u is a leaf:*

$$match(u) = \begin{cases} \lambda(u) & if\ \ \lambda(u) \neq' *' \\ \Sigma & if\ \ \lambda(u) =' *' \end{cases} \tag{5.1}$$

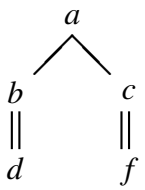


Figure 5.2: The tree pattern corresponding to the XPath queries in Example 5.3.

- *If u is not a leaf:*

$$match(u) = \begin{cases} \bigcup_{v_i \in C_u} match(u, v_i) & if \;\; u.type = \vee \\ \bigcap_{v_i \in C_u} match(u, v_i) & if \;\; u.type = \wedge \end{cases} \tag{5.2}$$

where $C_u = \{v_i | Child(u, v_i) \vee Desc(u, v_i)\}$ *and*

$$match(u, v_i) = \begin{cases} \{x | \lambda(u) = x \vee \lambda(u) =' *', \exists y \in match(v_i) s.t. (x, y) \in E\} \\ \qquad\qquad if \;\; Child(u, v_i) \\ \{x | \lambda(u) = x \vee \lambda(u) =' *', \exists y \in match(v_i) s.t. hasPath(x, y)\} \\ \qquad\qquad if \;\; Desc(u, v_i) \end{cases} \tag{5.3}$$

Based on the definition of $match(.)$, the query p is satisfiable if $match(r) \neq \emptyset$ where r is the root of p; it is unsatisfiable otherwise. Therefore, in the rest of the proof, it is enough to show that $match(r)$ can be determined in polynomial time. To that end, we present a polynomial-time method based on dynamic programming that determines $match(u)$ for each node u in p, moving from the leaves to the root. The proposed algorithm is shown in Figure 5.3.

The algorithm goes through each level of the tree from the leaves, at level L_{max}, to the root, where $L = 0$ (lines 2–33). At each level L, it goes through every node u_j and calculates $match(u_j)$ using dynamic programming (lines 3–32). To that end, if u_j is a leaf-node, it is straightforward to calculate $match(u_j)$ (lines 5–11). If u_j is not a leaf, the algorithm first calculates, for each child v_i of u_j, $match(u_j, v_i)$ (lines 13–25) and then determines $match(u_j)$ (lines 26–30), following Definition 5.2. Having calculated $match(u)$

Algorithm *CalculateMatch*
Input: Query p, Graph $G(V,E)$
Output: $r.match$, where r is the root of p

1: Let L_{max} be the depth of the tree pattern p
2: **for** $l = L_{max}$ **to** 0 **do**
3: **for all** $u_j, j = 1,\ldots,m$ in level l **do**
4: **if** u_j is a leaf-node **then**
5: **if** $\lambda(u_j) \in V$ **then**
6: $u_j.match \leftarrow \{\lambda(u_j)\}$
7: **else if** $\lambda(u_j) = *$ **then**
8: $u_j.match \leftarrow \Sigma$
9: **else**
10: $u_j.match \leftarrow \emptyset$
11: **end if**
12: **else**
13: **for all** v_i such that $Child(u_j,v_i) \vee Desc(u_j,v_i)$ **do**
14: Let E_{ji} be the direct edge from u_j to v_i
15: **if** $\lambda(u_j) \in \Sigma$ **then**
16: Let a be the label of u_j
17: **if** $\exists b \in v_i.match$ such that $((a,b) \in E \wedge Child(u_j,v_i)) \vee (hasPath(a,b) \wedge Desc(u_j,v_i))$ **then**
18: $E_{ji}.match \leftarrow \{a\}$
19: **else**
20: $E_{ji}.match \leftarrow \emptyset$
21: **end if**
22: **else if** $\lambda(u_j) = *$ **then**
23: $E_{ji}.match \leftarrow \{a \in \Sigma \mid \exists b \in v_i.match; ((a,b) \in E \wedge Child(u_j,v_i)) \vee (hasPath(a,b) \wedge Desc(u_j,v_i))\}$
24: **end if**
25: **end for**
26: **if** $u_j.type = \vee$ **then**
27: $u_j.match \leftarrow \bigcup_{v_i \in C_{u_j}} E_{ji}.match$
28: **else if** $u_j.type = \wedge$ **then**
29: $u_j.match \leftarrow \bigcap_{v_i \in C_{u_j}} E_{ji}.match$
30: **end if**
31: **end if**
32: **end for**
33: **end for**
34: **return** $r.match$

Figure 5.3: The pseudo-code of algorithm *CalculateMatch* which calculates $r.match$, given a query p rooted at r.

for every node u in the tree, the algorithm finally returns $match(r)$ (line 34).

We now present two lemmas, the first to prove that the algorithm, *CalculateMatch*, is correct, i.e. it returns $match(r)$ for the root r of the given query p, and the second to show that it runs in polynomial time.

Lemma 5.3 *Given a tree pattern query p rooted at r, algorithm* C*alculateMatch returns* $match(r)$.

Proof. We prove that the algorithm calculates, for each node u_j in p, $u_j.match$ as the set $match(u_j)$. Then the lemma will immediately follow for $u_j = r$. We prove this by

induction on the level L of u_j, from L_{max} to 0.

Base Case: When the level of u_j is L_{max}, u_j is a leaf. In this case, the label of u_j is either a specific label, say a, or the wildcard "$*$". In these cases, the algorithm sets $u_j.match$ to $\{a\}$ and Σ, respectively, at lines 6 and 8, which is trivially consistent with the definition of $match(u_j)$.

Induction hypothesis: We assume that the algorithm sets $u_j.match$ to $match(u_j)$, for each node u_j in some level $L \geq 1$.

Induction step: We prove that the algorithm sets $u_j.match$ to $match(u_j)$, for each node u_j in level $L-1$. If the node u_j is a leaf node, the proof is similar to that in the base case. Otherwise, the set C_{u_j} is nonempty and, by the induction hypothesis, the algorithm has already set $v_i.match$ to $match(v_i)$, for each node v_i in C_{u_j}. Also in this case, the algorithm determines $E_{ji}.match$, for each such a node v_i in C_{u_j} in lines 13 to 25. Assume that this part of the algorithm sets $E_{ji}.match$ to $match(u_j, v_i)$. Then, the algorithm, in lines 26 to 30, simply sets $u_j.match$ to $match(u_j)$ based on Definition 5.2. Therefore, we only need to show that lines 13 to 25 set $E_{ji}.match$ to $match(u_j, v_i)$, for each node in C_{u_j}.

To that end, we consider all possible cases with respect to possible labels of u_j and v_i. Note that $match(u_j, v_i)$ depends on $\lambda(u_j)$, which is either a specific label, say a, or the wildcard. First assume that $\lambda(u_j) = a$. In this case, by definition, $match(u_j, v_i) = a$ if, and only if, there is some label, say b, in $match(v_i)$ such that either there is an edge in G from a to b and the edge in p from u_j to v_i is a child edge or there is a path in G from a to b and the edge in p from u_j to v_i is a descendant edge; otherwise, $match(u_j, v_i)$ will be empty. This compound condition is exactly what is captured in line 17, based on the definitions of the $Child(.,.)$, $hasPath(.,.)$, and $Desc(.,.)$ predicates. The next case is when the label of u_j is "$*$". This case is a generalised form of the previous case, where the label of u_j can be any label and not just a specific one. Therefore, $match(u_j, v_i)$ in this case may be considered as the union of that computed in the previous case, where now the label of u_j iterates over all alphabet symbols in Σ as opposed to a specific one; this is exactly what is

computed in line 23. Finally, the algorithm returns *r.match*, which completes the proof. ■

Lemma 5.4 *Algorithm* C*alculateMatch runs in polynomial time.*

Proof. Let n be the number of nodes in p and Σ be the alphabet. First, note that the function $hasPath(.,.)$ can be computed in $O(n^3)$, e.g. using Warshall's algorithm [76]. It can be precomputed and represented as a two-dimensional array so each subsequent instance $hasPath(a,b)$, $\forall a \in \Sigma$, $\forall b \in \Sigma$ is decided in $O(1)$. In addition, the adjacency matrix of the graph G can be precomuted in $O(n^2)$, so the precomuptation of both $hasPath(.,.)$ and the adjacency matrix has a one-off time cost of $O(n^3)$, which we will add to the complexity of the code residing in lines 1 to 34.

As can be seen, except for line 1 and line 34, which run in $O(1)$ time, the code resides within two nested for-loops starting at lines 2 and 3. These two for-loops iterate n times in total. So, we first determine the time complexity of the code from line 4 to line 31 and then multiply it by n.

The code fragment from line 4 to line 31 is an "if-else" statement, with its "if" branch being the code within lines 4–12 and the else branch within lines 12–31. The complexity of the "if" branch, which is in turn an "if-else" statement, is $O(\Sigma)$ (for its "else" branch). As we shall see this complexity is less than that of the code within lines 12–31, and therefore can be ignored. This latter code consists of a "for" loop (lines 13–25) followed by an "if-else" statement (lines 26–30), which we analyse in turn. The "for" loop at line 13 iterates once per internal node of p, which is $O(n)$ altogether. Line 14 runs in $O(1)$ and the rest of the code inside the "for" loop is a series of nested "if-else" statements. So, to analyse the complexity of the "for" loop (lines 13–25), we determine the complexity of each of the branches within the nested "if-else" statement and multiply the worst of them by n (for an upper bound on the number of iterations).

The condition of the "if" statement at line 17, which uses the $hasPath(.,.)$ function at most $|v_i.match|$ times, is determined in $O(|\Sigma|)$, because $|v_i.match| < |\Sigma|$ and each evalua-

tion of whether an edge is in E can be performed in $O(1)$ using the adjacency matrix of G. The statements at lines 18 and 20 run in $O(1)$ time and hence do not add to the complexity of the "if" statement. Therefore, the complexity of the code within lines 17–21 is $O(|\Sigma|)$. However, this is dominated by the complexity of the branch specified at line 23, because the latter corresponds to the same amount of computation for each of the alphabet symbols and, therefore, is more than the former by a factor of $|\Sigma|$. That is, line 23 runs in $O(|\Sigma|^2)$.

So far, we have obtained the complexity of the code within the lines 14–24 as $O(|\Sigma|^2)$. On the other hand, the code from line 26 to line 30 determines the union or the intersection of $|C_{u_j}|$ sets, each of which is of size $O(|\Sigma|)$, which can be performed in $O(|C_{u_j}| \cdot \Sigma^2) = O(n \cdot \Sigma^2)$. Therefore, the complexity of the code from line 13 to line 30 will be $O(n \cdot \Sigma^2 + n \cdot \Sigma^2) = O(n \cdot \Sigma^2)$. This implies that the complexity of Lines 1–34 is $O(n^2 \cdot \Sigma^2)$. By adding the one-off time cost of $O(n^3)$ for precomputing $hasPath(.,.)$ and the adjacency matrix of G, the total complexity of the algorithm CalculateMatch is $O(n^2 \cdot |\Sigma|^2 + n^3)$, which is polynomial in n and $|\Sigma|$. ■

Note that it is possible to obtain a tighter bound on the time complexity of the algorithm. However, the obtained bound is sufficient for the purpose of Lemma 5.4.

Corollary 5.2 *Given a query p in $XP^{\{/,[],*,//,\cup\}}$ and a DTD D, SAT($XP^{\{/,[],*,//,\cup\}}$) is in PTIME if each rule in D in which a symbol from p appears is covering.*

Corollary 5.3 *Given a query p in $XP^{\{/,[]\}}$ and a DTD D, SAT($XP^{\{/,[]\}}$) is in PTIME if each rule in D is either covering or duplicate-free.*

We know so far that SAT($XP^{\{/,[],*,//,\cup\}}$) is PTIME under covering DTDs but NP-complete under general non-covering DTDs. We now show that our PTIME algorithm for SAT($XP^{\{/,[],*,//,\cup\}}$) under covering DTDs remains complete, though unsound, for non-covering DTDs. That is, given a query p in $XP^{\{/,[],*,//,\cup\}}$ and a non-covering DTD D,

our PTIME algorithm returns true if p is satisfiable under D. Note that the algorithm is unsound, i.e. it may still return true even if p is not satisfiable under D.

Notation 5.1 *Let R be a non-covering regular expression. We use R_{cov} to denote the covering regular expression $(R|w)$, where w is a string of all the symbols in R. Let D be a non-covering DTD, i.e. a DTD which includes at least one non-covering DTD rule, which is a rule whose regular expression is non-covering. We use D_{cov} to denote the DTD obtained from D by replacing each non-covering regular expression R in D with R_{cov}.*

Notation 5.2 *Let t be a document tree. We use $V(t)$ to denote the set of the nodes in t. For each v in $V(t)$, we use C_v to denote the bag $\{b \mid u$ is a child of v, $\lambda(u) = b\}$.*

Proposition 5.3 *Let D be a DTD, R^a be the regular expression corresponding to the content model for the label a, and t be a document tree. Then:*

$$t \in SAT(D) \text{ iff } (\forall v \in V(t), \exists w \in L(R^{\lambda(v)}) \text{ such that } C_v \subseteq [w]).$$

This result follows directly from the definition of a tree satisfying a DTD.

Notation 5.3 *Let q be an XPath query and D be a DTD. By $unSAT(q,D)$ we mean the problem of deciding whether q is not satisfiable under D, that is, for every document tree t whether $t \not\models (q,D)$.*

Theorem 5.4 *Let q be an XPath query in $XP^{\{/,[],*,//,\cup\}}$ and D be a non-covering DTD. Then,*

$$unSAT(q,D_{cov}) \Rightarrow unSAT(q,D).$$

Proof.

$$\begin{aligned} unSAT(q,D_{cov}) &\Rightarrow \forall t, t \not\models q \vee t \not\models D_{cov} \\ &\Rightarrow \forall t, t \not\models q \vee (\exists v \in V(t), \nexists w \in L(R_{cov}^{\lambda(v)}), C_v \subseteq [w])) \end{aligned}$$

$$\Rightarrow \quad \forall t, t \not\models q \vee (\exists v \in V(t), \nexists w \in L(R^{\lambda(v)}), C_v \subseteq [w]))$$

$$(because\ L(R^{\lambda(v)}) \subseteq L(R^{\lambda(v)}_{cov})$$

$$\Rightarrow \quad \forall t, t \not\models q \vee t \not\models D$$

$$\Rightarrow \quad unSAT(q, D)$$

■

Based on the above theorem, given a query Q in $XP^{\{/,[],*,//,\cup\}}$ and a non-covering DTD D, the following PTIME algorithm may be used to return $true$ if $unSAT(Q,D)$ is $true$:

ALGORITHM *unSatAlg*

Input: an XPath query Q, a DTD D

Output: Boolean value stating if Q is D-unsatisfiable

Step 1. Form D_{cov} from D by replacing each content model R^a in D
with $R^a|w^a$, where w^a is a string of all symbols in R^a.

Step 2. Using a PTIME algorithm to decide $unSAT(Q, D_{cov})$,
return $true$ if $unSAT(Q, D_{cov})$ is so.

End *unSatAlg*

Obviously, both Step1 and Step 2 can be performed in PTIME. Algorithm *UnSatAlg* is sound (but not complete) for non-covering DTDs. More specifically, when the algorithm returns true, then the query is indeed unsatisfiable, but if it returns false then the query may or may not be satisfiable.

5.4 Conclusion

This chapter was concerned with discovering properties of real-world DTDs and their impact on the satisfiability problem for XPath. The motivation behind this was our belief

that although common XPath problems are of high complexity, e.g. NP-hard, in general, real-world applications usually provide simpler structures under which such otherwise hard problems could be performed in PTIME.

In particular, we examined several real-world DTDs and discovered a new property, called covering, which most of them satisfied. We observed that even the minority of the examined real DTDs which did not possess the covering property were duplicate-free. We showed that the satisfiability problem for the XPath fragment $XP^{\{/,[],*,//,\cup\}}$ can be solved in PTIME when the underlying DTD has the covering property. We also showed that the satisfiability problem for the fragment $XP^{\{/,[]\}}$ can also be solved in PTIME when the underlying DTD is duplicate-free. These problems were previously shown to be NP-hard under general DTDs.

Chapter 6

XPath Containment under DTDs

In this chapter we study the problem of containment for various fragments of XPath using only the most common axes, / and //, wildcard($*$), union ($\cup$), and filter($[\,]$). In all cases, we are interested in the containment problem for XPath queries considered together with a DTD D (which we refer to as D-containment): given two XPath queries q and p and a DTD D, whether or not, on every document tree t which satisfies D, every answer of q on t is also an answer of p on t. The problem is in general coNP-hard, and extensive research has been conducted to determine special cases where it becomes tractable.

In this chapter, we first introduce a new DTD property, called *well-behaved*. Then, we prove that BSCs and BFCs, defined in Chapter 4, are necessary and sufficient to capture D-containment of queries in $\mathrm{XP}^{\{/,[]\}}$under *well-behaved DTDs*. Finally, we show that, given the set BSCs and BFCs, D-containment of queries in $\mathrm{XP}^{\{/,[]\}}$ under a special case of well-behaved DTDs, called *well-formed* DTDs, is tractable. An investigation of real-world DTDs shows that well-formed DTD rules arise frequently in practice.

This chapter shows how the chase method can be applied to decide D-containment when both DTDs and queries have duplicates. In particular, this chapter introduces a new concept of MinChildBag. Although the idea of using MinChildBag and chase procedure to decide D-contanment is not limited to the queries in $\mathrm{XP}^{\{/,[]\}}$, we used this fragment

because it is sufficient to demonstrate the approach and the concept of MinChildBag and also allows for simpler proofs.

6.1 The well-behaved property

Recall the definitions relating to bags from Chapter 4. In particular, recall that if w is a string of symbols over alphabet Σ, then we use the notations $[w]$ and $\{w\}$ to denote, respectively, the bag and the set of symbols appearing in w.

The existence of duplicate elements in queries, particularly duplicate siblings, complicates the D-containment problem. The main difficulty is related to the question of whether or not two or more such siblings in a query could map to the same node in a document tree. For this reason, in some research the D-containment problem is studied under duplicate-free DTDs [92]. A duplicate-free DTD is one in which no regular expression uses the same symbol more than once. When the DTD is duplicate-free, either all the duplicate siblings must always map to the same document node (detected by a functional constraint (FC) implied by the underlying DTD) or there can be an arbitrary number of such document nodes. In the first case, all the duplicate siblings in a query should be merged, while in the latter case no action is needed. Although the duplicate-free property of DTDs simplifies the problem, it is rather restrictive. That is, each duplicate-free rule is syntactically limited to use each symbol only once. A less restrictive idea is to limit the rules *semantically*. In other words, it is only important whether each label of a set of siblings can occur either only once or infinitely often. Based on this idea, we define the well-behaved property below and introduce two types of DTD constraints which are necessary and sufficient to decide D-containment under such DTDs.

Definition 6.1 *A DTD is* well-behaved *if for each symbol $a \in \Sigma$ and each $b \in \Sigma^a$, whenever there exists a string $u \in L(R^a)$ such that $|b|_{[u]} = k$, $k > 1$, then $\forall n > k$, $\exists v \in L(R^a)$ such*

that

$$
\begin{aligned}
&i) \quad |b|_{[v]} \geq n \\
&ii) \quad \{u\} = \{v\} \\
&iii) \quad \forall c \in [u], |c|_{[u]} \leq |c|_{[v]}
\end{aligned}
$$

Informally speaking, the above definition says that if a string u in $L(R)$ has more than one instance of a symbol b, then we can find another string, still in $L(R)$, by increasing the number of bs to any value (with possibly increasing the number of other symbols in u and irrespective of the order). Intuitively, this means that if in a query there are some duplicate b-siblings and there is no FC to merge *all* of them, then we cannot merge any number of them either. That is, if two distinct b-siblings are allowed, then any number of them are allowed too.

Example 6.1 *The following DTD rule is well-behaved:*

$$a \rightarrow (b,e,f?) \mid (f*,(e*\mid b*),d) \mid (d,b*)$$

However, the following DTD rule is not:

$$a \rightarrow (b,e,e) \mid (e,b,b)$$

6.2 Determining Minimum Numbers of Query Nodes

When an XPath query q containing no duplicate occurrences of element names is matched against a document tree t, every node in q is mapped to a distinct node in t. This simplifies the analysis of D-containment, as opposed to the case when a node in a query has children labeled with the same name; these children may or may not always be mapped to distinct nodes in a document tree $t \in SAT(D)$, depending on D.

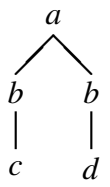


Figure 6.1: An XPath query expressed as a tree pattern.

Example 6.2 *Consider the query tree q shown in Figure 6.1. If the rule for b in DTD D is*

$$b \rightarrow (c \,|\, d)$$

then the b-nodes in q must always be mapped to distinct nodes in any tree $t \in SAT(D)$. On the other hand, if the rule for b in D is

$$b \rightarrow (c?, d?)$$

then, when evaluating q on a tree $t \in SAT(D)$, the b-nodes in q may sometimes be mapped to the same node in t and sometimes to different ones.

It is essential to know whether or not such nodes in a query q must be mapped to distinct nodes in a tree t because it can affect whether other nodes must exist in t and hence can be added to q (by applying DTD constraints). For example, consider the query tree q shown in Figure 6.1. If the rule for a in DTD D above is

$$a \rightarrow (b, e) \,|\, (f, b*)$$

then in the case of the first rule for b, since we know that the b-nodes in q will always be mapped to distinct tree nodes, we also know that the b-nodes must have an f-node as a sibling. Hence an f-node can be added to q as a child of a. However, in the case of the second rule for b, we could not add an f-child to the a-node because the b-nodes in q may sometimes map to the same b-node in a tree in $SAT(D)$. This means when we want to

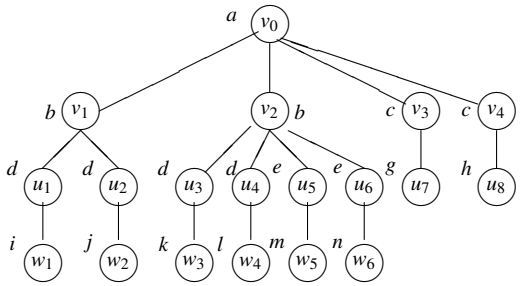


Figure 6.2: The query tree used in Example 6.3 and Example 6.4.

apply a BSC $a : B \Downarrow f$ to an a-node in a query q, we must make sure that the minimum bag of labels of children of the a-node includes B. The minimum bag of labels of children of a node v is called the *MinChildBag* of v, which is formally defined in this section.

Definition 6.2 *Let r be a node in a query q and u and v be two nodes in $sub(r)$. We say u and v are* Potential Siblings (PS) *in t, written $PS(u,v,r)$, if (1) $u = v$, (2) u and v are siblings, or (3) $(\lambda(parent(u)) = \lambda(parent(v))$ and $PS(parent(u), parent(v), r)$ and $u \neq r$ and $v \neq r$.*

The $PS(.,.,r)$ relation is reflexive ($PS(v,v,r)$, $\forall v$ in t), symmetric ($PS(v,u,r) \rightarrow PS(u,v,r)$, $\forall v,u$ in t), and transitive ($PS(v,u,r) \wedge PS(u,w,r) \rightarrow PS(v,w,r)$, $\forall v,u,w$ in t). Therefore, it is an equivalence relation. We also use $PS(*,v,r)$ to indicate the set of potential siblings of v in $sub(r)$, i.e. $PS(*,v,r) = \{u \mid PS(u,v,r)\}$.

Definition 6.3 *Let r be a node in a query q and v be a node in $sub(r)$ with set of children $children(v)$. Then* Potential Children *$PC(v,r)$ is the set of nodes defined as follows:*

$$PC(v,r) = \begin{cases} \emptyset & \text{if } v \text{ is a leaf} \\ PS(*,u,r), \text{ for some } u \in children(v) & \text{otherwise} \end{cases}$$

We also define $LPC(v,r) = \{\lambda(u) \mid u \in PC(v,r)\}$ as the set of labels of the nodes in $PC(v,r)$.

Example 6.3 *Consider the query tree shown in Figure 6.2 and the subtree $sub(v_0)$. The following results hold:*

$PC(v_0, v_0) = \{v_1, v_2, v_3, v_4\}$

$PC(v_1, v_0) = PC(v_2, v_0) = \{u_1, u_2, u_3, u_4, u_5, u_6\}$

$PC(v_3, v_0) = PC(v_4, v_0) = \{u_7, u_8\}$

$PC(u_1, v_0) = PC(u_2, v_0) = PC(u_3, v_0) = PC(u_4, v_0) = \{w_1, w_2, w_3, w_4\}$

$PC(u_5, v_0) = PC(u_6, v_0) = \{w_5, w_6\}$

Now consider the subtree $sub(v_2)$. *We have:*

$PC(v_2, v_2) = \{u_3, u_4, u_5, u_6\}$

$PC(u_3, v_2) = \{w_3, w_4\}$

$PC(u_4, v_2) = \{w_3, w_4\}$

$PC(u_5, v_2) = \{w_5, w_6\}$

$PC(u_6, v_2) = \{w_5, w_6\}$

Definition 6.4 *Given a tree t and a node w in t, by* $sub(w)$, *we mean the subtree of t rooted at w. Given a DTD D, a document tree* $t \in SAT(D)$, *and a node w in t, by* $SAT(w)$ *we mean the set of all subtrees s such that the root of s is labeled* $\lambda(w)$ *and the tree obtained by replacing* $sub(w)$ *in t with s is still in* $SAT(D)$.

Definition 6.5 *Let r be a node in a query q and v be a node in* $sub(r)$. *Also let* $\{u_1, \ldots, u_n\} \subseteq PC(v, r)$ *such that* $\lambda(u_i) = \alpha$ *where* $1 \leq i \leq n$ *and* $\alpha \in \Sigma$. *Then* $merge(\alpha, v, r)$ *is defined as a tree whose root* v' *is labeled with* $\lambda(v)$ *and has n children* $u'_1, \ldots, u'_n$ *such that* $sub(u'_i)$ *in* $merge(\alpha, v, r)$ *is identical to* $sub(u_i)$ *in q,* $i = 1, \ldots, n$.

Example 6.4 *Figure 6.3(a) and (b) depict* $merge(d, v_1, v_0)$ *and* $merge(e, v_1, v_0)$, *respectively, which are the result of merging the b-nodes in the query shown in Figure 6.2.*

Definition 6.6 *Let D be a well-behaved DTD, and q be a D-satisfiable query in* $XP^{\{/,[]\}}$. *Then the MinChildBag of v in* $sub(r)$ *in q is denoted by* $MCB(v, r)$ *and is defined as follows:*

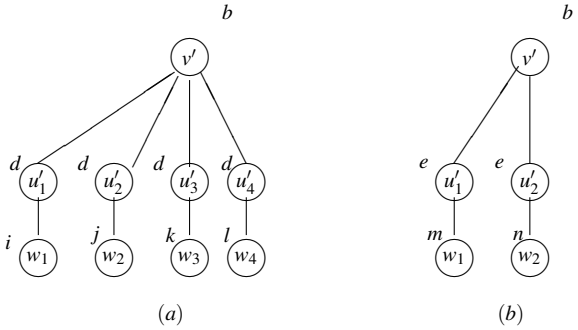


Figure 6.3: The merged trees used in Example 6.4.

1. *if $v = r$ then $MCB(v,r) = \{c_1^{k_1}, \ldots, c_n^{k_n}\}$, where each $c_i \in LPC(v,r)$, $i = 1, \ldots, n$, and k_i is* 1 *if there is a tree t in $SAT(D)$ on which q has a nonempty answer such that all the c_i-children of v can be mapped to the same node in t; k_i is > 1 otherwise.*

2. *if $v \neq r$ then $MCB(v,r) = \bigcup_{\alpha \in LPC(v,r)} MCB(v', v')$ where v' is the root of $merge(\alpha, v, r)$.*

First note that $MCB(v,r) = \emptyset$ if v does not have any children. Also note that, for a label c_i, $i = 1, \ldots, n$, if all the c_i-children of v are leaves then $k_i = 1$. Finally, by the MinChildBag of v, we simply mean $MCB(v,v)$.

Example 6.5 *Consider the query tree q in Figure 6.2 and the following DTD D:*

$$
\begin{aligned}
a &\rightarrow (b, c*) \,|\, (c, b*) \\
b &\rightarrow (d, e) \,|\, (e, d*, f) \\
c &\rightarrow (g \,|\, h) \\
d &\rightarrow (i \,|\, (j?, k?) \,|\, l) \\
e &\rightarrow (m?, n?)
\end{aligned}
$$

We will have $MCB(v_0, v_0) = \{b, c^{>1}\}$ because no document tree t in $SAT(D)$ exists on which q has a nonempty answer such that both c-nodes map to the same node in t (because of the D-rule for c and the children of the c-nodes in q). Moreover, $MCB(v_1, v_0) = merge(d, v_1, v_0) \cup merge(e, v_1, v_0)$ for the subtrees shown in Figure 6.3(a) and (b). Since

Algorithm *calculateMCB*
Input: v and r; v is a node in $sub(r)$
Output: $MCB(v,r)$

if $v = r$ **then**
  Calculate $PC(u,r)$ and $LPC(u,r)$ for each node u in $sub(r)$
  {To be used also in all the subsequent recursive calls}
end if
$B = \emptyset$
for all $b \in LPC(v,r)$ **do**
  if all b-nodes in $PC(v,r)$ are leaves **then**
    $B = B \cup \{b\}$
  else
    Let u be an arbitrary b-node in $PC(v,r)$
    $tmpMCB = calculateMCB(u,r)$
    if $\delta_{tmpMCB} R^b \neq \emptyset$ **then**
      $B = B \uplus \{b\}$
    else
      $B = B \uplus \{b^{>1}\}$
    end if
  end if
end for
return B

Figure 6.4: The pseudo-code of Algorithm *calculateMCB* which calculates $MCB(v,r)$, where v is a node in $sub(r)$.

the d-nodes can not be mapped to the same node in a document tree in $SAT(D) \cap SAT(q)$*,therefore,* $MCB(v_1, v_0) = \{d^{>1}, e\}$.

We now present Algorithm *calculateMCB* which, given v and r, calculates $MCB(v,r)$ where v is a node in $sub(r)$. The algorithm is given in Figure 6.4. In the special case where v is a leaf, the for-loop is not entered and the algorithm returns the empty bag B, which is consistent with the definition of $MCB(v,r)$. Otherwise, the bag B will be constructed from the labels in $LPC(v,r)$; the only question is, for each such label b, whether the multiplicity is 1 (i.e., $b \in B$) or >1 (i.e., $b^{>1} \in B$). Of course, by the definition of $MCB(v,r)$, the multiplicity is 1 if all the b-nodes in $PC(v,r)$ are leaves, which is captured by the outer if-statement. However, if at least one of these nodes is not a leaf, then the $else$-branch of the outer if-statement is entered, which is the recursive part of the algorithm.

In this part of the algorithm, first $MCB(u,r)$, where u is a b-node (for some label b) in $PC(v,r)$, is calculated and called $tmpMCB$. Then, the concept of derivatives introduced in Chapter 4 is used to determine whether one or more occurrences of b is required in B. More specifically, by the definition of derivatives, a b-node in a document tree in $SAT(D)$

can have children specified in *tmpMCB* if, and only if, the derivative of the regular expression R^b of the DTD rule for b with respect to *tmpMCB* is nonempty. This is exactly what is examined by the inner if-statement. If the derivative is nonempty, one b, or otherwise more than one (> 1) b, will be added to B. However, the problem here seems to be that *tmpMCB* is an infinity, as opposed to an ordinary, bag. The key point here lies in the well-behavedness of the underlying DTD D, where a b-node in a document tree in $SAT(D)$ can have children specified in an infinity bag *tmpMCB* if, and only if, it can have children specified in the corresponding ordinary bag, say *tmpMCB-ordinary*, where *tmpMCB-ordinary* is the bag obtained from *tmpMCB* by replacing every multiplicity which is > 1 in *tmpMCB* with multiplicity 2 in *tmpMCB-ordinary*.

Example 6.6 *Consider the query tree q in Figure 6.2 and the DTD in Example 6.5. Assume that the algorithm is called to compute* $MCB(v_0, v_0)$. *Because* v_0 *is not a leaf, the* for*-loop is entered. The sets* $PC(v_0, v_0)$ *and* $LPC(v_0, v_0)$ *are* $\{v_1, v_2, v_3, v_4\}$ *and* $\{b, c\}$, *respectively. Let* v_1 *be the arbitrary node chosen, so* $calculateMCB(v_1, v_0)$ *is called. The new parameters are* $LPC(v_1, v_0) = \{d, e\}$ *and* $PC(v_1, v_0) = \{u_1, u_2, u_3, u_4, u_5, u_6\}$. *Let d be the first label considered by the* for*-loop. Now, let* u_1 *be the arbitrary d-node chosen. We have* $LPC(u_1, v_0) = \{i, j, k, l\}$ *and* $PC(u_1, v_0) = \{w_1, w_2, w_3, w_4\}$. *Then the algorithm returns* $\{i, j, k, l\}$ *because all children in* $PC(u_1, v_0)$ *are leaves. Therefore,* $\{d^{>1}\}$ *since* $\delta_{\{i,j,k,l\}} R^d = \emptyset$. *Similarly, when considering label e next in the* for*-loop, let* u_5 *be the arbitrary e-node chosen. The algorithm returns* $B = \{m, n\}$. *Because* $\delta_{\{m,n\}} R^e \neq \emptyset$, *e is added to B to give* $\{d^{>1}, e\}$. *Now returning to the first invocation and considering label c in the* for*-loop, we find out* $MCB(v_3, v_0) = \{g, h\}$ *(because the children of the c-nodes are leaves). Finally, we discover that* $MCB(v_0, v_0) = \{b, c^{>1}\}$ *because* $\delta_{\{d^{>1}, e\}} R^b \neq \emptyset$ *and* $\delta_{\{g,h\}} R^c = \emptyset$.

Theorem 6.1 *Let D be a well-behaved DTD, q be a D-satisfiable query in* $XP^{\{/,[]\}}$, *r be a node in q, and v be a node in* $sub(r)$. *Then the bag B returned by the algorithm calculateMCB on the inputs v and r is* $MCB(v, r)$.

Proof. Let L be the level at which the node v is located in $sub(r)$, having defined the level of r as 0. Let L_{max} be the depth of the subtree, i.e. the level of the deepest node in $sub(r)$. We use induction on L, starting from $L = L_{max}$ as the base case and proving that if the theorem holds for the nodes at a level $L > 0$, it will also hold for the nodes at the level $L-1$.

Base case: In this case $L = L_{max}$ and v and all nodes u such that $PS(v,u,r)$ are leaves. Therefore, $merge(-,v,r)$ will be a singleton tree, which implies that $MCB(v,r) = \emptyset$. As can be seen, in this case the algorithm returns $B = \emptyset$ as well.

Inductive hypothesis: We assume that the algorithm properly returns $MCB(v,r)$ where v is at a level $0 < L \leq L_{max}$.

Induction step: We prove that the algorithm properly returns $MCB(v,r)$ where v is located at the level $L-1$. To that end, we need to show that the returned bag B is equal to $MCB(v',v')$, where v' is the root of $merge(-,v,r)$. By Definition 6.6, if v' does not have any children, i.e. $PC(v,r) = LPC(v,r) = \emptyset$, then $MCB(v',v')$ will be $\emptyset$. In this case, the algorithm does not execute the for-loop and $B = \emptyset$ is returned.

If $PC(v,r) \neq \emptyset$, then $MCB(v',v')$ is a bag composed of the labels of the children of v', each with either 1 or > 1 as its multiplicity. Similarly, the bag returned by the algorithm is composed of the labels in $LPC(v,r)$, which are the same as the labels of the children v', by the definition of $merge(-,v,r)$. Therefore, we only need to show that, for each label b in $LPC(v,r)$, the multiplicity of b in B is the same as that in $MCB(v',v')$. However, because the multiplicity is either 1 or > 1, we need to show that there is one b in B if and only if there is one b in $MCB(v',v')$.

First assume that there is one b in $MCB(v',v')$. Then all the b-children of v' can be mapped to the same b-node in a tree $t' \in SAT(merge(b,v,r)) \cap SAT(D)$. By the inductive hypothesis, we conclude that the bag of labels of the children of the single b-node in t' must at least include $MCB(u,r)$, where u is a b-child of the root v' (note that $MCB(u_i,r) = MCB(u_j,r)$ for all pairs u_i and u_j of b-children of v'). However, by the definition of

derivatives, this implies $\delta_{MCB(u,r)}R^b \neq \emptyset$. In this case the condition of the if-statement in the algorithm evaluates to true and the multiplicity of b in B will also be 1.

Conversely, assume that the multiplicity of b in B is 1. Then, the condition of the if-statement in the algorithm must have evaluated to true, i.e., $\delta_{MCB(u,r)}R^b \neq \emptyset$, where u is an arbitrary b-node in $PC(v,r)$. By the definition of derivatives, this implies the existence of a tree in $SAT(D)$ in which a b-node exists whose bag of labels of children includes the bag $MCB(u,r)$. By the induction hypothesis, we conclude that a tree $t' \in SAT(merge(b,v,r)) \cap SAT(D)$ must exist such that all the b-children of v' are mapped to the same b-node in t'. That is, the multiplicity of b in $MCB(v,r)$ is also 1. ■

6.3 Chasing the queries

Given two XPath queries p and q in $\mathrm{XP}^{\{/,[]\}}$, p contains q iff there is a containment mapping from p to q [65] (for the definition of containment mapping see Chapter 2). However, before we can check the existence of a containment mapping from p to q, we need to apply the constraints inferred from the underlying DTD to q. This can be done by the well-known chase procedure adapted from [92]. Let $\mathbb{C}$ be a set of BSCs and BFCs, the chase $q_{\mathbb{C}}$ of q by $\mathbb{C}$ is obtained by repeatedly applying the constraints in $\mathbb{C}$ to each node v labeled, say a, in q, as shown in the following steps:

1. Let $f \in \mathbb{C}$ be a BFC of the form $a : B \downarrow b$. If $B \subseteq MCB(v,v)$ and v also has distinct children u_1 and u_2 both labeled with b, then the BFC f is applicable to q. The result of applying f to v is a query identical to q but with the nodes u_1 and u_2 being merged.

2. Let $s \in \mathbb{C}$ be a BSC of the form $a : B \Downarrow c$. If $B \subseteq MCB(v,v)$ and v does not have a child labeled c, then the BSC s is applicable to q. The result of applying s to v is a query identical to q but with v having an additional child labeled c.

The result of chasing q by $\mathbb{C}$ is a sequence $q_0,\ldots,q_k$ such that $q_0 = q$, q_{i+1} is the result of applying some constraints in $\mathbb{C}$ to q_i, and no constraint can be applied to q_k. Because each q_{i+1}, $i = 0,\ldots,k-1$, is obtained from applying a DTD constraint in $\mathbb{C}$ to its predecessor q_i, the final query $q_k = q_{\mathbb{C}}$ will be D-equivalent to $q_0 = q$ (also see Proposition 6.2 in Section 6.4). The following lemma shows that chasing sequences are finite.

Lemma 6.1 *Let $\mathbb{C}$ be the set of BSCs and BFCs implied by DTD D, and q be an XPath query in $XP^{\{/,[]\}}$. Every chasing sequence of q by $\mathbb{C}$ is finite.*

Proof. A BFC can only be applied to a pair of original nodes (or nodes resulting from the merging of two original nodes) in q since applying a BSC does not introduce sibling nodes with duplicate labels. In addition, applying a BFC reduces the number of nodes in q by one. Hence, at some point in the chasing sequence, no BFC in $\mathbb{C}$ will be applicable to q. Each BSC in $\mathbb{C}$ can be applied at most once to each original node in q. Only child constraints can be applied to newly added nodes since when they are first introduced such nodes have no children. Since DTD D is satisfiable, there can be no cycle of child constraints. Hence, each child constraint can be applied only a finite number of times. ■

6.4 Using BSCs and BFCs for *D*-containment

In this section, we prove that BFCs and BSCs are necessary and sufficient constraints to capture D-containment for XPath queries in $\text{XP}^{\{/,[]\}}$ when the underlying DTD is well-behaved. Before presenting the main theorem, we present the following propositions, each holding trivially. In each case, we assume that q is an XPath query, D is a DTD, $\mathbb{C}$ is the set of constraints implied by D, and $q_{\mathbb{C}}$ is the chase of q by $\mathbb{C}$.

Proposition 6.1 $q \equiv_{SAT(\mathbb{C})} q_{\mathbb{C}}$ *(regardless of the XPath fragment q is in and the types of constraints in $\mathbb{C}$).*

The proof follows from the fact that $q_{\mathbb{C}}$ is obtained by sound chase operations, none of which affects the result of executing q over any document tree in $SAT(\mathbb{C})$.

Proposition 6.2 *$q \equiv_{SAT(D)} q_{\mathbb{C}}$.*

The proof follows from the fact that $SAT(D) \subseteq SAT(\mathbb{C})$ and Proposition 6.1.

Proposition 6.3 *$q \subseteq_{SAT(\mathbb{C})} p$ if $q_{\mathbb{C}} \subseteq p$.*

Proof.

$$\begin{aligned} q_{\mathbb{C}} \subseteq p \quad &\equiv \quad q_{\mathbb{C}} \subseteq_{SAT(\mathbb{C})} p \\ &\Rightarrow \quad q \subseteq_{SAT(\mathbb{C})} p \text{ (since } q \equiv_{SAT(\mathbb{C})} q_{\mathbb{C}}, \text{ by Proposition 6.1)} \end{aligned}$$

■

We now introduce some terminology used in the proof of the theorem below. Given a D-satisfiable query q in $\mathrm{XP}^{\{/,[]\}}$ and a document tree t in $SAT(D)$, we use g to denote a homomorphism from q to t (since q has a nonempty answer on t, such a homomorphism exists). We use $g_{\mathbb{C}}$ to denote a homomorphism from $q_{\mathbb{C}}$ to t such that $g_{\mathbb{C}}(v) = g(v)$ for all v in both q and $q_{\mathbb{C}}$ (since $q_{\mathbb{C}}$ is the chase of q such a homomorphism exists). Furthermore, we will use two different colours, namely white and black, to distinguish between the nodes in t, as follows:

- every node which is in the codomain of g or $g_{\mathbb{C}}$ is coloured black.
- all other nodes are coloured white.

Theorem 6.2 *Let q and p be two queries in $XP^{\{/,[]\}}$, D be a well-behaved DTD, and $\mathbb{C}$ be the set of BSCs and BFCs implied by D. If q is D-satisfiable, then $q \subseteq_{SAT(D)} p$ if and only if $q \subseteq_{SAT(\mathbb{C})} p$.*

Proof. The (if) direction follows from the fact that $SAT(D) \subseteq SAT(\mathbb{C})$. In order to prove the (only if) direction, it is enough, by Proposition 6.3, to prove that $q \subseteq_{SAT(D)} p$ implies $q_{\mathbb{C}} \subseteq p$. To that end, we provide a containment mapping from p to $q_{\mathbb{C}}$, given that $q \subseteq_{SAT(D)} p$. More specifically, we show that there exists a document tree $t \in SAT(D)$, a one-to-one mapping $g_{\mathbb{C}}$ from $q_{\mathbb{C}}$ to the document tree t, and a homomorphism h from p to t such that the following hold:

1. h is covering, i.e. it maps every node in p to a black node in t.

2. h maps every child edge in p to a pair of nodes (z', z) in t such that z' is the parent of z and there is a pair of nodes (u', u) in $q_{\mathbb{C}}$, where u' is the parent of u and $z' = g_{\mathbb{C}}(u')$ and $z = g_{\mathbb{C}}(u)$.

The composite function $g_{\mathbb{C}}^{-1} \circ h$ will become a containment mapping from p to $q_{\mathbb{C}}$ (note that in order for $g_{\mathbb{C}}^{-1} \circ h$ to be a function, as opposed to a relation in general, $g_{\mathbb{C}}$, but not necessarily h, must be one-to-one).

We first prove that there exists a pair $< t, g_{\mathbb{C}} >$, where t is a finite document tree in $SAT(D)$ and $g_{\mathbb{C}}$ is a one-to-one mapping from $q_{\mathbb{C}}$ to t such that $g_{\mathbb{C}}(v) = g(v)$ for all nodes v in both q and $q_{\mathbb{C}}$. We already know that there exists a document tree t in $SAT(D)$ on which q has some nonempty answer. Then there must be a homomorphism g from q to t. Because $q_{\mathbb{C}}$ has the same answer as q on t, there must also be a homomorphism from $q_{\mathbb{C}}$ to t. Let us define a homomorphism $g_{\mathbb{C}}$ from $q_{\mathbb{C}}$ to t such that $g_{\mathbb{C}}(v) = g(v)$ for all nodes v in both q and $q_{\mathbb{C}}$. If $g_{\mathbb{C}}$ is not one-to-one, then there must be two sibling nodes u_1 and u_2 in $q_{\mathbb{C}}$ that are mapped to a same node z in t. Let b be the label of z. Because the DTD is well-behaved there must be another tree in $SAT(D)$ that is the same as t but has an additional node labeled b allowing for u_1 and u_2 to map to distinct nodes. If such a tree does not exist, then u_1 and u_2 must have already been merged as the result of chasing q (because of a BFC), which is a contradiction. Similarly, we can obtain a document tree such that all siblings in $q_{\mathbb{C}}$ with the same label can map to distinct nodes in the tree. Therefore,

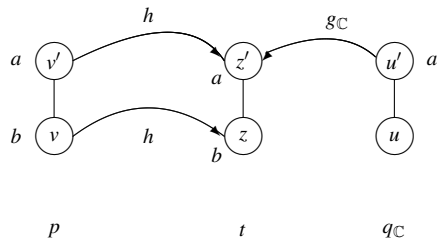


Figure 6.5: Queries p and $q_{\mathbb{C}}$ and tree fragment t in case (i)

there must exist a tree in $SAT(D)$ and a one-to-one mapping from $q_{\mathbb{C}}$ to that tree whose codomain is the same as the codomain of g. We still refer to them as t and $g_{\mathbb{C}}$, respectively. We colour the nodes of t based on g and $g_{\mathbb{C}}$ as described above.

Now we show that there exists a covering homomorphism h from p to t whose codomain does not include any white node in t. Assume that there is no covering homomorphism from p to t that maps every node in p to a black node in t. We already know that there must be a covering homomorphism from p to t because p D-contains q and must have a nonempty answer on t. Let h be a covering homomorphism from p to t. Assume that there is a node v in P such that $h(v)$ is white. Since the roots of p, q, and $q_{\mathbb{C}}$ are mapped to the root of each document tree in $SAT(D)$, v cannot be the root of p, hence it has a parent. Let v' be the parent of v, $z' = h(v')$, $z = h(v)$, and u' be the node in $q_{\mathbb{C}}$ which is mapped by $g_{\mathbb{C}}$ to z' (see Figure 6.5). Also let the labels of v' and v, be a and b, respectively (in the special case, $a = b$). There are two possible cases as follows:

i) z' in t does not have any black b-child.

ii) z' in t has some black b-children which are siblings of z.

Case i) In this case, u' in $q_{\mathbb{C}}$ cannot have a b-child. Hence, there can be no BSC requiring a b-child to be present when all of the other children of u' are present. This means that z and all other b-children of z' in t (and possibly other children too) must be present in t because of D and possibly children of u' in $q_{\mathbb{C}}$, but they do not always have to appear (otherwise there must be a BSC implied by D for a). Hence, we must be able to replace

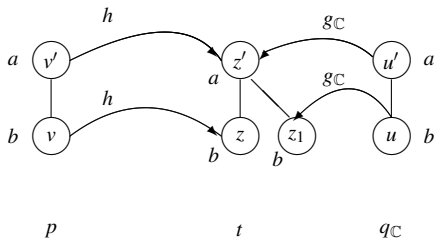


Figure 6.6: Queries p and $q_{\mathbb{C}}$ and tree fragment t in case (ii)

$sub(z')$ in t with a subtree s in $SAT(z')$ whose root has no b-child, and therefore there is no longer a homomorphism from p that maps v' to z'.

Case ii) Now consider case (ii), where z' in t has at least one black b-child z_1 (see Figure 6.6). By assumption, no homomorphism from p can map v to z_1, but h maps v to some sibling z of z_1. Since z_1 and z are both b-nodes, we can replace $sub(z)$ (and that rooted at any other white b-child of z') with that rooted at z_1 to obtain a tree that is still in $SAT(D) \cap SAT(q)$ and to which $g_{\mathbb{C}}$ is still a one-to-one mapping from $q_{\mathbb{C}}$, but to which there is no homomorphism from p that maps v' to z'.

Finally, for every homomorphism from p to t, we can perform the replacements identified in case (i) or (ii) to t in order to obtain a tree t' in which every a-node to which v' was previously mapped by a homomorphism either has no b-children or has only b-children to which v cannot be mapped. Hence, there is no homomorphism from p to t'. Since $t' \in SAT(D) \cap SAT(q)$ and there is still a one-to-one mapping from $q_{\mathbb{C}}$ to t', we have that p does not D-contain q, a contradiction. We conclude, therefore, that there must be a homomorphism h and a one-to-one mapping $g_{\mathbb{C}}$ such that $h(v)$ is a black node. ■

The above result, together with the chase procedure presented in Section 6.3, which in turn uses the algorithm *calculateMCB*, suggests that the D-containment $q \subseteq_{SAT(D)} p$ can be determined by first obtaining the chase $q_{\mathbb{C}}$ of q, followed by deciding the containment $q_{\mathbb{C}} \subseteq p$. However, this method is based on finding the derivative of a regular expression with respect to a bag, as part of the *calculateMCB* procedure, which was shown in Chapter

4 to be exponential in the input size in the worst case. Therefore, we define, in the next section, a special class of well-behaved DTDs for which D-containment of XPath queries in $\text{XP}^{\{/,[]\}}$ can be tested in PTIME.

6.5 Tractability of D-containment for queries in $\text{XP}^{\{/,[]\}}$

In this section, we introduce a type of DTDs, called well-formed DTDs, which are well-behaved and for which the D-containment of XPath queries in $\text{XP}^{\{/,[]\}}$ is in PTIME provided that the set of BSCs and BFCs implied by D is given.

Definition 6.7 *A regular expression R over alphabet Σ^R is* well-formed *if for each subexpression p,q in R, $\Sigma^p \cap \Sigma^q = \emptyset$. A DTD D is* well-formed *if each content model (i.e., regular expression) in D is well-formed.*

Example 6.7 *The following DTD is well-formed:*

$$a \;\rightarrow\; (b,e) \,|\, (d,e^*)$$

However, the following DTD is not well-formed as there exists a concatenation of two occurrences of "e":

$$a \;\rightarrow\; (b,e,e) \,|\, (d,e^*)$$

Well-formed regular expressions occur often in practice. We have examined 100 real-world DTDs with more than 5000 rules. Our experiments reveal that about 92 percent of the content models in the DTDs are well-formed. Table C.1 in Appendix C shows our results.

We now present several theoretical results.

Theorem 6.3 *Every well-formed regular expression is also well-behaved.*

Proof. Let R be a well-formed regular expression over Σ^R, $b \in \Sigma$, and u be a string in $L(R)$ such that $|b|_{[u]} = k$ where $k > 1$. Since R is well-formed, by Definition 6.7, there is no sub-expression p, q in R such that $b \in \Sigma^p$ and $b \in \Sigma^q$. Therefore, the only way to have more than one b in u is to have a Kleene-star in R applied to either b (i.e. $b*$) or a sub-expression containing b. In the first case, we can simply make another string v, still in $L(R)$, from u by increasing the number of bs to any value $n > k$, thereby meeting the first requirement in Definition 6.1. Moreover, the same symbols appear in both u and v, i.e. $\{u\} = \{v\}$, which means the second requirement is also met. Finally, the number of occurrences of every symbol other than b in $[u]$ remains intact in v, i.e. $\forall c \in [u], |c|_{[u]} \leq |c|_{[v]}$, therefore the third requirement is also met, and R is well-behaved.

In the second case, suppose that R' is the sub-expression containing b to which a Kleene-star is applied, and $u = xu'y$, where u' is the substring of u that matches $(R')^*$. We form another string v, still in $L(R)$, by increasing the number of times the Kleene-star is applied to R', i.e. by iterating u' n times, $n > k$. Therefore, the first and third requirements are met ($|b|_{[v]} \geq n$ and $\forall c \in [u], |c|_{[u]} \leq |c|_{[v]}$). Moreover, the same symbols appear in both u and v, i.e. $\{u\} = \{v\}$, which means the second requirement is also met. Therefore, R is well-behaved. ■

Definition 6.8 *Let R be a regular expression and B be a bag of symbols. We define a function f as follows:*

$$f(R,B) = \begin{cases} 1 & \text{if } \delta_B R \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

When $f(R,B) = 1$, we say R covers *B.*

In Theorem 6.4 we will show that $f(R,B)$ can be determined in PTIME, when R is well-formed. To that end, we first present some lemmas. The first lemma obviously holds:

Lemma 6.2 *Let R be a regular expression over alphabet Σ^R, and B be a bag of symbols in Σ^R. Then, $f(R^*, B) = 1$ if and only if for all $b \in B$, b appears in R.*

In the following, for completeness, we allow for the derivative of a regular expression with respect to the empty set, defined as leaving the regular expression intact. That is, $\delta_\emptyset R = R$.

Lemma 6.3 *Let $R = R_1, R_2$ be a nonempty well-formed regular expression over Σ^R, and B be a nonempty bag of symbols in Σ^R. Then, $\delta_B R = (\delta_{B_1} R_1), (\delta_{B_2} R_2)$ where $B_i = \{b^k | b \in \Sigma^{R_i}$ and $k = |b|_B\}$, $i = 1, 2$.*

Proof. First note that $B_1 \cap B_2 = \emptyset$, because of the well-formedness of R. We now prove the lemma by induction on $|B|$.

Base Case: Let B be a singleton $\{b\}$, i.e. $|B| = 1$. Without loss of generality, assume $b \in \Sigma^{R_1}$ (and, therefore, $b \notin \Sigma^{R_2}$). Then, $B_1 = \{b\}$ and $B_2 = \emptyset$. Therefore,

$$\begin{aligned} \delta_B R &= \delta_B(R_1, R_2) \\ &= (\delta_B R_1), R_2 \\ &= (\delta_{B_1} R_1), (\delta_{B_2} R_2). \end{aligned}$$

Induction hypothesis: We assume that for all $|B| \leq n$, $\delta_B R = (\delta_{B_1} R_1), (\delta_{B_2} R_2)$.

Induction step: We assume that we are given $B \uplus \{b\}$ of size $n+1$ and also, without loss of generality, that b appears in R_1, and hence not in R_2. We show that $\delta_{B \uplus \{b\}} R = (\delta_{B_1 \uplus \{b\}} R_1), (\delta_{B_2} R_2)$.

$$\begin{aligned} \delta_{B \uplus \{b\}} R &= \delta_{B_1 \cup B_2 \uplus \{b\}} R \\ &= \delta_{B_1 \cup B_2 \uplus \{b\}}(R_1, R_2) \\ &= \delta_{B_1 \cup B_2}((\delta_b R_1), R_2) \text{ using Equation 4.2, because } \delta_b R_2 = \emptyset \end{aligned}$$

First, consider the case where $B_1 \cup B_2$ is a bag of symbols in $\Sigma^{\delta_b(R_1,R_2)}$. In this case, using the induction hypothesis,

$$\begin{aligned}\delta_{B\uplus\{b\}}R &= (\delta_{B_1}(\delta_b R_1)),(\delta_{B_2}R_2)\\ &= (\delta_{B_1\uplus\{b\}}R_1),(\delta_{B_2}R_2)\end{aligned}$$

Now consider the case where $B_1 \cup B_2$ is not a bag of symbols in $\Sigma^{\delta_b(R_1,R_2)}$. In this case, obviously, $\delta_{B_1\cup B_2}((\delta_b R_1),R_2) = \emptyset$. We now show that in this case also $(\delta_{B_1\uplus\{b\}}R_1),(\delta_{B_2}R_2) = \emptyset$ to complete the proof. To that end, note that $B_1 \cup B_2$ is a bag of symbols in Σ^{R_1,R_2} and the only reason for $B_1 \cup B_2$ not being a bag of symbols in $\Sigma^{\delta_b(R_1,R_2)}$ is that $b \in \Sigma^{R_1}$, $b \notin \Sigma^{\delta_b R_1}$, and $|b|_{B_1} > 0$. However, in this case, $(\delta_{B_1}(\delta_b R_1)) = \emptyset$, which implies $(\delta_{B_1\uplus\{b\}}R_1),(\delta_{B_2}R_2) = \emptyset$. ■

Lemma 6.4 *Let $R = R_1|R_2$ be a nonempty well-formed regular expression over Σ^R, and B be a nonempty bag of symbols in Σ^R. Then, $\delta_B R = (\delta_B R_1)|(\delta_B R_2)$.*

Proof. We use induction on $|B|$.

Base Case: Let B be a singleton $\{b\}$, i.e. $|B| = 1$. Then,

$$\begin{aligned}\delta_B R &= \delta_B(R_1|R_2)\\ &= (\delta_B R_1)|(\delta_B R_2)\end{aligned}$$

Induction hypothesis: We assume for $|B| = n$, $\delta_B R = (\delta_B R_1)|(\delta_B R_2)$.

Induction step: We show that $\delta_{B\uplus\{b\}}R = (\delta_{B\uplus\{b\}}R_1)|(\delta_{B\uplus\{b\}}R_2)$.

$$\begin{aligned}\delta_{B\uplus\{b\}}R &= \delta_{B\uplus\{b\}}(R_1|R_2)\\ &= \delta_B((\delta_b R_1)|(\delta_b R_2)) \text{ using Equation 4.2}\\ &= (\delta_B(\delta_b R_1))|(\delta_B(\delta_b R_2)) \text{ by the induction hypothesis}\end{aligned}$$

$$= \quad (\delta_{B\uplus\{b\}}R_1))|(\delta_{B\uplus\{b\}}R_2)$$

■

Theorem 6.4 *Let R be a nonempty well-formed regular expression over Σ^R, and B be a nonempty bag of symbols in Σ^R. Then, $f(R,B)$ can be determined in polynomial time.*

Proof. We use strong induction on $|R|$.

Base case: Assume $|R| = 1$. In this case, trivially, $f(R,B) = 1$ if and only if B is a singleton that contains the only symbol in R; this can be decided in polynomial time. Let $B = \{a\}$; based on the answer to whether or not the symbol a occurs in R, $f(R,B)$ returns 1 or 0 in $O(|R|)$.

Induction hypothesis: We assume that $f(R,B)$ is decidable in polynomial time, for all $|R| \leq n$, for some $n \geq 1$.

Induction step: We prove that $f(R,B)$ is decidable in polynomial time, where $|R| = n+1$. To that end, we note that $f(R,B) = 1$ if and only if $\delta_B R$ is not empty; it is 0 otherwise. Therefore, we show that we can decide in polynomial time whether $\delta_B R$ is not empty. Since $|R| > 1$, R is not a singleton and one of the following three cases must hold, where R_1 and R_2 are in turn well-formed regular expressions with size no greater than n:

1. $R = R_1, R_2$. In this case, by Lemma 6.3, $\delta_B R = (\delta_{B_1} R_1), (\delta_{B_2} R_2)$, where $B_i = \{b^k | b \in \Sigma^{R_i}$ *and* $k = |b|_B\}$, $i = 1,2$. Therefore, $\delta_B R$ is not empty if and only if neither $\delta_{B_1} R_1$ nor $\delta_{B_2} R_2$ is empty. That is, $f(R,B) = f(R_1,B_1), f(R_2,B_2)$. On the other hand, both the bags B_1 and B_2 and, by the induction hypothesis, both the values $f(R_1,B_1)$ and $f(R_2,B_2)$ can be determined in polynomial time. That is, $f(R,B)$ is decidable in polynomial time.

2. $R = R_1|R_2$. In this case, by Lemma 6.4, $\delta_B R = (\delta_B R_1)|(\delta_B R_2)$. Therefore, $\delta_B R$ is not empty if and only if either $\delta_B R_1$ or $\delta_B R_2$ is not empty. That is, $f(R,B) = 1$ if, and

only if, $f(R_1,B)+f(R_2,B)>0$. On the other hand, by the induction hypothesis, both the values $f(R_1,B)$ and $f(R_2,B)$ can be determined in polynomial time. That is, $f(R,B)$ is decidable in polynomial time.

3. $R=(R_1^*)$. In this case, by Lemma 6.2, $\delta_B R=1$ if and only every symbol in B appears in R_1, which is decidable in polynomial time.

■

Given a set $\mathbb{C}$ of BSCs and BFCs implied by DTD D and queries p and q in XP$^{\{/,[]\}}$, we will prove, in Theorem 6.5, that deciding D-containment is in PTIME. To that end, we first show in Lemma 6.5 that $MCB(v,v)$, for each node v in q, can be calculated in PTIME. Then, in Lemma 6.6, we prove that the chase of q by $\mathbb{C}$ can be determined in PTIME.

Lemma 6.5 *Let D be a well-formed DTD and q be a D-satisfiable query in XP$^{\{/,[]\}}$ whose root is r. Then $MCB(r,r)$ can be determined in polynomial time.*

Proof. First, we show that the calculation of the sets $PC(v,r)$ and $LPC(v,r)$ for all nodes v in the tree q rooted at r can be accomplished in polynomial time. These sets are calculated only once in the first (the highest) recursive call, when $v=r$, and used in all the subsequent recursive calls, where v is a non-root node in the tree q rooted at r. We only show that the calculation of $PC(v,r)$, for all the nodes v in the tree q, can be accomplished in polynomial time, because the calculation of $LPC(v,r)$ will then be simply performed in polynomial time too.

To that end, we process the nodes v in the tree from the root r to the leaves. That is, for each level L of the tree in increasing order, starting from $L=0$ for the root, we calculate $PS(*,v,r)$ and $PC(v,r)$ for every node v in level L. For each node v in a level L, $PS(*,v,r)$ is obtained by going through all the nodes v' in the level L of the tree and checking whether v' is a potential sibling of v, i.e. whether $PS(v,v',r)$ holds. That is, we need to determine whether $parent(v')$ has the same label as $parent(v)$ and belongs to $PS(*,parent(v),r)$.

Because of processing the nodes from top to bottom (in a dynamic programming fashion), we already know $PS(*, parent(v), r)$. Because $PS(*, parent(v), r) = O(|q|)$, it takes $O(|q|)$ to decide whether v' belongs to $PS(*, parent(v), r)$. Similarly, because there are $O(|q|)$ nodes which we go through, it takes $O(|q|^2)$ to determine $PS(*, v, r)$. Finally, the definition of $PC(v, r)$ implies that it will be the empty set if v is a leaf. Otherwise, let u be a child of v (which resides in the level $L+1$). Then, $PC(v, r)$ will be the set $PS(*, u, r)$, which we just showed can be determined in polynomial time.

Having calculated $PC(v, r)$ (and consequently $LPC(v, r)$) in polynomial time, we now show the rest of the algorithm also runs in polynomial time. To that end, we use induction to prove the proposition that each invocation of $calculateMCB(v, r)$, where v is a node in the tree q rooted at r, runs in polynomial time. Let L be the level at which the node v is located in the tree and L_{max} be the depth of the tree, i.e. the level of the deepest node in q. We use induction on L, this time starting from $L = L_{max}$ as the base case and proving that if the proposition holds for the nodes at a level greater than 0, it will also hold for the nodes at the level $L-1$.

Base case: In this case $L = L_{max}$ and v is a leaf ($LPC(v, r) = \emptyset$), and the algorithm returns $\emptyset$ in $O(1)$.

Induction hypothesis: We assume that the algorithm is run in polynomial time, where v is at a level $0 < L \leq L_{max}$.

Induction step: We prove that the algorithm runs in polynomial time, where v is located at level $L-1$. If v is a leaf, then the algorithm runs in $O(1)$ time. Otherwise, $LPC(v, r)$ is nonempty and the for-loop iterates $|LPC(v, r)| = O(|q|)$ times. At each iteration of the for-loop, the first if-condition is evaluated in $|PC(v, r)| = O(|q|)$ and if evaluated to true, the algorithm returns in polynomial time. Otherwise, the first else-block is run, which includes a recursive invocation of the algorithm, which is performed in polynomial time by the induction hypothesis. The rest of the code (including the second if-statement, based on Theorem 6.4) also runs in polynomial time. Therefore, each iteration of the for-

loop runs in polynomial time, and there are $O(|q|^2)$ of these iterations, which completes the proof. ■

Lemma 6.6 *Let D be a well-formed DTD, $\mathbb{C}$ be the set of BSCs and BFCs implied by D, and q be a tree in XP$^{\{/,[]\}}$. Then the chase $q_{\mathbb{C}}$ of q by $\mathbb{C}$ can be determined in polynomial time.*

Proof. In order to obtain the chase of q by $\mathbb{C}$, we go through each vertex v in q, from level 0 to the last level of the tree of q, and check whether there are some constraints in $\mathbb{C}$ applicable to v. In particular, for each BFC $a : B \downarrow b$ (respectively BSC $a : B \Downarrow c$) in $\mathbb{C}$, we first check whether $B \subseteq MCB(v,v)$. Consequently, two or more children of v may be merged (respectively, a child may be added). However, each merge procedure (respectively, addition of a child) is performed in polynomial time, calculation of $MCB(v,v)$ is performed in polynomial time by Lemma 6.5, checking for $B \subseteq MCB(v,v)$ is decided in $O(|MCB(v,v)| \times |B|) = O(|q| \times |\Sigma|)$, where Σ is the alphabet, and there are $|C|$ constraints to check. Therefore, the whole process of checking and applying constraints, if any, to a node v is performed in polynomial time. Since the number of nodes in q is $O(|q|)$, the chase of q by $\mathbb{C}$ is obtained in polynomial time. ■

Theorem 6.5 *Let D be a well-formed DTD and $\mathbb{C}$ be the set of BSCs and BFCs implied by D. Then the D-containment of queries in XP$^{\{/,[]\}}$ can be decided in PTIME.*

Proof. In (the only-if part of) Theorem 6.2, we showed that $q \subseteq_{SAT(D)} p$ implies $q_{\mathbb{C}} \subseteq p$. Conversely, $q_{\mathbb{C}} \subseteq p$ also implies $q \subseteq_{SAT(D)} p$, because $q_{\mathbb{C}} \subseteq p$ implies $q_{\mathbb{C}} \subseteq_{SAT(\mathbb{C})} p$ which (by Proposition 6.1) implies $q \subseteq_{SAT(\mathbb{C})} p$, which in turn implies $q \subseteq_{SAT(D)} p$ (because $SAT(D) \subseteq SAT(\mathbb{C})$). Therefore, to decide the D-containment $q \subseteq_{SAT(D)} p$, we only need to decide the containment $q_{\mathbb{C}} \subseteq p$. By Lemma 6.6, $q_{\mathbb{C}}$ is obtained in polynomial time.

Finally, because the containment of XPath queries in $\text{XP}^{\{/,[]\}}$is in PTIME [65], the containment $q_{\mathbb{C}} \subseteq p$ can be decided in polynomial time, which completes the proof. ■

Theorem 6.6 *Under well-formed DTDs, the containment problem for XPath queries in each of the following fragments is coNP-hard:*

1. $XP^{\{/,[],*\}}$
2. $XP^{\{/,[],//\}}$
3. $XP^{\{/,[],\cup\}}$

Proof. The proof follows immediately from Theorem 5.2 and the fact that each duplicate-free DTD is also well-formed and the satisfiability problem is reducible to the complement of containment problem [7]. ■

6.6 Intractability results regarding covering DTDs

In this section, we present several theoretical results regarding the containment problem in the presence of covering DTDs.

The following theorem shows that, under covering DTDs, the *D*-containment of queries in $\text{XP}^{\{/,[]\}}$ is intractable, unless $P = NP$, even though the *D*-satisfiability of such queries under covering DTDs was shown in the previous chapter to be tractable. This result is important because it implies that for any other fragment that include child axis and filter operators the problem, under covering DTDs, remain coNP-hard. This is the case in practical applications, and most real world XPath queries contain these operators.

Theorem 6.7 *XPath containment under covering DTDs for* $XP^{\{/,[]\}}$ *is coNP-hard.*

We prove the theorem by reduction from the Sibling Constraint Implication (SC IMP) problem, which has been shown to be coNP-hard for general DTDs [91]. First, we show that it is also coNP-hard for covering regular expressions.

Let R^a be the regular expression representing the content model of an element a in a DTD D, Σ^a be the set of element names appearing in R^a, and B be a bag of symbols in Σ^a. R^a implies the sibling constraint $a : B \Downarrow c$, denoted $R^a \models a : B \Downarrow c$, if $\forall w \in L(R^a)$, when the bag $[w]$ contains the bag B then it also contains c. In other words, we have $\forall w \in L(\delta_B R^a) : c \in [w]$.

Now we prove that SC IMP is coNP-hard for covering regular expressions.

Lemma 6.7 *SC IMP for covering regular expressions is coNP-hard.*

Proof. We show that SC IMP remains coNP-hard for covering regular expressions by reduction from the SC IMP problem.

Let R^a be the regular expression representing the content model of an element a in a DTD D, Σ^a be the set of element names appearing in R^a, B be a bag of symbols in Σ^a, and $c \in \Sigma^a$ be a symbol such that $c \notin B$ (when c is an element of B, the SC is trivial, i.e. always holds independently of the regular expression, in both covering and non-covering regular expressions). Also let $\overline{R^a} = (R^a|w)$, where $[w] = \Sigma^a$. So $\overline{R^a}$ is a covering regular expression. $\overline{R^a}$ can easily be obtained by adding w to R^a in PTIME. Now we show that $\overline{R^a} \models a : B \Downarrow c$ if and only if $R^a \models a : B \Downarrow c$.

We first note that:

$$\delta_B \overline{R^a} = (\delta_B R^a)|(\delta_B w), \text{ where } \delta_B w = \begin{cases} [w] - B & \text{if } B \subseteq [w] \\ \emptyset & \text{otherwise} \end{cases} \tag{6.1}$$

The only-if direction (trivial): Assume that $\overline{R^a} \models a : B \Downarrow c$. This means $\forall w' \in L(\delta_B \overline{R^a}) : c \in [w']$. Using Equation 6.1: $\forall w' \in L((\delta_B R^a)|(\delta_B w)) : c \in [w']$. This means: $\forall w' \in L(\delta_B R^a) : c \in [w']$, i.e. $R^a \models a : B \Downarrow c$.

The if direction: Assume that $R^a \models a : B \Downarrow c$. There exist two possible cases:

1. $B \not\subseteq [w]$ (trivial)

   This means that $\forall w' \in L(\delta_B R^a) : c \in [w']$, which is equivalent to $\overline{R^a} \models a : B \Downarrow c$ when $B \not\subseteq [w]$ (Equation 6.1).

2. $B \subseteq [w]$

   This means:

   $$\forall w' \in L(\delta_B R^a) : c \in [w'] \tag{6.2}$$

   On the other hand:

   $$\text{if } c \in \Sigma^a \text{ and } c \notin B, \text{ then } c \in [w] - B \tag{6.3}$$

   because $[w] = \Sigma^a$. Now consider $\overline{R^a}$ and an arbitrary string $w' \in L(\delta_B \overline{R^a})$:

   $$\begin{aligned} w' \in L(\delta_B \overline{R^a}) &\Rightarrow w' \in L(\delta_B R^a | \delta_B w) \\ &\Rightarrow (w' \in L(\delta_B R^a)) | (w' \in L(\delta_B w)) \\ &\Rightarrow (c \in [w']) | (c \in [w']) \text{ using Equations 6.2 and 6.3} \\ &\Rightarrow c \in [w'] \end{aligned}$$

   This means $\overline{R^a} \models a : B \Downarrow c$.

■

Now we prove Theorem 6.7:

Proof. Given an instance of SC IMP:

Instance: A covering expression R over alphabet Σ^R, $B = \{b_1, \ldots, b_k\} \subseteq \Sigma^R$, and $c \in \Sigma^R$.

Question: Does every string $w \in L(R)$ that contains B, i.e. $B \subseteq [w]$, also contain the symbol c (denoted $R \models a : B \Downarrow c$)?

We construct the following instance of the D-containment problem for $\mathrm{XP}^{\{/,[]\}}$. Assume

that D is a DTD with just one rule $a \rightarrow R$, and $p = a[b_1]\ldots[b_k]$ and $q = a[c]$ are two XPath queries which are D-satisfiable. Then $a[b_1]\ldots[b_k] \subseteq_{SAT(D)} a[c]$ if and only if $R \models a : B \Downarrow c$. According to Lemma 6.7 deciding whether $R \models a : B \Downarrow c$ is coNP-hard, and the construction of the D-containment instance is accomplished in polynomial time. Therefore, deciding $a[b_1]\ldots[b_k] \subseteq_{SAT(D)} a[c]$ is also coNP-hard. ■

6.7 Conclusion

In this chapter, we first introduced a new DTD property, called well-behaved, which is a semantic rather than a syntactic property. It is a generalisation of the duplicate-free property. However, syntactically, a symbol may appear more than once in a well-behaved DTD content model. We then showed that BFCs and BSCs (DTD constraints defined in Chapter 4) are necessary and sufficient for deciding D-containment of XPath queries in $\mathrm{XP}^{\{/,[]\}}$ under well-behaved DTDs.

Furthermore, we introduced a subclass of well-behaved DTDs, called well-formed DTDs, and formally proved the tractability of D-containment of XPath queries in $\mathrm{XP}^{\{/,[]\}}$ under such DTDs, given the set of BFCs and BSCs. Moreover, the tractability result still holds for other classes of well-behaved DTDs as long as determining whether the content model of a DTD rule covers a bag of symbols can be accomplished in polynomial time. Therefore, a potential avenue for future work is to determine other subclasses of well-behaved DTDs that meet this requirement.

Chapter 7

Conclusions

This chapter summarises the contributions of the thesis and introduces some possible avenues for future work.

7.1 Summary

In this thesis, we have studied different aspects of the XPath query containment and satisfiability problems under DTDs (Document Type Definitions). In general, there are no efficient solutions for these problems, so the main purpose of this study is to determine special cases in which the problems become tractable.

We first defined derivatives of regular expressions, with respect to bags of symbols. Derivatives of regular expressions had been previously defined but with respect to strings of symbols. However, we want to use derivatives in the static analysis of XPath queries posed on XML documents that are valid with respect to some DTD D. More specifically, we want to determine, when querying documents that are valid with respect to D, whether a given node v in an XPath query can have a set of child nodes whose labels form the bag B, irrespective of the order of the child nodes. This turns out to be equivalent to determining whether the derivative of the content model (regular expression) for v in D with respect to B is non-empty. We called this the DERIVATIVE NON-EMPTINESS problem. We proved

that DERIVATIVE NON-EMPTINESS is NP-complete.

Previous studies have introduced constraints inferred from DTDs to decide containment in terms of the chase procedure. Several procedures for different fragments of XPath have been proposed and different DTD constraints have been applied. However, the results are still not satisfactory, partially because the proposed DTD constraints do not include all the DTD information. Motivated by such difficulties, in Chapter 4, we studied constraints implied by DTDs. We defined more general constraints based on bags, as opposed to sets, of element names and proved some of their properties. In particular, two types of DTD constraints, *Bag Sibling Constraints (BSCs)* and *Bag Functional Constraints (BFCs)* were introduced.

One of our concerns was discovering properties of real-world DTDs and their impact on the satisfiability and containment problems for XPath queries. The XPath satisfiability problem is, given a DTD D and an XPath query q, to decide whether or not the query q can ever return a non-empty answer over any document preserving the DTD constraints expressed by D. In other words, it is to decide whether q is *consistent* with D. The XPath D-containment problem, on the other hand, is, given a DTD D and two XPath queries p and q, whether or not every answer returned by evaluating p over a document preserving D is also an answer returned by evaluating q over that document. The problem is in general coNP-hard, and extensive research has been conducted to determine special cases where it becomes tractable.

The satisfiability problem for the fragment $\mathrm{XP}^{\{/,[]\}}$, denoted by $\mathrm{SAT}(\mathrm{XP}^{\{/,[]\}})$, is NP-hard in general. This result follows from a result in [91]. A contribution of this thesis regarding the satisfiability problem is the discovery of a property we called the *covering* property. We showed that XPath satisfiability for the fragment $\mathrm{XP}^{\{/,[],*,//,\cup\}}$ is in PTIME for covering DTDs. We also showed that $\mathrm{SAT}(\mathrm{XP}^{\{/,[]\}})$ is decidable in PTIME for duplicate-free DTDs. The notion of a duplicate-free DTD was introduced in [91]. Our investigation of real-world DTDs, which led to the discovery of the prevalent property of

covering, revealed the pleasing fact that for many real-world cases, the XPath satisfiability problem can be solved in PTIME [68].

One of the situations that increases the complexity of XPath containment is the existence of duplicate elements in queries, particularly duplicate siblings. The main difficulty with this situation is related to the question of whether or not two or more such siblings could map to the same node in a document tree. However, by examining real-world DTDs, we observed that it is only important whether each label of a set of siblings can occur either only once or infinitely often. Based on this idea, we defined the *well-behaved* property and showed that Bag Sibling Constraints (BSCs) and Bag Functional Constraints (BFCs) are necessary and sufficient to decide D-containment under such DTDs. Unfortunately, the well-behaved property does not allow for an efficient test for D-containment, so we defined a subclass of well-behaved DTDs called *well-formed* DTDs.

In particular, we showed that DERIVATIVE NON-EMPTINESS for well-formed regular expressions can be solved in polynomial time. This tractability result has promising application in deciding whether or not a regular expression covers a bag of symbols, an important problem in the context of deciding containment of XPath queries [92]. Finally, we showed that, given the set BSCs and BFCs implied by a DTD D, D-containment of queries in $\mathrm{XP}^{\{/,[]\}}$ under well-formed DTDs is tractable.

7.2 Future Work

The presented work is just a starting point in the direction of discovering features of real-world applications and deriving low-cost algorithms for such problems as query satisfiability, containment, and optimisation. Among possible avenues for further research in this regard are the following.

1. In this thesis, we found sets of constraints implied by certain types of DTD that are sound and complete for deciding containment for $\mathrm{XP}^{\{/,[]\}}$ using the chase procedure

and containment mappings. Corresponding results in the following areas remain to be investigated:

- Extending the analysis to larger XPath fragments.
- Extending the analysis to other, more general, types of DTDs and showing that the restricted nature of the DTD results in particular constraints being sufficient and necessary to capture all possibilities.

2. It would be interesting to investigate a precise classification of XPath features and determine, for each class, the complexity of containment under well-behaved/well-formed DTDs, so it is known which classes yield tractability, NP-hardness, or even undecidability. Similarly, such a classification is open for the satisfiability problem under covering DTDs.

3. Further work is needed to investigate the problem of satisfiability in the presence of other axes, for example sibling-axes, for which the problem remains tractable.

4. The number of all constraints implied by a given DTD may be enormous, some of which are redundant. Another challenge is to investigate the methods to derive a set of constraints with no redundancy.

5. In Chapter 3, we used a counter-example to show that the constraints proposed by [53] are not sufficient to capture containment when the underlying DTD is not duplicate-free. It would be interesting to prove that they are sufficient when the underlying DTD is duplicate-free.

6. The experimental results in this thesis showed that when DTDs are classified as non-covering, it is mostly due to only a few of their rules being non-covering. This suggests for the notion of 'locally', vs. 'globally', covering DTDs. Consequently, a possibility for future work is to determine features of queries for which the satisfia-

bility problem under locally covering DTDs (respectively, the containment problem under locally well-behaved DTDs) remains tractable.

7. It would be worthwhile to further investigate real-world schema constraints, including DTDs, and discover new features; and also to investigate the effect of such features on achieving PTIME algorithms for common XPath problems. For such research which is highly dependent on real-world data, it is first required to determine how to obtain reliable real-world XPath DTDs, or in general schema constraints, as well as real-world XPath queries.

8. Another suggestion for future work is to extend the constraints, for the purpose of containment under constraints, from simple DTDs to more general schema languages such as XML schema or even Relax NG, as these have not yet been studied sufficiently in the literature.

Appendix A

DTDs and their application domains

Table A.1: DTD names and their application domains

DTD Name	Application Domain
Oagis	Open Applications Group archives
Odm1-1-0	Optimal Design Markup Language
LevelOne	HL7 Clinical document Architecture
Ecoknowmics	Economic Knowledge Management
XML Schema	XML Schema
HP	HL7 Document architecture
Meerkat-xml-flavour	Storehouse of News about Technological Developments
OSD	Open Software Description
Opml	Outline Processing Markup Language
Rss-091	XML vocabulary for describing metadata about websites
TV-Schedule	TV Schedule
Xbel 1.0	XML Bookmarks Exchange Language
XHTML1-strict	Extensible HTML version 1.0 Strict
Newspaper	Newspaper
DBLP	Digital Bibliography Library Project
Music ML	Music Digital Library
XMark	DTD XML Benchmark
Yahoo	Yahoo auction data
Reed	Courses from Reed College
Nlm	Medline National Library of Medicine
SigmodRecord	Index of articles from SIGMOD Record
Ubid	UBid auction data
Ebay	EBay auction data
News ML	News
PSD	Protein Sequence Database
Mondial 3.1	World geographic database

Table A.1: (continued)

DTD Name	Application Domain
321gone	Auction
ADL-Access-Report	ADL Access Report Model
Docbooks.dtd	DocBook XML DTD
XHTML1-Frameset	Extensible HTML Frameset
XHTML1-Strict	Extensible HTML Strict
XHTML1-Transitional	Extensible HTML Transitional
E-Invoice	Electronic invoice
Hibernate-mapping-2.0	Java object/relational mapping tool
Imagelib	Image Library used in Docbook
RecipeML	Recipe Markup Language
RSS-2.0	Really Simple Syndication - Web content syndication format
TieXLite	SGML TEI Lite
Tr9401	Oasis Catalog Standard
PFA Legal XML	JNET Messages
PropertyList	Apple's developer tools
RDF SGML	Resource Description Framework
MathML	Mathematical Markup Language
DSSSL	DSSSL Architectural Forms
Web-app-2-3	Web Application 2.3 Model
XTM1	XML Topic Map DTD
Olinksum	OLINK Summary Information
QAML	Frequently Asked Questions
Repository	the grammar of the Descriptor repository
Soextblx	Exchange Table Model
Interim Legal XML	JNET Messages
FOT DTD	DSSSL Flow Object Tree
FGDC-1.00	Declaration for formal metadata
Arrest	Legal XML - JNET Messages
Catalog	OASIS XML catalog
ComicsML	Comics Markup Language
web-facesconfig-1-1	JavaServer Faces Application Configuration File
Ag-1.1	Annotation Graphs
springbeans	Namespace for JavaBeans Objects
article-R.1.3.dtd	Scientific and scholarly articles
autoupdate-catalog-1	Update Center Modules
Oil	Ontology Integration Language OIL
cml-10	Chemical Markup Language
document-info-1.3	Store information about documents
docutils	Store information about documents
DS-DevInf-V1-2	SyncML Device Information
ebBPSS	Execution of business transactions
Groupstats	Group Statistics
Userstats	User Statistics
HDF5-File-1-2-2	NCSA Hierarchical Data Format

Table A.1: (continued)

DTD Name	Application Domain
ibtwsh	Itsy Bitsy Teeny Weeny Simple Hypertext
karbon-1.3	Karbon Markup Language
kdatabase-1.2	KDatabase document format
kformula-1.3	Math symbols and formulas
kontour-1.2	KIllustrator Markup Language
kpresenter-1.3	KPresenter document format
kspread-1.3	KSpread document format
kugartemplate-1.3	Kugar template
kword-1.3	KWord document format
oebdoc12	Open eBook Publication Structure
request	Electronic communications
Resume	Resume Document Type Definition
Shoe-xml	An extension to HTML
RTML-2.1	Remote Telescope Markup Language
rfc2629	RFC document series
Sodaconstructor	Browser-based applet
TMML-1.0	Turing Machine Markup Language
videoCD	VCDImager VideoCD XML
VOTable	Virtual Observatory Tabular Format
wddx-dtd-10	Web Distributed Data Exchange
xbn	XML Belief Network file Format
xmlTV	TV listings representation
zthes-05	Thesaurus Navigation
Fo2000	XSL FO documents
SYNCML-METINF-1.1	SyncML Representation Protocol
DMDDFDTD-1.2	OMA DM Device Description Framework
pap-2.1	Push Access Protocol
tabletemplatestes-1.3	Tablestyle Marlup Language
CDisc-11	Clinical Data Interchange Standards Consortium
Sun-domain-1.20	Glassfish configuration

Appendix B

Covering DTDs

Table B.1: The classification of DTD rules

DTD Name	Number of Rules	Non-covering		Covering	
		Dup-free	Dup	Dup-free	Dup
ADL-Access-Report	19	2	0	16	1
Docbooks.dtd	360	18	7	314	21
XHTML1-Frameset	91	1	0	88	2
XHTML1-Strict	77	1	0	74	2
XHTML1-Transitional	89	1	0	86	2
E-Invoice	66	0	0	66	0
Hibernate-mapping-2.0	49	3	0	38	8
Imagelib	7	0	0	7	0
RecipeML	68	3	4	58	3
RSS-2.0	30	0	0	29	1
TieXLite	143	26	6	109	2
Tr9401	7	0	0	7	0
PFA	24	0	0	24	0
PropertyList	11	0	0	10	1
RDF	11	0	0	11	0
MathML	39	0	0	39	0
DSSSL	16	0	0	16	0
Web-app-2-3	77	0	0	74	3
XTM1	19	0	0	15	4
Olinksum	8	0	0	8	0
FAQ	28	0	0	26	2
Repository	25	0	1	24	0
soextblx	7	0	0	7	0
Iterim	11	0	0	11	0
FOT	83	0	0	83	0
FGDC-1.00	340	5	2	315	18
Arrest	38	0	0	38	0
Catalog	6	0	0	6	0

Table B.1: (continued)

DTD Name	Number of Rules	Non-covering		Covering	
		Dup-free	Dup	Dup-free	Dup
ComicsML	31	0	0	31	0
web-facesconfig-1-1	80	0	0	76	4
Ag-1.1	9	0	0	9	0
springbeans	22	0	0	18	4
article-R.1.3.dtd	74	1	0	68	5
autoupdate-catalog-1-0	14	0	0	11	3
Oil	49	2	1	45	1
cml-10	24	0	0	24	0
Document-info-1.3	15	0	0	15	0
Docutils	89	3	0	86	0
DS-DevInf-V1-2	47	0	0	47	0
ebBPSS	28	1	0	27	0
Groupstats	14	0	0	14	0
Userstats	17	0	0	17	0
HDF5-File-1-2-2	48	0	0	39	9
Ibtwsh	37	1	0	36	0
karbon-1.3	10	0	0	10	0
Kdatabase-1.2	16	0	0	16	0
kformula-1.3	27	0	0	27	0
kontour-1.2	26	0	0	20	6
kpresenter-1.3	86	0	0	86	0
Sodaconstructor	11	0	0	11	0
TMML-1.0	8	0	0	8	0
videoCD	45	0	0	44	1
voTable	24	0	0	23	1
wddx-dtd-10	16	0	0	14	2
Xbn	28	0	0	28	0
xmlTV	40	0	0	40	0
zthes-05	15	0	0	15	0
Fo2000	56	1	0	52	3
SYNCML-METINF-1.1	17	0	0	17	0
DMDDFDTD-1.2	56	0	0	48	8
pap-2.1	18	0	0	17	1
tabletemplates-1.3	8	0	0	7	1
CDisc-11	85	0	0	84	1
Sun-domain-1.20	109	0	0	108	1
Oagis	617	161	18	422	16
Geophysical ML	444	0	1	414	29
LevelOne	31	0	0	29	2
Ecoknowmics	224	1	0	221	2
XML Schema	26	1	0	19	6
HP	59	0	0	59	0

Table B.1: (continued)

DTD Name	**Number of Rules**	**Non-covering**		**Covering**	
		Dup-free	Dup	Dup-free	Dup
Meerkat	14	0	0	14	0
OSD	15	0	0	14	1
Opml	15	0	0	15	0
RSS-091	24	0	0	24	0
TV-Schedule	10	0	0	10	0
Xbel-1.0	9	0	0	9	0
PRI-PSD	56	0	0	56	0
Newspaper	7	0	0	7	0
DBLP	37	0	0	37	0
Music ML	12	3	0	9	0
XMark	77	1	0	75	1
Yahoo	32	0	0	32	0
Reed	16	0	0	16	0
NLM Medline	41	0	0	41	0
Sigmod Record	11	0	0	11	0
Ubid	32	0	0	32	0
Ebay	32	0	0	32	0
News ML	116	0	0	112	4
PSD	66	0	0	64	2
Mondial 3.1	23	0	0	23	0
321gone	32	0	0	32	0
kspread-1.3	36	0	0	35	1
kugartemplate-1.3	13	0	0	13	0
kword-1.3	79	0	0	76	3
oebdoc12	66	1	0	64	1
Request	6	0	0	5	1
Resume	106	0	0	97	9
Shoe-xml	17	1	0	16	0
RTML-2.1	39	0	0	37	2
Rfc2629	46	0	0	46	0
Total	5534	236	44	5053	201
Percentage	100%	4.3	0.8	91.3	3.6%

Appendix C

Well-formed DTDs

Table C.1: The classification of DTD rules

DTD Name	Number of Rules	Number of well-formed Rules
ADL-Access-Report	19	18
Docbooks.dtd	360	332
XHTML1-Frameset	91	89
XHTML1-Strict	77	75
XHTML1-Transitional	89	87
E-Invoice	66	66
Hibernate-mapping-2.0	49	41
Imagelib	7	7
RecipeML	68	61
RSS-2.0	30	29
TieXLite	143	135
Tr9401	7	7
PFA	24	24
PropertyList	11	10
RDF	11	11
MathML	39	39
DSSSL	16	16
Web-app-2-3	77	74
XTM1	19	15
Olinksum	8	8
FAQ	28	26
Repository	25	24
soextblx	7	7
Iterim	11	11
FOT	83	83
FGDC-1.00	340	324
Arrest	38	38
Catalog	6	6

Table C.1: (continued)

DTD Name	Number of Rules	Number of well-formed Rules
ComicsML	31	31
web-facesconfig-1-1	80	76
Ag-1.1	9	9
springbeans	22	18
article-R.1.3.dtd	74	69
autoupdate-catalog-1-0	14	12
Oil	49	47
cml-10	24	24
Document-info-1.3	15	15
Docutils	89	89
DS-DevInf-V1-2	47	47
ebBPSS	28	28
Groupstats	14	14
Userstats	17	17
HDF5-File-1-2-2	48	44
Ibtwsh	37	37
karbon-1.3	10	10
Kdatabase-1.2	16	16
kformula-1.3	27	27
kontour-1.2	26	26
kpresenter-1.3	86	86
Sodaconstructor	11	11
TMML-1.0	8	8
videoCD	45	44
voTable	24	23
wddx-dtd-10	16	16
Xbn	28	28
xmlTV	40	40
zthes-05	15	15
Fo2000	56	54
SYNCML-METINF-1.1	17	17
DMDDFDTD-1.2	56	55
pap-2.1	18	18
tabletemplates-1.3	8	8
CDisc-11	85	85
Sun-domain-1.20	109	108
Oagis	404	364
Geophysical ML	444	418
LevelOne	31	29
Ecoknowmics	224	223
XML Schema	26	21
HP	59	59

Table C.1: (continued)

DTD Name	Number of Rules	Number of well-formed Rules
Meerkat	14	14
OSD	15	15
Opml	15	15
RSS-091	24	24
TV-Schedule	10	10
Xbel-1.0	9	9
PRI-PSD	56	56
Newspaper	7	7
DBLP	37	37
Music ML	12	12
XMark	77	77
Yahoo	32	32
Reed	16	16
NLM Medline	41	41
Sigmod Record	11	11
Ubid	32	32
Ebay	32	32
News ML	116	112
PSD	66	65
Mondial 3.1	23	23
321gone	32	32
kspread-1.3	36	36
kugartemplate-1.3	13	13
kword-1.3	79	78
oebdoc12	66	64
Request	6	6
Resume	106	98
Shoe-xml	17	17
RTML-2.1	39	37
Rfc2629	46	46
Total	5321	4944
Percentage	100%	92.3%

References

[1] The OASIS cover pages: The online resource for markup language technologies. Available at http://www.oasis-open.org/cover/schemas.html, 2003.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[3] A. Aho, Y. Sagiv, and J. D. Ullman. Equivalence of relational expressions. *SIAM J. Computing*, 8(2):218–246, 1979.

[4] J. Albert. Algebraic properties of bag data types. In *Proc. 17th Int. Conf. on Very Large Data Bases*, pages 211–219, 1991.

[5] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *The VLDB Journal*, 11(4):315–331, 2002.

[6] T. Amoth, P. Cull, and P. Tadepalli. Exact learning of unordered tree patterns from queries. In *Proc. of the twelfth conference on Computational Learning Theory*, pages 323–332, 1999.

[7] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2):1–79, 2008.

[8] A. Berglund. Extensible Stylesheet Language (XSL), Version 1.1. Available at `http://www.w3.org/TR/xsl`, December 2006.

[9] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language (XPath) 2.0. Available at `http://www.w3.org/TR/xpath20`, January 2007.

[10] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *Proc. 32th Int. Conf. on Very Large Data Bases*, pages 115–126, 2006.

[11] G. J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML schema: A practical study. In *Proc. Seventh Int. Workshop on the Web and Databases*, pages 79–84, 2004.

[12] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. Available at http://www.w3.org/TR/xmlschema-2/, October 2004.

[13] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. Available at `http://www.w3.org/TR/xquery`, January 2007.

[14] A. Bonifati, S. Ceri, and S. Paraboschi. Active rules for XML; a new paradigm for e-services. *The VLDB Journal*, 10(1):39–47, 2001.

[15] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Y. eds. Extensible markup language (XML) 1.0 (fifth edition). Available at `http://www.w3.org/TR/2008/REC-xml-20081126/`, November 2008.

[16] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2002-0, Hong-Kong University of Science and Technology, April 2001.

[17] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.

[18] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *Proc. 10th Int. Conf. on the World Wide Web*, pages 201–210, 2001.

[19] D. Calvanese, G. De Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *Proc. Seventeenth ACM Symp. on Principles of Databases Systems*, pages 149–158, 1998.

[20] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query answering and query containment over semi-structured data. In *Proc. 8th Int. Workshop on Database Programming Languages*, pages 40–61, 2001.

[21] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. of the 9th ACM symposium on Theory of Computing*, pages 77–90, 1977.

[22] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. 11th Int. Conf. on Data Engineering*, pages 190–200, 1995.

[23] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *Proc. 12th ACM Symp. on Principles of Databases Systems*, pages 59–70, 1993.

[24] D. Chen and C. Y. Chan. Minimisation of tree pattern queries with constraints. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 609–622, 2008.

[25] Z. Chen, H. Jagadish, L. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: on efficient evaluation of XQuery. In *29th International Conference on Very Large Data Bases*, pages 237–248, 2003.

[26] B. Choi. What are real DTDs like? In *Proc. Fifth Int. Workshop on the Web and Databases*, pages 43–48, 2002.

[27] J. Clark. XML path language (XPath), version 1.0. Available at `http://www.w3.org/TR/xpath`, November 1999.

[28] J. Clark. XSL Transformations (XSLT), Version 1.0. Available at `http://www.w3.org/TR/xslt`, November 1999.

[29] J. Clark. TREX - Tree Regular Expressions for XML., 2001. Available at `http://www.thaiopensource.com/trex`.

[30] J. Clark and M. Makoto. Relax-NG Tutorial, March 2003. Available at `www.oasis-open.org/committees/relax-ng/tutorial.html`.

[31] B. Den Cate and C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. In *Proc. Twenty-sixth ACM Symp. on Principles of database systems*, pages 73–82, 2007.

[32] S. DeRose, R. Daniel, P. Grosso, E. Maler, J. Marsh, and N. Walsh. XML Pointer Language (XPointer), Version 1.0. Available at `http://www.w3.org/TR/xptr`, August 2002. W3C Working Draft.

[33] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath fragments. In *Proc. 8th Int. Workshop on Knowledge Representation Meets Databases*, 2001.

[34] A. Deutsch and V. Tannen. XML queries and constraints, containment and reformulation. *Theoretical Comput. Sci.*, 336(1):57–87, 2005.

[35] B. Ducharme. Filling in the DTD gaps with Schematron. Available at: http://www.xml.com/lpt/a/968, 2002.

[36] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer. Available at `http://www.w3.org/TR/xmlschema-0`, October 2004.

[37] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. *J. ACM*, 55(1):2, 2008.

[38] D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proc. Seventeenth ACM Symp. on Principles of Databases Systems*, pages 139–148. ACM Press, 1998.

[39] D. Florescu, L. Rashid, and P. Valduriez. Answering queries using OQL view expressions. In *Workshop on Materialised Views in conjunction with ACM SIGMOD*, pages 84–90, 1996.

[40] M. Friedman and D. Weld. Efficient execution of information gathering plans. In *Int. Joint Conf. on Artificial Intelligence*, pages 785–791, 1997.

[41] F. Geerts and W. Fan. Satisfiability of XPath queries with sibling axes. In *Proc. 10th Int. Workshop on Database Programming Languages*, pages 122–137, 2005.

[42] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. on Database Syst.*, 29(4):752–788, December 2004.

[43] J. Groppe and S. Groppe. Filtering unsatisfiable XPath queries. *Data and Knowledge Engineering*, 64(1):134–169, 2008.

[44] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *Proc. 13th ACM Symp. on Principles of Databases Systems*, pages 45–55, 1994.

[45] J. Hidders. Satisfiability of XPath expressions. In *Proc. 9th Int. Workshop on Database Programming Languages*, pages 21–36, 2003.

[46] Y. Ishihara, T. Morimoto, S. Shimizu, K. Hashimoto, and T. Fujiwara. A tractable subclass of DTDs for XPath satisfiability with sibling axes. In *Proc. 14th Int. Workshop on Database Programming Languages*, pages 68–83, 2009.

[47] H. Ishikawa and M. Ohata. An active web-based distributed database system for e-commerce. In *Proc. First Int. Workshop on Web Dynamics*, page 27, 2001.

[48] R. Jelliffe. The current state of the art of schema languages for XML. In *Presentation at XML Asia Pacific Conference*, 2001.

[49] R. Jelliffe. The Schematron assertion language 1.6, October 2002. Available at `http://xml.ascc.net/resource/schematron/Schematron2000.html`.

[50] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28(1):167–189, 1984.

[51] A. C. Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, 1988.

[52] L. Lakshmanan, G. Ramesh, H. Wang, and Z. Zhao. On testing satisfiability of tree pattern queries. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 120–131, 2004.

[53] L. Lakshmanan, H. Wang, and Z. Zhao. Answering tree pattern queries using views. In *Proc. 32th Int. Conf. on Very Large Data Bases*, pages 571–582, 2006.

[54] D. Lee and W. W. Chu. Comparative analysis of six XML schema languages. *SIGMOD RECORD*, 29(3):76–87, September 2000.

[55] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. Fourteenth ACM Symp. on Principles of Databases Systems*, pages 95–104, 1995.

[56] A. Y. Levy, A. Rajaraman, and J. Ordille. Query heterogeneous information sources using source descriptions. In *Proc. 22th Int. Conf. on Very Large Data Bases*, pages 251–262, 1996.

[57] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *Proc. 19th Int. Conf. on Very Large Data Bases*, pages 171–181, 1993.

[58] A. Y. Levy and D. Suciu. Deciding containment for queries with complex objects and aggregations. In *Proc. Sixteenth ACM Symp. on Principles of Databases Systems*, pages 20–31, 1997.

[59] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.

[60] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[61] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Trans. on Database Syst.*, 4(4):455–469, September 1979.

[62] M. Makoto. RELAX (REgular LAnguage description for XML), 2000. Available at `http://www.xml.gr.jp/relax`.

[63] W. Martens, F. Neven, and G. J. Bex. Expressiveness and complexity of XML schema. *ACM Trans. on Database Syst.*, 31(3):770–813, September 2006.

[64] M. Marx. XPath with conditional axis relations. In *Proc. 9th Int. Conf. on Extending Database Technology*, pages 477–494, 2004.

[65] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, January 2004.

[66] T. Milo and D. Suciu. Index structure for path expressions. In *Proc. 7th Int. Conf. on Database Theory*, pages 277–295, 1999.

[67] M. Montazerian and P. T. Wood. Chasing one's tail: XPath containment under cyclic DTDs. In *Proc. 15th Int. Workshop on Database Programming Languages*, 2011.

[68] M. Montazerian, P. T. Wood, and S. R. Mousavi. XPath query satisfiability is in PTIME for real-world DTDs. In *Proc. 5th Int. XML Database Symposium*, pages 17–30, 2007.

[69] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. on Internet Tech.*, 5(4):1–45, 2005.

[70] A. Nakhimovsky and T. Myers. *XML Programming: Web Applications and Web Services with JSP and ASP*. Apress, 2002.

[71] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *ACM Trans. on Database Syst.*, 31(3):770–813, 2006.

[72] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, and E. Kogan. XQuery implementation in a relational database system. In *Proc. 31th Int. Conf. on Very Large Data Bases*, pages 1175–1186, 2005.

[73] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. Nineteenth ACM Symp. on Principles of Databases Systems*, pages 35–46, 2000.

[74] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 299–309, 2002.

[75] E. Robertsson. Combining Schematron with other XML schema languages, 2002. Available at `http://www.topologi.com/public/Schtrn XSD/Paper.html`.

[76] K. H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 2007.

[77] Y. Sagiv and M. Yannakakis. Equivalence among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1981.

[78] T. Schwentick. XPath query containment. *SIGMOD RECORD*, 33(1):101–109, 2004.

[79] B. Simon. Smart space for learning; a mediation infrastructure for learning services. In *Proceedings of the Twelfth International Conference on World Wide Web*, pages 20–24, 2003.

[80] L. J. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proc. of the 5th ACM symposium on Theory of Computing*, pages 1–9, 1973.

[81] N. Suzuki and Y. Fukushima. Satisfiability of simple XPath fragments in the presence of DTDs. In *Proceedings of the Eleventh International Workshop on Web Information and Data Management*, pages 15–22, 2009.

[82] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Definition Language (XSD)- Part I: Structures, June 2008. Available at `http://www.w3.org/TR/xmlschema11-1`.

[83] E. van der Vlist. ISO DSDL overview. Available at `http://www.idealliance.org/papers/dx-xml03/papers /04-03-03 /04-03-03.html`, 2003.

[84] P. Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies*, pages 183–202, 1999.

[85] N. Walsh and J. Cowan. Schema language comparison. Available at http://nwalsh.com/xml2001/schematownhall/slides/, December 2001.

[86] J. Wang and X. Yu. Chasing tree pattern under recursive DTDs. In *Database Systems for Advanced Applications*, pages 250–261, 2010.

[87] F. Wei and G. Lausen. Conjunctive query containment in the presence of disjunctive integrity constraints. In *Computer Science in Perspective*, pages 231–244, 2003.

[88] P. T. Wood. Optimizing web queries using document type definitions. In *Proc. 2nd Int. Workshop on Web Information and Data Management*, pages 28–32, 1999.

[89] P. T. Wood. On the equivalence of XML patterns. In *Proc. 1st Int. Conf. on Computational Logic*, pages 1152–1166, 2000.

[90] P. T. Wood. Rewriting XQL queries on XML repositories. In *Proc. 17th British National Conf. on Databases*, pages 209–226, 2000.

[91] P. T. Wood. Minimising simple XPath expressions. In *Proc. Fourth Int. Workshop on the Web and Databases*, pages 13–18, 2001.

[92] P. T. Wood. Containment for XPath fragments under DTD constraints. In *Proc. 9th Int. Conf. on Database Theory*, pages 300–314, 2003.

[93] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th Int. Conf. on Very Large Data Bases*, pages 82–94, 1981.

[94] X. Zhang and Z. M. Ozsoyoglu. Implication and referential constraints: A new formal reasoning. *IEEE Trans. on Knowledge and Data Eng.*, 9(6):894–910, 1997.