Data Definition Languages for XML Repository Management Systems

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r N a t u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät Informatik

von

Matthias Niewerth

Dortmund

2015

Tag der mündlichen Prüfung:	27. März 2015
Dekan:	Gernot A. Fink
Gutachter:	Thomas Schwentick Angela Bonifati

Preface

Already when I got my first computer at the age of six, I was fascinated by the possibilities offered by an 8-bit processor using 65536 bytes of random access memory. It soon became very clear that I wanted to do "something with computers". This is a decision that I hardly ever questioned and that I never regretted. This thesis is the climax of a way that I started very young. It is my dearest wish to thank all persons who have accompanied me on this way. As this is a big number of people, I have to apologize to those who should be mentioned, but are not.

First, I thank Thomas Schwentick for being a great advisor. He introduced me to the habits of scientific research and to the field of database theory. I remember countless occasions, where we discussed the details of some proof, how to present some work on a conference, or what is the best way to represent constraints for XML documents, but also sometimes controversies about priorities of work or how to document my progress. In retrospect, the creation of this thesis was a wonderful time and Thomas Schwentick has contributed a lot to this positive feelings.

I am very thankful to Wim Martens, who always was helpful when I had questions and who has given me a new professional home in Bayreuth. I joyfully remember many hours we shared discussing scientific topics, the newest academic gossip, or just the news of the day.

I thank Angela Bonifati for providing a secondary evaluation of my thesis and Jens Teubner and Boris Düdder for taking part in my dissertation commission.

Of course, I want to thank all my colleagues for the wonderful time. This includes my colleagues, both in Dortmund and in Bayreuth, but also numerous persons from other universities, which I met at many different places during workshops, conferences and summer schools. In order to not doing injustice to someone, I refrain from giving a list.

Also, the colleagues and professors who guided me through my undergraduate studies are too countable to list them all. However, one name has to be mentioned here: Ingo Wegener was an outstanding teacher and scientist. I myself enjoyed several courses given by him. What is most remarkable to me is, that he, shortly before he lost his fight against cancer, agreed to supervise my diploma thesis. Sadly, he was not given the time to finish this process.

I also have to mention my school time, where I had a few problems to integrate within rules that I could not understand at this time. In retrospect, of course, I see that most of the rules were there for good reasons. I am very thankful to my school directors Bernhard Sporkmann and Wolfgang Gorniak, who allowed me to bend some of these rules, such that I could follow my passion together with some colleagues. I will never forget the long afternoons, where we optimized the school network, enriched our knowledge of Linux, or sometimes just played using a network, which was much bigger than what would have

been possible at home. The freedom I had there is definitely not what is usual. Norbert Stäbler should be mentioned for very motivating and dedicated computer science classes and for supporting our extra-curricular activities, wherever possible. While I already decided a long time ago, that computer science is my preferred field, the classes given by him were a good preparation and strengthened my decision to study computer science.

Finally, my warmest thanks go to my family. To my uncle who stired my passion for computers and to my parents who supported me on the whole way. My uncle rised my interest by giving me the 8-bit computer, I mentioned in the very beginning. We had many interesting discussions since then, starting technically when I was young and getting more theoretical and philosophical during my studies. I cannot say enough thanks to my parents who supported me on the whole way and who enabled me to start my academic career. There are no words to express what they mean to me.

One last thing: Often forgotten, there are the institutions who enable young scientists to start an academic career by providing the necessary finances and without whom many careers would not be possible. In my case, this is the Future and Emerging Technologies (FET) programme of the European Commission, which supported me during my time in Dortmund under the FET-Open grant agreement FOX, number FP7-ICT-233599 and the Deutsche Forschungsgemeinschaft (DFG), which supported me during my time in Bayreuth under the grant MA 4938/21 (Emmy Noether Nachwuchsgruppe).

Contents

Introduction

10

1 Introduction

XML is nowadays the de facto standard for data exchange on the web. However, while XML is widely used for data exchange, its role in data storage is quite small. As a consequence, data has to be converted between XML and other formats, especially relational databases, many times.

Many of these conversion steps could be avoided if the data model of the database systems would be XML. In this thesis we will study certain aspects of database systems dedicated to manage big amounts of XML data. We call such systems *XML Repository Management Systems* (XRMS).

Of course, we will not be able to cover all aspects of XRMS's, as the topic is very broad and touches almost every part of database theory of the last four decades. Instead, we will concentrate on data description languages for XRMS's. Data description languages are a very important part of the interface between a database administrator or programmer and a database management system.

We will distinguish two very different aspects of data description languages. The first aspect is the ability to describe structure of the data. In classical relational database management systems this basically boils down to describing which attributes each relation has and which domain each of the attributes uses. In XRMS's, the structure of the data can be much more complicated, because the underlying data model is a tree.

The second aspect of a data description language is its ability to express semantic constraints of the data. The most prominent example is that IDs should be unique.

The intuitive difference between both aspects is that semantic constraints look at the data values itself and can compare them, usually for equality, while structural definitions do not compare data values.

We give examples on both aspects after we have introduced our running example. First, we want to highlight the questions, we are going to analyze.

1.1 Questions

As already said, we are going to analyze data definition languages for XRMS's. In particular, we will investigate the following questions, that arise in the context of XRMS's.

- How can the structure of XML databases and documents be described?
- What are the properties of regular expressions, as they are used in existing XML schema languages?

- How to design schemas in the case that XML databases are distributed?
- How can semantic constraints on XML databases be described?

Towards the first question, we will propose the BonXai schema language that allows to describe the structure of documents based on simple rules instead of a complex type system. We will compare BonXai with existing XML schema languages and analyze the conversion algorithms between XML Schema and BonXai.

We will cover the second question only very briefly, as the main work in this area has been done by Katja Losemann.

The third question covers a very broad topic. We will concentrate on one concrete question: Given a schema S and a root document D that references documents $D_1, \ldots, D_n$, how should good schemas for the referenced documents look like, such that the combined document obtained by inserting the child documents in the root document satisfies the original schema.

The investigation of the last question almost covers half of the thesis. We will investigate two quite different approaches. The first approach is by designing a general framework for studying XML integrity constraints. Based on this framework, we will analyze how well existing constraint definition languages fit into the framework. Furthermore, we will investigate the complexity of the implication problem for XML integrity constraints for constraints that are the XML analog to relational functional dependencies and relational key constraints.

The second approach is to add key constraints to two variable first order logic. Two variable first order logic has already some applications in XML theory.

In the next section, we present the running example, that we use to investigate the questions.

1.2 Running Example: A Content Management System

We start with a simple example, which will be used as running example throughout the thesis. Instead of writing HTML documents by hand, nowadays many web publishers use content management systems (CMS's) to manage their web data.

Usually these systems deliver XML data[1] to the viewers of the web pages. However, most of these systems store the data inside a relational database, i.e. as blobs of text. It seems to be a natural choice to store this data using an XRMS, once such systems become ready for production use.

In Figure 1.1 we depicted parts of a simple XML database of such a content management system. The depicted database has two parts, one part stores the content or documents with some meta-information and the other part contains the user data.

The user part stores user IDs together with associated persons (only first and last name for simplicity reasons) and the documents part stores documents together with the user ID, which created the document. Of course, in a real system we would also

[1] XHTML is one out of many document standards based on XML.

```
<root>
  <documents>
    <document>
      <owner>user42</owner>
      <content>
        <!-- Here should be the content of the first document. -->
      </content>
    </document>
    <document>
      <owner>user23</owner>
      <content>
        <!-- Here should be the content of the second document. -->
      </content>
    </document>
  </documents>
  <persons>
    <person>
      <firstname>Joe</firstname>
      <lastname>Smith</lastname>
      <user-id>user42</user-id>
    </person>
    <person>
      <firstname>Ann</firstname>
      <lastname>Brown</lastname>
      <user-id>user23</user-id>
    </person>
  </persons>
</root>
```

Figure 1.1: Running Example: Database of a Content Management System (CMS)

need to store password hashes, access rights, and much more, but we feel, for the sake of this thesis, the example should be as simple as possible while still capturing some of the challenges, XRMS's have to deal with.

Since we will always interpret XML documents as trees for the ease of algorithmic analysis, we additionally depicted a tree view of the XML database in Figure 1.2. For a formal definition of our tree model the reader is directed to Chapter 3.

We will give English language descriptions of the database and its constraints. Of course, these descriptions are neither complete nor suited for any actual database system,

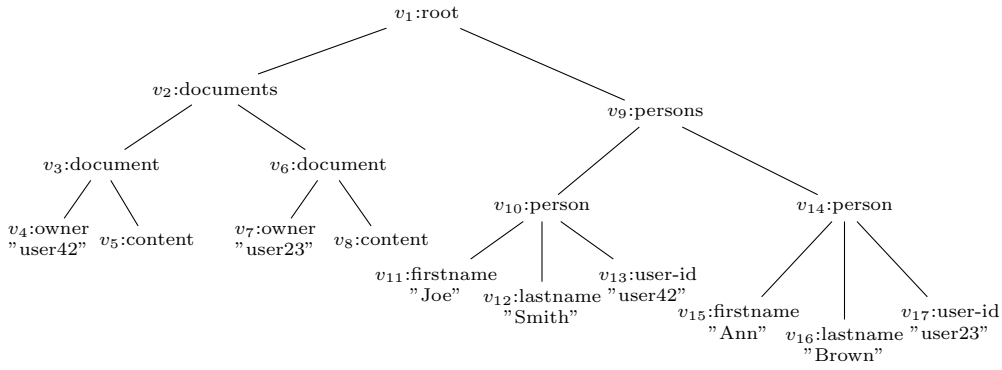


Figure 1.2: CMS Database from Figure 1.1 as a tree

but during the thesis we will see how these descriptions can be formulated using formal languages.

a) The root element of the XML document is labeled `root` and has two child elements labeled persons and documents.

b) The `persons` element has arbitrary many child elements all labeled `person`.

c) Each `person` element has two child elements labeled `firstname` and `lastname` respectively and arbitrarily many child elements labeled `user-id`

d) Elements labeled `firstname`, `lastname` or `user-id` do not have child elements.

e) Elements labeled `firstname` or `lastname` have a string as data value which only consists of letters.

f) Elements labeled `user-id` have a string as data value which consists of letters and digits and starts with a letter.

g) User IDs should be unique, i.e. there should be no two persons which use the same user ID.

h) The user IDs, which own some documents should exist, i.e. for every user ID which occurs as the owner of a document, there has to be a person associated with this ID.

It can be observed that the given constraints are of different nature. The constraints (a–d) only talk about the structure of the XML database (the XML tree) and do not refer to the data stored inside the elements (nodes). We will refer to these as *syntactic* constraints.

The constraints (e–f) define the domain used for data values of different nodes. While these constraints restrict the possible data values, we still call them syntactic. As we

will represent all domains by anonymous uncountable sets throughout the thesis, we will concentrate on those syntactic constraints, which describe the structure of the tree.

Finally, there are the constraints (g–h), that compare data values. The constraint g) compares all data values of `user-id` nodes to rule out any identical data values among these nodes, while the constraint h) expresses that data values occurring in some part of the database also have to occur at another place. We call these types of constraints *semantic* constraints or *integrity* constraints. The constraints g) and h) represent the most widely used integrity constraints, namely key constraints and inclusion constraints. Of course, one might think of different semantic constraints such as a constraint enforcing a chronological order on timestamps of the stored documents. However these constraints are out of the scope of this thesis.

1.3 Structure of the Thesis

In Chapter 2, we depict the components of database management systems and comment on the differences between relational database management systems and XML repository management systems. We use this description to clarify which aspects of XRMS's are covered by this thesis. Afterwards, Chapter 3 establishes some definitions and notations used throughout the thesis.

The thesis is divided into three parts, not counting introduction and conclusion. In the first and second part, we will look at syntactic descriptions and semantic constraints of XML documents, respectively, while in the third part, we will briefly describe a software library, which contains many prototypical algorithms for dealing with XML schemas and documents.

As already mentioned, the first part of this thesis covers syntactical definitions, that is describing constraints like the constraints (a–f) from our running example. After introducing existing XML schema languages in Chapter 4, we introduce the pattern-based schema language BonXai in Chapter 5 and compare it with traditional schema languages, especially XML Schema and Document Type Definitions. In Chapter 6 we provide a solid theoretical background for the BonXai schema language. Many schema languages, including Document Type Definitions, XML Schema and BonXai, use deterministic regular expressions to describe content models of nodes. We will give a short overview over this class of regular expressions in Chapter 7. We close the part about syntactic definitions in Chapter 8 with some results on distributed XML design. We research how a syntactic definition for a global document can be split into several syntactic definitions for local documents such that recombining local documents (according to a given rule) generates a document, which is valid according to the global definitions.

In the second part, we continue with semantic constraints on XML databases, i.e. how to describe constraints similar to the constraints (g) and (h) of our running example. We start with repeating the basics of relational integrity constraints and sketching some challenges for the design of XML integrity constraint languages in Chapter 9. In Chapter 10, we will depict a new framework for XML integrity constraints. Furthermore, we will show how constraints depicted in existing constraint languages can be represented

using this framework. The following Chapter 11 is entirely destined for analyzing the complexity of the implication problem of XML integrity constraints. We will present complexity results for some instantiations of the framework presented in Chapter 10. At the end of the second part, we present some older work in Chapter 12, which analyzes the combination of key constraints of arbitrary arity with first order sentences using only two variables.

The third part contains only one chapter presenting the software library FoXLib. This library contains many prototypical algorithms for analyzing XML Schemas and documents and developing XML Schemas. Especially it contains algorithms to convert XML Schema definitions into BonXai schemas and vice versa.

1.4 Contributions by other Authors

Most of the results presented in this thesis are joined work with other authors.

The work on the BonXai schema language presented in Chapters 4 to 6 is based on joined work with Wim Martens, Frank Neven and Thomas Schwentick. Parts of the content of Chapter 5 were presented as a demo at the 38th International Conference on Very Large Databases (VLDB 2012) [MNNS12]. The initial draft of BonXai (presented in Chapter 5) was designed by a student group at TU Dortmund University [DGG+09]. The main work on the prototype implementation presented a VLDB 2012 was done by the author of this thesis. The remaining content of Chapters 4 to 6 is not yet published.

Chapter 7 mostly summarizes literature work for completeness and reference reasons. The results on closure properties and descriptional complexity of DREs are joined work with Katja Losemann and Wim Martens [LMN12]. Katja Losemann is the main author of this work. The hardness result for DRE minimization is the work of this author. It has not been published before.

The work about schema decompositions presented in Chapter 8 has been presented at the 29th Symposium on Principles of Database Systems (PODS 2010) [MNS10]. The author of this thesis is the main author of this work with contributions from Wim Martens and Thomas Schwentick.

The work about integrity constraints presented in Part II is based on joined work with Thomas Schwentick. A first version of the results on first order logic with two variables in Chapter 12 has been presented at the 14th International Conference on Database Theory (ICDT 2011) [NS11]. The revised version presented in Chapter 12 is by the author of the thesis. It uses a clearer model for representing the constraints imposed by $\mathrm{FO}^2(\sim, +1)$-formulas and provides a better upper bound in the case without key constraints. The underlying concepts are the same as in [NS11].

The work on XML2Relational constraints from Chapters 10 and 11 has been presented at the 17th International Conference on Database Theory (ICDT 2014) [NS14]. A journal paper of this work has been submitted to the Theory of Computing Systems journal [NS15]. Unfortunately, there was a big flaw in one of the proofs of the conference version. Therefore Section 11.4 has been rewritten by the author of this thesis. As the new version did not undergo a review process before completion of the thesis, it was of

significantly lower quality then the remainder of the thesis. Therefore, the published version of the thesis contains a version of the proof extracted from the submitted journal version [NS15] that has been created together with Thomas Schwentick based on the work of the author.

The software library FoXLib presented in Chapter 13 contains contributions from different sources. Details are given in Section 13.4.

2 Database Management Systems

Even if this thesis will only cover a small fraction of an XML Repository Management System, we want to use this chapter to take a look at the complete system. Our understanding of an XML Repository Management System is a Database Management System (DBMS), where the underlying data format is XML data.

To understand what an XRMS is, it is obviously necessary to understand what a DBMS is. Therefore we recapitulate the features and components of a DBMS in this chapter.

We will give a general overview over Database Management Systems according to Garcia-Molina, Ullman and Widom [GMUW02] and Ramakrishnan and Gehrke [RG03].

According to Garcia-Molina, Ullman and Widom [GMUW02],

> *a DBMS is a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time, safely.*

A DBMS is expected to

1. *Allow users to create new databases and specify their schema (logical structure of data), using a specialized language called a data-definition language.*
2. *Give users the ability to query [...] and modify the data, using an appropriate language, often called a query language or data-manipulation language.*
3. *Support the storage of very large amounts of data [...] over a long period of time, keeping it secure from accident or unauthorized use and allowing efficient access to the data for queries and database modifications.*
4. *Control access to data from many users at one, without allowing the actions of one user to affect other users and without allowing simultaneous accesses to corrupt the data accidentally.*

Figure 2.1 lists the components of a DBMS. The figure is based on Figure 1.3 in [RG03]. In the following we will investigate the individual components depicted in Figure 2.1. We will explain their role mostly on the basis of usual relational database systems based on the SQL standard. Furthermore we will explain some differences between existing implementations based on the SQL standard and an imaginary XML Repository Management System. Our considerations are general enough so that we do not need to distinguish different flavors (implementations) of the SQL standard.

Quite obviously most of the necessary changes apply to the high-level components of the DBMS while the low-level components are more independent from the structure of the data (relational vs. XML trees).

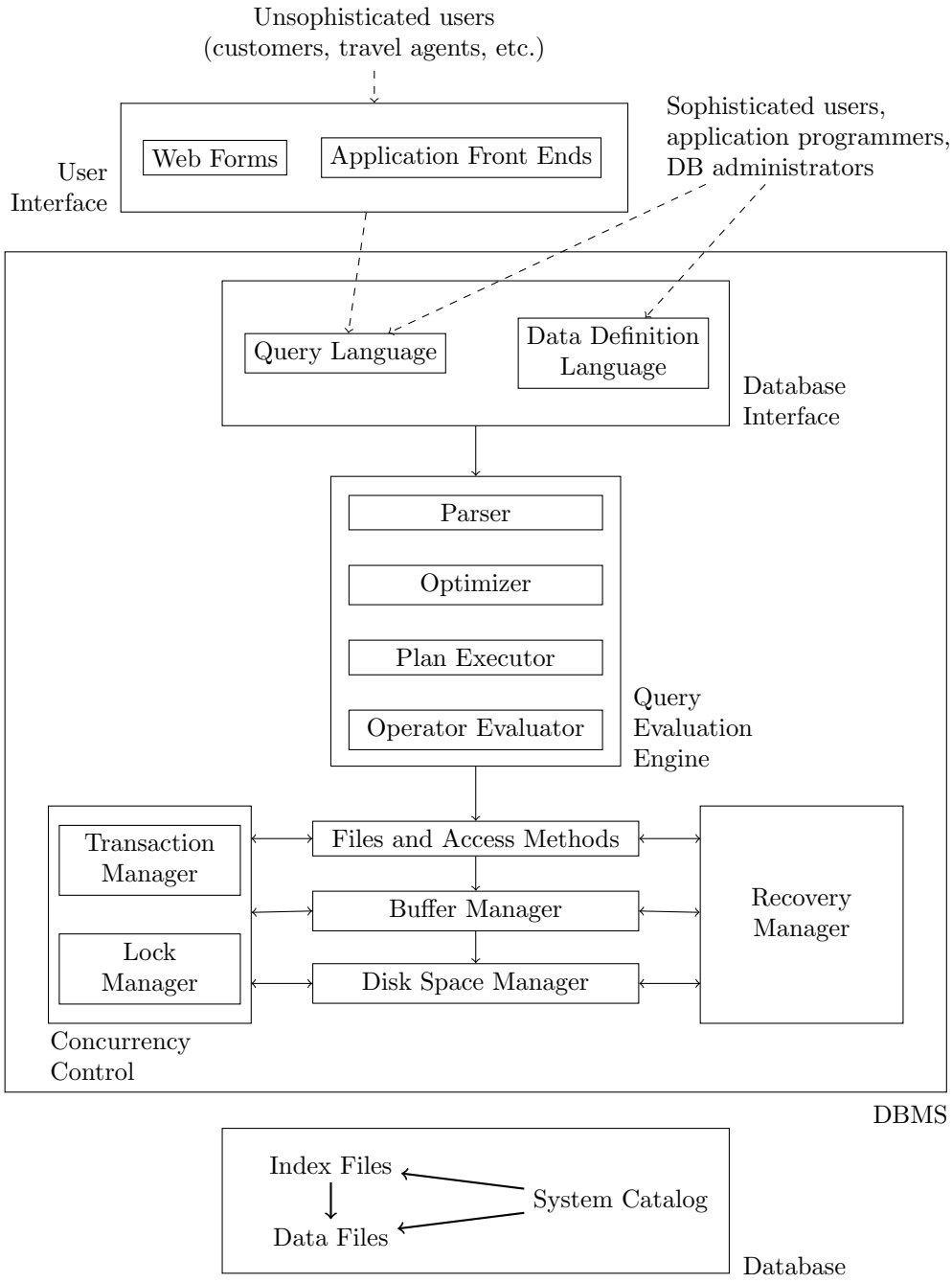


Figure 2.1: Architecture of a DBMS (Based on Figure 1.3 in [RG03])

2.1 Database Interface

For a user, the first visible part of a database management system is its interface. The database interface consists of a query language, which is used to query and update the data and a data definition language, which describes the structure of the database.

Ordinary users usually only interfere with the query language, usually through an application front end or web service. They are interested in retrieving data, adding new data or editing existing data through database queries. Application programmers and database administrators additionally use the data description language to specify the structure of the data before the database can actually be used to store data.

Most parts of this thesis deal directly or indirectly with the definition of the database interface for XML Repository Management Systems. More precisely we have a look at the data definition language for XML databases. The first part of this thesis deals with syntactical definitions of XML databases and the second part deals with semantic constraints on XML databases. Both aspects are part of the data definition language. In Part III, we present a prototype software library. One of the main features of this software library is a conversion routine between two different data definition languages for XML documents.

Query Languages

Query languages are used to retrieve and update the data. We will distinguish two kinds of queries. Retrieval queries do not change the database and just return the queried data. In SQL, these queries start with the keyword `SELECT`. On the other hand, update queries usually do not return any data. Instead, they change the database. In SQL, these queries start with the keyword `UPDATE` or `INSERT`.

For relational databases both types of queries are important, while many applications dealing with XML data do not support update queries at all. This is because in many cases XML is only used to exchange and transfer data but not to actually store data. In these cases the data is typically stored in a relational database management system. Therefore, it is never necessary to update XML data itself. Updates are processed by the underlying relational database and the (changed) XML document is recreated from this database whenever it is needed. For an XRMS, update queries are much more important as the data is stored in XML databases and updates have to be directly applied to the XML database.

A big difference between relational and XML query languages is, that XML query languages are usually recursive, e.g., it is possible to state queries like "Give all a-nodes that are below some b-node.". This query implicitly needs the transitive closure of the edge-relation of the underlying tree. General query languages for XML have to be recursive, as the trees can be arbitrarily deep.

The most prominent query languages for XML are XPath and XQuery. The XPath query language allows to query for a set of nodes, while the XQuery language can return XML documents, constructed by the query. XQuery is build on top of XPath.

Diego Figueira has done a lot of research on XPath satisfiability. The results are nicely presented in his PHD thesis [Fig10].

Although query languages are very important for XML processing in general and XRMS's in particular, they are out of scope of this thesis, which focuses on syntactical description and semantic constraints of data. In Chapters 10 and 11 we use tree patterns as query language to define semantic constraints for XML databases.

Data Definition Languages

Data definition languages describe how data should look like. In the SQL standard the `CREATE TABLE` command defines the structure of a relation.

To create a relational database which are capable of holding the information given in the XML database in Figure 1.1, we can use the following SQL commands.

```
CREATE TABLE users (user-id CHAR(20), name CHAR(50));
CREATE TABLE documents (owner CHAR(20), content TEXT);
```

These commands create two tables/relations with the names users and documents. The users table contains tuples where the first entry is a string of at most 50 characters and the second entry is a string of at most 20 characters. The documents table contains tuples where the first entry is a string of up to 20 characters and the second entry is some text of arbitrary length. Note that table definitions in SQL can be much more complicated than in this simple example.

While the structure of SQL based databases is defined by a series of `CREATE TABLE` commands, the structure of XML documents is usually defined by a schema. There is a wide variety of existing schema description languages. The most important ones are Document Type Definitions (DTDs) [BPSM$^+$08] and XML Schema [GSMT$^+$12]. These two schema languages are standardized by the W3C. Other proposed schema languages include Relax NG [CM01], Schematron [Sch99] and Document Structure Descriptions [DSD02]. We will see some example schemas for our CMS example in Chapter 4.

In theoretic work, schemas are usually described by regular tree languages, which can for example be represented by various classes of tree automata or by monadic second order logic over trees. An important difference to the afore mentioned schema languages is that these models usually use a more abstract tree model that does not deal with all subtleties of the XML standard.

In Chapter 5, we will introduce BonXai as a pattern based schema language. We will compare BonXai to DTDs and XML Schema and have a look at the problems of converting XML Schema descriptions to BonXai schemas and vice versa.

Integrity Constraints

Integrity constraints describe semantic constraints on the data. They are part of the data definition language.

We can formulate the two example constraints from the introduction using the following SQL commands:

```
CREATE TABLE users (user-id CHAR(20) PRIMARY KEY, name CHAR(50));
CREATE TABLE docs (doc-id INT, owner CHAR(20) REFERENCES users
(user-id));
```

These commands necessarily include the definitions of the tables, as SQL integrity constraints are part of the syntactic definitions. The addition of `PRIMARY KEY` behind the declaration of `user-id` specifies that there should not be two different tuples with the same user ID, while the specification `REFERENCES users (user-id)` behind the declaration of `owner` specifies that every entry in the column `owner` should have a corresponding entry in the column `user-id` of the `users` table.

For XML, many approaches for integrity constraints have been proposed. On the practical side there is the minimalistic ID/IDREF mechanism of DTDs and there are the much more powerful identity constraint definitions from the XML Schema standard. On the theoretical side there are many different approaches showing that there is no clear consensus yet on the definition of XML integrity constraints, see e.g. [AL04, HL03, KW07, LLL02].

We will have a closer look on XML integrity constraints in the second part of this thesis. Where we give a unifying framework for some of these approaches.

The following section about query evaluation explains why it is desirable to solve the implication problem of integrity constraints with low complexity.

2.2 Query Evaluation Engine

The query evaluation engine is responsible for parsing and evaluation of queries. It consists of a parser which translates the query into an initial query plan. Afterwards, the optimizer can modify the query plan for shorter execution times. For complex queries there might be many different equivalent query plans with big differences in execution times. The plan executor is responsible for executing a given query plan by calling the operator evaluator for all operations inside the query plan. The operator evaluator contains all the necessary algorithms for performing database operations occurring in query plans like performing a join or applying some projection or selection.

The most sophisticated part of the query evaluation engine is the optimizer. We give one example on how the optimizer can optimize a query. The following SQL query returns all persons owning some document.

```
SELECT name FROM users,docs WHERE user-id=owner;
```

An initial query plan for this query might look as follows:

1. Compute the Cartesian product of users and docs:
   $\texttt{tmp} \leftarrow \texttt{users} \times \texttt{docs}$
2. Select all tuples where user-id equals owner:
   $\texttt{tmp2} \leftarrow \{(w,x,y,z) \mid (w,x,y,z) \in \texttt{tmp} \wedge w = z\}$

3. Project to the column name:
   $\texttt{output} \leftarrow \{(x) \mid \exists w, y, z.\ (w, x, y, z) \in \texttt{tmp2}\}$

Note that we use temporary tables for clarity while a real database system would not employ intermediate tables but instead execute all steps at the same time directly using the output from one step as the input to the next step. Temporary tables are usually only used if the applied operations do not allow otherwise.

It is easy to see that this query plan can have quadratic running time in the size of the database due to the computation of the Cartesian product. Much time can already be saved by projecting the docs relation to the owner column before computing the Cartesian product. Note that the projection will not only remove the column with the document ids but will additionally keep only one entry for every user owning some documents instead of one entry per document.

However the query can be optimized even more. It is a waste of computation time to compute the complete Cartesian product if we only need those tuples where the `user-id` and `owner` attributes coincide. The following query plan will also return the correct result.

1. Project the docs relation to the owner column:
   $\texttt{tmp} \leftarrow \{(y) \mid \exists x.\ (x, y) \in \texttt{docs}\}$

2. For each owner, scan the users relation for tuples with matching entries in the user-id column and output the corresponding name:
   for each $x \in \texttt{tmp}$ **do** $\texttt{output} \leftarrow \texttt{output} \cup \{(y) \mid (x, y) \in \texttt{users}\}$

Optimization can go even further, taking integrity constraints into account. If `user-id` is a key, the database engine knows that every user ID can occur only once and can stop scanning the `users` relation in step 2 of the query plan after one match is found.

If there is an index for the `user-id` column available (which is reasonable to assume if `user-id` is the primary key of the relation), the table scan can be replaced by an index look-up, which usually needs constant to logarithmic time depending on whether the index is a hash table or some tree based data structure.

Altogether the expected execution time of the query can be reduced from quadratic to linear given an appropriate database structure with indexes. As databases can be very huge it is reasonable to take big efforts in optimizing a query before its execution.

Query optimization on itself is out of the scope of this thesis. Nevertheless, there has been research on semantic query optimization, i.e., optimizing queries by taking semantic information like integrity constraints into account [Kin81, CGM88]. Newer research even looks on semantic query optimization in the context of XML [SRM05]. An important subproblem in semantic query optimization is the inference of integrity constraints that hold on the database from the constraints explicitly given by the creator of the database. In the second part of the thesis, we will have a look on integrity constraints for XML. In Chapter 11 we will especially analyze the complexity of the implication problem of integrity constraints for XML.

2.3 Low-Level Features

We will just shortly discuss the lower layers of a database management system. These components are out of the scope of this thesis and are just discussed for completeness. The information given is summarized from [GMUW02] and [RG03]. We note, that data gets more and more distributed. We mainly discuss these lower layers to give an idea of the difference between the relational and the XML data model. For an XRMS that deals with distributed data, additionally some new layer has to be introduced that distributes queries across several machines.

Files and Access Methods

The content of the database is stored in small portions called pages. These pages are stored on secondary memory. The files and access methods take care about which page holds what piece of information. Furthermore, they manage the inner structure of pages.

The pages do not only contain the data itself, but also contain indexes, which are necessary to efficiently find the requested information.

The files and access methods layer probably needs to be adapted for XML Repository Management Systems, as trees have different storage requirements than relations. Even if every tree could be stored in a relational database by storing the edge relation, the data relation (i.e. which node has which data value) and the label relation (i.e. which node has which label), this is certainly not the most efficient way to store or query the data. Querying data using these relations would require a number of join operations that is linear in the depth of the tree.

Buffer Manager

The buffer manager is responsible for transferring pages between main memory and secondary memory. This does not only involve copying pages to main memory which are requested by the upper layers and writing modified pages back to secondary storage. For better performance the buffer manager should predict, which pages are needed next and prefetch them to the main memory. On the other side the buffer manager need to choose which pages to remove from main memory when space needs to be freed.

While the basic techniques of the buffer manager will not differ between relational and XML databases, it might be necessary to adapt the prefetching and replacing strategies to different access patterns of an XML Repository Management System.

Disk Space Manager

The disk space manager is responsible for allocating space in secondary memory. This is necessary when the database grows or to store log files (see recovery).

The tasks of the disk space manager can be entirely performed by the underlying operating system. In the case when the relations, indexes and log files are stored as separate files in some filesystem, the operating system will take care of allocating the necessary space when these files grow.

However, for big databases it is prudent to replace the generic algorithms of the operating system by specialized algorithms of the DBMS, which can predict the growth of the separate files. In this case the disk space manager has to manage the space on behalf of the DBMS.

Concurrency

When several persons or programs are accessing the database at the same time they should always see a consistent state of the database. This also holds in case one or more concurrent accesses do modify the database.

Modifications to the database should be atomic, i.e. it should not be possible to view an intermediate state of the database. This is ensured by transactions, whereby a transaction can consist of one or several update queries. Even if one transaction does modify several relations, all concurrent queries should either see the state before or after these modifications. In no case they should see some intermediate state. This can be accomplished by either locking the parts of the database that are to be modified or by creating a copy of these parts. In the first case, all concurrent queries which need access to the locked parts have to wait until the transaction is complete. In the second case, all concurrent queries will only see the copy of the data and therefore the old state of the database. The transaction and lock managers are responsible for providing a consistent view of the database for all queries.

Recovery

The state of the database should also be consistent after a system crash or power failure. Therefore the recovery manager will log all modifications to the database to some separate space. After a crash, the recovery manager will compare the logs with the state of the database and ensure that for each transaction it holds that it is either completely applied to the database or that all modifications that have already been done to the database are reverted to restore the state before the start of the transaction.

2.4 Distributed Data

Distribution of data is not mentioned in Figure 2.1, as database management systems have mostly been studied and designed for standalone use. However, information has become more and more distributed since the arrival of the Web. The distribution of XML data is even essential in many areas such as e-commerce, collaborative editing, or network directories [AGM09]. When dealing with such distributed XML data, it is desirable to have a system that can grant a large amount of independence to individual peers, while at the same time also being able to deal with the data as a whole. These demands create new challenges for DBMS in general and for XRMS in particular. In Chapter 8, we analyze how a good schema design for such distributed XML databases can be achieved.

3 Preliminaries and Notation

In this chapter, we introduce the necessary notation to allow a formal exploration of the topic starting with the definition of regular expressions and finite automata in Section 3.1. Most importantly, we will introduce our tree model in Section 3.2. Afterwards, we will give some formal definition of tree languages, which are mainly used in the second part of the thesis. At last we will introduce tiling problems, which have no direct connection to XML databases. However we will use them for several reduction proofs throughout the thesis. At the end of this chapter we give an (incomplete) list of our used notation.

Table 3.1 at the end of this chapter gives an overview over commonly used symbols and the (type of) objects that are referenced by this symbols.

3.1 Regular Languages

A *language* is a (possibly infinite) set of strings over a finite alphabet Σ.

For any language L, we define

- $\text{first}(L) = \{a \in \Sigma \mid \exists w \in \Sigma^*.\ aw \in L\}$ to be the set of first symbols of L,
- $\text{followlast}(L) = \{a \in \Sigma \mid \exists v \in L, w \in \Sigma^*.\ vaw \in L\}$ to be the set of symbols that can follow after a string of L

Finite Automata

A *(nondeterministic, finite) automaton* (or *NFA*) $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, I, F)$, where

- Q is a set of states;
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function;
- I is the set of initial states; and
- F is the set of accepting states.

By δ^* we denote the extension of δ to strings, i.e., $\delta^*(q, w)$ is the set of states that can be reached from q by reading w. If an automaton has a single initial state we usually denote this state by q_0. We define the size of an automaton to be the number of its states.

An automaton is deterministic (or a *DFA*), if $I = \{q_0\}$ is a singleton and $|\delta(q, a)| \leq 1$, for all $q \in Q, a \in \Sigma$.

For simplicity we allow in DFA that for some q, a, $\delta(q, a) = \emptyset$. Thus, we do not need non-accepting sink states but rather use undefined transitions to "stop" a run of an

automaton. As a consequence, in a minimal DFA, from all states an accepting state is reachable.

For ease of notation we interpret δ as a (partial) function from $Q \times \Sigma$ to Q in the case of DFAs.

An *alternating (finite) automaton* (or *AFA*) $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, I, F)$, which is defined just as in an NFA but where Q is partitioned into E (existential states) and U (universal states). The existential states behave exactly as in an NFA. That is, for an existential state q, if $\delta(q, a) = P$, there exists an accepting run for the remainder of the input word, starting from at least one state in P. The universal states q require that, if $\delta(q, a) = P$, there exists an accepting run for the remainder of the input word, starting from *every* state in P. For details we refer to, e.g., [Yu97].

A *regular language* is a language that can be denoted by a finite automaton.

Regular Expressions

The regular expressions over Σ are defined as follows: ε, $\emptyset$ and every Σ-symbol is a regular expression; and whenever R and S are regular expressions, then so are (RS), $(R + S)$, and $(R)^*$. For readability, we usually omit parentheses in examples. Sometimes we write $R \cdot S$ to emphasize that two expressions are concatenated. If $X = \{a_1, \ldots, a_n\} \subseteq \Sigma$ is a set of symbols then we may use X as an abbreviation of the expression $a_1 + \cdots + a_n$.

The language defined by a regular expression R, denoted by $L(R)$, is defined as usual.

We denote the set of all regular expressions with RE. We consider two possible extensions of regular expressions, as they are used in XML schema languages.

The first extension allows the one-or-more operator. Let R be a regular expression, then $(R)^+$ is a regular expression, where $L((R)^+) = L((R)(R)^*)$. We denote the set of regular expressions with the additional one-or-more operator with RE^+.

The second extension are counters. Let n, m be natural numbers with $0 \leq n < m$ and R be a regular expression (with counters), then $R^{[n,m]}$ and $R^{[n,*]}$ are regular expressions with counters, where

- $L(R^{[n,m]}) = L(\underbrace{RR\ldots R}_{n \text{ times}} \underbrace{(\varepsilon + R(\varepsilon + R(\ldots)))}_{m-n \text{ times}})$; and
- $L(R^{[n,*]}) = L(\underbrace{RR\ldots R}_{n \text{ times}} R^*)$.

It is well-known that XML schema languages use deterministic (sometimes also called one-unambiguous) regular expressions [BKW98]. We have a closer look at this class of regular expressions in Chapter 7.

We define the size $|r|$ of a regular expression r to be the number of occurrences of alphabet symbols, i.e. $|\varepsilon| = |\emptyset| = 0$, $|a| = 1$ for every alphabet symbol a, $|(r_1 + r_2)| = |(r_1 r_2)| = |r_1| + |r_2|$ and $|(r_1)^*| = |(r_1)^{[n,m]}| = |r_1|$ for regular expressions r_1 and r_2.

3.2 Tree Model

For XML, the W3C published the Document Object Model (DOM) as an official standard [WLA+00]. It captures all features of XML documents, e.g., it distinguished element, attribute text and entity nodes.

As most parts of this thesis focus on theoretic analysis, we prefer a simpler model, which captures the intrinsic difficulty of the DOM model but allows for simpler reasoning with less cases that need to be distinguished. In this model we do not distinguish different types of nodes, instead we assume that each node has a node id, a label and a data value.

In detail, we consider labeled directed trees with data values. To this end, we assume pairwise disjoint, infinite sets $\mathcal{V}$ of nodes, $\mathcal{D}$ of data values and $\mathcal{L}$ of labels.

Definition 3.1 An *XML tree* t is a tuple $(V, E, \mathrm{lab}, \mathrm{dv}, \prec_c)$, where

- $V \subseteq \mathcal{V}$ is a finite set of *nodes*,
- $E \subseteq V \times V$ is a set of *edges*,
- $\mathrm{lab} : V \to \mathcal{L}$ is a *labeling function*,
- $\mathrm{dv} : V \to \mathcal{D}$ is a function assigning to every node a data value, and
- $\prec_c$ is a partial order that orders the children of each node linearly.

We further require that $t = (V, E)$ is a directed tree with a unique root, denoted $\mathrm{root}(t)$, such that all edges are directed away from $\mathrm{root}(t)$.

We refer to the set of labels of a tree t by $\mathrm{lab}(t)$, and to the set of data values by $\mathrm{dv}(t)$. We often omit $\prec_c$ from tree descriptions, when the order is of no importance.

If $(u, v) \in E$ then we say that u is the *parent* of v and v is a child of u. The *descendant* relation E^+ is the transitive closure of E and the *ancestor* relation the reversal of the descendant relation.

The child string child-string(v) of a node v is the list of labels of the children of v ordered according to $\prec_c$.

The ancestor string ancestor-string(v) of a node v is the list of labels of the ancestors of the node starting with the root and ordered according to the descendant relation. We demonstrate these definitions based on the tree representation in Figure 1.2. The ancestor string of the node v_{10} is `root persons person` and the child string of node v_6 is `owner content`.

Even if we assume that every node has a data value, we will not always depict every data value. We will usually only depict those data values that are important for a specific example.

To define integrity constraints, we use the equality relation on data values. However, we do not assume any other relation on data values such as linear orders.

In Chapters 4 and 5, were we deal with actual schema languages for XML, we will additionally use some terminology from the DOM model. Especially we have to distinguish between element and attribute nodes and to consider different namespaces in these chapters.

3.3 Tree Languages

A tree language is a set of trees. Tree languages can for example be specified by various types of schema languages, by tree automata or by some logic. We usually denote tree languages by the capital letter T. If S is some schema in some schema language, we denote the tree language specified by S with $T(S)$. We denote the set of all possible trees with $\mathcal{T}$.

We will have a closer look at schema languages in the first part of this thesis. As this part concentrates on syntactical description of XML documents, we will mostly ignore the data values of trees in this part of the thesis.

A very important schema language are Document Type Definitions. A *Document Type Definition (DTD)* $D : \Sigma \to 2^{\Sigma^*}$ over a set of element labels Σ is a function that assigns to every element label $a \in \Sigma$ a regular language L_a.

A tree $t = (V, E, \text{lab}, \text{dv}, \prec_c)$ is *valid* wrt. a DTD D denoted by $t \models D$, if for every node $v \in V$ it holds that child-string$(v) \in D(\text{lab}(v))$.

A DTD is usually defined by a set of productions

$$a \rightsquigarrow R,$$

where $a \in \Sigma$ and R is a regular expression. We write $a \rightsquigarrow R \in D$ to denote that the rule $a \rightsquigarrow R$ is contained in the definition of the DTD.

For our investigations of the implication problem for X2R-constraints in the second part of this thesis, we will consider two kinds of schema languages for XML-documents. As a schema language with large expressiveness we use the class Reg of regular tree languages. However, very often schemas for XML documents only restrict the set of allowed elements in a content model in a simple fashion. We mainly concentrate on a setting where the order of siblings in an XML document is ignored and thus we use the following important restriction of DTDs: simple DTDs. We use the definition of [KW07], which we basically repeat here.

Given an alphabet Σ, a regular expression over Σ is called *simple*, if it is of the form $s_1 \cdots s_n$, where for each s_i, there is a letter $a_i \in \Gamma$ such that s_i is either a_i, $a_i?$, a_i^+ or a_i^* and for $i \neq j$, $a_i \neq a_j$. A *simple DTD* (sDTD) is a DTD where the right-hand-side of each production is simple.

For all our lower bound results even the following further restriction of simple DTDs suffices. In an *extremely simple DTD* (esDTD), only the set of allowed labels is fixed, that is every content model has the same regular expression $(a_1^* \dots a_\ell^*)$, where $\{a_1, \dots, a_l\}$ is the set of allowed labels.

Simple DTDs have unique minimal models in the following sense as already observed and used in [KW07].

Lemma 3.2 *Let D be an* sDTD *and ℓ a label that occurs in some (finite) tree that conforms to D. Then there exits a unique (with respect to structure and labels) minimal tree t_ℓ such that for every tree t and every induced subtree t' with a root node labeled ℓ,*

- *t_ℓ can be obtained by removing some nodes from t', and*

- *if t' is replaced by t_ℓ in t, the resulting tree still conforms to D.*

We refer to the trees of the form t_ℓ as *minimal D-trees*. It should be noted that if a label c occurs in a minimal D-tree t_ℓ then its induced subtree in t_ℓ is just t_c.

It can easily be tested whether for some label ℓ from an sDTD the tree t_ℓ actually exists and, therefore, ℓ can occur in a (finite) model of D. We therefore assume throughout that an sDTD only contains useful labels.

For our reasoning algorithms, we are interested in small (representations of) counter-example trees. It is easy to see that t_ℓ can be of exponential size in the size of D. Thus, an sDTD alone can already enforce minimal models of exponential size. We will therefore use a compact representation of trees conforming to an sDTD D to be defined next.

For a given tree t and sDTD D we define the *D-expansion* $[t]_D$ as the tree resulting from t by application of the following process. If there is a node v with label ℓ with a child u that has a label ℓ' that is disallowed below an ℓ-node by D then $[t]_D$ is undefined. Otherwise, as long as there are nodes v with some label ℓ such that for some $\ell' \in D(\ell)$ v has no child with label ℓ', a copy of $t_{\ell'}$ (as guaranteed by Lemma 3.2), in which all nodes have new, pairwise distinct data values, is added below v. If it exists, $[t]_D$ is the unique minimal tree conforming to D and containing t as a subtree. We note that in $[t]_D$, every node v can uniquely be identified by a pair (u, w), where u is a node from t and w a (possibly empty) sequence of labels from D of length at most $|D|$.

3.4 Tiling Problems

We will use tiling problems in reduction proofs throughout the thesis.

A *tiling instance* is a tuple $\mathcal{U} = (U, u_0, u_F, V, H)$, where

- U is a finite set of tiles;
- u_0 is the *first* tile;
- u_F is the *final* tile
- $H \subseteq U \times U$, are the *horizontal constraints*; and
- $V \subseteq U \times U$, are the *vertical constraints*.

Given a tiling instance $\mathcal{U} = (U, u_0, u_F, V, H)$, a *tiling* is a mapping

$$\lambda : \{0, \ldots, n\} \times \{0, \ldots, m\} \to U.$$

A tiling is valid, if

- the first tile is u_0 and the last tile is u_F, i.e. $\lambda(0,0) = u_0$ and $\lambda(n,m) = u_F$; and
- the horizontal and vertical constraints are met, i.e. $(\lambda(i,j), \lambda(i+1,j)) \in H$ for $i \in [0,n)$, $j \in [0,m]$ and $(\lambda(i,j), \lambda(i,j+1)) \in V$ for $i \in [0,n]$, $j \in [0,m)$.

We consider the following problems:

Tiling	
Given:	a tiling instance $\mathcal{U} = (U, u_0, u_F, V, H)$
Question:	Does there exist a valid tiling λ?

CorridorTiling	
Given:	a tiling instance $\mathcal{U} = (U, u_0, u_F, V, H)$, a number n in unary
Question:	Does there exist a valid tiling λ that has width n?

ExponentialCorridorTiling	
Given:	a tiling instance $\mathcal{U} = (U, u_0, u_F, V, H)$, a number n in unary
Question:	Does there exist a valid tiling λ that has width 2^n?

Theorem 3.3 ([van97, CGLV02])

(a) Tiling *is undecidable.*

(b) CorridorTiling *is* PSPACE*-complete.*

(c) ExponentialCorridorTiling *is* EXPSPACE*-complete.*

The intuitive idea behind the hardness proofs for tiling problems is that each row of a tiling represents a configuration of a Turing machine and a complete tiling represents a run of a Turing machine. The horizontal and vertical constraints ensure that the run is correct. In the literature often an initial and last row encode the initial and final configuration of the Turing machine. In the case of Tiling and ExponentialCorridorTiling, the input only contains the parts the first and last row encoding the input and output. The Tiling problem has therefore the same complexity as the acceptance problem of a Turing machine and the CorridorTiling and ExponentialCorridorTiling problems have the same complexity as the acceptance problem of space bounded Turing machines.

For simplicity reasons, we define our tiling instances with an initial and a final tile instead of an initial and a final row. It can be easily seen that these variants are inter-reducible, by first changing the tiling instance such that tiles in the first and last row occur exactly once and cannot occur anywhere else (by adding copies of these tiles to U and modifying H and V accordingly) and then changing H in such a way that the first tile uniquely identifies all tiles in the first row.

In some proofs, we assume that for each valid tiling instance it holds that $(\lambda(i, n), \lambda(i + 1, 0)) \in H$ for each row $i < m$. This allows us to represent a tiling as a string without the need to take special care for the horizontal constraints across row borders. Again, it is easy to see that this can be acomplished by adjusting the input instance accordingly, i.e., by adding another tile that works as row separator and corresponding changes in H and V.

Symbol	used for
Σ	set of integrity constraints (only in Part II)
Σ, Γ	finite alphabet (Σ only used in Part I)
$\mathcal{A}$, $\mathcal{B}$	finite automata
A	edge relation in patterns
D, S	DTD, Schema, regular tree language
$\mathcal{D}$	set of all data values
E	edge relation in tree
H, V	constraints in tiling problems
I	problem instances
K, $\mathcal{K}$	sets of key constraints
L	string language
$\mathcal{P}$	set of propositions
T	set of trees
$\mathcal{T}$	set of all trees
U	set of tiles
$\mathcal{U}$	tiling instances
V	node set in trees
X	set of variables in tree patterns
a, b, c	symbols from finite alphabet
d, e	data values
f, g, h	functions
i, j, k	natural numbers
ℓ	natural number, abstract label
n, m	natural numbers
p, q	states in finite automata
p	tree patterns
t	trees
u, v, w	nodes in a tree
u	tile from a tiling instance
δ	transition function of a finite automaton
σ	integrity constraints
τ	target integrity constraint in implication instances (only in Part II)
τ	typing (only in Chapter 8)
κ	key constraint (over strings)
π	embeddings of tree patterns in trees
ρ	relational constraint
μ	XML-to-relational mapping
$[i, j], [i, j)$	intervals of natural numbers
$[v, w]$	path in a tree (from node v to node w)
2^M	powerset (of M)

Table 3.1: Symbols and notation used in the thesis.

Part I

Schema Definition Languages

4 Defining the Structure of Trees

In the introduction we have described our running example of a content management system. We have given a few syntactic constraints, to which the database for this system should obey. In Part I of this thesis, we have a closer look on how to specify such types of constraints, that is, how to specify the structure of trees. In this chapter we will see how syntactic constraints can be specified using Document Type Definitions (DTDs) or XML Schema Definitions (XSDs). We will therefore extend our running example to see the difference in expressiveness between those two schema specification languages. In Chapter 5, we will introduce the pattern based schema specification language BonXai, which combines the simplicity of DTDs with the expressivity of XML Schema. All these schema languages have in common that they use deterministic regular expressions to describe the allowed sequences of children of nodes in the tree. In Chapter 7, we will have a brief look at some properties of this type of regular expressions. We will close this part of the thesis in Chapter 8 by looking at schema design for distributed XML repositories. We are especially interested in schema designs so that validity of a global combined document can be checked locally at all peers contributing to this document.

We start by giving a DTD and an XML Schema, so that valid trees fulfill the following constraints, which we already know from the introduction. Note that as we are mostly interested in describing the structure of the tree, we have left out constraints that only restrict the domain of data values of some nodes.

- The root element of the XML document is labeled root and has two child elements labeled users and documents.
- The users element has arbitrarily many child elements all labeled person.
- Each person element has two child elements labeled firstname and lastname respectively and arbitrarily many child elements labeled user-id

Document Type Definitions constitute the first schema language for XML and are most well-known for their simplicity. Basically, DTDs are a grammar-based formalism where element declarations are entirely context insensitive. That is, the content-model for an element is solely dependent on the name of that element. In Figure 4.1, we have depicted a DTD such that all valid trees obey the given constraints.

From the theoretical point of view, a DTD is a function $D : \Sigma \to 2^{\Sigma^*}$ mapping labels (of nodes) to allowed sequences of children. A document is valid wrt. a DTD, if for every node v it holds that the child string of v is contained in $D(\text{lab}(v))$, where $\text{lab}(v)$ is the label of v. In DTDs the allowed sequences of children are specified by regular

```
<!ELEMENT root      (users, documents)>
<!ELEMENT users     (person)*>
<!ELEMENT person    (firstname, lastname, (user-id)*)>
<!ELEMENT documents (document)*>
<!ELEMENT document  (owner, real-document)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname  (#PCDATA)>
<!ELEMENT user-id   (#PCDATA)>
<!ELEMENT owner     (#PCDATA)>
```

Figure 4.1: A DTD describing our CMS database

expressions[1]. Regular expressions use the operators concatenation (`,`), disjunction (`|`), zero-or-more (`*`), one-or-more (`+`) and optional (`?`). Sub-expressions can be grouped with round brackets. The rule

```
<!ELEMENT person (firstname, lastname, (user-id)*)>
```

from the DTD in Figure 4.1 expresses that every node labeled person has as first child a node with label `firstname` followed by a child with the label `lastname` followed by arbitrarily many nodes with the label `user-id`. The keyword `#PCDATA` used in the last four rules of the DTD says that the matched nodes are text nodes which are nodes without a label just containing some text. We mostly ignore these text nodes, as they are not interesting for the structure of the tree. Rules with the keyword `ATTLIST` instead of `ELEMENT` can be used to specify the list of allowed attributes for an element node.

4.1 Example: A Toy Markup Language

We will now discuss a fictional (toy) markup language that we will use to discuss the main features of XML Schema. Furthermore we will use this example to introduce the BonXai schema language in the next chapter. We first describe the markup language and an example document informally and then we will define a DTD and an XML Schema for it.

Example 4.1 (An example document) *Consider the XML fragment in Figure 4.2 with content formatted in a fictional markup language. The* `document` *is divided into* `template`, `userstyles`, *which contains user-defined style definitions, and* `content`. *The* `content` *part contains the actual text of the document, with markup (bold, font changes, etc.).*

[1] To be more precise: by deterministic regular expressions. We have a look on this class of regular expressions in Chapter 7.

```
<document xmlns="http://mydomain.org/namespace">
  <template>
    <section>
      <titlefont name="SomeFont" size="42"/>
      <style><font name="Times" size="12"/></style>
      <section>
        <titlefont size="23"/>
      </section>
    </section>
  </template>
  <userstyles>
    <style name="userdefined1">
      <font name="MyFancyFont"/>
      <color color="red"/>
    </style>
    <style name="...">
      ...
    </style>
  </userstyles>
  <content>
    <section title="Introduction">
      In this paper we discuss ...
      <section title="Motivation">
        Our problem is important because ...
        <bold>This text is bold</bold><italic>and this is italic</italic>
        <style name="userdefined1">
          This text is red and uses a different font.
        </style>
      </section>
    </section>
    <section title="...">
      ...
    </section>
  </content>
</document>
```

Figure 4.2: An XML-document

Below the `content` *node, the text is structured by* `section` *elements, which can be nested to form, subsections, subsubsections, etc.*

The `template` *element should describe the default formatting of the text within* `content`. *One could think that* `template` *defines ACM Journal style, for example. Within* `template`, *the default formatting of sections is specified within the* `section` *child of* `template` *and the*

default formatting of subsections within the `section` *grandchild. So, a difference between* `template` *and* `content` *is that, in* `template`*, there is at most one* `section` *element per nesting depth. For the sake of the example, the rationale is that the default formatting of all sections at the same level should be the same. Furthermore,* `template` *does not contain text since all the actual text is within* `content`*.*

The `userstyles` *element contains a list of* `style` *elements. Each such* `style` *element should be thought of as being either some user-defined style (e.g., a fancy font for bold mathematics). Each* `style` *element has a unique name, which can be referred to from within* `content`*. In our example, we only declared one user-defined style* `userdefined1`*.*

4.2 A DTD for the Markup Language

We chose our example such that there are elements within `content` and within `template` that have the same element names but different semantics, notably, the `section` element. Similarly, `style` has a different semantics if it is used within `userstyles`, within `template`, or within `content`.

DTDs do not have the expressive power to take these differences into account and must define a common content model for all elements with the same name. That is, a DTD can only define one rule for `section` independent of where a `section` element occurs in the XML-document.

Example 4.2 (An example DTD for Example 4.1) *Assuming an entity named* `markup`*, defined as*

```
<!ENTITY % markup "bold|italic|font|style|color">
```

we could define `section` *from Example 4.1 in a DTD as follows:*

```
<!ELEMENT section    (#PCDATA|section|%markup;)*>
<!ATTLIST section    title CDATA #IMPLIED>
```

A complete DTD for which the XML-document is valid is given in Figure 4.3. We present this entire DTD because it is instructive to compare it with the XSD which we expose next and with the BonXai schema which we define later and is equivalent to the XSD.

4.3 An XML Schema for the Markup Language

Figure 4.4 depicts a tree representation of the XML document of Figure 4.2. Nodes labeled `text` in the figure are text nodes as described in the DOM model. We do not care about the contents of these nodes. The tree representation is crucial for understanding the expressiveness of XML Schema (and, therefore, also the expressiveness of BonXai in the next chapter). Intuitively, XML Schema can distinguish between elements of the same name, when the sequence of labels that occur on the path from the root to the element is different. This means that XML Schema can distinguish the `section` elements

```
<!ELEMENT document   (template, userstyles, content)>
<!ELEMENT template   section>
<!ELEMENT userstyles style*>
<!ELEMENT content    section*>

<!ENTITY % markup "bold|italic|font|style|color">

<!ELEMENT section    (#PCDATA|titlefont|section|%markup;)*>
<!ATTLIST section    title CDATA #IMPLIED>

<!ELEMENT bold       (#PCDATA|%markup;)*>
<!ELEMENT italic     (#PCDATA|%markup;)*>

<!ELEMENT font       (#PCDATA|%markup;)*>
<!ATTLIST font       name CDATA #IMPLIED
                     size CDATA #IMPLIED>

<!ELEMENT style      (#PCDATA|%markup;)*>
<!ATTLIST style      name CDATA #IMPLIED>

<!ELEMENT titlefont  EMPTY>
<!ATTLIST titlefont  name CDATA #IMPLIED
                     size CDATA #IMPLIED>

<!ELEMENT color      (#PCDATA|%markup;)*>
<!ATTLIST color      color CDATA #REQUIRED>
```

Figure 4.3: A DTD describing the XML document in Figure 4.2.

within `content` from those within `template`, for example. Indeed, the former have the labels `section content document` on the path to the root, whereas the latter have `section template document`. (Similarly, XML Schema can also distinguish between `style` within `userstyles`, within `template`, or within `content`.)

We next develop an XSD for our example markup language which will be able to differentiate the elements with the same name but different semantics. XSDs can take context into account through the explicit use of types.

Example 4.3 (An example XSD for Example 4.1) *An XSD describing the markup language of Example 4.1 is presented in Figures 4.5 to 4.7. Figure 4.5 contains the definition of the root* `document` *node and the definition of the* `markup` *group, which we defined similarly to the markup entity in the DTD to avoid any unnecessary verbosity.*

Types in XML Schema can be specified in two different ways. They can be defined anonymous directly below some element. We have done this with the types of the root node

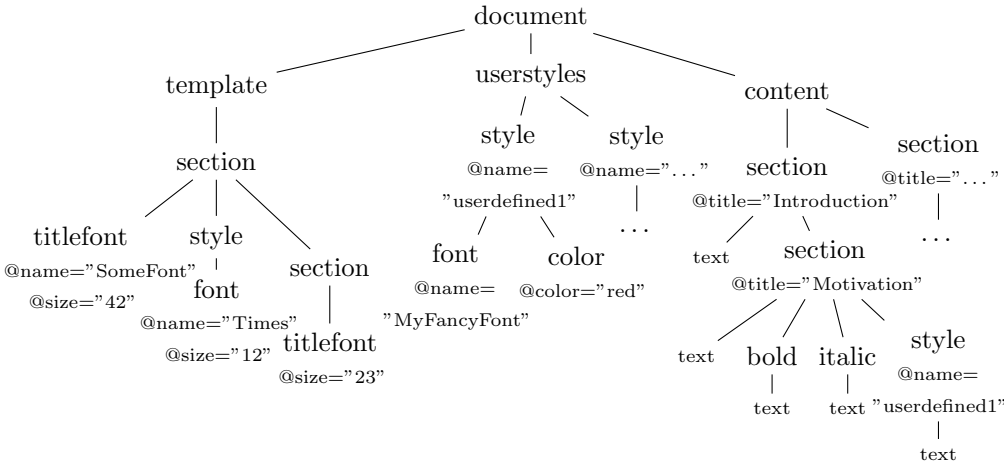


Figure 4.4: Tree representation of the XML document in Figure 4.2.

and the nodes directly below the root node. On the other hand they can be defined with names. These types are defined directly below the `xs:schema node` *and can be referenced using a type attribute inside* `xs:element` *nodes. This is the way how we defined all other types. All our type names start with a capital* T *so that the reader can easily distinguish them from element names.*

The XSD distinguishes between two types *of sections:* Tsection *and* TtemplateSection*. The former should be used within* `content` *and the latter one within* `template`*. It is instructive to view this in terms of the tree representation of our sample document, depicted in Figure 4.4. The* type *of a* `section` *element is determined by the type of its parent. That is, when the parent of such an element is labeled* `content` *or is a* `section` *element with type* Tsection*, the section can contain text and markup. On the other hand, if the parent is labeled* `template` *or is a* `section` *with type* TtemplateSection*, the* `section` *element cannot contain text, it can only contain formatting instructions. Similarly, the XSD contains three types that can be used for* `style`*:* TtemplateStyle *(for* `style` *elements below* `template`*),* TnamedStyle *(for* `style` *elements below* `userstyles`*, and* TstyleRef *(for* `style` *elements below* `content`*).*

The use of types in XSDs to define context is not unrestricted. The Element Declarations Consistent constraint, which is enforced by the XML Schema Specification [GSMT+12, Section 3.8.6.3] prohibits the use of the same element occurring in the same content model with different types.[2] One consequence of this constraint is that XSDs can only identify the context of an element based on the labels of elements occurring on the path from the root to that element, the so-called ancestor path. In [MNSB06], it was shown that the kind of constraint which can be put on such an ancestor path by an XSD can

[2] A detailed discussion on the implications of this constraint can be found in [MNSB06, MNS07].

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns="http://mydomain.org/namespace"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://mydomain.org/namespace"
           elementFormDefault="qualified">

  <xs:element name="document">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="template">
          <xs:complexType>
            <xs:sequence minOccurs="0">
              <xs:element name="section" type="TtemplateSection"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="userstyles">
          <xs:complexType>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
              <xs:element name="style" type="TnamedStyle"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="content">
          <xs:complexType>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
              <xs:element name="section" type="Tsection"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:group name="markup">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="bold"   type="Tmarkup"/>
      <xs:element name="italic" type="Tmarkup"/>
      <xs:element name="style"  type="TstyleRef"/>
      <xs:element name="font"   type="Tfont"/>
      <xs:element name="color"  type="Tcolor"/>
    </xs:choice>
  </xs:group>
```

Figure 4.5: An XSD describing the XML document in Figure 4.4 — part 1.

```
<xs:complexType name="TtemplateSection">
  <xs:sequence>
    <xs:element name="titlefont" type="TtemplateFont"    minOccurs="0"/>
    <xs:element name="style"     type="TtemplateStyle"   minOccurs="0"/>
    <xs:element name="section"   type="TtemplateSection" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="TtemplateFont">
  <xs:attribute name="name" type="xs:string"  use="optional"/>
  <xs:attribute name="size" type="xs:integer" use="optional"/>
</xs:complexType>

<xs:complexType name="TtemplateStyle">
  <xs:all>
    <xs:element name="font"  type="TtemplateFont"  minOccurs="0"/>
    <xs:element name="color" type="TtemplateColor" minOccurs="0"/>
  </xs:all>
</xs:complexType>

<xs:complexType name="TtemplateColor">
  <xs:attribute name="color" type="xs:string"/>
</xs:complexType>

<xs:complexType name="TnamedStyle">
  <xs:all>
    <xs:element name="font"  type="TtemplateFont"  minOccurs="0"/>
    <xs:element name="color" type="TtemplateColor" minOccurs="0"/>
  </xs:all>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Tsection" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="markup"/>
  <xs:element name="section" type="Tsection"/>
  </xs:choice>
  <xs:attribute name="title" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Tmarkup" mixed="true">
  <xs:group ref="markup"/>
</xs:complexType>
```

Figure 4.6: An XSD describing the XML document in Figure 4.4 — part 2.

```
  <xs:complexType name="TstyleRef" mixed="true">
    <xs:group ref="markup"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="Tcolor" mixed="true">
    <xs:group ref="markup"/>
    <xs:attribute name="Tcolor" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="Tfont" mixed="true">
    <xs:group ref="markup"/>
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="size" type="xs:integer" use="optional"/>
  </xs:complexType>
</xs:schema>
```

Figure 4.7: An XSD describing the XML document in Figure 4.4 — part 3.

always be captured by a regular expression and usually even by so-called linear XPath expressions [KS07], which are Core XPath expressions that do not branch.[3] The latter insight influences the design of BonXai, as we will see in the next chapter, to make such contexts explicit through the addition of patterns over ancestor paths.

[3]Consequently, linear XPath expressions can only reason about *paths* in trees. In the context of tree pattern queries, linear XPath expressions are sometimes also referred to as *path queries*.

5 BonXai

In this chapter, we will introduce the BonXai schema language and compare it with existing schema languages, especially Document Type Definitions (DTDs) and XML Schema.

In Section 5.1, we will therefore show two BonXai schemas for the toy markup language introduced in the previous chapter, a simple one, showing the basic BonXai syntax and a slightly more complex one exploiting the extended expressivity compared to DTDs. Afterwards, in Section 5.2, we treat the principles behind BonXai by means of several examples mostly based on the toy markup language. We explore in Section 5.3, how BonXai can be used as an analysis tool for XML Schema. The design of BonXai is influenced by existing XML schema languages. We discuss these in Section 5.4.

In the next chapter we provide a solid theoretical background for BonXai and in Chapter 13 we describe the FoXLib library, which contains — among other algorithms — an example implementation of the conversion algorithms between BonXai and XML Schema and tools helping debugging XML Schema and BonXai definitions.

5.1 BonXai Schemas for the Markup Language

We now discuss some BonXai schemas for the markup language described in Chapter 4.1. Actually, we give two BonXai schemas. The BonXai schema in Figure 5.1 is equivalent to the DTD given in Figure 4.3, while the BonXai schema in Figure 5.2 exploits the additional expressiveness of BonXai to precisely match our running example's markup language. It is equivalent to the XSD given in Figures 4.5–4.7.

Both examples use a compact syntax inspired by Relax NG [CM01]. Like a DTD, a BonXai schema is a collection of rules. The right-hand side of a rule denotes a content model as usual. The left-hand side can be either a label or a regular expression if more expressiveness is needed. We use a regular expression syntax which resembles XPath expressions since this allows users to also write linear XPath expression on left-hand sides. The semantics is that for an XML document to match the schema, the children of nodes in the document selected by a left-hand side expression when evaluated from the root, should match the content model denoted in the right-hand side of the rule. For instance, the rule

```
template//section = { element titlefont?, element style?, element section? }
```

stipulates that `section` elements occurring somewhere below a `template` element can contain a `titlefont` child, a `style` child, and a `section` child, whereas the rule

```
content//section = mixed { attribute title, (element section | group markup)* }
```

```
target namespace http://www.example.com/MyDocument
namespace xs = http://www.w3.org/2001/XMLSchema

global { document }

groups {
  group markup = { element bold | element italic | element font |
                   element style | element color }
}

grammar {
  document   = { element template, element userstyles, element content }
  template   = { element section }
  userstyles = { (element style)* }
  content    = { (element section)* }

  section    = mixed { attribute title, (element titlefont | element section
                                                          | group markup)* }
  bold       = mixed { (group markup)* }
  italic     = mixed { (group markup)* }
  font       = mixed { attribute name, attribute size, (group markup)* }
  style      = mixed { attribute name, (group markup)* }
  titlefont  =       { attribute name, attribute size }
  color      = mixed { attribute color, (group markup)* }

  @title     = { type xs:string }
  @name      = { type xs:string }
  @size      = { type xs:string }
  @color     = { type xs:string }
}
```

Figure 5.1: A BonXai schema equivalent to the DTD in Figure 4.3.

stipulates that elements occurring somewhere below a `content` element should contain a title and may contain text (indicated by the keyword `mixed`) with markup. The keyword `mixed` allows mixed content, i.e., it is allowed to interleave text with XML tags. In the BonXai schema in Figure 5.2, `/` and `//` stand for the XPath axes "child" and "descendant", respectively. We denote concatenation, disjunction, Kleene star, and "optional" by "`,`", "`|`", "`*`", and "`?`", as in DTDs. The operator "`&`" stands for unordered concatenation, which is known as `xs:all` in XSD. If an expression does not start with `/` or `//`, we implicitly assume that it starts with `//`. This way simple labels are just a special case of regular expressions.

```
target namespace http://www.example.com/MyDocument
namespace xs = http://www.w3.org/2001/XMLSchema

global { document }

groups {
  attribute-group fontattr = { attribute name?, attribute size? }
  group markup = { ( element bold | element italic | element style |
                     element font | element color )* }
}

grammar {
  document     = { element template, element userstyles, element content }
  content      = { (element section)* }
  template     = { (element section)? }
  userstyles   = { (element style)* }

  content//section = mixed { attribute title,(element section|group markup)* }
  content//style   = mixed { attribute name, group markup }
  content//font    = mixed { attribute-group fontattr, group markup }
  content//color   = mixed { attribute color, group markup }
  (bold|italic)    = mixed { group markup }

  template//section = { element titlefont?, element style?, element section? }
  template//style   = { element font? & element color? }

  userstyles/style  = { attribute name, element font? & element color? }
  (userstyles|template)//color            = { attribute color }
  (userstyles|template)//(font|titlefont) = { attribute-group fontattr }

  @name   = { type xs:string }
  @color  = { type xs:string }
  @title  = { type xs:string }
  @size   = { type xs:integer }
}
```

Figure 5.2: A BonXai schema equivalent to the XSD in Figures 4.5–4.7.

The main difference with the corresponding XSD is that contexts are now defined explicitly. Another way of viewing the difference between XSD and BonXai is top-down versus bottom-up. XSDs carry all relevant information about the root-path in a top-down fashion, encoded in types, while BonXai, instead, looks upward from a node, thus separating types from their inference. Furthermore, as XSDs employ types, context has

to be specified in terms of automata, while BonXai can use the more user-friendly regular expressions or linear XPath expressions.

5.2 BonXai at a Glance

BonXai schemas consist of up to five blocks. First, there is a *namespace block*, declaring all namespaces used in the schema. The second block is called the *global block* and specifies which element names can occur at the root of documents that match the schema. Third, there is an optional *group block*, which can declare the equivalent of XSD groups. The fourth block is called the *grammar block* and is the actual core of the schema. The grammar block contains the definitions of the rules that define the structure of documents. Finally, there is an optional *constraints block* which defines integrity constraints.

This chapter is not intended to discuss all details of the BonXai language and how they correspond to XML Schema. Instead, we provide a high-level overview and refer the reader to [MMN+14] for further details. We first discuss a few BonXai-specific matters (ancestor patterns, child patterns, and priorities) and then we show how BonXai seamlessly incorporates most of XML Schema language features (like differentiation between elements/attributes, simple types, element- and attribute groups, namespaces, constraints, schema imports, mixed types, default values, wildcard patterns).

Global Element Names Elements that are declared `global` in a BonXai schema can occur as root elements in XML documents that match the schema. In our running example, there is a single such element, called `document`. Global elements are the only elements that can be referenced from foreign namespaces. The list of global elements corresponds to the elements declared directly below the `xs:schema` node of an XML Schema.

Ancestor Patterns A rule within the grammar-block of a BonXai schema is of the form

```
<ancestor pattern> = <child pattern>
```

The ancestor pattern (left of the equality sign) describes the context of the rule and should be matched against paths in the tree that start from the root. Ancestor patterns are variants of regular expressions, built from element names and attribute names (i.e. names starting with @). The regular expressions have the operators union (|), concatenation (/), descendant (//), Kleene star (∗), one-or-more (+), and zero-or-one (?). Sub-patterns can be grouped using round brackets.

It should be noted that attribute names should only occur at the end of ancestor patterns as in XML documents attributes cannot have any children. Therefore ancestor patterns like `/a/@b/c` cannot match any nodes in a well formed XML document.

For convenience, a pattern that does not start with either /or // is implicitly assumed to start with //. This allows to just use an element name as ancestor pattern to match all elements of this name, as in DTDs.

Child Patterns In its simplest form, a child pattern is a regular expression describing the content model of a set of elements. To allow some other features (e.g. groups) and not introducing ambiguity, all element names have to be prefixed with the keyword `element`. Regular expressions in child patterns are built using concatenation (`,`), union (`|`), interleaving(`&`), Kleene closure (`*`), one-or-more (`+`), zero-or-one (`?`) and counting (`{n,m}`). The upper bound of counters may be `*` instead of a number to express that there is no upper bound. Sub-expressions can be grouped using round brackets. The use of the interleaving operator is restricted, to reflect the restrictions imposed by the all-pattern of XML Schema. (The restrictions for XML Schema are described in Section 3.8.2 in [GSMT+12].) In plain words, these restrictions say that no content model should use an interleaving operator and at the same time a union or a concatenation operator. Furthermore, in content models containing an interleaving operator, counters are only allowed directly above element declarations in the syntax tree of the regular expression.

Instead of a child pattern there can be a reference to a type described in an XML schema, as described below under "References to foreign namespaces". For compatibility with XML we require that rules where the ancestor pattern contains any attribute names, the child pattern is a reference to a simple type. Again, the reason is that attributes are not allowed to have any children in XML documents.

Attributes Attributes are specified at the beginning of child patterns, that is, child patterns can have an optional list of attribute declarations before the start of the element declarations. Attributes are separated by comma and can be followed by a `?`, indicating that the attribute is optional.

Priorities It is possible to define BonXai rules such that two or more rules match the same path. When such a multiple match occurs, BonXai gives priority to the rule that occurs lowest in the schema. To illustrate this, assume that we would change the ancestor pattern `content//section` to `section`. Then we would have the rules

```
section           = mixed { attribute title, (element section | group markup)* }
template//section = { element titlefont?, element style?, element section? }
```

in the schema. Both rules are matched by a `section` element that is below a `template` element. In this case, the rule for `template//section` takes priority and therefore the semantics of the modified schema are the same as the semantics of the original schema. The rationale behind priorities is that a developer can first write down rules that generally apply in the schema and write down the special cases and exceptions later. We introduced priorities in BonXai because they were required for ensuring full compatibility with XML Schema's expressive power. We explain this matter in more detail in Chapter 6.3.

Groups Groups can be used in BonXai to abbreviate parts of child patterns that are common to several different patterns (similar to entities in DTD and groups in XML Schema). In our running example, we use the group `markup`, to abbreviate the disjunction of the elements `bold`, `italic`, etc. Groups are declared in the groups block and can be used using the keyword `group` inside child patterns.

Attribute groups can be used analogously to groups. They are prefixed by the keyword `attribute-group`, as for example in the rules for `font`-elements in Figure 5.2.

Namespaces BonXai has full namespace support. The target namespace is declared using the keyword `target namespace`. Other namespaces can be declared using the syntax `namespace <prefix> = <namespace URI>`. The target namespace will be used as default namespace for all names, which are not prefixed with a namespace prefix. Names in other namespaces can be expressed by `<prefix>:<local name>`, as in XML Schema.

Mixed and nillable content models Mixed or nillable content models are declared using the keyword `mixed`, respectively, `nillable`, in front of the child pattern. Both keywords can be combined.

Default and fixed values Default values for attributes and elements using a simple type can be declared using the syntax `type <typename> default "<value>"`. Fixed values can be declared analogously.

Integrity Constraints BonXai allows to express the same integrity constraints as XML Schema (i.e., unique, key, and keyref). The term "keyref" is taken from XML Schema, where it denotes a foreign key constraint. As in XML Schema, keys should have a name, so that keyrefs can refer to them. The general syntax of key constraints is

`key <name> <ancestor pattern> { <selector> { <fields> } }`,

where the ancestor pattern is used to select the elements for which the key should be defined and selector and fields have the same meaning as in XML Schema. The syntax for unique constraints is the same, apart from the fact that unique constraints do not have a name. In a keyref, the semantics of `<name>` is that it should be the name of the key it refers to.

Example 5.1 (Keys for Example 4.1) *To express in our running example that names of user-defined styles should be unique, we can use the key constraint*

`key stylekey /document { //userstyles/style { @name } }`.

It says that, below the `document` *root, paths that match the linear XPath expression* `//userstyles/style/@name` *uniquely identify paths that match* `//userstyles/style` *(as in XML Schema). Finally, we can express that every* `style` *used in* `content` *should be declared in* `userstyles` *by the foreign key constraint*

`keyref stylekey /document { //style { @name } }`.

References to foreign namespaces BonXai allows to refer to content of foreign XML Schemas, so that content that is defined elsewhere does not need to be re-defined within the BonXai schema. In particular, it is possible to refer to foreign elements, attributes, and XML Schema simple- or complex types. We explain how foreign content can be referenced and how we intend the use of foreign references in BonXai.

```
target namespace http://www.example.com/MyCMS
namespace doc=http://www.example.com/MyDocument

global { root }

grammar {
  root      = { element users, element documents }
  users     = { element person* }
  person    = { element name, element user-id* }
  documents = { element document* }
  document  = { element owner, elementref doc:document }
  name      = { type xs:string }
  user-id   = { type xs:string }
}
```

Figure 5.3: BonXai schema describing our CMS database.

Global elements of foreign namespaces can be referenced by using the `elementref` keyword inside child patterns. (In XML Schema it is only possible to refer to foreign elements if they are global elements in the foreign schema. We inherit this restriction.) In Figure 5.3 we have depicted a BonXai schema for our content management system. The rule

```
document = { element owner, elementref doc:document }
```

says that documents inside the CMS should be validated against the (global) element `document` from the namespace `http://www.example.com/MyDocument`, which is declared in the BonXai schema from Figure 5.2. Note that in the content management system there are `document` nodes from two different namespaces. There are `document` nodes from the CMS namespace itself, which have as children an `owner` node (from the CMS namespace) and a `document` node from the markup language namespace containing the actual document.

To provide another example, we want to extend our markup language with support for SVG vector images. We can accomplish this by adding the namespace declaration `namespace svg=http://www.w3.org/2000/svg` and extending the group `markup` with `elementref svg:svg`.

Similarly, foreign global attributes can be referenced by using the `attributeref` keyword. For example, if we want to be able to add XLink[1] references to documents, this can be accomplished by adding `namespace xlink=http://www.w3.org/1999/xlink` to the namespace declarations and extending the content model of the `document` rule with `attributeref xlink:href?`. (We explain how to import a bigger fragment of the XLink language when we discuss wildcards next.)

[1]XLink is a language intended to allow embedding of hyperlinks and some other meta-information to arbitrary XML documents in a standardized way.

References to types (in foreign XML Schemas) are mainly intended to refer to simple types like `xs:integer` and `xs:string`. Type references are expressed by replacing the right-hand side of a rule with { `type ns:typename` }, where `type` is a keyword, `ns` should be a declared namespace, and `typename` the name of the target type inside namespace `ns`. In our running example, the rule `@title =` { `type xs:string` } express that all `title` attributes should use the type `string` which is declared in the XML Schema namespace.

In general, it is also possible (but perhaps not encouraged) to refer to foreign XML Schema complex types. For example, the rule `//foo =` { `type svg:svgType` } would state that each element with name `foo` has the type `svgType` of the `svg` namespace. (However, we feel that using `elementref svg:svg` instead, whenever possible, is more elegant.)

In summary, although BonXai is intended to be a language that reduces the use of types to a minimum, we do allow references to foreign types. The reasons for this decision are that it allows the use of XSD simple types and that we like to allow users to easily import (e.g., well-known, standard) types which are defined elsewhere. It should be noted that, whenever an element is declared to have an (XSD-)type, no BonXai rules are applied to nodes below this element, as the set of allowed subtrees for this element is entirely determined by the type.

Wildcards Wildcards are expressed by any-patterns in XML Schema. Note that XML Schema wildcards can be restricted to certain namespaces and it can be declared whether elements matched by any-patterns should be checked against some schema declaration or not. BonXai provides the same mechanism for wildcards. For example, to allow arbitrary foreign markup, we could extend the `markup` group with `any` {`lax namespace` {`##other`}}, meaning, that elements from other namespaces are allowed and should be validated, if a declaration is present. As in XML schema, the validation policy can be changed to `strict` (a declaration has to be present) or `skip` (the subtree below matched elements is not validated at all).

It is also possible to allow arbitrary attributes using the keyword `anyattribute`. As for arbitrary elements, the wildcard can be restricted to certain namespaces. For example to allow arbitrary XLink information to be added to document roots, we can extend the childpattern of the `document` rule by

`anyattribute` {`strict namespace` {`http://www.w3.org/1999/xlink`}}.

The `strict` keyword says that the content should be validated and validation should fail if the XLink declaration is not present.

Annotations Annotations can be used to add further information to a schema. In BonXai, annotations can be added before every rule. Annotations have no semantic meaning for the schema. However they might have a meaning for software used to create and edit BonXai schemas.

Our implementation (see Chapter 13) uses annotations to preserve type names when converting XSDs to BonXai schemas. This way the user can easily grasp the correspon-

dence between XML Schema complex types and BonXai rules. When converting BonXai schemas to XSDs, these annotations are used to generate meaningful XSD complex type names. For example, our implementation uses the annotation

```
@typename=MyTypename
//a = { ... }
```

with the meaning that a complex type created for the rule `//a = { ... }` should be named `MyTypename` when converting to XML Schema. In theory, it may be possible that more than one XSD complex type needs to be created for a single BonXai rule. In this case our implementation adds numbers after the given name.

Unconstrained Elements It is theoretically possible to write BonXai schemas which do not constrain certain ancestor paths. For example, if a BonXai schema would only have the two rules

```
/a = { element b, element c}
//b = { ... }
```

then the *c*-child of the root in a corresponding document does not have a matching ancestor pattern. In this case, BonXai allows any content below this `c`-child. Concretely, we translate this case to XML Schema's *anytype*, which is the most general type in XML Schema [PGM+12, Section 3]. We treat such elements the same as elements that refer to an XSD-type (see the last paragraph of *References to foreign namespaces*). Therefore, as a consequence, no BonXai rules are matched against descendants of the `c`-child of the root.

5.3 BonXai at Work

We now discuss a few more specific use cases for BonXai to illustrate that BonXai is not just a "readable syntax for XSDs" but can also be used to perform some more serious tasks more efficiently.

Analyzing existing XSDs Existing XSDs can be converted to BonXai to analyze their structural complexity. Such a BonXai inspection can, e.g., give an idea of the amount of structural expressiveness which goes beyond DTDs and where it sits. In addition, the selection patterns provided by BonXai provide direct insight into the definition of elements depending on their context. As such, the BonXai translation, converting the machine readable syntax of XSDs in the more human-readable compact syntax of BonXai, and the associated highlighting features in our GUI help users to understand schema definitions more quickly and easily. The selection patterns in the left-hand sides of BonXai rules give users immediate insight on where a given complex type is used in an XML document. Since such selection patterns are basically specified in a fragment of XPath, users familiar with XML technology can already benefit from this feature without having to learn yet another standard.

Example 5.2 *In our running example, the BonXai rules*

```
template//section = { element titlefont?, element style?, element section? }
content//section  = mixed { attribute title, (element section | group markup)* }
```

give immediate insight in the difference between the complex types `TtemplateSection` *and* `Tsection` *from Figure 4.6. The former specifies the structure of* `section`*-descendants of* `template` *elements in the tree; and the latter of* `section`*-descendants of* `content` *elements.*

Evolving from a DTD to an XSD BonXai can be used to move from a DTD to an XSD rather painlessly while, at the same time, taking advantage of the extra expressiveness. One can convert the given DTD into BonXai, add the desired extra structural features directly in the BonXai schema, and convert the result to XSD.

Example 5.3 *The BonXai schema in Figure 5.1 is equivalent to the DTD in Figure 4.3. By only a few modifications it can be extended to the BonXai schema from Figure 5.2, which can then be exported to an XSD equivalent to the one in Figures 4.5 to 4.7.*

Schema Evolution Schema evolution refers to updating a schema to reflect a re-structuring of the underlying data. We distinguish two use cases regarding schema evolution, depending on whether we want to modify an existing XSD or an existing BonXai schema using our system. In the latter case, schema evolution can simply be done by editing the BonXai schema. In the former case, the workflow is roughly the following: Convert the XSD to BonXai; alter the schema by specifying additional constraints or changing some content models; and re-export the schema to XSD.

The highlighting features of the system, mapping patterns in BonXai rules to complex types in the generated XSD fragment provide the developer with control to inspect the induced changes in the original XSD more rapidly and accurately.

Especially the priority system used by BonXai can be very helpful in schema evolution. For example, in our running example, sections can be nested arbitrarily deeply. Assume that we want to change the schema such that the nesting depth of sections is at most three. In the BonXai schema in Figure 5.2, this can be achieved by inserting the rule

```
content/section/section/section = { attribute title, group markup }
```

at the end of the rules that start with `content`. The semantics of this rule would be that subsubsections only have a title attribute and markup, but no `section` children.

If one would want to perform the equivalent change directly in XML Schema, one would be required to make three complex types for sections below `content`: one for each allowed nesting depth. Incidentally, when converting the updated BonXai schema back to XSD, the converter produces exactly these three complex types.

Debugging invalid XML documents w.r.t. an XSD When an XML document is invalid with respect to an XSD, BonXai can offer a transparent explanation when the mismatch is caused by a complex type violation. To this end, the existing XSD can be converted to BonXai. The system can highlight where an element mismatch occurs. The left-hand sides of the BonXai rules can offer more insight in terms of simple patterns for which kinds of elements are affected than the complex-type names provided by the XSD. (In this respect, tracing complex-type definitions in large XSDs to find such a source of errors is much like debugging source code that consists of GOTO-statements.) Again, the highlighting features of the system can aid the developer to understand how changes in patterns affect the invalid XML document.

Developing new Schemas / Using BonXai Stand-Alone As mentioned before, BonXai is not primarily meant as a replacement for XSDs, but to a large extent it can be used as such. The system can be used to develop schemas from scratch and to debug them. When the schema is finished, XML documents can be validated directly against the BonXai schema. Of course, the BonXai schema can also be exported to an XSD as well and XML documents can then be validated against the XSD using state of the art XML validators for XSDs.

For stand-alone use, BonXai's main strength lies in its succinct and transparent way for defining the *structure* of XML documents. BonXai does not (yet) have a syntax for defining XML Schema simple types. Therefore, simple types always need to be imported from an existing XSD. One way to do this is to write a structurally very simple XSD that only defines a set of simple types (that is, without complex types). This XSD can be imported into the BonXai schema, which can then use the simple types from the XSD and define structural aspects through its grammar.

5.4 A Comparison with Other Schema Languages for XML

As already stated before, BonXai borrows concepts from several existing schema languages for XML. The purpose of this section is to give an overview of the most well-known of those languages and discuss their relationship with BonXai.

Following [MS06], DSD2 [DSD02] (Document Structure Description 2.0) is a language developed by the University of Aarhus and AT&T Research Labs whose primary goal is to be simple yet expressive. Like BonXai, DSD2 is based on rules which must be satisfied for every element in the input document. BonXai and DSD2 are incomparable in how context is defined. While DSD2 is far more expressive than DTDs, its exact expressiveness in formal language theoretic terms is unclear. It allows context to be defined in terms of Boolean expressions which can refer to structural predicates like *parent* and *ancestor*, but, unlike BonXai, also allows to look downward using predicates like *child* and *descendant*. BonXai on the other hand harnesses the full power of regular languages on the ancestor

path, while DSD2 seems to remain within the star-free regular languages (on the ancestor path). For this reason, DSD2, on a structural level, is incomparable to XML Schema.

Relax NG [CM01] has been developed within the Organization for the Advancement of Structured Information Standards (OASIS). Like DSD2, its main goal is to combine simplicity with expressivity. In formal language theoretic terms, the expressiveness of Relax NG corresponds to the unranked regular tree languages which strictly includes XML Schema [MLMK05, MNSB06]. Like XML Schema, Relax NG is grammar based and utilizes types to define context. However, Relax NG schemas are not restrained by the Unique Particle Attribution constraint or the Element Declarations Consistent constraint. So, unlike XSDs and therefore BonXai, the context of an element in Relax NG can depend on the complete tree. As BonXai strives for simplicity it utilizes a readable compact syntax which is inspired by that of Relax NG.

Schematron [Sch99] is a rule-based language based on patterns, rules and assertions. Basically, an assertion is a pair (ϕ, m) where ϕ is an XPath expression and m an error message. The error message is displayed when ϕ fails. A rule groups various assertions together and defines by means of an XPath expression a context in which the grouped assertions are evaluated. Patterns then group various rules together. Schematron is not so much intended as a stand-alone schema language but can be used in cooperation with existing schema languages. BonXai shares the use of XPath-expressions with Schematron, although BonXai restricts them to a very small subset (linear expressions) to ensure compatibility with XML Schema.

Co-constraints is an overloaded term which generally refers to a mechanism for verifying data interdependencies. While DSD, Schematron, and Relax NG quite naturally allow to express co-constraints, XSDs are rather limited in this respect. The latter motivated the formulation of extensions of DTDs and XSDs, named DTD++ [FGMV04] and SchemaPath [CMV04], with XPath expressions to express co-existence and co-absence of element names and attributes. These extensions share with BonXai the use of XPath to express conditions but differ from BonXai in that they increase the expressiveness beyond that of XML Schema.

6 The Theory Underlying BonXai: Core XSD and Core BonXai

In this chapter, we explain the underlying theory of BonXai. In particular, we provide

- a compact and clear formal model of core BonXai schemas;
- a theoretical foundation for the BonXai priority system;
- a formal back and forth translation procedure between core XML Schema and core BonXai;
- an analysis of the blow-up of these conversions;
- practically relevant fragments of core XML Schema and core BonXai that can be efficiently translated into the other formalism; and
- proof of worst-case optimality for the conversions.

Our aim is to provide a precise mathematical description of BonXai's core which abstracts away from unavoidable cosmetics like namespaces and data types, and which offers a quick understanding of the essentials of the language. The presentation of the translations between BonXai and XML Schema fulfills a similar purpose and, in addition, makes the relation between BonXai and XML Schema apparent. In particular, the translation provides insight to where one language can be more succinct than the other.

To concentrate the presentation on the logical core of the translation instead of on bells and whistles, we introduce a high-level abstraction of BonXai and utilize an abstraction of XML Schema which is standard in the literature.

6.1 A Formal Model for XML Schema Definitions

An XML Schema uses a finite set of element names and complex type names. We therefore fix finite sets EName and Types of *element names* and *complex type names*, respectively. The set TEName of *typed element names* is then defined as $\{a[t] \mid a \in \mathsf{EName}, t \in \mathsf{Types}\}$. In an XML Schema, a typed element name $a[t]$ could, for example, be written as `<xs:element name="`a`" type="`t`"/>`. Our abstraction of an XML Schema closely follows the definition from [MLMK05, MNSB06] and is also used in [MNS07]:

Definition 6.1 An *XSchema Definition (XSD)* is a tuple $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ where EName and Types are finite sets of elements and types, respectively, ρ is mapping from Types to regular expressions over alphabet TEName, and $T_0 \subseteq \mathsf{TEName}$ is a set of typed start elements. Furthermore, the following two conditions hold:

Element Declarations Consistent (EDC) There are no typed elements $a[t_1]$ and $a[t_2]$ in a regular expression $\rho(t)$ with $t_1 \neq t_2$. Furthermore, there are no typed elements $a[t_1]$ and $a[t_2]$ in T_0 with $t_1 \neq t_2$.

Unique Particle Attribution (UPA) Each regular expression $\rho(t)$ is deterministic.

The EDC constraint can be found in [GSMT+12, Section 3.8.6.3] and the UPA constraint in [GSMT+12, Section 3.8.6.4].

We sometimes also refer to $\rho(t)$ as the *content model associated to* t. For ease of notation we extend the definition of ρ to typed element names as follows:

$$\rho(a[t]) = \rho(t) \text{ for every } a[t] \in \mathsf{TEName}.$$

A *typing* of an XML document D w.r.t. X associates, to each node u of D, a type of the schema. Formally, a typing of D w.r.t. X is a mapping μ from $\mathrm{Nodes}(D)$ to TEName. A typing μ is *correct* if it satisfies the following three conditions:

- $\mu(\mathrm{root}(D)) \in T_0$.
- For each node $u \in \mathrm{Nodes}(D)$, we have $\mu(u) \in \{\mathrm{lab}(u)[t] \mid t \in \mathsf{Types}\}$.
- For each node $u \in \mathrm{Nodes}(D)$ with children $u_1, \ldots, u_n$ from left to right, we have $\mu(u_1) \cdots \mu(u_n) \in L(\rho(\mu(u)))$.

An XML document D conforms to an XSD X if there exists a *correct typing* μ *of* D *w.r.t.* X. Notice that typings are unique due to the EDC condition, that is, there can be at most one correct typing for a given document D w.r.t. a given XSD X.

Example 6.2 *We present an XSchema Definition for a fragment*[1] *of the XML Schema in Figures 4.5 to 4.7 to illustrate XSchemas. We can abstract the schema as XSD* $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$*, where*

- $\mathsf{EName} = \{$*document, content, section, style, bold, italic, font, color, template, userstyles*$\}$
- $\mathsf{Types} = \{\mathsf{Tdocument}, \mathsf{Ttemplate}, \mathsf{Tuserstyles}, \mathsf{Tcontent}, \mathsf{TtemplateSection}, \mathsf{TtemplateStyle}, \mathsf{TtemplateFont}, \mathsf{TtemplateColor}, \mathsf{TnamedStyle}, \mathsf{Tsection}, \mathsf{Tmarkup}, \mathsf{TstyleRef}, \mathsf{Tfont}, \mathsf{Tcolor}\}$

[1] We focus on the elements of the schema; since this is the part where the complexity lies when converting between XML Schema and BonXai.

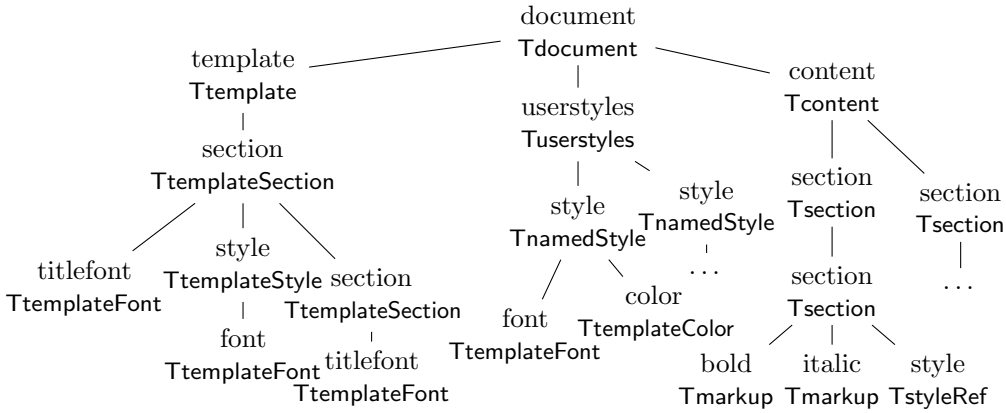


Figure 6.1: Typing for the XML document in Figure 4.4 and the XSD in Figures 4.5 to 4.7.

- ρ *is defined as follows (some parts omitted):*
  $Tdocument \rightarrow template[\mathsf{Ttemplate}]\ userstyles[\mathsf{Tuserstyles}]\ content[\mathsf{Tcontent}]$
  $Ttemplate \rightarrow (section[\mathsf{TtemplateSection}])?$
  $Tuserstyles \rightarrow (style[\mathsf{TnamedStyle}])^*$
  $Tcontent \rightarrow (section[\mathsf{Tsection}])^*$
  $Tsection \rightarrow (bold[\mathsf{Tmarkup}]/ + \cdots + color[\mathsf{Tcolor}] + section[\mathsf{Tsection}])^*$
  $\ldots$

- $T_0 = \{\mathsf{Tdocument}\}$

In the XML Schema, we defined some types inline. These types are called anonymous types in XML Schema and do not have a name. In this example, we denote these types by Tdocument, Ttemplate, Tuserstyles, and Tcontent. We did not specify the function ρ completely since it would make the example rather verbose. Notice that we also omitted rules from this example that would use the xs:all operator (respectively, the &-operator in BonXai). Indeed, we do not consider this operator in the present chapter to simplify presentation. The operator & could, in fact, simply be added to the regular expressions that we use for specifying content models in XSchemas and BonXai. It would make the discussion in this chapter more verbose (because we have different regular expressions for ancestor strings than for child strings) but the translation between XSchema and BonXai that we present further on would be essentially the same. For similar reasons, we do not consider attributes and minoccurs/maxoccurs constraints in XML Schema.

The correct typing for the XML document in Figure 4.4 according to the XSchema in Example 6.2 is denoted in Figure 6.1. A typing of the XML document according to the XML Schema in Figures 4.5 to 4.7 would look very similar.

6.2 A Formal Model for BonXai Schemas

A BonXai schema is abstracted as follows.

Definition 6.3 A *BonXai Schema Definition (BXSD)* is a pair $B = (\mathsf{EName}, S, R)$ where $S \subseteq \mathsf{EName}$ is a set of start elements and R is an ordered list $r_1 \to s_1, \ldots, r_n \to s_n$ of rules, where

- all r_i are regular expressions over the alphabet EName and
- all s_i are deterministic regular expressions over the alphabet EName.

For each $i = 1, \ldots, n$, we say that the rule $r_i \to s_i$ has *index* i. Let D be an XML document and u a node of D. A rule $r_i \to s_i$ is *relevant* for u if i is the largest index such that anc-str$^D(u) \in L(r_i)$. Notice that a node u has at most one relevant rule in B. An XML document D conforms to the BXSD B if the label of root(D) is in S and, for each node $u \in \text{Nodes}(D)$, if $r_i \to s_i$ is relevant for u, then ch-str$^D(u) \in L(s_i)$. The definition of relevant rules reflects the priority system in BonXai: rules with a higher index have higher priority.

Our abstraction of BonXai Schema Definitions requires expressions s_i to be deterministic. This restriction corresponds to the Unique Particle Attribution (UPA) restriction from XSDs [GSMT$^+$12, Section 3.8.6.4]. In particular, this restriction is necessary to make BXSDs expressively equivalent to XSDs.

Example 6.4 *The formal abstraction of the BonXai schema in Figure 5.2 is the BXSD* $B = (\mathsf{EName}, S, R)$ *where*

- $\mathsf{EName} = \{$*document, template, userstyles, content, section, style, title*$\}$
- $S = \{$*document*$\}$
- R *is the ordered list containing rules (some parts omitted):*

$$
\begin{aligned}
&//\mathit{document} &&\to \mathit{template\ userstyles\ content}\\
&//\mathit{content} &&\to \mathit{section}^*\\
&//\mathit{template} &&\to \mathit{section}\\
&//\mathit{userstyles} &&\to \mathit{style}^*\\
//\mathit{content}//\mathit{section} &&\to (\mathit{bold} + \cdots + \mathit{color} + \mathit{section})^*\\
&\cdots\\
//\mathit{template}//\mathit{section} &&\to \mathit{titlefont?\ style?\ section?}\\
&\cdots
\end{aligned}
$$

Here, we wrote the left-hand-sides of BonXai rules as in the previous chapter. Formally, in this chapter, // abbreviates the regular expression EName^*.

If we ignore the types in Figure 6.1, it represents a tree (with some parts omitted) that would be valid against this BXSD.

6.3 Priorities in BonXai

In this section we explain some fine points of the priority-based semantics of rules in BonXai schemas. Priorities were mainly introduced to avoid compatibility problems with XML Schema. However, we think they can also be convenient as we will explain below.

In the theory of pattern-based schemas for XML (of which BonXai is an example), two alternative semantics for multiple matches of rules have been investigated [GN11, KS07]: existential semantics and universal semantics. We say that the *ancestor-pattern of rule* `r = {s}` *matches a node* n in an XML tree, if the string of element names from the root of the document to n matches the regular expression `r`. The two semantics can now informally be defined as follows:

- Universal semantics: for each node n in the XML tree and each rule `r = {s}` for which the ancestor pattern matches n, the children of n must match `s`.

- Existential semantics: for each node n in the XML tree, there must be at least one rule `r = {s}` for which the ancestor pattern matches n and the children of n match `s`.

Thus, under universal semantics, we would require a matching element to match *all* content model definitions of relevant rules and under existential semantics, we would require a matching element to match *at least one* content model definition of a relevant rule. However, in practice, we cannot apply any of these two semantics if we want to be compatible with the Unique Particle Attribution rule of the W3C XML Schema specification. The Unique Particle Attribution rule requires content model definitions to be *deterministic regular expressions* (sometimes also called *one-unambiguous regular expressions* [BKW98]).

One can show that both universal and existential semantics would give BonXai expressivity beyond XML Schema. The intuitive reason both existential and universal semantics would make BonXai too powerful is that, for existential semantics to be translatable in XSDs, languages that can be defined by deterministic regular expressions would need to be closed under finite unions. To be able to translate universal semantics in XSDs, they would need to be closed under intersection. However, they are closed under neither operation (see Chapter 7 for references), which rules out universal semantics and existential semantics as equally expressive candidates.

A "quick and dirty" solution to deal with this problem could be to require ancestor patterns in rules to have an empty intersection. However, we feel that this would be very user-unfriendly. Consider again our running example in Figure 5.2. The two ancestor patterns `template//section` and `content//section` have a non-empty intersection since both could, in theory, match a word that has an occurrence of `template`, followed by `content`, followed by `section` (even though such a word cannot occur as a path in trees defined by the schema). Changing the two ancestor patterns to mend this problem would make the schema less readable. We therefore feel that this option would lead to unreadable schemas and a requirement that users would have to be experts in formal

language theory (rewriting regular expressions such that they have an empty intersection and still state what is meant).

We show in the next Section that the priority-based semantics of BonXai does not have the expressivity problems of universal or existential semantics, by giving conversion algorithms from the core of BonXai to XML Schema and back; and by observing that the Unique Particle Attribution constraint is preserved. Furthermore, we already explained in the last chapter that priorities are a useful feature of the BonXai schema language.

6.4 Translations Between Schemas

In this section, we discuss how to translate back and forth between XML Schema and BonXai. We discuss the translation from XML Schema to BonXai first and the converse later.

Translation from XML Schema to BonXai

We present a translation algorithm from XSDs to BXSDs. This algorithm is the core of a procedure that we implemented to translate XML Schema into BonXai [MNNS12]. The algorithm consists of two phases. The first phase converts an XSD into an intermediate data structure, which is called a *DFA-based XSD*. We will define such a DFA-based XSD formally, because it is a representation of schemas that is very convenient in proofs. In the second phase, the DFA-based XSD is translated to the BXSD.

DFA-based XSDs were introduced in [MNS07] (Definition 6) as an alternative characterization of XML Schema Definitions. We now define DFA-based XSDs as in [MNS07], with a minor difference: we require their content models to be deterministic regular expressions. This extra condition is necessary to reflect the UPA condition of XSD.

Definition 6.5 A *DFA-based XSD (with deterministic content models)* is a tuple $(\mathcal{A}, S, \lambda)$, where $\mathcal{A} = (Q, \mathsf{EName}, \delta, q_0)$ is a DFA with initial state q_0 and without final states such that q_0 has no incoming transitions, $S \subseteq \mathsf{EName}$ is the set of allowed root element names and λ is a function mapping each state in $Q \setminus \{q_0\}$ to a deterministic regular expression over EName. Furthermore, for every state $q \in Q$ and every element name a occurring in $\lambda(q)$, we have that $\delta(q, a)$ is non-empty.

In the remainder of this chapter, S usually equals $\{a \mid \delta(q_0, a) \text{ is non-empty}\}$. (The intuition is that, for each element $a \in S$, the automaton $\mathcal{A}$ can read a string that starts with a. Since S is simply the set of root elements, λ does not map q_0 to a regular expression.) However, we sometimes use fully defined DFAs (which are DFAs in which $|\delta(q, a)| = 1$ for every state q and label a) and therefore we need to explicitly mention S in general. Since we *only* consider DFA-based XSDs with deterministic content models in this thesis, we henceforth simply refer to them as *DFA-based XSDs*.

An XML document D *satisfies* $(\mathcal{A}, S, \lambda)$ if the root node is labeled with an element name from S and, for every node u, $\delta^*(\text{anc-str}^t(u)) = \{q\}$ implies that $\text{ch-str}^D(u)$ is in the language defined by $\lambda(q)$.

We now explain how to translate a given XSD $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ into an equivalent DFA-based XSD in linear time. The procedure is an adaptation from procedures in [MNSB06, GN11] which were developed for slightly different models of XSDs.[2]

Lemma 6.6 (Adapted from Lemma 7 in [GN11]) *Each XSD can be translated into an equivalent DFA-based XSD in linear time.*

Proof. Let $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ be an arbitrary XSD. The equivalent DFA-based XSD $(\mathcal{A}, S, \lambda)$ with $\mathcal{A} = (\mathsf{EName}, Q, \delta, q_0)$ is constructed by Algorithm 1. We provide additional explanation for the algorithm. In line 3, $\delta(q_0, a)$ is well-defined thanks to the EDC constraint for XSDs (that states that t is uniquely determined by a). Similarly, in line 4 we have that X fulfills the EDC constraint. Therefore, $\delta(t_1, a)$ is well-defined and $\mathcal{A}$ is guaranteed to be a deterministic automaton. Finally, in line 5, $\mu(\rho(t))$ denotes the regular expression obtained from $\rho(t)$ by replacing every typed element $a[t]$ by the element a. Notice that, since X fulfills the UPA constraint, we have that $\mu(\rho(t))$ is a deterministic regular expression. Therefore, $(\mathcal{A}, S, \lambda)$ is a DFA-based XSD and has deterministic content models. The fact that $(\mathcal{A}, S, \lambda)$ can be constructed from X in linear time is immediate from the algorithm. The equivalence between $(\mathcal{A}, S, \lambda)$ and X is easily seen. □

We now show how to translate DFA-based XSDs into equivalent BXSDs. The translation is similar to the proof of Theorem 7.1 ((a) ⇒ (d)) in [MNSB06].

Lemma 6.7 *Each DFA-based XSD $(\mathcal{A}, S, \lambda)$ can be translated into an equivalent BXSD B with linearly many rules in $|\mathcal{A}|$.*

Proof. Let $(\mathcal{A}, S, \lambda)$ be a DFA-based XSD with $\mathcal{A} = (\mathsf{EName}, Q, \delta, q_0)$. Algorithm 2 specifies how to obtain the equivalent BXSD $B = (\mathsf{EName}, S, R)$. In line 2, the regular expression r_q defines the language of the DFA $\mathcal{A}$ in which q is the only accepting state, i.e., the language of the automaton $(\mathsf{EName}, Q, \delta, q_0, \{q\})$. Since each expression s_q on line 3 is deterministic, the right-hand sides of rules in R are deterministic as well. Finally, R contains the rules $r_q \to s_q$, for each $q \in Q$, in arbitrary order. □

The reason why the ordering of the rules in R in the proof of Lemma 6.7 is not important is that, for each pair of states $q_1 \neq q_2$ from $\mathcal{A}$, we have that $L(r_{q_1}) \cap L(r_{q_2}) = \emptyset$. The latter holds because $\mathcal{A}$ is a DFA. Furthermore, the BXSD B can have regular expressions that are exponentially larger than $|\mathcal{A}|$ in general. This cannot be avoided[3] because $\mathcal{A}$ is a DFA and the worst-case conversion from a DFA to a regular expression is well-known to be exponential [EZ76]. In Section 6.5 we discuss classes of schemas that capture most cases in practice and that do not lead to such a blow-up.

[2] One consequence of the slightly different models of XSDs is that the translation in [GN11] is quadratic, whereas it is linear in our case.

[3] Proving that an exponential blow-up cannot be avoided is more technical than just this observation, see Section 6.6.

Algorithm 1 Translating an XSD to an equivalent DFA-based XSD.

Input: XSD $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$
Output: DFA-based XSD $(\mathcal{A} = (Q, \mathsf{EName}, \delta, q_0), S, \lambda)$ equivalent to X
1: $S := \{a \mid \exists t \in \mathsf{Types} \text{ such that } a[t] \in T_0\}$
2: $Q := \{q_0\} \uplus \mathsf{Types}$
3: For each $a[t] \in T_0$, $\delta(q_0, a) := t$
4: For each $t_1 \in \mathsf{Types}$ and $a \in \mathsf{EName}$ such that $a[t_2]$ occurs in $\rho(t_1)$, $\delta(t_1, a) := t_2$
5: For every $t \in \mathsf{Types}$, $\lambda(t) := \mu(\rho(t))$ ▷ $\mu(\rho(t))$ is obtained from $\rho(t)$ by replacing every $a[t]$ with a

Algorithm 2 Translating a DFA-based XSD into an equivalent BXSD.

Input: DFA-based XSD $(\mathcal{A} = (Q, \mathsf{EName}, \delta, q_0), S, \lambda)$
Output: BXSD $B = (\mathsf{EName}, S, R)$ equivalent to X
1: **for** every state $q \in Q$ **do**
2: $r_q :=$ a regular expression for the DFA $(Q, \mathsf{EName}, \delta, q_0, \{q\})$
3: $s_q := \lambda(q)$
4: $R := r_{q_1} \to s_{q_1}, \ldots, r_{q_n} \to s_{q_n}$, where $\{q_1, \ldots, q_n\} = Q$

Translation from BonXai to XML Schema

The translation from BonXai to XML Schema follows a similar overall outline as the reverse translation. Again, we use DFA-based XSDs as an intermediate representation in the translation. That is, we first translate BXSDs into DFA-based XSDs and translate the latter to XSDs. However, the present translation is more technical than the one before.

We first give the translation steps and we prove later that they are worst-case optimal.

Lemma 6.8 *Each BXSD B can be translated into an equivalent DFA-based XSD $(\mathcal{A}, S, \lambda)$ for which $|\mathcal{A}|$ is at most exponential in $|B|$.*

Proof. Let $B = (\mathsf{EName}, S, R)$ be a BXSD, where $R = r_1 \to s_1, \ldots, r_n \to s_n$. We translate B into $(\mathcal{A}, S, \lambda)$ as described in Algorithm 3. On line 2 we want the DFAs $\mathcal{A}_i = (\mathsf{EName}, Q_i, \delta_i, q_0^i, F_i)$ to be *minimal* and *complete*. Here, a DFA $\mathcal{A}_i$ is *complete* when $\delta_i(q, a)$ is defined for every $q \in Q_i$ and $a \in \mathsf{EName}$. A DFA can be made complete by adding an extra state to which all previously non-defined transitions lead. Furthermore, it is well-known that every regular language has a unique minimal, complete DFA. (Notice that, since regular expressions are exponentially more succinct than deterministic finite automata, $\mathcal{A}_i$ can be exponentially larger than r_i in the worst case.)

The DFA-based XSD $(\mathcal{A}, S, \lambda)$ is then constructed through a product automaton: in line 3, we define $\mathcal{A}$ to be the product $\mathcal{A}_1 \times \cdots \times \mathcal{A}_n$. More precisely, $\mathcal{A} = (Q, \mathsf{EName}, \delta, q_0)$, where $Q = Q_1 \times \cdots \times Q_n$, $q_0 = (q_0^1, \ldots, q_0^n)$ and, for every state $(p_1, \ldots, p_n) \in Q$ and every $a \in \mathsf{EName}$, we have $\delta((p_1, \ldots, p_n), a) = (q_1, \ldots, q_n)$ where, for every i, $\delta(p_i, a) = q_i$. Notice that $\mathcal{A}$ can be exponentially larger than $|B|$ and does not have accepting states.

Algorithm 3 Translating a BXSD to an equivalent DFA-based XSD.

Input: BXSD $B = (\mathsf{EName}, S, R)$, where $R = r_1 \to s_1, \ldots, r_n \to s_n$
Output: DFA-based XSD $(\mathcal{A}, S, \lambda)$ equivalent to B, with $\mathcal{A} = (Q, \Sigma, \delta, q_0)$

1: **for** each $i = 1, \ldots, n$ **do**
2: $\quad \mathcal{A}_i :=$ minimal complete DFA $(Q_i, \mathsf{EName}, \delta_i, q_0^i, F_i)$ for $L(r_i)$
3: $\mathcal{A} := \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ ▷ $\mathcal{A}$ has state set $Q_1 \times \cdots \times Q_n$
4: **for** each $(q_1, \ldots, q_n) \in Q_1 \times \cdots \times Q_n$ **do**
5: $\quad$ **if** $\exists i \in \{1, \ldots, n\}$ such that $q_i \in F_i$ **then**
6: $\quad\quad i :=$ largest number such that $q_i \in F_i$
7: $\quad\quad \lambda((q_1, \ldots, q_n)) := s_i$
8: $\quad$ **else**
9: $\quad\quad \lambda((q_1, \ldots, q_n)) := (\mathsf{EName})^*$

The content models of the DFA-based XSD are defined in lines 7 and 9. Line 7 is the case where at least one of the automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ accepts, i.e., at least one BXSD rule matches. The content model of the relevant state in the DFA-based XSD is then defined to be the content of the highest-priority matching BXSD rule. Line 9 is the case where no BXSD rule matches. Here, according to the definition of BXSDs, every child-string should be allowed. We therefore must allow the content $(\mathsf{EName})^*$. It can be shown that B is equivalent to $(\mathcal{A}, S, \lambda)$. □

It should be noted that Algorithm 3 is optimized for readability and not for efficiency. It is straightforward to change Algorithm 3 such that it only computes reachable states of $\mathcal{A}$. Note that whether a state is reachable also depends on the right-hand sides of the rules, because a transition $\delta(p, a)$, for which the label a does not occur in $\lambda(p)$, can never be taken in a conforming document.

Lemma 6.9 (Adapted from Lemma 7 in [GN11]) *Each DFA-based XSD can be translated into an equivalent XSD in linear time.*

Proof. Let $(\mathcal{A}, S, \lambda)$ be a DFA-based XSD, where $\mathcal{A} = (\mathsf{EName}, Q, q_0, \delta)$. We construct an equivalent XSD $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ in Algorithm 4. In line 4 of the algorithm we implicitly use that $\delta(q, a)$ is non-empty for every state q and every element name a occurring in $\lambda(q)$. □

We note that the resulting XSD can be "minimized" efficiently using a minor adaptation of the minimization algorithm for XSDs from [MN07]. (More formally, it is possible to efficiently produce an XSD such that the set Types is minimal among all equivalent XSDs. Also, the expressions r_q do not become larger.) The difference with the minimization algorithm from [MN07] would be that the deterministic regular expressions r_q should not be minimized. (In fact, minimizing deterministic regular expressions is NP-hard, as we will show in Chapter 7.4.

Algorithm 4 Translating a DFA-based XSD to an equivalent XSD.

Input: DFA-based XSD $(\mathcal{A}, S, \lambda)$ with $\mathcal{A} = (Q, \mathsf{EName}, \delta, q_0)$
Output: XSD $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ equivalent to $(\mathcal{A}, S, \lambda)$
1: $\mathsf{Types} := Q$
2: $T_0 := \{a[\delta(q_0, a)] \mid a \in S, \delta(q_0, a) \neq \emptyset\}$
3: **for** each state $q \in Q$ **do**
4: $\quad r_q :=$ expression obtained from $\lambda(q)$ by replacing each symbol a with $a[\delta(q, a)]$
5: $\quad \rho(q) = r_q$

6.5 Efficient Translations for Fragments

Even though the translations between XSD and BonXai given in the previous Section are provably optimal, as we will see in Section 6.6, they can be exponential in the worst case. In this section, we argue why we do not expect this to be a problem in practice. In particular, we prove that the translation is polynomial for a restriction of XSDs that accounts for the overwhelming majority of schemas in practice. An examination of 225 XSDs from the Web revealed that in more than 98% the content model of an element only depends on the label of the element itself, the label of its parent, and the label of its grandparent [MNSB06]. This motivates the study of the following class of DFA-based XSDs.

Definition 6.10 A DFA-based XSD is *k-suffix*, if the type of an element only depends of the last k symbols of its ancestor string. More precisely, a DFA-based XSD $(\mathcal{A}, S, \lambda)$ with $\mathcal{A} = (Q, \mathsf{EName}, \delta, q_0)$ is k-suffix based if $\mathcal{A}(w_1 a_1 \cdots a_k) = \mathcal{A}(w_2 a_1 \cdots a_k)$ for all strings w_1, w_2 over EName and symbols $a_1, \ldots, a_k \in \mathsf{EName}$.

Hence, 98% of the XSDs in the aforementioned study have a corresponding 3-suffix DFA-based XSD. Actually, this DFA-based XSD can be obtained simply by applying the construction of Lemma 6.6 to the given XSD. Furthermore, according to Lemmas 6.6 and 6.9, the translations between XSDs and DFA-based XSDs are straightforward and very efficient. We therefore do not revisit these constructions and focus on translations between (k-suffix) DFA-based XSDs and BXSDs. The BXSDs corresponding to this class of schemas can be defined as follows.

Definition 6.11 A regular language L is a *suffix language* if $L = \{w\}$ or $L = L(\mathsf{EName}^* w)$ for some word w. It is a *k-suffix language* if, additionally, $|w| \leq k$. A BXSD (EName, S, R) is *k-suffix based* if, for every rule $r \to s$ in R, the left-hand side r is a k-suffix language.

The following theorem considers the translation from k-suffix based BXSDs and k-suffix DFA-based XSDs. It is similar in flavor to Proposition 5.2 in [KS07], but considers rules with a priority system as in BonXai. Kasneci and Schwentick avoided this issue by assuming that rules have pairwise disjoint left-hand-side languages.

Theorem 6.12 *Each k-suffix based BXSD can be translated in polynomial time into an equivalent k-suffix DFA-based XSD of linear size.*

Proof. Let $B = (\mathsf{EName}, S, R)$ be a k-suffix based BXSD with $R = (w_1 \to s_1, \ldots, w_\ell \to s_\ell, //w_{\ell+1} \to s_{\ell+1}, \ldots, //w_n \to s_n)$. So, each string $w_1, \ldots, w_n$ has length at most k.

The equivalent k-suffix DFA-based XSD $D = (\mathcal{A}, S, \lambda)$ with $\mathcal{A} = (Q, \mathsf{EName}, \delta, q_\varepsilon)$ can be defined as follows. Let $P = \{w \mid \exists \text{ string } v \text{ over } \mathsf{EName} \text{ for which } wv \in \{w_1, \ldots, w_n\}\}$ be the set of prefixes of all w_i and let $Q := \{(q_w, j) \mid w \in P, j \in \{0, 1\}\}$ be a set of states representing all prefixes and indicating whether the "current prefix" is still a prefix of the whole word. Then we define

$$\delta((q_w, j), a) = \begin{cases} (q_v, j) & \text{if } wa = v \\ (q_v, 1) & \text{otherwise} \end{cases}$$

where v is the longest suffix of wa in P. Furthermore we let $\lambda((w, 1)) = s_i$, where $i > \ell$ is the highest index such that w_i is a suffix of w, and $\lambda((w, 0)) = s_i$, where $i \leq \ell$ is the highest index such that $w_i = w$. The construction of D from B is easily seen to be polynomial. Equivalence between B and $\mathcal{A}$ can be immediately seen since $\mathcal{A}$ follows the standard approach for pattern matching with automata. Furthermore, D fulfills the k-suffix property by definition. □

We now consider the reverse direction. An important difference with Theorem 6.12 is that this direction is exponential in k, that is, it needs k to be constant in order to be polynomial. However, as we noted before, in 98% of the schemas occurring in the practical study of [MNSB06], we see that $k \leq 3$.

Theorem 6.13 *Let k be a constant. Each k-suffix DFA-based XSD can be translated in polynomial time into an equivalent k-suffix based BXSD.*

Proof. Let $D = (\mathcal{A}, S, \lambda)$ with $\mathcal{A} = (Q, \mathsf{EName}, \delta, q_0)$ be a k-suffix DFA-based XSD. The BXSD $B = (\mathsf{EName}, S, R)$, where B consists of the rules

$$//a_1/a_2/\ldots/a_k \to \alpha, \text{ for which } \lambda(\delta(q_0, a_1a_2\ldots a_k)) = \alpha, \text{ and}$$

$$/a_1/a_2/\ldots/a_\ell \to \alpha, \text{ for which } \ell < k \text{ and } \lambda(\delta(q_0, a_1a_2\ldots a_\ell)) = \alpha.$$

Note that the ordering of the rules does not matter as the ancestor patterns describe pair-wise disjoint languages, where the first kind of rules describes all elements of depth at least k and the second kind of rules describes all elements at depths less than k.

By construction it is obvious that the BXSD B is equivalent to D, as D is k-suffix based. It is easy to see that B contains less than $|\mathsf{EName}|^{(k+1)}$ rules and that B can be computed in polynomial time if k is fixed. □

Finally, we note that it is easy to decide if a given XSD can be translated efficiently into a BXSD, i.e., whether it corresponds to a k-suffix DFA-based XSD (where k can either be fixed in advance or not). Questions of this kind were investigated in [CMM13].

6.6 Worst-Case Optimality of the Translation Algorithms

We now prove that both translation algorithms are worst-case optimal wrt. the size of the resulting schemas. In particular, we show that both conversions from Section 6.4 can lead to exponential size blow-ups in general, i.e. when not restricting to the fragments given in Section 6.5.

From BonXai to XML Schema

We prove that the translation from BXSDs to XSDs is worst-case optimal.

Theorem 6.14 *There exists a family of BXSDs $(B_n)_{n\in\mathbb{N}}$ such that, for each n, the BXSD B_n has size $O(n)$ but the smallest XSD equivalent to B_n has size at least 2^n.*

Proof sketch. Let $n \in \mathbb{N}$ be arbitrary. Let $B_n = (\mathsf{EName}_n, S_n, R_n)$ be the BXSD with $\mathsf{EName}_n = \{a, a_1, \ldots, a_n, b_1, \ldots, b_n\}$, $S_n = \{a_1, \ldots, a_n\}$, and R_n consisting of the following rules:

$$\begin{array}{rcl}
//a & \to & \varepsilon \\
//(b_1 + \cdots + b_n) & \to & \varepsilon \\
//(a_1 + \cdots + a_n) & \to & (a + a_1 + \cdots + a_n) \\
//a_1//a_1//a & \to & b_1 \\
//a_2//a_2//a & \to & b_2 \\
\vdots & \vdots & \vdots \\
//a_n//a_n//a & \to & b_n
\end{array}$$

Again we wrote the regular expressions on the left-hand-side of rules with // as an abbreviation for EName^*. This schema defines a set of unary (i.e., non-branching) trees and its semantics is the following. If the ancestor path of an a-element contains, for each $1 \leq i \leq n$, at most one a_i element, its content model is ε. Otherwise, if j is the largest number such that a_j occurs at least two times on the path to the a element, then this a element has b_j as a child.

It can be proved with techniques from [MN07] that the smallest XSD equivalent to the above BXSD is exponentially large in n. Intuitively, in order to decide which b_i is the child under an a, the types of the XSD needs to keep track of the largest j, for which a_j has already occurred twice, and, worse, the set of $i > j$, for which a_i has already occurred once. □

From XML Schema to BonXai

When converting an XML Schema (XSD) to a BonXai Schema Definition (BXSD) using the procedures in Lemmas 6.6 and 6.7 it is possible that the BXSD is exponentially larger than the XSD. The source of this exponential blow-up lies in Algorithm 2 which is used in Lemma 6.7. More precisely, line 1 constructs a regular expression equivalent to a DFA, which is well known to be exponential in the worst case [EZ76].

We will now show that this blow-up cannot be avoided in general, which means that, in this sense, our conversion algorithm is worst-case optimal. Recall, however, that our

conversion which we showed in Lemma 6.7 does not produce a large number of rules in the BXSD. Indeed, if the DFAs that Algorithm 2 encounters on line 2 only produce polynomially large regular expressions, then the whole conversion is polynomial as well. We discussed a particularly relevant such case in Section 6.5.

The proof of the following Theorem is rather technical. It is based on the proof in [EZ76]. The hard part of our proof is to show that the exponential blowup cannot be avoided by a clever use of the priorities in BonXai.

Theorem 6.15 *There exists a family $(X_n)_{n\in\mathbb{N}}$ of XSDs such that, for each n, X_n has size $O(n^2)$ but the smallest BXSD equivalent to X_n has size at least $2^{\Omega(n)}$.*

Before we can give the proof of Theorem 6.15, we need a lemma that bounds the size of regular expressions for left derivatives of languages (left derivatives were defined by Brzozowski [Brz64]). To this end, the *left derivative of a string language L with respect to a string w*, denoted by $\partial_w L$, is defined as

$$\partial_w L = \{v \mid wv \in L\}.$$

The *left derivative of a language L with respect to a language X*, denoted by $\partial_X L$, is defined as

$$\partial_X L = \{v \mid \exists w \in X \text{ such that } wv \in L\}.$$

We denote by $\text{depth}(\alpha)$ the depth of the parse tree for α.

Lemma 6.16 *Let α be a regular expression and X be an arbitrary language. Then there exists a regular expression α' for the language $\partial_X L(\alpha)$, such that $|\alpha'| \in O(\text{depth}(\alpha)|\alpha|)$.*

Proof. For a language L, let $\text{prefix}(L) = \{v \mid \exists w.\ vw \in L\}$ be the set of all prefixes of strings in L. We construct α' inductively as follows.

$$\begin{aligned}
\partial_X \emptyset &= \emptyset \\
\partial_X \varepsilon &= \begin{cases} \varepsilon & \text{if } \varepsilon \in X \\ \emptyset & \text{otherwise} \end{cases} \\
\partial_X a &= \begin{cases} \varepsilon + a & \text{if } X \cap \{\varepsilon, a\} = \{\varepsilon, a\} \\ a & \text{if } X \cap \{\varepsilon, a\} = \{\varepsilon\} \\ \varepsilon & \text{if } X \cap \{\varepsilon, a\} = \{a\} \\ \emptyset & \text{otherwise} \end{cases} \\
\partial_X (\alpha_1 + \alpha_2) &= \partial_X \alpha_1 + \partial_X \alpha_2 \\
\partial_X (\alpha_1 \cdot \alpha_2) &= (\partial_{X_1} \alpha_1) \cdot \alpha_2 + \partial_{\partial_{L(\alpha_1)} X} \alpha_2 \\
\partial_X \alpha^* &= (\partial_{\partial_{L(\alpha^*)} X} (\varepsilon + \alpha)) \cdot \alpha^*
\end{aligned}$$

It can be shown by a straightforward induction that all inductive definitions given above are correct.

It remains to show that $|\alpha'| \leq \text{depth}(\alpha)|\alpha|$. We show $|\alpha'| \leq \text{depth}(\alpha)|\alpha|$ by an induction on the structure of α. For the induction base case, we observe that $|\alpha| = |\alpha'| = 1$ in

the cases where $|\alpha|$ is an atomic expression. Applying the induction hypothesis to the equations above gives us

- $|\alpha'| \leq \text{depth}(\alpha_1)|\alpha_1| + \text{depth}(\alpha_2)|\alpha_2|$ in the case $\alpha = \alpha_1 + \alpha_2$;
- $|\alpha'| \leq \text{depth}(\alpha_1)|\alpha_1| + |\alpha_2| + \text{depth}(\alpha_2)|\alpha_2|$ in the case $\alpha = \alpha_1 \cdot \alpha_2$; and
- $|\alpha'| \leq (\text{depth}(\alpha) - 1)|\alpha_1| + |\alpha_1|$ in the case $\alpha = \alpha_1^*$.

In all three cases we can conclude $|\alpha'| \leq \text{depth}(\alpha)|\alpha|$ using the fact that $|\alpha_1| + |\alpha_2| \leq |\alpha|$ and $\max(\text{depth}(\alpha_1), \text{depth}(\alpha_2)) = \text{depth}(\alpha) - 1$. This concludes the proof. □

Now we are ready to prove Theorem 6.15.

Proof of Theorem 6.15. We leverage a technique by Ehrenfeucht and Zeiger [EZ76], who showed that there exists a class of languages $(Z_n)_{n\in\mathbb{N}}$, such that Z_n can be accepted by a DFA of size $O(n^2)$ but cannot be defined by a regular expression of size smaller than 2^{n-1}.

For every $n \in \mathbb{N}$ we let $\Sigma_n = \{a_{ij} \mid i, j \in \{1, \ldots, n\}\}$. We call i the *source* and j the *target* of a symbol a_{ij}. We define Z_n as

$$Z_n = \{w_1 \cdots w_m \in \Sigma_n^* \mid \forall i \in \{1, \ldots, m-1\}, \exists j, k, l \text{ such that } w_i w_{i+1} = a_{jk} a_{kl}\}.$$

That is, in every word in Z_n, the target of a symbol and the source of the following symbol must be equal. Every word $w \in \Sigma_n^* \setminus Z_n$ has a first symbol $a_{i\ell}$ whose target ℓ does not coincide with the source of the following symbol. We call ℓ the error index of w.

We now construct a family $(X_n)_{n\in\mathbb{N}}$ of XSDs, such that X_n is of size $O(n^2)$ and the smallest BXSD equivalent to X_n has size $2^{\Omega(n)}$. We define X_n by its DFA-based XSD $(\mathcal{A}_n, S_n, \lambda_n)$. To this end, we let $S_n = \Sigma_n$ and choose the components of $\mathcal{A}_n = (Q \cup Q', \Sigma_n, \delta, q_1)$ as follows.

- $Q = \{q_i \mid 1 \leq i \leq n\}$ and $Q' = \{q'_i \mid 1 \leq i \leq n\}$;
- for every $q_i \in Q$ and $a_{j\ell} \in \Sigma$, $\delta(q_i, a_{j\ell}) = \begin{cases} q_\ell & \text{if } i = j \\ q'_i & \text{if } i \neq j \end{cases}$
- and, for every $q'_i \in Q'$ and $a_{j\ell} \in \Sigma$, $\delta(q'_i, a_{j\ell}) = q'_i$,
- for every $q_i \in Q$, $\lambda(q_i) = \varepsilon \cup \Sigma$,
- for every $q'_\ell \in Q$, $\lambda(q'_\ell) = \varepsilon \cup \Sigma \cup \{a_{\ell\ell} a_{\ell\ell}\}$.

In other words, $\mathcal{A}_n$ is a DFA that tests whether a word is in Z_n and remembers, for words not in Z_n, their error index.

The documents valid with respect to X_n are thus characterized by the following two properties.

- All label sequences over Σ_n are allowed in paths.

- The only allowed kind of branching is binary branching of the form $a_{ij} \to a_{\ell\ell}a_{\ell\ell}$ below nodes whose ancestor path contains a Z_n-error with error index ℓ.

We note that, as branching can only take place below an error, and the first error of a path is unique, in every document there can be binary branching $a_{\ell\ell}a_{\ell\ell}$ with at most one kind of symbols.

It is straightforward that X_n is of size $O(n^2)$. To show that every BXSD B equivalent to $(\mathcal{A}_n, S_n, \lambda_n)$ is of size $2^{\Omega(n)}$ we prove that B must have at least one ancestor pattern of size $2^{\Omega(n)}$.As already mentioned, it is known from [EZ76] that every regular expression for Z_n is of size $2^{\Omega(n)}$. Actually Ehrenfeucht and Zeiger prove a stronger result:

Proposition 6.17 ([EZ76, Theorem 4.1]) *For every $n \in \mathbb{N}$, there is a string $g \in Z_n$, such that every regular expression α with $vgw \in L(\alpha)$ for some v and w and $L(\alpha) \subseteq Z_n$ is of size $2^{\Omega(n)}$.*

For our purposes, we need a slightly stronger version:

Proposition 6.18 *For every $n \in \mathbb{N}$, there are strings $g_1, \ldots, g_n \in Z_n$ such that $h = g_1 g_2 \ldots g_n \in Z_n$ and for every $i \in \{1, \ldots, n\}$,*

- *g_i contains no symbol from $\{a_{1i}, \ldots, a_{ni}\}$; and*
- *every regular expression α_i with $vg_iw \in L(\alpha_i)$ for some v and w and $L(\alpha_i) \subseteq Z_n$ is of size $2^{\Omega(n)}$.*

Proof of Proposition 6.18. First we note that Proposition 6.17 still holds, if we replace the condition $L(\alpha) \subseteq Z_n$ by $L(\alpha) \subseteq Z_m$, for any $m > n$. This is because symbols outside Σ_n are useless for strings from Z_n, and therefore any regular expression for Z_n over Σ_m could be translated into an expression of (at most) the same size over Σ_n by replacing every symbol outside Σ_n with $\emptyset$.

By the same kind of reasoning it follows that, for every $i \in \{1, \ldots, n\}$, Proposition 6.17 also holds with respect to strings in Z_n over $\Sigma_n^{(i)} = \Sigma_n \setminus \{a_{ij}, a_{ji} \mid j \leq n\}$ and expressions over Σ_n. Let thus, for every i, $h_i \in Z_n$ be a string over $\Sigma_n^{(i)}$ such that every regular expression α with $v_i h_i w_i \in L(\alpha)$ for some v_i and w_i and $L(\alpha) \subseteq Z_n$ is of size $2^{\Omega(n)}$. By choosing v_i and w_i as suitable one-letter strings we obtain strings $g_i = v_i h_i w_i$ with the stated properties. □

Let now B be a BXSD for $(\mathcal{A}_n, S_n, \lambda_n)$. Our goal is to show that B has at least one ancestor pattern of size $2^{\Omega(n)}$. We can assume w.l.o.g. that B does not contain any rule with a child expression allowing content models $a_{ii}a_{ii}$ and $a_{jj}a_{jj}$, for $i \neq j$. To this end, let us assume such a rule α exists and there is a string $z = a_1 \ldots a_m$ matching the left hand side of α such that some document in $L(B)$ contains z as its ancestor path. If no such z exists, α can be deleted from B without changing its language. On the other hand, if such a document exists, α allows the document in which below the z-path two leaves labeled a_{ii} occur and the document in which below the z-path two leaves labeled a_{jj} occur, contradicting the definition of the language of X_n.

We call any rule allowing a content model $a_{ii}a_{ii}$ a t_i-rule and any other rule a t-rule. We emphasize that, as we just showed, a rule can only be a t_i-rule, for *one* index i.

We consider strings (as ancestor paths) from Z_n of the form $s = h^k s'$, with h from Proposition 6.18, $k \geq 1$ and $s' \in \Sigma_n^*$. Clearly, strings can be matched by several rules, but for each string s, B must have a last rule $r_s : \alpha_s \to \beta_s$ whose left hand side matches s. However, several strings can possibly share the same last rule.

Let, for every such s,

$$\alpha'_s = \partial_{h^{k-1} g_1 \dots g_j}\ \alpha_s,$$

where $j = 0$ if α_s is a t-rule and $j = i - 1$ if α_s is a t_i-rule. We note that $g_{j+1} \dots g_n s' \in L(\alpha'_s)$ by construction. By Lemma 6.16, it follows that $|\alpha'_s| = O(|\alpha_s|^2)$ and therefore $|\alpha_s| = \Omega(\sqrt{|\alpha'_s|})$.

For each string $s = h^k s' \in \Sigma_n^*$ one of the following conditions must hold, for some $\ell \in \{1, \dots, n\}$.

(1a) $L(\alpha'_s) \subseteq Z_n$.

(1b) $L(\alpha'_s) \not\subseteq Z_n$, r_s is a t_ℓ-rule, and every string in $L(\alpha'_s) \setminus Z_n$ has error index ℓ.

(2a) $L(\alpha'_s) \not\subseteq Z_n$, r_s is a t_ℓ-rule, and there exists a string in $L(\alpha'_s) \setminus Z_n$ with error index $j \neq \ell$.

(2b) $L(\alpha'_s) \not\subseteq Z_n$ and r_s is a t-rule.

Let us assume first that, for some $s = h^k s' \in \Sigma_n^*$, one of the cases (1a) or (1b) holds.

In case (1a), we can conclude from Proposition 6.18 that α'_s is of size $2^{\Omega(n)}$. Therefore α_s is of size $\sqrt{2^{\Omega(n)}} = 2^{\Omega(n)}$.

In case (1b), we construct a regular expression γ from α'_s by replacing each occurrence of a symbol $a_{i\ell}$ with $i \in \{1, \dots, n\}$ by $\emptyset$. By construction, γ has the following properties:

- $|\gamma| \leq |\alpha'_s|$;
- $L(\gamma) \subseteq Z_n$, since every string in $L(\alpha'_s) \setminus Z_n$ has a symbol $a_{i\ell}$ for some i; and
- $g_\ell \in L(\gamma)$, as $g_\ell \in L(\alpha'_s)$ and g_ℓ contains no symbol $a_{i\ell}$ by definition.

We can conclude from Proposition 6.18, that γ and therefore α'_s is of size $2^{\Omega(n)}$. We can conclude again that α_s is of size $2^{\Omega(n)}$, as well.

We can thus assume from now on that, for every $s = h^k s' \in \Sigma_n^*$, one of the cases (2a) or (2b) applies. We are going to show next that this implies that the number of rules in B must be infinite, a contradiction from which we can conclude the statement of the theorem. More precisely, we show that for each string of the form $s = h^k s' \in \Sigma_n^*$, there is a string $z = h^{k-1} z' \in \Sigma_n^*$ such that r_z comes *strictly after* r_s in the list of rules of B. Clearly, repeated application of this statement yields a sequence of at least k rules with ascending indexes. As the process can be started with an arbitrary k, we get the desired contradiction.

Let thus $s = h^k s' \in \Sigma_n^*$, for some $k \geq 1$. By our assumption, either condition (2a) or (2b) holds for α'_s.

We first consider the case that r_s is a t_ℓ-rule, for some $\ell \in \{1, \ldots, n\}$ and (2a) holds with some string $w \in L(\alpha'_s) \setminus Z_n$ with error index $j \neq \ell$. Let us assume towards a contradiction that r_s is the last rule (in the order of rules) matching $z = h^{k-1}g_1 \ldots g_{\ell-1}w$. Then the document consisting of a path with label sequence z arriving at some node v with two leaf children labeled by $a_{\ell\ell}$ below v, is valid for B, a contradiction as the error index of z is not ℓ. Therefore, there must be another rule in B *after* r_s whose left hand side matches z and whose right hand side does *not* allow the content model $a_{\ell\ell}a_{\ell\ell}$.

We next consider the remaining case that r_s is a t-rule and (2b) holds. Let $w \in L(\alpha'_s) \setminus Z_n$ with some error index j and let us assume towards a contradiction that r_s is the last rule matching $z = h^{k-1}w$. Then the document consisting of a path with label sequence z arriving at a node v with two leaf children labeled by a_{jj} is not valid for B, a contradiction.

Therefore, again there must be another rule in B *after* r_s whose left hand side matches z and whose right hand side *allows* the content model $a_{jj}a_{jj}$.

Thus, we have shown that for each string of the form $s = h^k s' \in \Sigma_n^*$, there is a string $z = h^{k-1}z' \in \Sigma_n^*$ such that r_z comes *strictly after* r_s in the list of rules of B, and we are done.

This completes the proof that B has size $2^{\Omega(n)}$. □

6.7 Further Research on the BonXai Schema Language

In Chapter 5, we introduced the BonXai schema language, which should aid in developing good schemas for XML documents and databases. Despite we have given a profound theoretical background for the BonXai schema language in the current chapter, the language is still a working draft, which requires further work. For a fully functional schema language at least support for specifying simple types is missing. Other required and/or desirable features need to be identified by actual XML designers. Therefore, the BonXai editor and the underlying FoXLib library that will be presented in Chapter 13 need to be deployed to a wider audience. Until now, only a few university researchers have used the software.

Additionally, also the theory behind BonXai can be improved. While we have presented fragments of BonXai and XML Schema that can be efficiently converted back and forth, we would like to have better algorithms for converting more general schemas. Czerwinski et al. [CMM13] have researched the separability problem of regular languages. To create a BonXai schema, we do not need exact representations of the regular languages in the left-hand sides of the rules. It is sufficient, if we can separate the languages used in different rules. Czerwinski et al. concentrated on the decision problem (Can two languages be separated using a simpler language?). It is not immediately clear how to compute the separating language itself. Further research in this direction might lead to nicer BonXai schemas, which use simpler expressions in the left-hand sides of the rules.

7 Deterministic Regular Expressions

In this chapter we will have a closer look on the Unique Particle Attribution enforced on all DTD and XML Schema definitions. The BonXai schema specification language, we have introduced in Chapter 5 inherited this restriction from XML Schema to be compatible with XML Schema.

In the literature this kind of regular expressions is known as one-unambiguous regular expressions or deterministic regular expressions (DREs) [BKW98, BGMN09]. We will continue to use the term deterministic regular expression.

Intuitively, a regular expression is deterministic when the following holds. When reading the input string from left to right, the expression always allows to match each symbol of that string uniquely against a position in the expression, without looking ahead.

Formally, let $\bar{r}$ stand for the regular expression obtained from r by annotating every alphabet symbol with its position in the expression. For example, for $r = b^*a(b^*a)^*$ we have $\bar{r} = b_1^*a_2(b_3^*a_4)^*$. A regular expression r is (weakly) *deterministic* if there are no strings wa_iv and wa_jv' in $L(\bar{r})$ such that $i \neq j$ for no $a \in \Sigma$. We denote the class of deterministic regular expressions (without counters) by DRE. For an overview over different classes of (deterministic) regular expressions, we direct the reader to the next section.

The expression $(a + b)^*a$ is not deterministic as already the first symbol in the string aaa could be matched by either the first or the second a in the expression. The equivalent expression $b^*a(b^*a)^*$, on the other hand, is deterministic. Brüggemann-Klein and Wood showed that not every (non-deterministic) regular expression is equivalent to a deterministic one [BKW98]. Thus, semantically, not every regular language can be defined with a deterministic regular expression.

We call a regular language *DRE-definable* if there exists a deterministic regular expression defining it. The classical example for a regular language that is not DRE-definable is $(a + b)^*a(a + b)$.

In Section 7.1, we will explain the differences between several classes of regular expressions that have been investigated in the literature. Section 7.2 looks at the problem given a language L, can this language be defined with deterministic regular expressions. We therefore introduce the orbit property originally introduced by Brüggemann-Klein and Wood [BKW98], which is a powerful tool to show that some regular language cannot be described by deterministic regular expressions. We give a summary over known results on closure properties and descriptional complexity of deterministic regular expressions in Section 7.3. Finally, we show in Section 7.4 that minimization of deterministic regular expressions is NP-complete.

7.1 Weak vs. Strong Determinism

In the literature, additional to (weakly) deterministic regular expressions, as they are used in W3C standards, strongly deterministic regular expressions are considered [GGM12].

A regular expression is strongly deterministic, if it is deterministic and additionally for every sub-expression r^* or $r^{[n,m]}$ that has as topmost operation a Kleene star or a counter, it holds that $\text{first}(r) \cap \text{followlast}(r) = \emptyset$.

Intuitively this restriction enforces that it is always clear which Kleene star or counter is used if the previous input symbol was matched by some symbol a_i of the expression and the current symbol is matched by a symbol symbol b_j with $j \leq i$.

The simplest regular expression, which is not strongly deterministic, is $(a^*)^*$ as it is not clear whether the inner or outer Kleene star should be used. Formally, it can be observed that $\text{first}(a^*) \cap \text{followlast}(a^*) = \{a\} \neq \emptyset$. The expression $(a^{[2,3]})^{[2,3]}$ is not strongly deterministic as it is not clear which counter to increment when reading the third a in the string $aaaa$. The equivalent expression $a^{[4,9]}$ is strongly deterministic.

With RE, $\text{RE}^{\#}$, DRE_w, $\text{DRE}_\text{w}^{\#}$, DRE_s, $\text{DRE}_\text{s}^{\#}$ we denote the different classes of (deterministic) regular expressions, where # denotes a class with counters and the subscripts w and s denote weak and strong determinism respectively.

Document Type Definitions use expressions from DRE_w and XML Schema definitions use expressions from $\text{DRE}_\text{w}^{\#}$ for the description of their content models. If we use the term deterministic regular expression or DRE without further specification, we always refer to DRE_w.

Let $\mathcal{L}(X)$ be the class of languages that can be expressed with X, where X is a class of regular expressions. The following theorem characterizes the relative expressiveness of the considered classes of regular expressions.

Theorem 7.1 ([BKW98, GGM12]) *It holds that*

$$\mathcal{L}(DRE_w) = \mathcal{L}(DRE_s) = \mathcal{L}(DRE_s^{\#}) \subsetneq \mathcal{L}(DRE_w^{\#}) \subsetneq \mathcal{L}(RE) = \mathcal{L}(RE^{\#}).$$

We want to shortly explain all inequivalences and equivalences in this theorem. For a more detailed analysis of weak and strong determinism, we refer to [GGM12].

The canonical expression showing that $\mathcal{L}(\text{DRE}_\text{w}) \neq \mathcal{L}(\text{DRE}_\text{w}^{\#})$ is $(a^{[2-3]}(b+\varepsilon))^*$. Obviously this expression is in $\text{DRE}_\text{w}^{\#}$ (as no alphabet symbol occurs twice). However there is no equivalent expression in DRE_w. This can be shown by applying Algorithm 5, which is described below.

It is known that every language over a unary alphabet that can be described by an expression from $\text{DRE}_\text{w}^{\#}$ can also be described by an expression from DRE_w. Therefore the language $L((aaa)(a+\varepsilon))$ which is not in $\mathcal{L}(\text{DRE}_\text{w})$ (this can again be proved with Algorithm 5) shows that $\mathcal{L}(\text{DRE}_\text{w}^{\#}) \neq \mathcal{L}(RE)$.

A regular expression $r \in L(\text{RE}^{\#})$ can be converted to a regular expression without counters by replacing every sub-expression $r^{[n,*]}$ with $r^n r^*$ and every sub-expression $r^{[n,m]}$ with $r^n(\varepsilon + r(\varepsilon + r(\ldots)))$ with $m-n$ occurrences of $(\varepsilon + r \ldots)$. The same construction works to convert expressions from $\text{DRE}_\text{s}^{\#}$ to expressions from DRE_s. The definition of

	RE	$\mathrm{DRE}_{\mathrm{w}}^{\#}$	$\mathrm{DRE}_{\mathrm{w}}$, $\mathrm{DRE}_{\mathrm{s}}$, $\mathrm{DRE}_{\mathrm{s}}^{\#}$
DFA	•	unknown	$\in$ PTIME [BKW98]
DFA with logsize alphabet	•	unknown	$\in$ NLOGSPACE [LBC14]
NFA, RE	•	unknown	PSPACE-C [CDLM13, LBC14, BGMN09]
$\mathrm{RE}^{\#}$	•	unknown	EXPSPACE-C [CDLM13]
$\mathrm{DRE}_{\mathrm{w}}^{\#}$	•	•	$\in$ EXPSPACE
$\mathrm{DRE}_{\mathrm{w}}$, $\mathrm{DRE}_{\mathrm{s}}$, $\mathrm{DRE}_{\mathrm{s}}^{\#}$	•	•	•

Table 7.1: Complexities for the problem given a language using the formalism on the right, can the language be expressed using the formalism on the top.

strong determinism ensures that the resulting expression is still strongly deterministic. This shows the equivalence $\mathcal{L}(\mathrm{DRE}_{\mathrm{s}}) = \mathcal{L}(\mathrm{DRE}_{\mathrm{s}}^{\#})$.

The equivalence $\mathcal{L}(\mathrm{DRE}_{\mathrm{w}}) = \mathcal{L}(\mathrm{DRE}_{\mathrm{s}})$ stems from the fact that every expression from $\mathcal{L}(\mathrm{DRE}_{\mathrm{w}})$ can be converted to star normal form[1] and every expression in star normal form is strongly deterministic.

7.2 Orbit Property and DRE Definability

Given a regular expression or a finite automaton, it is a natural problem to compute an equivalent regular expression which lies inside some class X or to ask whether such an expression exists at all.

In Table 7.1, we have summarized known complexities for the decision problem given a regular expression r or finite automaton $\mathcal{A}$ in class X does there exist a regular expression r' in class Y, such that $L(r) = L(r')$. In cases where $\mathcal{L}(X) \subseteq \mathcal{L}(Y)$ the answer is trivially true. These cases are marked with • in the table. The EXPSPACE upper bounds are by conversion to a regular expression without counters, which gives an exponential blowup.

It should be stressed that the class $\mathrm{DRE}_{\mathrm{w}}^{\#}$, which is used in XML Schema definitions, needs further research. In particular it is still not known whether it is decidable given a regular language L (as regular expression or finite automata), whether L is in $\mathcal{L}(\mathrm{DRE}_{\mathrm{w}}^{\#})$. As there are little to no results available for this class, we will concentrate on the other classes of deterministic regular expressions.

Brüggemann-Klein and Wood have shown that the class of DRE-definable languages is a strict subset of the regular languages [BKW98]. Towards this proof they have introduced the orbit property.

For a state q, the *orbit of* q, denoted $\mathcal{O}(q)$, is the strongly connected component of A that contains q. We call q a *gate* of $\mathcal{O}(q)$ if q is final, or q has an outgoing transition that leaves $\mathcal{O}(q)$. With $\mathcal{G}(q)$ we denote the set of all gates of $\mathcal{O}(q)$.

[1] The star normal form and the conversion of weakly deterministic regular expressions into star normal form are described in [BK92].

An automaton $\mathcal{A}$ has *the orbit property* if, for every pair of gates q_1, q_2 in the same orbit the following properties hold:

1. q_1 is final if and only if q_2 is final; and,
2. for all states q outside the orbit of q_1 and q_2, there is a transition (q_1, a, q) if and only if there is a transition (q_2, a, q).

Towards a decision algorithm for DRE-definability, we need some additional notation. The set of consistent symbols of an orbit $\mathcal{O}(q)$, denoted by S_q is the set

$$S_q = \{a \mid \exists q' \in \mathcal{O}(q).\ \forall q'' \in \mathcal{G}(q).\ \delta(q'', a) = q'\}.$$

With other words, starting from every gate q, a consistent symbol a always reaches the same state.

The cut-automaton $\mathcal{A}_S$ of an automaton $\mathcal{A}$ with only one orbit is derived from $\mathcal{A}$ by removing all transitions starting in a final state labeled with a consistent symbol. Note that in an automaton with only one orbit the gates coincide with the final states.

The *orbit language of* q is the language defined by the sub-automaton of $\mathcal{A}$ consisting of the orbit of q in which the initial state is q and the final states are the gates of $\mathcal{O}(q)$. The *orbit languages of* $\mathcal{A}$ are all orbit languages of q for all states q of $\mathcal{A}$.

The following theorem combines several results from [BKW98].

Theorem 7.2 ([BKW98])

(a) Not every regular language is DRE-definable.

(b) Let $\mathcal{A}$ be a minimal DFA. Then $L(\mathcal{A})$ is DRE-definable if and only if $\mathcal{A}$ has the orbit property and all orbit languages of $\mathcal{A}$ are DRE-definable. If $\mathcal{A}$ consists exactly of one orbit, $\mathcal{A}$ is DRE-definable if and only if the set S of consistent symbols is not empty and $L(\mathcal{A}_S)$ is DRE-definable.

(c) Let $\mathcal{A}$ be a DFA. Then it is decidable in quadratic time in the size of $\mathcal{A}$, whether the language of $\mathcal{A}$ is DRE-definable.

A consequence from Theorem 7.2 is that minimal DFAs which do not fulfill the orbit property cannot describe DRE-definable languages. Showing that the minimal DFA for some language does not fulfill the orbit property is therefore a canonical way of showing that a language is not DRE-definable.

Theorem 7.2 directly gives a decision algorithm for testing DRE-definability, which we depicted as Algorithm 5. This algorithm can be modified such that it computes a deterministic regular expression for $L(\mathcal{A})$ of at most exponential size in the case where $L(\mathcal{A})$ is DRE-definable [BKW98].

Building on top of the work from [BKW98], [CDLM13] and [LBC14] showed independently of each other that it is decidable in polynomial space, whether a language given by an NFA is DRE-definable.

Algorithm 5 BKW-Algorithm to test DRE-definability

```
1: function BKW(A = (Σ, Q, δ, q_0, Q_F))
2:     if A does not have the orbit property then reject
3:     if A has exactly one orbit then
4:         S := {a | ∃q' ∈ O(q).∀q'' ∈ G(q).δ(q'', a) = q'}
5:         if S = ∅ then reject
6:         if BKW(A_S) rejects then reject
7:     else
8:         for each orbit O of A do
9:             if BKW(A_O) rejects then reject
10:    accept
```

Theorem 7.3 ([CDLM13, LBC14]) *It is decidable in polynomial space in the size of a given NFA $\mathcal{A}$, whether $L(\mathcal{A})$ is DRE-definable.*

Furthermore, [LBC14] also gives an improved upper bound for DRE-definability of languages given by DFAs with small alphabets.

Theorem 7.4 ([LBC14]) *It is decidable in non-deterministic logarithmic space in the size of a given DFA $\mathcal{A}$ with alphabet Σ such that $|\Sigma| \in O(\log(|\mathcal{A}|))$, whether $L(\mathcal{A})$ is DRE-definable.*

7.3 Closure Properties and Descriptional Complexity of DREs

Unlike many other classes of languages, especially unlike regular languages, DRE-definable languages are not closed under many of the usually considered operations.

It has been observed that DRE-definable languages are not closed under union [BKW98], intersection [CHM11] or complement [GN08]. DRE-definable languages are also not closed under concatenation [BKW98], reversal[2] (the reverse of $L((a+b)a(a+b)^*)$ is not DRE-definable) or Kleene star [BKW98]. These results hold for alphabets with at least two symbols. For unary alphabets, the picture is a bit different. DRE-definable languages over a unary alphabet are trivially closed under reversal (the language does not change) and they are closed under intersection and Kleene star, as was shown in [LMN12]. They are not closed under union, concatenation and complement.

Even if the DRE-definable languages describe only a subset of the regular languages, it is nice to know which blow-ups to expect when converting a regular language given by a finite automaton or regular expression to a deterministic regular expression. It is also interesting to know which blow-ups to expect when applying certain operations on deterministic regular expressions, at least in the cases where the result is DRE-definable.

[2]The reversal of a language L is the set of strings $\{a_n \cdots a_1 \mid a_1 \cdots a_n \in L\}$.

			Finite Languages		Infinite Languages	
RE	DRE	DFA	Case exists?	Ref	Case exists?	Ref
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	yes	trivial	yes	trivial
$\Theta(n)$	$2^{\Omega(n)}$	$2^{\Omega(n)}$	yes	[KW80, BKW98]	yes	[LMN12]
$2^{\Omega(n)}$	$2^{\Omega(n)}$	$\Theta(n)$	no	[EKSW04]	?	
$\Theta(n)$	$2^{\Omega(n)}$	$\Theta(n)$	yes	unpublished[3]	yes	[LMN12]
$\Omega(n^{\log n})$	$\Omega(n^{\log n})$	$\Theta(n)$	yes	[GJ08]	?	

Table 7.2: Descriptional Complexity for DREs

These questions where investigated by Katja Losemann in her diploma thesis [Los10] and in continuation in [LMN12].

We present the known results for blow-ups caused by conversion between REs, DFAs and DREs in Table 7.2. A line of this table which is marked with "yes", expresses that there exists a family $(L_n)_{n\in\mathbb{N}}$ of (in)finite DRE-definable languages, such that minimal representations of L_n satisfy the given size bounds.

It is well known that there exist families of languages such that minimal DFAs are exponentially larger than minimal REs and vice versa. An interesting result from [LMN12] is that there exists a family of languages, such that minimal regular expressions and minimal DFAs are small but minimal deterministic expressions are exponentially larger. The intuitive reason is that the considered family of languages can be represented succinctly using regular expression and using determinism but not both at the same time. In a way, deterministic regular expressions inherit the disadvantages from both deterministic automata and regular expressions.

7.4 Minimization

It would be nice, if deterministic regular expressions could be efficiently minimized and if for each DRE-definable language there would exist a unique minimal DRE.

Unfortunately minimal DREs are not unique (not even up to permutation of unions) and minimization of DREs is NP-hard. A direct consequence is that there exist no unique minimal representation of DTDs and XML Schema and minimization of DTDs and XML Schema descriptions is NP-hard.

There are several canonical ways to phrase a decision problem for minimization. We use the following variant.[4]

[3] A proof will appear in the full version of [LMN12].

[4] Another variant would be to ask, whether a given expression is minimal. The complexity of both variants is the same.

MinDRE	
Given:	A deterministic regular expression α, a number k.
Question:	Does there exist a deterministic regular expression α', such that $L(\alpha') = L(\alpha)$ and $\|\alpha'\| \leq k$?

Theorem 7.5 MinDRE *is* NP*-complete.*

Proof. The upper bound follows is by guessing a deterministic regular expression of size at most k and checking whether it is equivalent to the given expression. Equivalence of DREs can be checked in polynomial time by conversion to DFAs.

The proof of the lower bound is by reduction from the NP-complete Independent Set Problem.

IndependentSet	
Given:	An undirected Graph $G = (V, E)$, a number k.
Question:	Does there exists a set $V_I \subseteq V$ of nodes in G, such that $\|V_i\| \geq k$ and there is no edge between nodes from V_I?

Given an instance $(G = (V, E), k)$ of the independent set problem with $V = \{v_1, \ldots, v_n\}$, we construct an instance (α, k') of the minimization problem.

Let $n = |V|$ and $m = |E|$. For the expression α, we will use the alphabet $\Sigma = V \cup E \cup X$, where $X = \{x_{1,1}, \ldots, x_{n,n}\}$ is a set of n^2 new symbols.

Let E_i be the set of edges incident to v_i and $X_i = \{x_{i,j} \mid \{v_i, v_j\} \notin E\}$. Let furthermore $\alpha_i = v_i \cdot (E_i + X_i)$ and $\beta_i = (v_i + \varepsilon) \cdot (E_i + X_i)$. We define α as

$$\alpha = \alpha_1 + \cdots + \alpha_n + E + X$$

and $k' = |\alpha| - kn$. Note that the symbols from X are only used to ensure that each subexpression α_i is of the same size. This finishes the construction of α.

We continue with the correctness proof. We first show that if there exists an independent set V_I of size k, then there exists a DRE γ of size k'.

Let V_I be an independent set of G of size k. We construct the deterministic expression

$$\gamma = \gamma_1 + \cdots + \gamma_n + \Sigma', \text{ with}$$

$$\gamma_i = \begin{cases} \beta_i & \text{if } v_i \in V_I \\ \alpha_i & \text{if } v_i \notin V_I \end{cases} \quad \text{and} \quad \Sigma' = \Sigma \setminus (\text{first}(\gamma_1) \cup \cdots \cup \text{first}(\gamma_n)).$$

It is easy to see, that γ is a regular expression for $L(\alpha)$. The size of γ is $|\alpha| - kn = k'$, as each use of β_i instead of α_i saves n alphabet symbols, which do not have to occur in Σ'. Furthermore β is deterministic, as for each two subexpressions β_i and β_j, $\text{first}(\beta_i) \cap \text{first}(\beta_j) = \emptyset$. Note that the intersection could only be nonempty if $\beta_i = \gamma_i$, $\beta_j = \gamma_j$ and v_i and v_j are adjacent in G. This would be a contradiction to V_I being an independent set of G.

We continue with showing that if there exists an expression γ of size at most k' for $L(\alpha)$, then there exists an independent set V_I of G of size k.

We say a regular expression δ is *similar* to a regular expression δ', if δ can be transformed into δ' using only

- the law of commutativity of $+$, i.e., changing the order of disjuncts;
- the law of associativity, i.e., adding or removing of unnecessary brackets.

We use the following claim, which we show afterwards.

Claim 7.6 *Every minimal DRE γ for the language $L(\alpha)$ is similar to an expression of the form*

$$\gamma_1 + \cdots + \gamma_n + \Sigma \setminus (\text{first}(\gamma_1) \cup \cdots \cup \text{first}(\gamma_n)),$$

where every γ_i is equal to either α_i or β_i.

Given a deterministic expression for $L(\alpha)$, which is of the form of Claim 7.6, we show that $V_I = \{v_i \mid \beta_i = \gamma_i\}$ is an independent set for G. Assume in contradiction, that $v_i \in V_I$, $v_j \in V_I$, and e_k is an edge connecting v_i and v_j in G. Then $e_k \in \text{first}(\beta_i) \cap \text{first}(\beta_j)$, as $\beta_i = \gamma_i$ and $\beta_j = \gamma_j$. This is a contradiction to the assumption that β is a DRE.

It remains to show Claim 7.6. First, we observe that γ does neither contain $\emptyset$ (as γ is minimal) nor a Kleene star (as the language is finite). As the language contains more than one string, γ cannot be an atomic regular expression and therefore either has a top-level concatenation or a top-level disjunction. For the ease of the proof, we allow disjunctions with only one disjunct and interpret the case where γ has a top-level concatenation as such a disjunction with only one disjunct. As we assume no unnecessary brackets, none of the disjuncts can have a top-level disjunction itself. Therefore every disjunct is atomic or has a top-level concatenation.

It remains to show that

(a) each of the disjuncts is either α_i or β_i (for some i) or an atomic symbol; and

(b) for each $i \in [1, n]$, there exists a disjunct, which is α_i or β_i.

We can observe that for each disjunct $\gamma_i = \eta_1\eta_2$ that has a top-level concatenation it holds that

- η_1 and η_2 are disjunctions of alphabet symbols and maybe ε, because the maximal length of strings in the language is 2;
- η_1 only contains symbols from V and maybe ε, because all other symbols only occur as first symbols in strings of length one;
- η_1 contains exactly one symbol v_i and maybe ε; and
- η_2 contains exactly the symbols E_i and X_i, where i is the index such that η_1 contains the alphabet symbol v_i.

The last two observations are because γ is a deterministic expression. Therefore, there cannot be two disjuncts γ_i and γ_j with $i \neq j$, such that $\text{first}(\gamma_i) \cap \text{first}(\gamma_j) \neq \emptyset$. It follows that η_2 has to contain exactly the symbols E_i and X_i if $v_i \in L(\eta_1)$. As $X_i \neq X_j$ for $i \neq j$ it follows that η_1 cannot contain v_i and v_j with $i \neq j$.

It can be easily seen that (a) and (b) follow from the last two observations. Note that there has to be one disjunct for every $v_i \in V$, as $V \subseteq \text{first}(\alpha)$. □

Note, that we have used a specific definition of the size of a regular expression (number of alphabet symbols). However the proof can be easily adopted to show hardness for other sensible notions of the size of a regular expression.

We can easily see from the proof that minimal DREs are not unique (up to permutation of disjunctions). We note, that different independent sets of the same size correspond to different DREs of the same size.

7.5 Further Research on Deterministic Regular Expressions

We have summarized known results about deterministic regular expressions, which are used in Document Type Definitions, XML Schema specifications and — for compatibility reasons — also in BonXai. Even if our understanding of these expressions has gotten better over the years, there are still unsolved problems. Especially the complexity of the following problem is still open: Given a regular language, can this language be described by a deterministic regular expressions with counters. It is even unknown whether this problem is decidable. As a consequence, it is also not known whether the following problem is decidable: Given a regular tree language S (by a tree automaton or by a RelaxNG schema), can this tree language be described by an XML Schema? Aside from this open problem, it would also be interesting to get a better understanding of which languages can be described by deterministic expressions without counters. Up to now, we only have the algorithm from Brüggemann-Klein and Wood [BKW98] to classify regular languages into DRE-definable and not DRE-definable. It would be nice to have a more intuitive description about which languages can be defined by DREs. However, such a more intuitive description does not necessarily exist.

8 Schema Decomposition

In this chapter we focus on an important part of distributed XML Repository Management Systems, namely on collections of XML documents and on schema design for such collections. We abstract collections of XML documents as *distributed XML documents*. These are XML documents that consist of several logical parts which are possibly located on different machines.

Following Abiteboul et al. [AGM09], a *distributed XML document* consists of a *root document* t, which is an XML tree that is stored locally at some site. Some of the leaves of t are labeled with references $\mathrm{f}_1, \ldots, \mathrm{f}_n$, which point to external resources $r_1, \ldots, r_n$. The extension $\mathrm{ext}(t)$ of t is then obtained by replacing each node f_i with the XML tree or XML forest provided by the resource r_i referenced by f_i. In other words, $\mathrm{ext}(t)$ is a large XML document that is distributed over $r_1, \ldots, r_n$. The root document t provides an interface to this large XML document and obtains through its pointers f_i the knowledge of where to get access to the different parts. These parts can be maintained by different peers and/or provided by programs or web service calls. We therefore sometimes also refer to the f_i as *function calls*.

We come back to our running example of a content management system. In such a system a university employee might want to create a personal page consisting of

- some profile information like name, telephone number, location of the office;
- provided lectures; and
- most recent published articles.

Of course, she could simply produce such a document and add it to the content management system. However she would need to update this information quite regularly. Therefore, she prefers to reuse content stored in different systems.

For the sake of this example, the personal data is stored on a server of the faculty, the provided lectures can be obtained from a central system of the university and her most recent articles can be obtained from a bibliographic server of the department.

A straightforward solution is to provide hyperlinks to these resources, but this is inconvenient as the reader of the page actually needs to follow these links to see the underlying data. A slightly more advanced technique would be to use inline frames[1] to embed the remote content into the page. However, this technique allows only very limited formatting. A much more convenient solution would be if the content management system stores only references to the foreign data and, whenever[2] the page is requested, fetches

[1] In HTML, an inline frame (iframe) creates a rectangular area to embed the referenced web page.
[2] Of course, some caching technique may be applied to reduce traffic and page load times.

an up to date copy of the data to create the profile page. This way, the fact that the information is stored on different servers would be invisible to readers of the page.

While it is easy to implement a system that just fetches the data and delivers it to the client, there are some complications. The first one is checking that the data is valid, i.e. that it satisfies a given schema. Usually, when a page is created or changed, the content management system will check whether the syntax of the page is valid and if necessary ask the editing person to correct any errors. If a page contains references, the referenced content may change without notice. Furthermore, people updating the bibliographic data or the list of lectures will not necessarily know about this profile page and not take care whether their changes will invalidate the profile page. Therefore it is not possible to validate the profile page at the point of time where the content of the page changes.

The solution is that the CMS has to check not only if the page is valid at the moment of creation, but also if the page is valid for every possible content from the referenced sources. It is reasonable to assume that the referenced pages do not contain garbage, but instead always return a document t which is valid according to some known schema S.

The problem whether a document is valid for all possible contents of the referenced documents is the soundness problem (of distributed documents).

Soundness of distributed documents	
Given:	a global schema S, a distributed document t with function calls $\mathrm{f}_1, \ldots, \mathrm{f}_n$, local schemas $S_1, \ldots, S_n$
Question:	Is the composed document $\mathrm{ext}(t)$ valid wrt. S for all possible documents $r_1, \ldots, r_n$ with $r_i \in L(S_i)$?

In this chapter, we do not focus on the soundness problem. Instead we focus on the more advanced problem of designing good schemas for distributed documents. We describe the problem based on the "profile page example": In the modified example all university employees have a profile page as described above, which is stored as a single document. These documents have to comply to some schema S. Now some webmaster wants to organize the data for better management; that is, creating separate databases for lectures, bibliographic information and administrative purposes. Then there should be one (template) distributed document t, where only the parameters of the function calls need to be adjusted for different persons.

There are several questions which arise in this setting. One of them is, given a distributed document t (with function calls $\mathrm{f}_1, \ldots, \mathrm{f}_n$) and a schema S, how to generate good schemas $S_1, \ldots, S_n$ for the queried web services. Abiteboul et al. have researched this question [AGM09]. They call a sequence of schemas $(S_1, \ldots, S_n)$ a *typing* τ and according to them there are several degrees of desirability for typings: local typings, maximal local typings and perfect typings.

Intuitively a typing is local, when all trees that can be constructed by replacing the function calls with trees from the schemas of the typing are valid wrt. S (i.e., the typing is sound) and all valid trees can be constructed this way (i.e., the typing is complete). A typing $\tau = (S_1, \ldots, S_n)$ is maximal local, if it is local and there exist no typing

$\tau' = (S'_1, \ldots, S'_n)$ such that $\tau \subsetneq \tau'$, where inclusion is defined componentwise, i.e., $\tau \subseteq \tau'$, if and only if $S_i \subseteq S'_i$ for all $i \in [1, n]$. A typing is perfect, if it is maximal and there exist no sound typing which is incomparable to the given typing.

This rises immediately six decision problems. Given a design and a typing one may ask whether the typing is local, maximal local or perfect and given a design one may ask if local, maximal local or perfect typings exist.

We study these decision problems and improve the results of [AGM09]. In the database theory context, there is a connection with the work of Calvanese et al. [CGLV02]. However, their intention is orthogonal to ours. Stated with our definitions, they would start from a global schema S and a typing τ and ask for a maximal schema of distributed documents S' for which τ is sound for $(S, t(\mathrm{f}_1, \ldots, \mathrm{f}_n))$ for every $t(\mathrm{f}_1, \ldots, \mathrm{f}_n) \in S'$.

8.1 From XML Documents to Strings

Abiteboul et al. studied the typing problems for DTDs, XML Schemas, and extended DTDs [PV00] as schema languages.

It is known that several decision problems for DTDs and XML Schemas can be reduced to corresponding problems on strings. For example, in [MNS09] it is shown that containment and equivalence testing for DTDs and XML Schemas over a class of regular expressions C has the same complexity as containment and equivalence testing for C. This result was extended by [AGM09] in the context of perfect and (maximal) local typings. In this sense, it follows from [AGM09, MNS09] that all the aforementioned problems have the same complexity for DTDs as for the regular expressions that these DTDs use. For this reason, as long as we are interested in DTD and XML Schema, we can safely focus our study to strings instead of trees. In other words, we study designs (R, w) and $(\mathcal{A}, w)$ where R is a regular expression, $\mathcal{A}$ is a finite automaton, and w is a *distributed string* $w_0\,\mathrm{f}_1\,w_1 \cdots \mathrm{f}_n\,w_n$ with function calls $\mathrm{f}_1, \ldots, \mathrm{f}_n$ and strings $w_0, \ldots, w_n$. It should be noted that the typing problems for Relax NG schemas [CM01] or extended DTDs cannot be reduced to the string case if EXPTIME $\neq$ PSPACE.

8.2 Notation and Algorithmic Problems

Let Σ be a finite alphabet and Σ_f be a set of *function calls*, typically written as f or $\mathrm{f}_1, \mathrm{f}_2$, etc. We recall the following notions from Abiteboul et al. [AGM09].

Definition 8.1 A *distributed string* is a string $w = w_0\,\mathrm{f}_1\,w_1 \ldots \mathrm{f}_n\,w_n$, where $n \in \mathbb{N}$, $w_i \in \Sigma^*$ and $\mathrm{f}_i \in \Sigma_\mathrm{f}$, for each i. We write $w(\mathrm{f}_1 \cdots \mathrm{f}_n)$ for w if we want to emphasize the function calls. A *design* is a pair (L, w) consisting of a language L and a distributed string w. We often specify designs as $(\mathcal{A}, w)$ or (R, w) for an automaton $\mathcal{A}$ or a regular expression R.

Definition 8.2 A *typing* τ for (L, w) is a sequence $(L_1, \ldots, L_n)$ of nonempty languages over Σ. We write $w(\tau)$ for the language

$$\{w_0 v_1 w_1 \cdots v_n w_n \mid v_i \in L_i, 1 \leq i \leq n\}.$$

Given a design (L, w) and a typing τ we call τ

- a *sound typing* for (L, w), if $w(\tau) \subseteq L$,
- a *complete typing* for (L, w), if $w(\tau) \supseteq L$,
- a *local typing* for (L, w), if $w(\tau) = L$, i.e., if it is sound and complete,
- a *maximal typing* for (L, w), if it is sound and there exists no sound typing τ' for (L, w), such that $\tau \subsetneq \tau'$, where inclusion is defined componentwise.
- a *perfect typing* for (L, w), if it is local and if for each sound typing τ' for (L, w) it holds $\tau' \subseteq \tau$.

In this chapter, we consider the following algorithmic problems. Given a design $D = (L, w)$ and a typing τ,

LOC: check whether τ is a local typing for D;

ML: check whether τ is maximal and local for D;

PERF: check whether τ is a perfect typing for D.

Given a design $D = (L, w)$,

$\exists$−LOC: check whether there exists a local typing for D;

$\exists$−ML: check whether there exists a maximal local typing for D;

$\exists$−PERF: check whether there exists a perfect typing for D.

For a $k \in \mathbb{N}$, we denote by $\exists$−kLOC (resp., $\exists$−kML) the problem $\exists$−LOC (resp., $\exists$−ML) where w in the given design (L, w) only contains k function calls.

The complexity of these problems might depend on the formalism in which the language L is given and in which the typing has to be specified. For simplicity, we only study cases where these two formalisms coincide. More precisely, we consider NFAs (as in [AGM09]), DFAs, and DREs as specification formalisms. We denote the resulting algorithmic problems as in LOC(DFA), where L and the target typing are specified by DFAs. Since not all regular languages can be defined by DREs, we need to make clear what we mean by ML(DRE). In ML(DRE) we want to know whether τ is local and there exists no sound *DRE-definable* typing τ' such that $\tau \subsetneq \tau'$.[3]

The Table 8.1 summarizes complexity results for these problems.

[3]One could define this problem in two different manners: either τ' can be regular, or needs to be DRE-definable. From our proof it follows that these two problems coincide.

	LOC	ML	PERF
NFA	PSPACE-c [AGM09]	**PSPACE-c** (8.28)	PSPACE-c [AGM09]
DFA	PSPACE-c [JR93]	**PSPACE-c** (8.28,8.30)	**in PTIME** (8.19)
DRE	**PSPACE-c** (8.27)	**PSPACE-c** (8.28,8.33)	**in PTIME** (8.21)

	∃−2LOC	∃−2ML	∃−PERF
NFA	PSPACE-h [AGM09] **in NEXPTIME** (8.36)		PSPACE-c [AGM09]
DFA	**PSPACE-c** (8.36,8.37)		**in PTIME** (8.19)
DRE	**PSPACE-h** (8.41)	**PSPACE-h** (8.41) **in EXPTIME** (8.42)	**in PTIME** (8.21)

Table 8.1: Summary of complexity results. Results of this thesis are highlighted. All results for REs are equal to the results for NFAs. The result for ML(NFA) was already stated in [AGM09], we present a corrected proof. All these results also hold for DTDs and XML Schemas using REs, NFAs, DFAs, and DREs as content models. The numbers between brackets indicate the theorem numbers in which the results are proved. We also prove that ∃−LOC(NFA) and ∃−ML(NFA) are in **EXPSPACE** in general (Theorem 8.34).

Typings and Regular Languages

We recall some results on language equations that have direct consequences for the typing problem. The next theorem follows immediately from Corollary 13 in [Bal04].

Theorem 8.3 ([Bal04]) *Let (L, w) be a design. If (L, w) has a local (even: non-regular) typing then it also has a regular, maximal local typing.*

This theorem holds independently of the formalism in which L is specified, as the considered problems are defined with respect to the languages. It gives a good reason to restrict attention to regular typings as was suggested in [AGM09] and is also done here. One particular consequence of this theorem is that the problems ∃−LOC(NFA) and ∃−ML(NFA) coincide. The same holds for ∃−LOC(DFA) and ∃−ML(DFA). However, the existence of local typings does not guarantee the existence of local typings specified by DREs (Theorem 8.38) and the existence of local typings specified by DREs does not guarantee the existence of maximal local typings specified by DREs (Theorem 8.40).

8.3 Connections to Language Theoretic Problems

The algorithmic problems are very related to language theoretic problems studied in the literature, especially ConcatenationEquivalence and Primality.

The ConcatenationEquivalence problem tests given languages $L_1, \ldots, L_n$ and L whether $L = L_1 \cdot L_2 \cdot \cdots \cdot L_n$, that is whether the concatenation of the languages L_1 to L_n is equal to the language L.

Obviously ConcatenationEquivalence is equal to the special case of LOC, where $w = \mathrm{f}_1 \ldots \mathrm{f}_n$, that is where all strings between function calls are empty. On the other hand, LOC can be easily reduced to ConcatenationEquivalence.[4] Therefore the complexity bounds for LOC are always the same as for the ConcatenationEquivalence problem for the respective formalism of specifying languages.

The Primality problem asks given a language L, whether it can be decomposed into two languages L_1 and L_2, such that $L = L_1 \cdot L_2$ and $L_1 \neq \{\varepsilon\} \neq L_2$.

In a similar way as LOC is related to the problem ConcatenationEquivalence, $\exists-$LOC is also related to the Primality problem.

The investigation of language decompositions goes back to Conway [Con71], who was interested in expressing a regular event E in the form $\mathrm{f}(F_1, F_2, \ldots)$, wherein f is a regular function and F_i are regular events. Language equations form a broad framework in formal language theory in which such kinds of questions are considered (see [Kun07] for a recent overview). The primality question for regular languages [SY99, Sal08] is a special case of a language equation, which has been studied in depth, both for finite and infinite languages [SY99, CFPR03, AF05, HSW06, SSY08, Sal08, Wie09].

The complexity of Primality(DFA) has been considered an open problem in Formal Language Theory since the late 90's (see Problem 2.1 in [Sal08]). Primality(DFA) is decidable but no further lower or upper bounds are known [Sal08]. We pinpoint the precise complexity of Primality(DFA) in Theorem 8.5: it is PSPACE-complete.

That the complexity of Primality was open for a long time indicates that it might be non-trivial to figure out the precise complexity of $\exists-$LOC(DFA) and $\exists-$LOC(NFA), as they are in a sense generalizations of Primality. As a step towards an answer to these complexity questions we determine the precise complexity of $\exists-$LOC(DFA) for distributed strings with at most two function calls, a case that already generalizes Primality(DFA).

Despite it is very connected to the $\exists-$2LOC problem, the connection is not as easy as above. Especially Primality is not equivalent to the special case of $\exists-$2LOC where $w = \mathrm{f}_1\,\mathrm{f}_2$, as in the $\exists-$2LOC problem languages are allowed to consist only of the empty word. Therefore the special case $w = \mathrm{f}_1\,\mathrm{f}_2$ of $\exists-$2LOC always has a trivial solution where one of the two languages is chosen to be $\{\varepsilon\}$. To overcome this problem, we define a slightly different version of the Primality problem called StrongPrimality, where the decomposed languages are not allowed to contain the empty word ε. Not very surprisingly, we can show the same complexity bounds for Primality and StrongPrimality using the same proof ideas.

In Lemma 8.29 we will show that there is a one-to-one correspondence between decompositions $L_1 \cdot L_2$ of a language L, where L_1 and L_2 do not contain ε and local typings for the design $(L_\#, \mathrm{f}_1\,\#\,\mathrm{f}_2\,\#)$, where $L_\# = \{a_1\#a_2\#\ldots a_n\# \mid a_1a_2\ldots a_n \in L\}$ results from L by adding a # after every symbol.

[4] Technically, this is only true if the used formalism for specifying languages is able to efficiently represent singleton languages, that is languages containing exactly one string.

8.3.1 Proof Strategies

As seen above, our problems are clearly connected to language theoretic problems. Therefore it is not surprising that some of our complexity bounds can be easily derived from the respective complexity bounds of the underlying language theoretic problems.

All our lower bounds are PSPACE bounds. They are either derived from the ConcatenationEquivalence problem or the StrongPrimality problem of the respective formalism.

While the details of the proofs depend on the formalism, the overall proof ideas for our upper bounds are more or less independent of the formalism used to specify languages. The ideas are as follows:

LOC: Trivial reduction to ConcatenationEquivalence

ML: Test locality using ConcatenationEquivalence. Compute an automaton which describes the empty language if and only if the typing is maximal.

PERF: Compute the unique perfect typing if it exists and compare this typing with the given one.

∃−LOC: Equivalent to ∃−ML for all formalisms which have at least the expressive power of regular languages.

∃−ML: Test all (finitely many) typings which respect a certain normal form for locality and maximality.

∃−PERF: Show that there always is only one candidate for a perfect typing. Compute this candidate and test whether it is perfect.

In the next section, we will give all proofs for the complexity of the underlying language theoretic problems (i.e. Primality and ConcatenationEquivalence). The section is mostly self contained, i.e. the proofs do not refer to results obtained in other sections. Afterwards we discuss the problems of testing whether a typing is perfect and whether there exists a perfect typing in Section 8.5. In Section 8.6 we discuss the normal forms needed for upper bound proofs. In Sections 8.7 and 8.8 we show the complexities for verifying a given typing and testing whether a typing exists, respectively.

8.4 The Language Primality Problem

The Primality problem for formal languages is defined as follows. A *non-trivial decomposition of a language* L is a pair (L_1, L_2) of languages, $L_1 \neq \{\varepsilon\} \neq L_2$ such that $L = L_1 \cdot L_2$. A language is called *prime* if it does *not* have a non-trivial decomposition. Primality($\mathcal{X}$) asks, given a representation X specified by formalism $\mathcal{X}$, whether $L(X)$ is prime.

We will show that Primality is PSPACE-complete for DFAs and DREs. Furthermore from our investigations of ∃−2LOC, we can easily conclude a NEXPTIME upper bound for NFAs and REs. For the PSPACE upper bound we will use the notion of a decomposition set as defined in [SY99].

Let $\mathcal{A} = (\Sigma, Q, \delta, q_0, Q_F)$ be a DFA. A subset Q' of Q is called a *decomposition set* of $\mathcal{A}$, if and only if

$$L(A) = \underbrace{L(\mathcal{A}_1)}_{L_1} \cdot \underbrace{\bigcap_{q \in Q'} L(\mathcal{A}_q)}_{L_2},$$

where $\mathcal{A}_1 = (\Sigma, Q, \delta, q_0, Q')$ and $\mathcal{A}_q = (\Sigma, Q, \delta, q, Q_F)$ for each $q \in Q'$.

For the upper bound we will use the following lemma.

Lemma 8.4 *Let L be a language given by a minimal DFA $\mathcal{A} = (\Sigma, Q, \delta, q_0, Q_F)$. If L is not prime, then there exists a decomposition set Q' of $\mathcal{A}$.*

Proof. Let L_a and L_b be languages, such that $L = L_a L_b$. We define Q' to be $Q' = \{q \mid \exists w \in L_a.\ \delta^*(q_0, w) = q\}$. It remains to show that Q' is a decomposition set. Let therefore be A_1, A_q, L_1 and L_2 be defined as in the definition of a decomposition set with respect to $\mathcal{A}$ and Q'.

$L \subseteq L_1 L_2$: Let w be a string from L and $w_1 \in L_a$ and $w_2 \in L_b$ be two strings such that $w = w_1 w_2$. Such strings exists as $L = L_a L_b$. By definition of $\mathcal{A}_1$ it is obvious that $w_1 \in L(\mathcal{A}_1) = L_1$. It remains to show that $w_2 \in L_2$, i.e., $w_2 \in L(\mathcal{A}_q)$ for every $q \in Q'$. Assume there exists a $q \in Q'$ such that $w_2 \notin L(\mathcal{A}_q)$. In this case let w_1' be a string from L_a with $\delta^*(q_0, w_1') = q$. The string $w_1' w_2$ is in $L_a L_b$ but not in L, as $\delta^*(q_0, w_1' w_2) = \delta^*(q, w_2) \notin Q_F$. This is a contradiction to the assumption that $L = L_a L_b$.

$L \supseteq L_1 L_2$: Let w_1 and w_2 be strings such that $w_1 \in L(\mathcal{A}_1)$ and $w_2 \in L(\mathcal{A}_q)$ for every $q \in Q'$. Then $\delta^*(q_0, w_1) = q$ for some $q \in Q'$ by the definition of $\mathcal{A}_1$ and $\delta^*(q, w_2) \in Q_F$ by the definition on $\mathcal{A}_q$. We can conclude that $w_1 w_2 \in L = L(\mathcal{A})$. This concludes the proof. □

For the lower bound, we use a result from literature. The problem ConcatenationUniversality is a special case of ConcatenationEquivalence. It asks, given two languages L_1 and L_2 over the alphabet Σ, whether $L_1 L_2 = \Sigma^*$. ConcatenationUniversality is PSPACE-complete, when both languages are given by DFAs [JR93].

Now we have the ingredients to show the PSPACE-completeness of Primality.

Theorem 8.5 Primality(DFA) *is* PSPACE*-complete.*

Proof. We first prove that Primality is PSPACE-hard. We use a polynomial time reduction from the complement of ConcatenationUniversality. Given two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, we construct a DFA $\mathcal{A}$, such that $L(\mathcal{A})$ is prime, if and only if $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2) \neq \Sigma^*$.

To this end, let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two arbitrary DFAs. Without loss of generality, we can assume that $L(\mathcal{A}_1)$ and $L(\mathcal{A}_2)$ are strict supersets of $\{\varepsilon\}$. Let Σ' be a disjoint copy of Σ, i.e., $\Sigma' = \{a' \mid a \in \Sigma\}$ and we assume that $\Sigma \cap \Sigma' = \emptyset$. Let \$ be a symbol not occurring in Σ or Σ'. By $\mathcal{A}_1'$ and $\mathcal{A}_2'$ we denote the DFAs resulting from $\mathcal{A}_1$ and $\mathcal{A}_2$ by replacing each character a from Σ with the corresponding character a' from Σ'. We denote the languages of $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_1'$ and $\mathcal{A}_2'$ by L_1, L_2, L_1' and L_2', respectively. We let $\mathcal{A}$ be an automaton for

$$L =_{\text{def}} \Sigma^* \cup L_1 \$ L_2' \cup L_1' \$ L_2 \cup L_1' \$\$ L_2'.$$

Claim 8.6 *Either there is no nontrivial decomposition of L or the only nontrivial decomposition is (L_a, L_b) with $L_a = L_1 \cup L_1'\$$ and $L_b = L_2 \cup \$L_2'$.*

Before we prove this claim, we first show that L is not prime, if and only if $L_1 \cdot L_2 = \Sigma^*$. If L is not prime, according to Claim 8.6, the only nontrivial decomposition is (L_a, L_b). Since $L_a \cap \Sigma^* = L_1$, $L_b \cap \Sigma^* = L_2$ and $L \cap \Sigma^* = \Sigma^*$, we can conclude that $L_1 \cdot L_2 = \Sigma^*$.

For the other direction we claim, that if $L_1 L_2 = \Sigma^*$, then (L_a, L_b) is a decomposition of L. Indeed, since each string in L can be written as a concatenation of a string in L_a and a string in L_b and conversely, we have that $L = L_a L_b$. This ends the proof of PSPACE-hardness.

We continue to show that Primality is in PSPACE. Let $\mathcal{A}$ be the DFA, for which we want to know whether $L(\mathcal{A})$ is prime.

The algorithm tests for each subset Q' of Q, whether Q' is a decomposition set. The correctness of the algorithm follows from Lemma 8.4. It only remains to show that testing whether Q' is a decomposition set can be done in polynomial space. To this end, let $\mathcal{B}$ be the following alternating automaton

(1) it simulates $\mathcal{A}$ on w and, whenever it enters a state from Q' it non-deterministically decides to continue the simulation or to proceed with (2),

(2) it verifies that the remainder w' of w is in $L(\mathcal{A}_q)$ for every $q \in Q'$ by universally branching to all states $q \in Q'$ and testing that $\delta^*(q, w') \subseteq Q_F$.

The equivalence of the AFA $\mathcal{B}$ with $\mathcal{A}$ can be tested in polynomial space (cf. [Var95]).

It remains to prove Claim 8.6: Let (L_c, L_d) be a decomposition of L. We first prove that $L_c \subseteq \Sigma^* \cup \Sigma'^*\$$ holds. First of all, since every string in L_c must be a prefix of a string in L, observe that no string in L_c can contain symbols from Σ and Σ' without having a \$-symbol in between. We now argue that

(a) no string in L_c can contain two \$-symbols; and

(b) no string in L_c can have a Σ-symbol followed by \$-symbol.

Indeed, towards a contradiction, if *(a)* or *(b)* would not be the case, then, since $L_c L_d \subseteq L$, L_d can only contain strings from Σ'^*, because only Σ'-symbols are allowed to occur after two \$-symbols in L and after a Σ-symbol followed by a \$-symbol. Since $\Sigma^* \subseteq L_c L_d$ we then have that $\Sigma^* \subseteq L_c$ and $\varepsilon \in L_d$. Since $L_d \neq \{\varepsilon\}$ ((L_c, L_d) is a nontrivial decomposition) this implies that $L_c L_d$ contains at least one string from $\Sigma^+\Sigma'^+$, which contradicts $L_c L_d \subseteq L$. By symmetry, L_d does not contain any strings with two \$ symbols either. We can conclude that both L_c and L_d contain strings with at least one \$ symbol.

To show that $L_c \subseteq \Sigma^* \cup \Sigma'^*\$$, it remains to prove that

(c) each string with a Σ'-symbol ends with a \$-symbol.

Towards a contradiction, assume that L_c has a string s with a Σ'-symbol, that does not end with a \$-symbol. Since L_c only contains prefixes of L and since *(a)* and *(b)* hold,

there are two possible cases: either $s \in L(\Sigma'^{+})$ or $s \in L(\Sigma'^{+}\$\Sigma^{+})$. The first case is impossible as L_d does not contain any strings with two \$ symbols. The second case is also impossible as concatenation of s with a string from L_b with a \$ symbol would yield a string outside L.

As L_d contains no strings with two \$ symbols, L_c contains at least one string in $\Sigma'^{*}\$$. Again by symmetry, we can conclude that $L_d \subseteq \Sigma^{*} \cup \$\Sigma'^{*}$ holds and that L_d contains at least one string in $\$\Sigma'^{*}$.

From $L_c L_d \cap \Sigma^{*}\$\Sigma'^{*} = L_1\$L_2'$ we now immediately get $L_c \cap \Sigma^{*} = L_1$ and, symmetrically, $L_d \cap \Sigma^{*} = L_2$.

Finally, from $L_c L_d \cap \Sigma'^{*}\$\$\Sigma'^{*} = L_1'\$\$L_2'$ we obtain $L_c \cap \Sigma'^{*}\$ = L_1'\$$ and $L_d \cap \$\Sigma'^{*} = \$L_2'$

Thus, $L_c = L_a$ and $L_d = L_b$ and (L_a, L_b) is the only nontrivial decomposition of L. □

This concludes the proof that Primality(DFA) is PSPACE-complete. It should be stressed that the complexity of deciding whether a language can be decomposed into three nontrivial languages is still unknown. The construction of an alternating automaton to test primality does not carry over to decompositions into three languages in the sense that the resulting automaton for the middle part can get exponentially large. Therefore our construction only leads to an EXPSPACE upper bound in the case of more than two languages.

In Section 8.8, we show an EXPSPACE upper bound for ∃−LOC(DFA) (Corollary 8.35). This bound easily carries over to the problem of deciding whether a language given by a DFA can be decomposed into three nontrivial languages. The used algorithm just needs to be adapted that it checks whether all languages do not equal $\{\varepsilon\}$.

The following proof for StrongPrimality uses the same ideas as the proof for Primality and only slightly different definitions of the languages to ensure that ε is not contained in the languages.

Theorem 8.7 StrongPrimality(DFA) *is* PSPACE*-complete.*

Proof. The proof of the lower bound is again by a reduction from ConcatenationUniversality. Thereto, let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two DFAs. We construct a DFA $\mathcal{A}$ such that $L(\mathcal{B}_1) \cdot L(\mathcal{B}_2) = \Sigma^{*}$ if and only if there exist languages F_1, F_2 such that $L(\mathcal{A}) = F_1 \cdot F_2$ with $\varepsilon \notin F_1$ and $\varepsilon \notin F_2$.

We first construct DFAs $\mathcal{A}_1$ for $L_1 = \Sigma \cdot L(\mathcal{B}_1)$ and $\mathcal{A}_2$ for $L_2 = L(\mathcal{B}_2) \cdot \Sigma$. Clearly, neither L_1 nor L_2 do contain the empty word. Furthermore, it is obvious that $L(\mathcal{B}_1) \cdot L(\mathcal{B}_2) = \Sigma^{*}$, if and only if $L_1 \cdot L_2 = \Sigma\Sigma^{*}\Sigma$.

Similarly as in the proof of Theorem 8.5 we let $\mathcal{A}$ be a DFA for

$$L =_{\text{def}} \Sigma\Sigma^{*}\Sigma \cup L_1\$L_2' \cup L_1'\$L_2 \cup L_1'\$\$L_2'.$$

As in the proof of Theorem 8.5 L_1' and L_2' are copies of L_1 and L_2, respectively, over an alphabet Σ'.

The proof of Claim 8.6 can almost literally be adapted to show that

- there is at most one possible decomposition of L into $L_a = L_1 \cup L_1'\$$ and $L_b = L_2 \cup \$L_2'$;

- (L_a, L_b) is a decomposition of L if and only if $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2) = \Sigma\Sigma^*\Sigma$; and
- neither L_a nor L_b contain the empty word.

The upper bound proof is identical to the proof in Theorem 8.5, except that we need to change the AFA $\mathcal{B}$ such that it additionally check that both languages do not allow the empty string. □

From our result that ∃−2LOC(NFA) and ∃−2LOC(RE) are in NEXPTIME (Theorem 8.36), we easily get an upper bound for Primality(NFA) and Primality(RE).

Theorem 8.8 Primality(NFA) *and* Primality(RE) *are in* NEXPTIME.

We do not give a proof here, as it is identical to the proof of Theorem 8.36 with the only difference, that the algorithm additionally needs to check that none of the two languages equals $\{\varepsilon\}$.

Concatenation Universality for DREs

Before we can show hardness results for primality for deterministic regular expressions, we need to show that ConcatenationUniversality is PSPACE-complete for DREs. Afterwards we use the same proof idea as we already used to show hardness for DFAs.

Theorem 8.9 ConcatenationUniversality(DRE) *is* PSPACE*-complete.*

Proof. The upper bound is easily obtained by transforming R_1 and R_2 into NFAs and the fact that ConcatenationUniversality is in PSPACE for NFAs. The lower bound is by reduction from the complement of PSPACE-complete CorridorTiling [van97]. We give a formal definition of CorridorTiling in Section 3.4.

Let $\mathcal{U} = (U, H, V, u_0, u_F, n)$ be an instance of the corridor tiling problem, where U is the set of tiles, H and V are the horizontal and vertical constraints, u_0 and u_F are the first and last tile[5] and n is the width of the tiling. As explained in Section 3.4, we can assume w.l.o.g. that for any valid tiling it holds that $(t, t') \in H$ for any pair of tiles such that t occurs at the end of some row and t' occurs at the beginning of the next row.

We construct DREs R_1 and R_2, such that $\Sigma^* = L(R_1) \cdot L(R_2)$, if and only if there is *no* corridor tiling for T. To this end, R_1, R_2 are designed such that $L(R_1) \cdot L(R_2)$ accepts all strings that do *not* encode valid corridor tilings, i.e., $L(R_1) \cdot L(R_2)$ catches all errors. The rationale of our reduction is that R_1 accepts all prefixes of encodings, and that R_2 checks whether an error occurs in the beginning of the string that it reads. In this way, R_1 can "guess" where the error should occur and R_2 can catch it.

Let $\Sigma = U$. We encode a corridor tiling as a string $v = v_1 v_2 \dots v_m$, where each v_i encodes one row of the tiling.

Let H_x denote the set of tiles that may be placed right of tile x and V_x the set of tiles, that may be placed above x. The respective sets of forbidden tiles are denoted by $\overline{H_x}$ and $\overline{V_x}$.

[5] Remember, that our definition of CorridorTiling only uses a first and last tile, instead of a first and last row.

R_1 is a regular expression which accepts any string that does not start with u_0 or does not use u_F.

$$R_1 = \varepsilon + \overline{u_0}\Sigma^* + u_0\overline{u_F}^*$$

With $\overline{u_0}$ and $\overline{u_F}$ we abbreviate the sets $\Sigma \setminus \{u_0\}$ and $\Sigma \setminus \{u_F\}$. R_2 checks for errors in horizontal or vertical constraints. It accepts the empty string and every string $w = a_1 a_2 \cdots a_k$, with $k \in \mathbb{N}$ such that either $(a_1, a_2) \notin H$ or $(a_1, a_{n+1}) \notin V$. More precisely,

$$R_2 = \varepsilon + \sum_{x \in \Sigma \setminus \{u_F\}} x(\overline{H_x}\Sigma^* + H_x\Sigma^{n-2}\overline{V_x}\Sigma^*).$$

By construction, R_1 and R_2 are deterministic expressions of quadratic size. We show that $\Sigma^* = L(R_1) \cdot L(R_2)$ if and only if there is no corridor tiling for T.

We first show that if there is a string v such that $v \notin L(R_1) \cdot L(R_2)$, then there is a valid tiling. First, we observe that v starts with u_0 and ends with u_F, as otherwise $v \in L(R_1)$. Furthermore v obeys all horizontal and vertical constraints. Otherwise, by construction of R_1 and R_2, the prefix of v up to the first error would be in $L(R_1)$ and the remaining suffix would be in $L(R_2)$.

On the other hand, if there exists a valid tiling of $\mathcal{U}$, then for the string v encoding this tiling it holds that $v \notin L(R_1) \cdot L(R_2)$: first of all, $v \notin L(R_1)$, as v starts with u_0 and contains u_F. It therefore suffices to show, that there is no prefix u of v with $u \in u_0\overline{u_F}^*$ such that the nonempty suffix w with $u = vw$ is in $w \in L(R_2)$. But this holds because all constraints are fulfilled, there is no such w, where the first tile in w violates the constraints regarding its right or top neighbor. □

As ConcatenationUniversality is a special case of ConcatenationEquivalence, the lower bound carries over to ConcatenationEquivalence. The upper bound proof still works for ConcatenationEquivalence. We get the following easy corollary.

Corollary 8.10 ConcatenationEquivalence(DRE) *is* PSPACE*-complete.*

Primality for Deterministic Expressions

Now we can continue to show PSPACE-completeness for Primality(DRE). We use the same proof idea as for DFAs (reduction from ConcatenationUniversality), however there is one additional complication: we need to show that the computed language is DRE-definable. To accomplish this we will not start from arbitrary instances of the ConcatenationUniversality problem, but instead use the instances produced by the reduction from CorridorTiling to ConcatenationUniversality given in the proof of Theorem 8.9.

Theorem 8.11 Primality(DRE) *and* StrongPrimality(DRE) *are* PSPACE*-complete.*

Proof. The PSPACE upper bounds are immediate from the upper bounds of Primality(DFA) and StrongPrimality(DFA), as deterministic regular expressions can be converted to equivalent DFAs in polynomial time (Theorem 7.2).

For the lower bound for Primality, we reduce from the CorridorTiling problem. We give a reduction function f, which results from composing the reduction function f_1 from the

reduction of CorridorTiling to ConcatenationUniversality in the proof of Theorem 8.9 with a function f_2, which we describe here. The domain of f_2 is the co-domain of f_1.

Let therefore R_1 and R_2 be the regular expressions from the proof of Theorem 8.9. With R_1' and R_2' we denote copies of R_1 and R_2 written with a disjoint copy Σ' of the alphabet Σ. Let L_1, L_2, L_1' and L_2' be the languages of R_1, R_2, R_1' and R_2'. We will show that there is a polynomial size DRE for the language

$$L = \Sigma^* \cup L_1\$L_2' \cup L_1'\$L_2 \cup L_1'\$\$L_2',$$

which concludes the proof, as we already know that L is prime if and only if $L_1 \cdot L_2$ is not the universal language (proof of Theorem 8.5) and that $L_1 \cdot L_2$ is not the universal language if and only if the tiling instance producing R_1 and R_2 has a valid corridor tiling (proof of Theorem 8.9).

We first note that

$$\underbrace{\Sigma^* + R_1\$R_2'}_{R_x} + R_1'\$(R_2 + \$R_2')$$

is a RE for L. This RE is not yet deterministic, as R_1 uses alphabet Σ. However, we can get a DRE R by transforming R_x into a DRE as described in the following.

We can describe the language $L(R_x)$ using the following expression

$$R_1(\Sigma^* + \$R_2') + \overline{R_1}\Sigma^*,$$

where $\overline{R_1} = \Sigma^* \setminus L(R_1)$. Furthermore we know from the proof of Theorem 8.9 that $R_1 = \varepsilon + \overline{u_0}\Sigma^* + u_0\overline{u_F}^*$ and $\overline{R_1} = u_0\overline{u_F}^*u_F\Sigma^*$, where $\overline{u_0} = \Sigma \setminus \{u_0\}$ and $\overline{u_F} = \Sigma \setminus \{u_F\}$. Hence, the language $L(R_x)$ can be defined by the following deterministic expression

$$\varepsilon + \$R_2' + \overline{u_0}\Sigma^*(\varepsilon + \$R_2') + u_0\overline{u_F}^*(\$R_2' + u_F\Sigma^*).$$

This concludes the proof for Primality(DRE).

We now show that StrongPrimality(DRE) is PSPACE-complete, again using a reduction from CorridorTiling, where the reduction function f is a composition of the reduction f_1 from CorridorTiling to ConcatenationUniversality and a function f_2, we give below. We use the construction from the proof of Theorem 8.7.

We only have to show, that there is a polynomial size DRE for the language

$$L = \underbrace{\Sigma\Sigma^*\Sigma + \Sigma L_1\$L_2'\Sigma'}_{L_x} + \Sigma' L_1'\$L_2\Sigma + \Sigma' L_1'\$\$L_2'\Sigma',$$

to apply the same arguments as in Theorem 8.7.

We therefore give a DRE S_2 for $L_2\Sigma$:

$$S_2 = u_F + \sum_{x \in \Sigma \setminus \{u_F\}} x(\varepsilon + \overline{H_x}\Sigma^+ + H_x\Sigma^{n-2}\overline{V_x}\Sigma^+).$$

As $\varepsilon \in L(R_2)$, each symbol from Σ is a string in $L_2\Sigma$. This is the reason, why u_F is added and the ε term is moved inside the brackets of the first sum: we need to accept every symbol from Σ. As every expression inside the first sum of R_2 ends with Σ^*, we can simply exchange Σ^* with Σ^+.

A DRE for L can be derived from R_1 and S_2 (and the corresponding expressions R_1' and S_2' that use Σ' instead of Σ) again by expressing L_x with a deterministic expression We can describe L_x by the expression

$$\Sigma\big(R_1(\Sigma^+ + \$S_2') + \overline{R_1}\Sigma^*\big)$$

and by the deterministic expression

$$\Sigma\big(\varepsilon + \$S_2' + \overline{u_0}\Sigma^*(\varepsilon + \$S_2') + u_0\overline{u_F}^*(\$S_2' + u_F\Sigma^*).$$

□

For our lower bound proofs we still need a slightly stronger version of the StrongPrimality problem, where the difference is, that we additionally require the factors to be definable by almost starless DREs.

Definition 8.12

- A regular expression R is *starless*, if it does not contain the Kleene star and $\varepsilon \notin L(R)$.
- A regular expression R is *almost starless*, if R is starless or R is in one of these forms:
  - $R = \varepsilon$,
  - $R = (a_1 + \cdots + a_n)^*$, for some symbols $a_1, \ldots, a_n$,
  - $R = R_1 + R_2$ for almost starless REs R_1 and R_2,
  - $R = R_1 \cdot R_2$, where R_1 is starless and R_2 is almost starless, or
  - $R = R_1 \cdot R_2$, where R_1 and R_2 are almost starless and $\varepsilon \notin L(R_2)$.

Corollary 8.13

(a) It is PSPACE*-hard to decide given an almost starless DRE R, if there exists a factorization (L_1, L_2) of $L(R)$, such that both languages can be defined by almost starless DREs and both languages do not contain ε.*

(b) It is PSPACE*-hard to decide given almost starless DREs R, R_1, R_2, whether (R_1, R_2) is a factorization of R.*

Proof. We have shown in the proof of Theorem 8.11 that the only possible nontrivial factorization of L (if L has a nontrivial factorization at all) is (L_a, L_b), with $L_a = L_1 + L_1'\$$ and $L_b = L_2 + \$L_2'$.[6] It is easy to verify that the expressions from the proof of Theorem 8.11 are almost starless. □

[6] L, L_a and L_b are defined exactly as in the proof of Theorem 8.11.

8.5 Perfect Typings

One of the main results of [AGM09] is that, if a perfect typing exists, there is only *one* candidate typing that needs to be checked and that an NFA can be efficiently constructed (the *perfect automaton* in [AGM09]) from which this typing can be directly inferred. If this typing is local then it is perfect. Therefore, PERF(NFA) can be solved by generating the candidate typing, testing whether it is local, and verifying whether it is equivalent to the typing in the input.

We recall the complexity results from Abiteboul et al. [AGM09]:

Theorem 8.14 ([AGM09])
(a) PERF(NFA) *is* PSPACE*-complete, and*
(b) $\exists-$PERF(NFA) *is* PSPACE*-complete.*

The results can be easily transferred to regular expressions.

Corollary 8.15
(a) PERF(RE) *is* PSPACE*-complete, and*
(b) $\exists-$PERF(RE) *is* PSPACE*-complete.*

Proof. The upper bounds are by reduction to the corresponding problems for NFAs. We just compute an NFA for every regular expression occurring in the input using polynomial time and apply the algorithm for NFAs.

The lower bound proofs of Theorem 8.14 given in [AGM09] work without modification for regular expressions. □

The PSPACE-hardness for these problems comes from testing whether the generated candidate typing is local. In other words, these problems are PSPACE-hard because testing language equivalence for NFAs and REs is PSPACE-hard.

This motivated us to study the perfect typing problems for DFAs and for deterministic regular expressions, which are known to have a PTIME language equivalence test.

8.5.1 Perfect Typings for DFAs

We first study the perfect typing problems for DFAs and prove that PERF(DFA) and $\exists-$PERF(DFA) can be solved in polynomial time. Our overall technique is reminiscent to the one used for proving Theorem 8.14, but the details are rather different. From a given design $D = (\mathcal{A}, w)$, where $\mathcal{A}$ is a DFA, a candidate automaton (i.e., *perfect automaton*) $\hat{\Omega}(\mathcal{A}, w)$ representing a typing τ can be computed in polynomial time such that D has a perfect typing if and only if $w(\tau) = L(\mathcal{A})$. However, two remarks are essential here, in order to understand the new difficulties: (1) the construction of $\hat{\Omega}(\mathcal{A}, w)$ is completely different from the construction in [AGM09] and (2) it is not straightforward to check $w(\tau) = L(\mathcal{A})$, because $w(\tau)$ is in general non-deterministic (this non-determinism arises from the freedom to choose between remaining in a type τ_i or reading the string w_i to advance to τ_{i+1}). Even if τ consists only of DFAs, the equivalence test $w(\tau) = L$ is

PSPACE-complete in general. We therefore need to adopt an approach in which we need more structural insight in the problem, which is exactly our challenge.

Given a design $D = (\mathcal{A}, w)$, where $\mathcal{A}$ is a DFA $(Q, \Sigma, \delta, s, F)$ and w is a distributed string $w = w_0\, \mathrm{f}_1 \ldots \mathrm{f}_n\, w_n$, we construct the candidate automaton $\Omega(D)$ as follows. We use the extended alphabet $\hat{\Sigma} = \Sigma \uplus \{\sigma_0, \ldots, \sigma_n\}$ and the homomorphism $h : \hat{\Sigma}^* \to \Sigma^*$, where $h(a) = a$ for any $a \in \Sigma$ and $h(\sigma_i) = w_i$ for any $i \in \{1, \ldots, n\}$.

By $\hat{\mathcal{A}}$ we denote the automaton derived from $\mathcal{A}$ by applying the inverse homomorphism h^{-1} to $\mathcal{A}$. More precisely, $\hat{\mathcal{A}} = (Q, \hat{\Sigma}, \hat{\delta}, s, F)$, where

$$\hat{\delta} = \delta \cup \{(q_a, \sigma_i, q_b) | q_b \in \delta^*(q_a, w_i)\}.$$

Since Σ and $\{\sigma_1, \ldots, \sigma_n\}$ are disjoint, $\hat{\mathcal{A}}$ is deterministic. Furthermore it can be constructed in polynomial time.

The *perfect automaton* $\hat{\Omega} = \hat{\Omega}(\mathcal{A}, w)$ is defined as the minimal DFA for

$$L(\hat{\mathcal{A}}) \cap L(\sigma_0 \Sigma^* \sigma_1 \Sigma^* \ldots \Sigma^* \sigma_n).$$

We can construct $\hat{\Omega}$ in polynomial time by performing the standard product construction on $\hat{\mathcal{A}}$ and the (trivial) linear size deterministic automaton for $\sigma_0 \Sigma^* \sigma_1 \Sigma^* \ldots \Sigma^* \sigma_n$. Recall our convention that minimal DFAs do not have (rejecting) sink states and therefore, for some q and σ_i, $\hat{\delta}(q, \sigma_i)$ might be empty. It should be noted that, as $\hat{\Omega}$ is minimal, $\hat{\Omega}$ only depends on the design $D = (L, w)$ and not on the concrete automaton representing L.

Example 8.16 *Figure 8.2 illustrates our construction with two designs. The DFA $\mathcal{A}_1$ of the design $D_1 = (\mathcal{A}_1, \mathrm{f}_1\, \mathrm{f}_2)$ is shown in Figure 8.2a (without the dashed transitions). $\hat{\mathcal{A}}_1$ results from adding the dashed self-loops, as the strings w_0, w_1 and w_2 are empty. The perfect automaton $\hat{\Omega}(D_1)$ is shown in Figure 8.2b. Later, we will see that this design does not have a perfect typing.*

The right half of Figure 8.2 gives a more complicated example, where a perfect typing actually exists. The design is $D_2 = (\mathcal{A}_2, \mathrm{f}_1\, bc\, \mathrm{f}_2)$, where $\mathcal{A}_2$ is the DFA of Figure 8.2d, without the dashed transitions. The DFA $\hat{\mathcal{A}}(D_2)$ is the automaton in Figure 8.2d with the dashed transitions. The two self-loops labeled with σ_0 and σ_2 at each state result again from the empty strings w_0 and w_2. The perfect automaton is shown in Figure 8.2e.

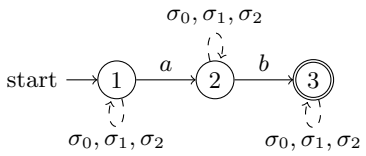
For $i \in [1, n]$, we define the *local candidate automaton* Ω_i as follows. First, let $\hat{\Omega}_i$ be the automaton obtained from $\hat{\Omega}$ by choosing

(i) as initial states those states q with some transition (r, σ_{i-1}, q), and

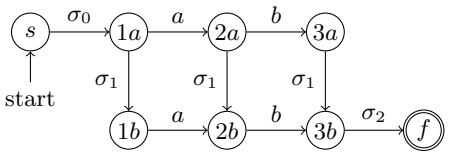
(ii) as final states those states p with some transition (p, σ_i, r').

Then, Ω_i is the automaton obtained from $\hat{\Omega}_i$ by removing all transitions labeled with some σ_j. Notice that, since $\mathcal{A}$ is deterministic, the only nondeterminism of Ω_i is the freedom to choose an initial state. We write $\vec{\Omega}$ for $(\Omega_1, \ldots, \Omega_n)$ and τ_Ω for the typing $(L(\Omega_1), \ldots, L(\Omega_n))$. Figures 8.2c and 8.2f display the respective local automata for the designs of Example 8.16.

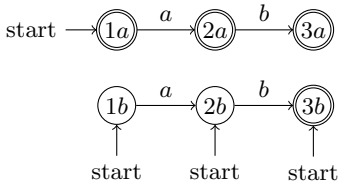
We need the following technical lemma:



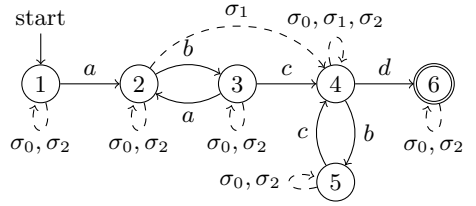
(a) DFA $\hat{\mathcal{A}}_1$ for the design $D_1 = (\{ab\}, \mathrm{f}_1\,\mathrm{f}_2)$



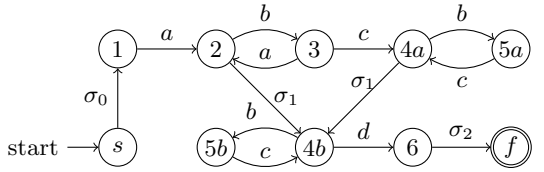
(b) Perfect automaton $\hat{\Omega}(D_1)$



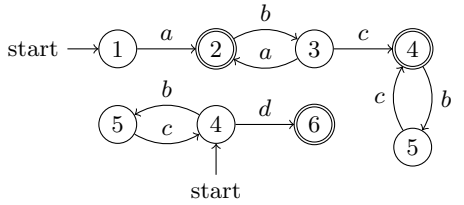
(c) Local candidate automata $\Omega_1(D_1)$ and $\Omega_2(D_1)$.



(d) DFA $\hat{\mathcal{A}}_2$ for the design $D_2 = ((ab)^+c(bc)^*d, \mathrm{f}_1\,bc\,\mathrm{f}_2)$



(e) Perfect automaton $\hat{\Omega}(D_2)$



(f) Local candidate automata $\Omega_1(D_2)$ and $\Omega_2(D_2)$.

Figure 8.2: Two designs: (a)–(c) has no perfect typing, (d)–(f) has a perfect typing.

Lemma 8.17 *Let $w = w_0\,\mathrm{f}_1 \ldots \mathrm{f}_n\,w_n$ be a distributed string, $\mathcal{A}$ be a DFA and $\tau = (L_1, \ldots, L_n)$ be a sound typing for $(\mathcal{A}, w)$. Let τ_Ω be the typing obtained from $\hat{\Omega}(\mathcal{A}, w)$ as described above. Then $\tau \subseteq \tau_\Omega$.*

Proof. To prove that $\tau \subseteq \tau_\Omega$, we show $L_i \subseteq L(\Omega_i)$, for $i \in [1, n]$. To this end, we fix i and $v_i \in L_i$. Furthermore, for each $j \neq i$, $1 \leq j \leq n$, let v_j be some string from L_j and let $\hat{v} = \sigma_0 v_1 \sigma_1 \cdots v_n \sigma_n$. As τ is sound, $v = w_0 v_1 w_1 \cdots v_n w_n \in L(\mathcal{A})$. Since $\hat{v} \in \sigma_0 \Sigma^* \sigma_1 \cdots \Sigma^* \sigma_n$, also $\hat{v} \in L(\hat{\Omega})$. Furthermore, the accepting run of $\hat{\Omega}$ on $\hat{v}$ induces a sequence of transitions from $\hat{\delta}^*$:

$$s_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{v_1}{}^* s_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{v_2}{}^* \cdots \xrightarrow{\sigma_{n-1}} q_n \xrightarrow{v_n}{}^* s_n \xrightarrow{\sigma_n} q_a$$

By definition of Ω_i we immediately get $v_i \in L(\Omega_i)$. Thus, $\tau \subseteq \tau_\Omega$. □

The following theorem is the technical core of this section. It proves that to test whether a design has a perfect typing it suffices to test whether all local candidate

automata have one initial state. Furthermore, the perfect typing is simply the vector of local candidate automata.

Theorem 8.18 *Let $w = w_0 \mathrm{f}_1 \dots \mathrm{f}_n w_n$ be a distributed string and $\mathcal{A}$ a DFA, such that $L(\mathcal{A}) \subseteq w_0\Sigma^* w_1 \cdots \Sigma^* w_n$. Let $\hat{\Omega}$, τ_Ω, and $\vec{\Omega} = (\Omega_1, \dots, \Omega_n)$ be defined as above. Then the following statements are equivalent.*

(a) There is a perfect typing for $(\mathcal{A}, w)$.

(b) τ_Ω is a perfect typing for $(\mathcal{A}, w)$.

(c) τ_Ω is a sound typing for $(\mathcal{A}, w)$.

(d) For each i, Ω_i has exactly one initial state.

Proof. We show the implications (a) ⇒ (d) ⇒ (c) ⇒ (b) ⇒ (a).
(a) ⇒ (d): Let $\tau = (L_1, \dots, L_n)$ be a perfect typing for $(\mathcal{A}, w)$. Towards a contradiction, assume that, for some $i \in [1, n]$, $p \neq q$ are initial states of Ω_i. Since by definition $\hat{\Omega}$ is minimal, there exists a string $u = u_i\sigma_i \cdots u_n\sigma_n$ such that $\delta^*(p, u) \in F$ and $\delta^*(q, u) \notin F$ or vice versa. We assume w.l.o.g. that $\delta^*(p, u) \in F$ — the other case is symmetric. Since by minimality of $\hat{\Omega}$ every state occurs in some accepting run and $L(\hat{\Omega}) \subseteq \sigma_0\Sigma^*\sigma_1 \cdots \sigma_{n-1}\Sigma^*\sigma_n$, there exist strings

- $v = v_i\sigma_i \cdots v_n\sigma_n$ with $\delta^*(q, v) \in F$,
- $u' = \sigma_0 u_1 \sigma_1 \cdots u_{i-1}\sigma_{i-1}$ with $\delta^*(s, u') = p$,
- $v' = \sigma_0 v_1 \sigma_1 \cdots v_{i-1}\sigma_{i-1}$ with $\delta^*(s, v') = q$.

Thus, $u'u$ and $v'v$ are both accepted by $\hat{\Omega}$ and therefore $(\{u_1\}, \dots, \{u_n\})$ and $(\{v_1\}, \dots, \{v_n\})$ are sound typings for $(\mathcal{A}, w)$. By perfectness of τ, both these typings are included in τ, hence, for each i, $\{u_i, v_i\} \subseteq L_i$. But this yields a contradiction as $u'v$ is not accepted by $\hat{\Omega}$, thus τ is not sound. Thus, we can conclude that (d) holds.
(d) ⇒ (c): Let, for each i, q_i be the unique[7] initial state of Ω_i. Let, for each i, $v_i \in L(\Omega_i)$. We need to show that $w_0 v_1 w_1 \cdots v_n w_n \in L(\mathcal{A})$.

For each i, let $s_i = \hat{\delta}^*(q_i, v_i)$. By construction of $\hat{\Omega}_i$ and uniqueness of initial states, we have that $\hat{\delta}^*(s_{i-1}, \sigma_{i-1}) = q_i$ for each i (where s_0 is interpreted as the initial state of $\hat{\Omega}$). Furthermore, $q_a = \hat{\delta}(s_n, \sigma_n)$ is the unique accepting state of $\hat{\Omega}$. Together,

$$s_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{v_1}{}^* s_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{n-1}} q_n \xrightarrow{v_n}{}^* s_n \xrightarrow{\sigma_n} q_a$$

is an accepting computation of $\hat{\Omega}$, hence for $\hat{A}$ and therefore

$$s_0 \xrightarrow{w_0}{}^* q_1 \xrightarrow{v_1}{}^* s_1 \xrightarrow{w_1}{}^* \cdots \xrightarrow{w_{n-1}}{}^* q_n \xrightarrow{v_n}{}^* s_n \xrightarrow{w_n}{}^* q_a$$

is an accepting computation of $\mathcal{A}$.

[7] It should be noted that by construction q_i is also the unique initial state of $\hat{\Omega}_i$.

(c) ⇒ (b): Let $v = w_0v_1w_1 \cdots v_nw_n \in L(\mathcal{A})$. It follows that $\tau = (\{v_1\}, \ldots, \{v_n\})$ is a sound typing for $(\mathcal{A}, w)$. By Lemma 8.17, $\tau \subseteq \tau_\Omega$ and thus $v \in w(\tau_\Omega)$, therefore τ_Ω is complete, hence local. Applying Lemma 8.17 again, immediately yields that τ_Ω is perfect.
(b) ⇒ (a): Immediate. □

Using Theorem 8.18, we can prove that the perfect typing problems are tractable for DFAs.

Theorem 8.19

(a) PERF(DFA) *is in* PTIME *and*

(b) ∃−PERF(DFA) *is in* PTIME.

Proof. We start with (b). To test whether $(\mathcal{A}, w)$ has a perfect typing for a given a distributed string w and a DFA $\mathcal{A}$ we first check if $L(\mathcal{A}) \subseteq w_0\Sigma^*w_1 \cdots \Sigma^*w_n$. This can be easily done in polynomial time, as there is a DFA of linear size in w, for the language $w_0\Sigma^*w_1 \cdots \Sigma^*w_n$. If the inclusion does not hold, there can be no perfect typing, as there can be no complete typing.

Next, we construct $\hat{\Omega}(\mathcal{A}, w)$ in polynomial time. We test in polynomial time if there is an $i \in [1, n]$ and (at least) two different states p and q, such that there are incoming transitions in p and q labeled by σ_i in $\hat{\Omega}$. By Theorem 8.18, $(\mathcal{A}, w)$ has a perfect typing if and only if there is no such i. This shows (b).

We continue with (a). To test whether a given typing τ is perfect for the design $(\mathcal{A}, w)$, we first check as in (b) whether a perfect typing exists. If this is the case, it remains to test whether $\tau = \tau_\Omega$. The latter can be done in polynomial time as it only involves equivalence tests for DFAs. □

8.5.2 Perfect Typings for Deterministic Regular Expressions

In real DTDs and XML Schema specifications content models are described by *deterministic* regular expressions (see Chapter 7). This raises the question how to solve the perfect typing problem for DREs. We first show that the case of deterministic regular expressions is quite different from the case of finite automata. In particular, there are designs with perfect typings that cannot be specified by deterministic regular expressions.

Theorem 8.20 *There is a design $D = (R, w)$ with a DRE R for which the (unique) perfect typing is not expressible by DREs.*

Proof. We show that the perfect typing for the design D_2 of Example 8.16 cannot be specified by DREs. The global schema of D_2 is specified by the DRE $R = (ab)^+c(bc)^*d$. As we argued in Example 8.16, the DFAs $\Omega_1 = \Omega_1(D_2)$ and $\Omega_2 = \Omega_2(D_2)$ describe a perfect typing $\tau = (L(\Omega_1), L(\Omega_2))$ for D_2, since they both have only one initial state.

We show that there can be no DRE for $L(\Omega_1)$, by showing that Ω_1 does not fulfill the orbit property as defined in Chapter 7.2. The minimal DFA for Ω_1 is depicted in Figure 8.2f and has state set $\{1, 2, 3, 4, 5\}$. Consider the orbit $\mathcal{O}(2) = \mathcal{O}(3)$. Both states

2 and 3 are gates of this orbit, as 2 is a final state and 3 has the transition $\delta(3, c) = 4$, which leaves the orbit.

In violation of the orbit property, 2 is final, but 3 is not. From Theorem 7.2 it follows that there can be no DRE for $L(\Omega_1)$. □

Theorem 8.20 shows that DREs require some care. However, computing whether there exist perfect typings is still feasible as stated in the next result.

Theorem 8.21

(a) PERF(DRE) *is in* PTIME *and*

(b) $\exists$−PERF(DRE) *is in* PTIME.

Proof. (a): Whether a typing $\tau = (L(R_1), \ldots, L(R_n))$ with DREs $R_1, \ldots, R_n$ is perfect for a design $D = (R, w)$ can be easily tested by translating R and each R_i into an equivalent Glushkov automaton (in quadratic time) and applying the algorithm of Theorem 8.18.

(b): Given a design $D = (R, w)$, where R is a DRE, an equivalent DFA A for R can again be computed in quadratic time. If a perfect typing $\tau = L_1, \ldots, L_n$ for (A, w) exists, it is unique and its DFA representation $A_1, \ldots, A_n$ can be computed in polynomial time (according to Section 8.5.1). Finally, it can be tested in PTIME if every $L(A_i)$ is DRE-definable [BKW98]. □

We note, that the proof does not show that perfect typings can be computed in polynomial time, as DREs can be exponentially larger than equivalent DFAs.

8.6 Normal Form Typings

For a given design there can be an infinite number of local typings. We show in this section that we can reduce the search space considerably by only considering typings of particular normal forms, which are based on automaton representations of the given languages.

Let, in the following, $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ always denote some NFA. An *$\mathcal{A}$-transformation* is a mapping $Q \to 2^Q$, i.e., a function that maps states of $\mathcal{A}$ to sets of states of A. For a string w, we denote by $T_w^{\mathcal{A}}$ the $\mathcal{A}$-transformation *induced by* w, i.e., the function $p \mapsto \delta^*(p, w)$. Given an $\mathcal{A}$-transformation T we write $L_{\text{trans}}(\mathcal{A}, T)$ for the set of strings w with $T_w^{\mathcal{A}} = T$. We say a typing $(L_1, \ldots, L_n)$ is in $\mathcal{A}$ *normal form ($\mathcal{A}$-NF)* if each L_i is a union of languages of the form $L_{\text{trans}}(\mathcal{A}, T)$.

If $\mathcal{A}$ is a DFA we consider a stronger normal form: For two sets X, Y of states of $\mathcal{A}$ let $L_\cap(\mathcal{A}, X, Y)$ denote the set of all strings w, for which $\delta^*(p, w) \in Y$, for every $p \in X$. A typing $(L_1, \ldots, L_n)$ is in *strong $\mathcal{A}$-normal form (strong $\mathcal{A}$-NF)*, if each L_i is of the form $L_\cap(\mathcal{A}, X_i, Y_i)$, for some $X_i, Y_i \subseteq Q$.

The idea behind the strong $\mathcal{A}$-NF is, that each set X is chosen as a subset of the initial states of the corresponding local automaton as constructed in Section 8.5. Each set Y is then chosen as the set of states, such that a state of the next X-set is reached by reading the next string w_i.

Remark *The specialization of the strong $\mathcal{A}$-NF with $i = 2$ and all $w_i = \varepsilon$ was already used by Salomaa et al. [SY99] under the term* decomposition sets. *We used this special case to show the* PSPACE *upper bound for* Primality *for DFAs in Section 8.4.*

Remark *It follows from the results of Section 8.5 that a perfect typing for a design $(\mathcal{A}, w)$ with a DFA $\mathcal{A}$ is always in strong $\mathcal{A}$-NF.*

Even though local typings do not need to be of this particular simple type (as we will see below), we show next that $\mathcal{A}$-NF typings and strong $\mathcal{A}$-NF typings deserve their names.

Theorem 8.22 *Let $\mathcal{A}$ be an NFA, and $\tau = (L_1, \ldots, L_n)$ a local typing for the design $D = (\mathcal{A}, w(\mathrm{f}_1 \cdots \mathrm{f}_n))$.*

(a) Then there exists an $\mathcal{A}$-NF local typing τ' for D such that $\tau \subseteq \tau'$.

(b) If $\mathcal{A}$ is a DFA there exists a strong $\mathcal{A}$-NF local typing τ' for D such that $\tau \subseteq \tau'$.

Proof. (a): For each $i = 1, \ldots, n$ we let

$$L'_i = \bigcup_{w \in L_i} L_{\mathrm{trans}}(\mathcal{A}, T^{\mathcal{A}}_w),$$

i.e., the set of strings for which there is some $w \in L_i$ with the same $\mathcal{A}$-transformation. Let $\tau' = (L'_1, \ldots, L'_n)$. As, in particular, each string $w \in L_i$ is in L'_i, we immediately get $\tau \subseteq \tau'$.

It remains to show that τ' is a sound typing for $(\mathcal{A}, w)$. To this end, let, for each i, $v_i \in L'_i$. For each i, there is some $u_i \in L_i$ with $T^{\mathcal{A}}_{u_i} = T^{\mathcal{A}}_{v_i}$. As τ is a sound typing, $\mathcal{A}$ has an accepting run on $w_0 u_1 w_1 \cdots u_n w_n$. As each v_i has the same $\mathcal{A}$-transformation as the respective u_i, $w_0 v_1 w_1 \cdots v_n w_n$ is accepted by $\mathcal{A}$ as well.

(b): Let $Y = \{q_0\}$ be the singleton set containing the initial state of $\mathcal{A}$. For each $i \in [1, n]$, we let $L'_i = L_\cap(A, X_i, Y_i)$, where $X_i = \bigcup_{q \in Y_{i-1}} \delta^*(q, w_{i-1})$ and $Y_i = \bigcup_{q \in X_i} \bigcup_{w \in L_i} \delta^*(q, w)$.

Let $\tau' = (L'_1, \ldots, L'_n)$. As, in particular, each string $w \in L_i$ is also in L'_i, we immediately get $\tau \subseteq \tau'$.

Clearly, τ' is in strong $\mathcal{A}$-NF. It remains to show that it is a sound typing for $(\mathcal{A}, w)$. Let therefore v_i be a string from L'_i for each $i \in [1, n]$. We use the following claim, which we prove later:

Claim 8.23 *For each $i \in [1, n]$ the following conditions hold:*

(i) For each $q \in Y_i$, there is a string $v \in w_0 L_1 \cdots w_{i-1} L_i$ such that $\delta^*(s, v) = \{q\}$.

(ii) $w_0 L'_1 w_1 L'_2 \cdots w_{i-1} L'_i \subseteq L(\mathcal{A}^{Y_i})$.

Here, $\mathcal{A}^{Y_i}$ denotes the automaton $\mathcal{A}$ with final state set Y_i, i.e., $\mathcal{A}^{Y_i} := (Q, \Sigma, \delta, q_0, Y_i)$.

By (ii) for every string $vw_n \in w_0L'_1w_1 \cdots w_{n-1}L'_nw_n$ we have $p = \delta^*(q_0, v) \in Y_n$. We can conclude by (i) that there is a string $v \in w_0L_1w_1 \cdots L_n$ such that $\delta^*(q_0, v) = p$. As τ is sound, $vw_n \in L(\mathcal{A})$ and thus $\delta^*(p, w_n) \in F$.

We still need to proof Claim 8.23: We let $Y_0 = \{q_0\}$ and prove (i) and (ii) by simultaneous induction on i, for every $i \in [0, n]$. Clearly (i) and (ii) hold for $i = 0$ (as they only refer to the empty string).

Now let $i \geq 1$ and $q \in Y_i$. By definition of Y_i there are $p \in X_i$ and $w \in L_i$ such that $\delta^*(p, w) = q$. By definition of X_i, $\delta^*(r, w_{i-1}) = p$, for some $r \in Y_{i-1}$. By induction, there is a string $v \in w_0L_1 \cdots w_{i-1}L_{i-1}$ such that $\delta^*(q_0, v) = r$. Thus, $\delta^*(q_0, vw_{i-1}w) = \{q\}$ and (i) follows.

Now let $w_0v_1 \cdots w_{i-1}v_i$ be a string in $w_0L'_1 \cdots w_{i-1}L'_i$ and $p = \delta^*(q_0, w_0v_1 \cdots v_{i-1})$. By induction, $p \in Y_{i-1}$. By definition of L'_i there is a state $q \in Y_i$ such that $\delta^*(p, w_{i-1}v_i) = q$. Thus, $\delta^*(q_0, w_0v_1 \cdots w_{i-1}v_i) = q \in Y_i$ and (ii) follows. □

Remark *Clearly, if τ is a maximal local typing then τ' is equivalent to τ, as $\tau' \subseteq \tau$ (τ is maximal) and $\tau \subseteq \tau'$ (by Theorem 8.22). Therefore, even if not every local typing* has *an equivalent normal form typing but only* is contained *in a sound (and maximal) normal form typing, we consider the term "normal form" adequate.*

Theorem 8.22 shows that, if one is interested in the existence of a (local, maximal local, perfect) typing, it is always sufficient to look for (strong) A-NF typings. Furthermore, it shows that *every* maximal local typing is equivalent to some typing in normal form.

The next theorem shows why normal forms are interesting from a complexity-theoretic point of view: we can define the languages in normal form typings by means of "small" finite automata.

Theorem 8.24 *Let $\mathcal{A}$ be an NFA, $D = (\mathcal{A}, w(\mathrm{f}_1 \cdots \mathrm{f}_n))$ a design, and $\tau = (L_1, \ldots, L_n)$ a typing for D.*

(a) If τ is in $\mathcal{A}$-normal form then, for each $i \in [1, n]$, there is a DFA $\mathcal{B}$ of exponential size in $|\mathcal{A}|$ such that $L(\mathcal{B}) = L_i$.

(b) If $\mathcal{A}$ is a DFA and τ is in strong $\mathcal{A}$-normal form, then, for each $i \in [1, n]$, there is an AFA $\mathcal{B}$ of polynomial size in $|\mathcal{A}|$ such that $L(\mathcal{B}) = L_i$.

Proof. (a) The DFA $\mathcal{B}$ simply keeps track of the transformation $T^{\mathcal{A}}_w$ induced by the input string.

Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be an NFA with state set $Q = \{q_1, \ldots, q_m\}$ and let $L = \bigcup_{j=1}^{k} L_{\mathrm{trans}}(\mathcal{A}, T_j)$ be an $\mathcal{A}$-NF language. The DFA $\mathcal{B}$ is defined as $((2^Q)^m, \Sigma, \delta_B, I_B, F_B)$, where

- the transition function δ_B is defined by

$$\delta_B((Q_1, \ldots, Q_m), a) = (\delta(Q_1, a), \ldots, \delta(Q_{|Q|}, a)).$$

  Here, as usual $\delta_A(Q_i, a) = \bigcup_{p \in Q_i} \delta_A(p, a)$, for every i;

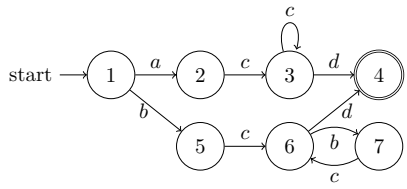


Figure 8.3: Automaton $\mathcal{A}$ of Example 8.25

- the initial state set I_B is $(\{q_1\}, \ldots, \{q_m\})$;
- F_B consists of all states $(T_j(q_1), \ldots, T_j(q_m))$, $j \leq k$.

(b) Let $\mathcal{A}$ be a DFA and $\tau = (L_1, \ldots, L_n)$ a typing for D in strong $\mathcal{A}$-normal form. For each i, $L_i = L_\cap(\mathcal{A}, X, Y)$ for some X and Y. Let $\mathcal{B}$ be the AFA that first universally branches to the states in X and then simulates deterministically (on all branches) $\mathcal{A}$ on w. Its accepting states are the states in Y. □

We note that the bounds of Theorem 8.24 do *not* apply to DFAs for the languages $w(\tau)$ as the concatenation of languages re-introduces nondeterminism. However, we can conclude a double-exponential size bound for DFAs for the languages $w(\tau)$.

It is tempting to hope for stronger normal forms for typings. For example, if in Theorem 8.22(b) all languages in τ' were of the form $L_\cap(\mathcal{A}, X, Y)$ with singleton X, then we could use polynomial-size NFAs instead of polynomial-size AFAs in Theorem 8.24(b). However, the following example shows that this is not possible and that therefore, our normal forms are, in a sense, optimal.

Example 8.25 *Let $\mathcal{A}$ be a DFA for the language $ac^+d + (bc)^+d$. and $D = (\mathcal{A}, \mathrm{f}_1\, c\, \mathrm{f}_2)$. $\mathcal{A}$ is depicted in Figure 8.3. The only local (and thus also maximal local) typing for D is $\tau = (ac^* + b(cb)^*, d) = (L_\cap(\mathcal{A}, \{1\}, \{2, 3, 5, 7\}), L_\cap(\mathcal{A}, \{3, 6\}, \{4\}))$.*

Notice that there is no single state q of $\mathcal{A}$ such that there exists a local typing of the form $(L_\cap(\mathcal{A}, X_1, Y_1), L_\cap(\mathcal{A}, \{q\}, Y_2))$ for D.

8.7 Verification of Typings

In this section, we study the complexity of testing whether a given typing is local or maximal local for a given design. To test whether a given typing is local has the same complexity as the ConcatenationEquivalence problem for the same formalism of specifying languages.

Lemma 8.26 *Let $\mathcal{X}$ be a formalism to specify languages such that, given a word w, one can construct a representation of the singleton language $\{w\}$ in $\mathcal{X}$ in logarithmic space. Then* LOC($\mathcal{X}$) *can be logspace reduced to* ConcatenationEquivalence($\mathcal{X}$) *and vice versa.*

Proof. Obviously ConcatenationEquivalence($\mathcal{X}$) is the special case of LOC($\mathcal{X}$), where all w_i are ε. For the other direction, let $(L, w_0\,\mathrm{f}_1\,w_1\,\mathrm{f}_2 \ldots \mathrm{f}_n\,w_n)$ be a design and $(L_1, L_2, \ldots, L_n)$ be a typing, where all languages are given by the formalism $\mathcal{X}$. For $i \in \{0, \ldots, n\}$ we compute representations X_i such that $L(X_i) = \{w_i\}$. The given typing is local, if and only if $L = X_0 L_1 X_1 \ldots L_n X_n$. By assumption, all X_i can be computed in LOGSPACE. □

Corollary 8.27 *The problems* LOC(NFA)*,* LOC(DFA)*,* LOC(RE)*, and* LOC(DRE) *are* PSPACE*-complete.*

Proof. The result follows from Lemma 8.26 and the fact that ConcatenationEquivalence is PSPACE-complete for DFAs [JR93] (and therefore also for NFAs and REs) and DREs (Corollary 8.10). □

The result for LOC(NFA) was already published in [AGM09].

Theorem 8.28 *The problems* ML(NFA)*,* ML(DFA)*,* ML(RE) *and* ML(DRE) *are in* PSPACE*.*

Proof. We start with ML(NFA). Let $D = (\mathcal{A}, w)$. We first show that a local typing $\tau = (L_1, \ldots, L_n)$ is *not* maximal for D if and only if there is an i, $1 \leq i \leq n$ and an $\mathcal{A}$-transformation T such that

(1) $(L_1, \ldots, L_{i-1}, L_{\mathrm{trans}}(\mathcal{A}, T), L_{i+1}, \ldots, L_n)$ is sound for D and

(2) $L_{\mathrm{trans}}(\mathcal{A}, T) - L_i \neq \emptyset$.

The "if" statement holds by definition of "maximal". For the "only if" statement let us assume that $\tau \subsetneq \tau''$, for some local typing τ''. By Theorem 8.22, there is an $\mathcal{A}$-NF typing $\tau' = (L'_1, \ldots, L'_n)$ such that $\tau'' \subseteq \tau'$, thus $\tau \subsetneq \tau'$. Therefore, there is some i such that $L_i \subsetneq L'_i$. By definition of $\mathcal{A}$-NF typings there is an $\mathcal{A}$-transformation T such that $L_{\mathrm{trans}}(\mathcal{A}, T) \subseteq L'_i$ but $L_{\mathrm{trans}}(\mathcal{A}, T) \not\subseteq L_i$.

Whether a given typing τ is maximal and local can thus be tested as follows.

(a) Test whether τ is local.

(b) For each i and T.

- Check (1) and (2) above.

If there exist i and T such that (1) and (2) do not hold, than τ is not maximal.

To test (1) it is sufficient to construct an NFA $\mathcal{A}'$ for

$$w(L_1, \ldots, L_{i-1}, L_{\mathrm{trans}}(\mathcal{A}, T), L_{i+1}, \ldots, L_n) \cap \overline{L}$$

and to verify that $L(\mathcal{A}') = \emptyset$. It is not hard to see, that there is such an NFA of exponential size which can be represented succinctly in polynomial space and therefore its non-emptiness can be tested in (nondeterministic thus also deterministic) polynomial space.

Condition (2) can be easily tested in polynomial space.

This finishes the proof for the upper bound of ML(NFA). The upper bound for ML(DFA) follows immediately, as it is just a special case of ML(NFA). The upper bound for ML(RE) follows easily by converting the regular expressions to polynomial size NFAs.

It remains to prove the upper bound of ML(DRE). For testing maximal locality, we translate the given DREs into NFAs and use the upper bound algorithm from above. It is not obvious that this is correct: a typing defined by DREs could be found non-maximal by the algorithm because there is a larger typing that is not DRE-definable. However, if there exists a larger typing that is not DRE-definable, then there is also a larger typing that is DRE-definable. The reason is that, for every DRE-definable language L and string w, the language $L \cup \{w\}$ is also DRE-definable (Lemma 10 in [BGMN09]). Let $\tau = (L_1, \ldots, L_n)$ be a DRE-definable typing and $\tau' = (L_1, \ldots, L'_i, \ldots, L_n)$ be a larger non DRE-definable typing. Than the typing $\tau'' = (L_1, \ldots, L_i \cup \{w\}, \ldots, L_n)$ with $w \in L'_i \setminus L_i$ is a larger typing than τ and DRE-definable. Thus τ cannot be a maximal DRE-definable typing. This shows that ML(DRE) is in PSPACE. □

We note that the result for ML(NFA) was already stated in [AGM09]. However, the presented proof for the upper bound is not correct. It claims that, for a design (L, w) a typing $\tau = (L_1, \ldots, L_n)$ is *not* maximal, if there is an i, such that

$$w(L_1, \ldots, L_{i-1}, \overline{L_i}, L_{i+1}, \ldots, L_n) \cap L \neq \emptyset.$$

This is not true: if $L = \{a, aa\}$ and $w = \mathrm{f}_1\,\mathrm{f}_2$, the typing $\tau = (L_1, L_2)$ with $L_1 = \{\varepsilon, a\}$ and $L_2 = \{a\}$ is maximal, even though the string aa is in $w(L_1, \overline{L_2}) \cap L$.

Lower Bounds for Maximal Local Typings

To show PSPACE lower bounds for ML(DFA) and ML(DRE), we exploit the PSPACE-hardness of StrongPrimality for DFAs and DREs.

For each language L with $\varepsilon \notin L$, let $L^{\#}$ be the language

$$L^{\#} = \{a_1 \# a_2 \# \ldots \# a_n \mid a_1 a_2 \ldots a_n \in L\}$$

and $L_{\#}$ be the language

$$L_{\#} = \{a_1 \# a_2 \# \ldots \# a_n \# \mid a_1 a_2 \ldots a_n \in L\},$$

where $\#$ is a fresh symbol that does not occur in L. Furthermore for each language L with $\varepsilon \notin L$, we define the design $D_L^{\#}$ to be $D_L^{\#} = (L_{\#}, \mathrm{f}_1 \,\#\, \mathrm{f}_2 \,\#)$. For the opposite transformation, we define for each string v and each language L, the string $v^{\natural}$ and language $L^{\natural}$, that result from v and L, by removing all occurrences of $\#$.

The following lemma will be crucial in our lower bound proofs for ML and $\exists-$ML.

Lemma 8.29 *For any language L, it holds that*

(a) every decomposition $L = L_1 L_2$ of L with $\varepsilon \notin L_1 \cup L_2$ translates to a local typing $(L_1^{\#}, L_2^{\#})$ for $D_L^{\#}$; and

(b) every local typing (L_1, L_2) *of* $D_L^\#$ *translates to a decomposition* $L = L_1^\natural L_2^\natural$ *of* L*, such that* $\varepsilon \notin L_1^\natural \cup L_2^\natural$.

Proof. We first prove (a). To this end, suppose that there is a nontrivial strong factorization (L_1, L_2) of L (in which neither L_1 nor L_2 contain ε). We claim that then, $(L_1^\#, L_2^\#)$ is a local typing for $D_L^\#$. To this end, for the direction $L_1^\# \cdot \# \cdot L_2^\# \cdot \# \subseteq L_\#$, let $u \in L_1^\#$ and $v \in L_2^\#$. Since $u \neq \varepsilon \neq v$, we have that both u and v start and end with Σ-symbols, $u\#v\#$ alternates between Σ-symbols and $\#$, and that $u^\natural v^\natural \in L$. Hence, $u\#v\# \in L_\#$ by definition of $L_\#$. Conversely, let s be a string in $L_\#$ and let $s^\natural$ be obtained from s by removing all occurrences of $\#$. By definition of $L_\#$, $s^\natural \in L$. Therefore, there exist $u \in L_1$ and $v \in L_2$ with $u \neq \varepsilon \neq v$ such that $uv = s$. Hence, $u^\# \# v^\# \# = s$ with $u^\# \in L_1^\#$ and $v^\# \in L_2^\#$.

Towards (b), suppose that there is a local typing $\tau = (L_1, L_2)$ for $D_L^\#$. We will prove that then $(L_1^\natural, L_2^\natural)$ is a nontrivial decomposition of L with $\varepsilon \notin L_1^\natural \cup L_2^\natural$. First of all, since all strings in $L_\#$ start with a Σ-symbol and alternate between Σ-symbols and $\#$, we have that all strings in $L_1^\natural$ and $L_2^\natural$ have length at least one.

So it remains to prove that $L = L_1^\natural L_2^\natural$. To this end, let s be a string in L and let $s_\#$ be its corresponding string in $L_\#$. Since (L_1, L_2) is a local typing, there exist $u \in L_1$ and $v \in L_2$ such that $s_\# = u \cdot \# \cdot v \cdot \#$. Since u and v start and end with a Σ-symbol, we have that $s = u^\natural v^\natural$ and $u^\natural \neq \varepsilon \neq v^\natural$. Conversely, let u be a string in $L_1^\natural$ and v be a string in $L_2^\natural$. Since (L_1, L_2) is a local typing, there exists a string $s_\# \in L_\#$ such that $u^\# \# v^\# \# = s_\#$, where $u^\#$ and $v^\#$ are the strings corresponding to u and v. But by definition, $s \in L$. Therefore we also have that $L_1 L_2 \subseteq L$ and (L_1, L_2) is indeed a nontrivial decomposition of L such that neither language contains ε. □

Now we can show lower bounds for the ML problem.

Lemma 8.30 ML(DFA) *is* PSPACE*-hard.*

Proof. The language $L = \Sigma\Sigma^*\Sigma \cup L_1\$L_2' \cup L_1'\$L_2 \cup L_1'\$\$L_2'$ constructed in the proof of Theorem 8.7 has the decomposition (L_a, L_b) with $L_a = L_1 \cup L_1'$ and $L_b = L_2 \cup \$L_2'$, if and only if $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2) = \Sigma^*$ and no nontrivial decomposition otherwise (proof of Theorem 8.7, Claim 8.6).

The reduction from the proof of Lemma 8.29 translates every nontrivial decomposition of the language L into a typing for the design $D_L^\# = (L_\#, \mathrm{f}_1 \,\#\, \mathrm{f}_2 \,\#)$ and vice versa.

We can therefore solve the original ConcatenationUniversality instance by testing whether $\tau = (L_a^\#, L_b^\#)$ is a local typing for $D^\#$. As there can be no other local typing for $D^\#$ (which follows from Claim 8.6), τ is a local typing for $D^\#$, if and only if it is a maximal local typing for $D^\#$.

As DFAs for the languages $L_\#$, $L_a^\#$ and $L_b^\#$ can be constructed from the given DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ in LOGSPACE, we have a LOGSPACE-reduction from the PSPACE-hard problem ConcatenationUniversality to ML(DFA). □

At first glance, it looks trivial to adopt the proof of Lemma 8.30 to ML(DRE). However it is not clear whether the used languages are definable by DREs and whether DREs for the languages can be computed in LOGSPACE.

Given a DRE R for a language L it is easy to compute a DRE $R_\#$ for the language $L_\#$ by replacing every occurrence of a symbol a with $a \cdot \#$. However it is more complicated to produce DREs for the language $L^\#$.

For the proof of Lemma 8.41 we need some preparation. We use the following fact:

Fact 8.31 ([BKW98]) *Let R_1 and R_2 be DREs (none of which denote $\emptyset$).*

(a) $R_1 + R_2$ is a DRE if and only if $\text{first}(R_1) \cap \text{first}(R_2) = \emptyset$.

(b) $R_1 \cdot R_2$ is a DRE if and only if $\text{followlast}(R_1) \cap \text{first}(R_2) = \emptyset$.

Lemma 8.32

(a) For any almost starless DRE R, the language $L^\# = \{a_1 \# \cdots \# a_n \mid a_1 \cdots a_n \in L(R)\}$ can be represented by a DRE $R^\#$ which is polynomial in the size of R.

(b) The DRE $R^\#$ can be constructed in polynomial time.

Proof. We first define a recursive function f, which defines a DRE for $L(R)^\#$ from an almost starless DRE R.

$$\mathrm{f}(R) = \begin{cases} R & \text{if } R = a \text{ or } R = \varepsilon \\ \varepsilon + X(\#X)^* & \text{if } R = X^* \\ \mathrm{f}(R_1) + \mathrm{f}(R_2) & \text{if } R = R_1 + R_2 \\ R_\# \, \mathrm{f}(R_2) & \text{if } R = R_1 R_2 \text{ and } \varepsilon \notin L(R_2) \\ \mathrm{f}(R_1)(\varepsilon + \# \, \mathrm{g}(R_2)) & \text{if } R = R_1 R_2 \text{ and } \varepsilon \in L(R_2) \end{cases}$$

where a is some symbol from Σ, R_1 and R_2 are almost starless DREs, and $X = (a_1 + \cdots + a_n)$ for some symbols $a_1, \ldots, a_n$. We note that by definition of almost starless DREs, R_1 has to be starless in the last case, i.e., $\mathrm{f}(R_1)$ does not end with $(\#X)^*$ in the last case.

The function g used in the definition of f is identical to f with the difference that it maps R to an expression that cannot generate ε.

$$\mathrm{g}(R) = \begin{cases} \mathrm{f}(R) & \text{if } \varepsilon \notin L(R) \\ \emptyset & \text{if } R = \varepsilon \\ \mathrm{g}(R_1) + \mathrm{g}(R_2) & \text{if } R = R_1 + R_2 \\ X(\#X)^* & \text{if } R = X^* \end{cases}$$

Clearly, f can be computed in logarithmic space. By induction (and case inspection) it is also easy to see that $\mathrm{f}(R)$ is indeed an RE for $L(R)^\#$.

It remains to show that $\mathrm{f}(R)$ is a DRE if R is an almost starless DRE. This is again shown by induction and the cases $R = a$, $R = \varepsilon$, $R = X^*$, are straightforward. The case $R = R_1 + R_2$ follows by Fact 8.31 (a): As $R_1 + R_2$ is a DRE, $\text{first}(R_1) \cap \text{first}(R_2) = \emptyset$.

However, $\text{first}(R) = \text{first}(\text{f}(R))$ holds for every DRE R and thus, again by Fact 8.31 (a), $\text{f}(R_1 + R_2)$ is also a DRE.

For starless DREs R it is easy to show that $\text{followlast}(R) = \emptyset$. Thus, Fact 8.31 (b) implies (in both possible cases) that $\text{f}(R_1 \cdot R_2)$ is a DRE. □

Now we can prove the lower bound for ML(DRE).

Lemma 8.33 ML(DRE) *is* PSPACE*-hard.*

Proof. Let R, R_1 and R_2 be almost starless DREs such that R has at most one nontrivial factorization. From Corollary 8.13b we know that testing whether $L(R) = L(R_1) \cdot L(R_2)$ is PSPACE-hard.

By Lemma 8.29 we know that $L(R) = L(R_1) \cdot L(R_2)$, if and only if $(L(R_1)^\#, L(R_2)^\#)$ is the only (and therefore maximal) local typing for $D^\#_{L(R)} = (L(R)_\#, \text{f}_1 \,\#\, \text{f}_2\,\#)$ and by Lemma 8.32 we can construct DREs for $L(R_1)^\#$ and $L(R_2)^\#$ in polynomial time. Furthermore it is easy to construct a DRE for $L(R)_\#$ in polynomial time. This concludes the proof. □

8.8 Existence of Typings

In the last section, we verified given typings. Now, we turn to the problem of testing whether there exists a typing of a certain kind.

We can conclude from Theorem 8.3, that the problems ∃−LOC and ∃−ML coincide for NFAs, DFAs and REs.

Theorem 8.34 *The problem* ∃−LOC(NFA) *is in* EXPSPACE*.*

Proof. According to Theorem 8.22, it suffices to test $\mathcal{A}$-normal form typings and according to Theorem 8.24, each language L_i in an $\mathcal{A}$-NF typing $(L_1, \ldots, L_n)$ can be represented by a DFA of exponential size in $|\mathcal{A}|$. We can conclude that $w_0 L_1 w_1 \cdots L_n w_n$ can be represented by an NFA of exponential size. Since equivalence between such an NFA and $\mathcal{A}$ can also be tested in exponential space, we can test ∃−LOC(NFA) in EXPSPACE by testing whether any $\mathcal{A}$-NF typing is local for $L(\mathcal{A})$. □

Corollary 8.35 *The problems* ∃−LOC(DFA)*,* ∃−LOC(RE)*,* ∃−ML(NFA)*,* ∃−ML(DFA) *and* ∃−ML(RE) *are in* EXPSPACE*.*

Proof. The result carries over to ∃−LOC(DFA), as DFAs are a special case of NFAs. It carries over to ∃−LOC(RE) by a simple reduction to ∃−LOC(NFA) which computes an NFA for the given regular expression. Finally the result carries over to ∃−ML(NFA), ∃−ML(DFA) and ∃−ML(RE) as these problems coincide with ∃−LOC(NFA), ∃−LOC(DFA) and ∃−LOC(RE). □

We can prove better upper bounds in the case of only two function calls.

Theorem 8.36

(a) $\exists$−2LOC(NFA) *is in* NEXPTIME.

(b) $\exists$−2LOC(RE) *is in* NEXPTIME.

(c) $\exists$−2LOC(DFA) *is in* PSPACE.

Proof. We start with (a). Let $D = (\mathcal{A}, w)$ be a design, where $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is an NFA and w is of the form $w_0\,\mathrm{f}_1\,w_1\,\mathrm{f}_2\,w_2$. The algorithm that decides whether such a design has a local typing simply guesses an $\mathcal{A}$-NF typing $\tau = (L_1, L_2)$ and verifies that it is indeed a local typing, i.e., that $L(\mathcal{A}) = w_0 L_1 w_1 L_2 w_2$.

As τ is in $\mathcal{A}$-normal form, L_1 and L_2 can be written as $L_1 = \cup_{T \in \mathcal{T}_1} L_{\mathrm{trans}}(\mathcal{A}, T)$ and $L_2 = \cup_{T \in \mathcal{T}_2} L_{\mathrm{trans}}(\mathcal{A}, T)$ for two sets $\mathcal{T}_1$ and $\mathcal{T}_2$ of A-transformations, respectively. Clearly, each of the sets $\mathcal{T}_1$ and $\mathcal{T}_2$ is of at most exponential size in $|\mathcal{A}|$ and can thus be guessed within exponential time.

Thus, it remains to be shown that the equality $L(\mathcal{A}) = w_0 L_1 w_1 L_2 w_2$ can be tested in exponential time. To this end, we construct an NFA $\mathcal{B}$ of exponential size in $|\mathcal{A}| + |w|$ that accepts the symmetric difference of $L(\mathcal{A})$ and $w_0 L_1 w_1 L_2 w_2$. Thus the equality holds if and only if $L(\mathcal{B}) \neq \emptyset$, which can be tested in logarithmic space in $|\mathcal{B}|$ and thus in exponential time.

The basic idea is that $\mathcal{B}$ simultaneously simulates $\mathcal{A}$ and tests whether the input string can be split into $w_0 v_1 w_1$ and $v_2 w_2$ such that $T_{v_1} \in \mathcal{T}_1$ and $T_{v_2} \in \mathcal{T}_2$. To this end, $\mathcal{B}$ maintains T_v after reading $w_0 v$ and it guesses the $\mathcal{A}$-transformation induced by v' of the remaining string $v' w_2$.

States of $\mathcal{B}$ are of the form $(D, T_1, T_2, m, Z_0, \ldots, Z_k, Y_1, \ldots, Y_l, f)$, where

- $D \subseteq Q$,
- T_1 and T_2 are $\mathcal{A}$-transformations,
- $Z_0, \ldots, Z_k, Y_0, \ldots, Y_l, f$ are from $\{0, 1\}$, where $k = |w_1|$ and $l = |w_2|$.
- $m \in \{0, \ldots, |w_0|\}$.

The intended meaning of a state $(D, T_1, T_2, m, Z_0, \ldots, Z_k, Y_1, \ldots, Y_l, f)$ is as follows:

- If $\mathcal{B}$ has only read a prefix v of w_0 so far then $D = \delta^*(s, v)$ and $m = |v|$ and we do not care about the other components.
- If $\mathcal{B}$ has read a string of the form $w_0 v$ then
  - $D = \delta^*(s, w_0 v)$;
  - $T_1 = T_v$;
  - T_2 is some A-transformation (which $\mathcal{B}$ guesses is $T_{v'}$ for some string v' such that the remainder string is $v' w_2$);
  - for every i, $Z_i = 1$ if and only if v is of the form xy, where y is the prefix of w_1 of length i and $T_1 \in \mathcal{T}_1$ was true after reading x;

- for every j, $Y_j = 1$ if and only if the previous j symbols formed the prefix of w_2 of length j and before that T_2 was the transformation mapping each state q to the set $\{q\}$.
- $f = 1$ if and only if at some point in the computation a state with $Z_k = 1$ and $T_2 \in \mathcal{T}_2$ occurred.

Note that $\mathcal{B}$ can determine all components of a state deterministically with the exception of T_2. An initial value for T_2 is guessed after reading the w_0 prefix of the input. Afterwards, the new value for T_2 has to be guessed consistently with respect to the previous value of T_2 and the current symbol a. I.e., $T_2'(p) = \cup_{q \in \delta(p,a)} T_2(q)$, for every state p, where T_2 denotes the new and T_2' the old value and a is the input symbol.

All states of $\mathcal{B}$ are accepting where either $D \cap F = \emptyset$, $Y_l = 1$ and $f = 1$ or $D \cap F \neq \emptyset$, $Y_l = 1$ and $f = 0$. In the former case a string in $L(\mathcal{A}) \setminus w_0 L_1 w_1 L_2 w_2$ was found, in the latter case a string in $w_0 L_1 w_1 L_2 w_2 \setminus L(\mathcal{A})$ was read.

This completes the proof of (a). Statement (b) again follows by an easy reduction which computes an NFA for the given regular expression. Therefore it only remains to show (c).

The proof of (c) is similar to the proof of (a), however it uses the strong normal form instead. Let $D = (\mathcal{A}, w_0\,\mathrm{f}_1\,w_1\,\mathrm{f}_2\,w_2)$ be a design, where A is a DFA. Thanks to Theorem 8.22 it is sufficient to consider typings $\tau = (L_1, L_2)$ in strong normal form, i.e., where $L_i = L_\cap(\mathcal{A}, X_i, Y_i)$, for $i \in \{1, 2\}$. Thus, the algorithm guesses such a typing and verifies that it is a local typing for D. As NPSPACE = PSPACE, it only remains to show that the latter can be done in polynomial space.

To this end, let $\mathcal{B}$ be the following alternating automaton

(1) It checks that its input is of the form $w_0 u$,

(2) it simulates $\mathcal{A}$ on u' and, whenever it enters a state from Y_1 it non-deterministically decides to continue the simulation or to proceed with (3),

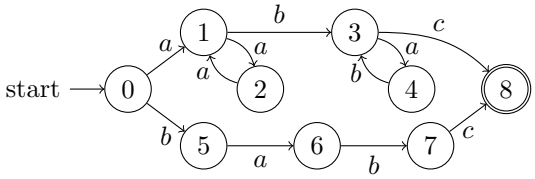
(3) it tests that the rest of the string is of the form $w_1 u'$, and

(4) it verifies that $u' \in L_\cap(\mathcal{A}, X_2, Y_2) \cdot w_2$ by universally branching to all states $p \in X_i$ and testing that $u' = u'' w_2$ with $\delta^*(p, u'') \subseteq Y_2$.

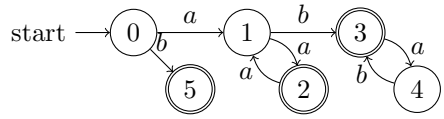
The equivalence of the AFA $\mathcal{B}$ with $\mathcal{A}$ can be tested in polynomial space (cf. [Var95]). □

Lower Bounds for ∃−LOC(DFA)

We clarify the relation between Primality and ∃−2LOC(DFA). Intuitively, one might assume that Primality can be logspace reduced to ∃−2LOC(DFA) by simply mapping the DFA $\mathcal{A}$ (the input to Primality) to the design $(\mathcal{A}, \mathrm{f}_1\,\mathrm{f}_2)$. However, a local typing for this design could yield the trivial decompositions (L_1, L_2) where $L_1 = \{\varepsilon\}$ and $L_2 = L(\mathcal{A})$ or vice versa. Therefore, we reduce from StrongPrimality, the variant of Primality which asks, given a DFA A whether there exists a non-trivial *strong* decomposition (L_1, L_2) of $L(A)$. i.e., where $\varepsilon \notin L_1$ and $\varepsilon \notin L_2$.



(a) Minimal DFA $\mathcal{A}$ for $L(R)$.



(b) Minimal DFA $\mathcal{A}_1$ for $L(R_1)$.

Figure 8.4: Minimal DFAs for the regular expressions R and R_1 in the proof of Theorem 8.38.

Lemma 8.37 $\exists-\mathsf{LOC(DFA)}$ *is* PSPACE*-hard, already for designs of the form* $(\mathcal{A}, \mathrm{f}_1\, a\, \mathrm{f}_2\, a)$.

Proof. We reduce from the complement of StrongPrimality(DFA). Let $\mathcal{A}$ be a DFA for a language L. By Lemma 8.29 we know that L can be decomposed to $L_1 L_2$ such that $\varepsilon \notin L_1 \cup L_2$ if and only if there exists a local typing $(L_1^{\#}, L_2^{\#})$ for the design $D_L^{\#} = (L_{\#}, \mathrm{f}_1 \,\#\, \mathrm{f}_2 \,\#)$. As we can compute a DFA for the language $L_{\#}$ in LOGSPACE, we have a logspace reduction from the complement of StrongPrimality(DFA) to $\exists-2\mathsf{LOC(DFA)}$. □

At this point, there is no matching lower bound. With our techniques, it can be shown that $\exists-\mathsf{LOC(DFA)}$ can be reduced to the problem whether there exists a nontrivial factorization of $L(\mathcal{A})$ into at least three languages, which is still open.

Deterministic Regular Expressions

Theorem 8.20 showed that there are designs with a perfect yet not DRE-expressible typing. We show next that there are even designs that have local typings, but none of the local typings is definable by deterministic regular expressions.

Theorem 8.38 *There is a design $D = (R, w)$, where R is a DRE, such that there exists a local typing for D, but there is no DRE-definable local typing for D.*

Proof. Let $w = \mathrm{f}_1\, ab\, \mathrm{f}_2$ and $R = a(aa)^*b(ab)^*c + babc$. Notice that R is a deterministic regular expression. The minimal DFA $\mathcal{A}$ for $L(R)$ is shown in Figure 8.4a.

We claim that D only has the local typing $(L(R_1), L(R_2))$ with $R_1 = (aa)^*(ab)^* + b$ and $R_2 = c$. Indeed, since $L(R)$ contains the string $babc$ and $w = \mathrm{f}_1\, ab\, \mathrm{f}_2$, we have, for every local typing $\tau = (L_1, L_2)$, that $c \in L_2$. Furthermore, L_2 cannot contain any other string than c: suppose, towards a contradiction, that L_2 contains $u \neq c$. But then $babu \in w(\tau') \setminus L(R)$ which contradicts that τ' is a local typing. Hence, L_2 must be $\{c\}$.

As $L_2 = \{c\}$ and τ is local, we immediately get $L_1 = \{v \mid v \cdot abc \in L(R)\}$ and thus $L_1 = L((aa)^*(ab)^* + b)$. The minimal DFA $\mathcal{A}_1$ for L_1 is shown in Figure 8.4b.

It only remains to show that $L(R_1) = L(\mathcal{A}_1)$ cannot be expressed by a DRE. By Theorem 7.2 it suffices to show that $\mathcal{A}_1$ does not fulfill the orbit property. The state set of $\mathcal{A}_1$ is $\{0, 1, 2, 3, 4, 5\}$. The states 1 and 2 are both gates of the orbit $\mathcal{O}(1) = \mathcal{O}(2)$, as 2

is final and 1 has the transition $\delta(1, b) = 3$, leaving the orbit. As 2 is final and 1 is not, the orbit property is not fulfilled.

Therefore, the only local typing for D is not DRE-definable. □

Since the typing τ in the proof of Theorem 8.38 is in strong $\mathcal{A}$-normal form we immediately get the following corollary.

Corollary 8.39 *Not every $\mathcal{A}$-NF typing, where $\mathcal{A}$ is the minimal DFA for the language $L(R)$ of a DRE, is DRE-definable.*

Due to Corollary 8.39, the upper bound for ∃−LOC(DFA) cannot be transferred to ∃−LOC(DRE), as it depends on $\mathcal{A}$-NF typings. It is possible, that there is a DRE-definable local typing τ for a design $D = (R, w)$, where R is a DRE, but the induced $\mathcal{A}$-NF typing is not DRE-definable, as can be seen from the following theorem. Note that maximal local typings are always in $\mathcal{A}$-NF.

We show that, unlike in the DFA case, ∃−LOC(DRE) is different from ∃−ML(DRE). Especially Theorem 8.3 does not hold, if the term regular is replaced by DRE-definable regular.

Theorem 8.40 *There is a design $D = (R, w)$ where R is a DRE such that D has a local DRE-definable typing, but no maximal local DRE-definable typing.*

Proof. In the proof of Theorem 8.20, we already showed, that the perfect typing $\tau = (L(\Omega_1), L(\Omega_2))$ for the design D_2 from Example 8.16 is not expressible by DREs. As τ is a perfect typing for D_2, there can be no other (possibly DRE-definable) maximal local typing for D_2. However, the DRE-definable typing $\tau_2 = (a(ba)^*, (bc)^*d)$ is a local typing for D_2.

On the other hand, a maximal DRE-definable typing $\tau = (L_1, L_2)$ cannot exist as otherwise $L(\Omega_1) \setminus L_1$ or $L(\Omega_2) \setminus L_2$ would contain a string w and DRE-definable languages are closed under adding a single string (Lemma 10 in [BGMN09]). □

Lemma 8.41 ∃−LOC(DRE) *and* ∃−ML(DRE) *are* PSPACE*-hard, already for designs of the form $(L, \mathrm{f}_1\, a\, \mathrm{f}_2\, a)$.*

Proof sketch. The statement can be easily concluded from Corollary 8.13 (Primality is hard for almost starless DREs), Lemma 8.29 (there is a one-to-one correspondence between decompositions and typings for the design $D_L^{\#} = (L_{\#}, \mathrm{f}_1 \,\#\, \mathrm{f}_2 \,\#)$) and Lemma 8.32 (the language $L(R)^{\#}$ is DRE-definable for every starless DRE R). □

Similarly as for DFAs, we can prove a better upper bound for ∃−ML(DRE) if the distributed string has only two function calls.

Theorem 8.42 ∃−ML(DRE) *is in* EXPSPACE *and* ∃−2ML(DRE) *is* PSPACE*-complete.*

Proof. The lower bound is immediate from Lemma 8.41.

For the upper bounds we use the following algorithm, which can be implemented to use only exponential space in the general case and polynomial space in the case with only 2 function calls.

(1) Compute a DFA $\mathcal{A}$ for the given DRE R.

(2) Compute the set T of all local typings in strong $\mathcal{A}$-NF for $(\mathcal{A}, w)$.

(3) For each typing $\tau \in T$:

  (4) Test, whether τ is maximal.

  (5) If τ is maximal, test whether τ is DRE-definable.

We argue why the algorithm is correct. Because of Theorem 8.22 we know that all maximal typings for $(\mathcal{A}, w)$ can be represented in strong $\mathcal{A}$-NF. Thus it is sufficient to test strong $\mathcal{A}$-NF typings. Furthermore, if there is a DRE-definable local typing τ, which is not maximal, there is also a local DRE definable typing τ' with $\tau \subsetneq \tau'$, as for every DRE-definable language L, the language $L \cup \{v\}$ is also DRE definable for any v (Lemma 10 in [BGMN09]).

The space complexities can be proved as follows. Step (1) is in quadratic time. Step (2) is in EXPSPACE in general: each typing in strong $\mathcal{A}$-NF can be represented by an n-tuple of exponentially large DFAs. Testing if such an n-tuple is a local typing costs EXPSPACE. Step (2) is in PSPACE if the distributed string has at most 2 function calls: In the proof of Theorem 8.36 we have shown, that testing whether a typing in strong $\mathcal{A}$-NF is local can be done in polynomial space in the size of $\mathcal{A}$, if the distributed string has at most 2 function calls. Thus we can test all (exponentially many) possible typings in strong $\mathcal{A}$-NF in PSPACE in step (2).

According to Theorem 8.28, computing whether a typing τ is maximal local in step (4), can be done in PSPACE (in the size of the typing).

According to Theorem 21 in [LBC14] it is possible to check whether a language of a DFA with log-size alphabet is DRE-definable in nondeterministic logarithmic space. For each DFA $\mathcal{A}$ occurring in some typing in $T_{\max}$, we can compute a DFA $\mathcal{A}$ of exponential size with a linear size alphabet for using only polynomial space. Using the result of [LBC14], it follows that we can check whether a typing in strong $\mathcal{A}$-NF is DRE-definable in nondeterministic polynomial space and therefore in PSPACE. Thus polynomial space suffices for step (4). □

8.9 Further Research on Distributed XML Design

In this chapter, we made an excursion into distributed XML repository management systems. However, we investigated mainly a single question: Given a distributed document and a schema, ho do good schemas for the referenced documents look like. There are obvious other questions in this area, like

We analyzed how to design good schemas for distributed documents. However, we have only looked at structural compatibility, i.e., we did not look at the data itself. While this might be enough for simple applications, more advanced applications will also demand semantic compatibility, i.e. consistency of integrity constraints over a distributed data set. Further research might reconsider the problem with integrity constraints in mind.

Part II

Integrity Constraints

9 Integrity Constraints for Relations and Trees

In Part II of this thesis we change our focus from syntactical descriptions to semantic constraints. We already pointed out the difference when we introduced our running example in the introduction. In contrast to syntactic definitions, which describe the structure of the data, semantic constraints look at the data itself.

In the introduction, we gave two examples of semantic constraints based on the running example of the content management system: The first one was that user IDs should be unique and the second one was that every user ID which owns some document should actually exist.

In this part of the thesis, we look at languages for specifying such constraints on XML documents and study algorithmic problems related to such constraints. Therefore, we first repeat the basics of integrity constraints on relational databases. This has two reasons. First, we want to see the similarities between both worlds. Second and more important, the framework for XML integrity constraints, which we will introduce in the next chapter, builds heavily on definitions of relational integrity constraints.

This part of the thesis is organized as follows. This chapter briefly sketches relational integrity constraints in Section 9.1 and identifies the challenges for XML integrity constraints in Section 9.2. Afterwards, we will introduce a general framework for XML integrity constraints in Chapter 10 and explore its possibilities when it comes to representing constraints formulated using existing standards or different frameworks, as proposed in the literature. In Chapter 11, we will analyze the complexity of the implication problem for integrity constraints for some instantiations of the framework. We will close this part in Chapter 12, where we will have a look at first order logic with two variables. In particular, we show there that the problem, given an $\mathrm{FO}^2(\sim, +1)$-formula and a key constraint, does there exists a tree satisfying the formula and the key constraint, is decidable on data words.

9.1 Relational Integrity Constraints

In this section, we give a very brief overview over relational integrity constraints. For a deeper introduction to the topic, we refer to [AHV94]. We want to start this section coming back to our running example. Figure 9.1 depicts a fragment of a relational version of our CMS database. We only look at two relations: The USER relation associates user IDs with persons (identified by their name) and the OWNER relation associates

USER

user-id	name
user2	Gene Hall
user3	Eve Johnson
user23	Ann Brown
user42	Joe Smith

OWNER

document-id	user-id
document1	user23
document2	user42
document3	user23
document4	user2

Figure 9.1: Relational database with two relations.

documents (represented by document IDs) with user IDs. A third relation that connects document IDs to the actual content of the document is of no importance, here.

As already mentioned, we want to express two semantic constraints. The first constraint is that user IDs are unique, the second is that every user ID referenced from a document exists.

One can verify that these two constraints can be expressed by the following first order sentences.

$$\Psi_{\text{uid-unique}} = \forall x_u, x_n, y_n.\ \ \text{USER}(x_u, x_n) \wedge \text{USER}(x_u, y_n) \ \rightarrow\ x_n = y_n$$
$$\Psi_{\text{uid-exists}} = \forall x_d, x_u.\ \ \text{OWNER}(x_d, x_u) \ \rightarrow\ \exists y_n.\ \text{USER}(x_u, y_n)$$

Given a set of constraints Σ, we will in particular look at two algorithmic problems: satisfiability and implication.

Satisfiability	
Given:	a set of constraints Σ
Question:	Does there exist a database instance I such that I satisfies every constraint $\sigma \in \Sigma$?

Implication	
Given:	a set of constraints Σ, a constraint τ
Question:	Does every database instace I that satisfies Σ also satisfy τ?

There is a long tradition in studying relational integrity constraints based on first-order sentences. They were first studied in [GM78]. As the general problem, given a first order sentence φ, does there exists a database satisfying the formulas, is clearly undecidable, there has been much research identifying fragments of first-order logic, such that the algorithmic problems (satisfiability, implication) are decidable and ideally tractable for these fragments.

The constraints $\Psi_{\text{uid-unique}}$ and $\Psi_{\text{uid-exists}}$ represent two very important classes of constraints, namely functional dependencies and inclusion constraints.

A *functional dependency* $\rho = Y \rightarrow B$ over a relation R consists of a set Y of attributes and a single attribute[1] B. It is satisfied by a relation R, if all tuples in R, which agree on the attributes in Y also agree on the attribute B. It is well known that functional dependencies can be rewritten as universal first order sentences. By FD we refer to the set of functional dependencies, as a relational constraint language. Usually, we denote sets of attributes by uppercase letters Y, Z and single attributes by uppercase letters B, C. Functional dependencies where first introduced by Codd [Cod72].

A functional dependency, $\rho = Y \rightarrow Z$, where $Y \cup Z$ covers the full set of attributes of a relation R is called a *key constraint* for R.

The example constraint of unique user IDs can be expressed by the key constraint user-id→name. We note that this dependency is equivalent to the first order sentence $\Psi_{\text{uid-unique}}$ depicted above.

We continue with the definition of inclusion constraints. An *inclusion constraint*

$$R[B_1, \ldots, B_m] \subseteq S[C_1, \ldots, C_m]$$

between two relations R and S is given by two ordered lists of attributes. It holds on a relational database, if the projection of R to the attributes $B_1, \ldots, B_m$ is included in the projection of S to the attributes $C_1, \ldots, C_m$, where the order of attributes is significant.

It is well known that deciding implication of functional dependencies and inclusion constraints is already undecidable [CV85]. In practical applications, one usually restricts to key constraints and foreign key constraints, as the implication of these constraints is decidable. A *foreign key constraint* is an inclusion constraint $R[B_1, \ldots, B_M] \subseteq S[C_1, \ldots, C_m]$, where the key constraint $C_1, \ldots, C_m \rightarrow S$ holds on S. Whether a given inclusion constraint is a foreign key constraint can therefore only be answered in the context of other integrity constraints that hold on a relational database and not on the structure of the database alone.

The Chase

A valuable tool in deciding implication of integrity constraints is the chase algorithm. The chase algorithm is a simple fixed-point algorithm that enforces constraints on a database instance I, by repeatedly applying some non-satisfied constraint σ on I. We first describe how the chase algorithm applies constraints. Afterwards, we will see, how the chase can be used to test implication of integrity constraints.

While the chase algorithm can be used for a wide range of constraints, we will only apply the chase algorithm in the context of functional dependencies and restrict this description of the chase to functional dependencies and inclusion constraints.

A non satisfied function dependency $\sigma = Y \rightarrow B$ is applied to a database relation R as follows: Let μ_1 and μ_2 be two tuples violating σ. Replace $\mu_1(B)$ wherever it occurs in I with $\mu_2(B)$. It is obvious that after application of the chase rule, μ_1 and μ_2 do no longer violate σ.

[1] As usual, we could allow a set of attributes instead of the single attribute B but such FDs can always be rewritten as a set of FDs with singleton attributes.

A non satisfied inclusion constraint $\sigma = R[B_1, \ldots, B_m] \subseteq S[C_1, \ldots, C_m]$ is applied to a database instance Ias follows: Let μ_1 be a tuple in R such that there is no tuple μ_2 in S with $\mu_2(C_i) = \mu_1(B_i)$ for $i \in \{1, \ldots, m\}$. Insert a tuple μ_3 in S, such that $\mu_3(C_i) = \mu_1(B_i)$ for $i \in \{1, \ldots, m\}$ and all other attributes of μ_3 have fresh data values. Again, it is obvious that after application of the chase rule, μ_1 does no longer violate σ.

If the chase algorithm terminates, by the termination condition, there are no violated constraints left. However, there is no guarantee, that the chase terminates. While the chase algorithm always terminates, if there are only functional dependencies, this is not the case when there are also inclusion constraints, as the newly inserted tuples can enforce further tuples to be added. In general it is not even possible to decide whether the chase algorithm terminates or not.

The chase has first been described by [MMS79] and has been generalized by [BV84]. There is still ongoing research on termination conditions for the chase algorithm [Mei10].

The chase algorithm can be used to test the implication problem of integrity constraints by starting with a most general counter-example for the target dependency and applying chase rules until all constraints from Σ are satisfied.

Whether a set Σ of functional dependencies implies a functional dependency $\tau = Y \rightarrow B$ can be decided with the chase algorithm by starting with a relation R that contains to tuples μ_1 and μ_2 such that $\mu_1(Y) = \mu_2(Y)$ and both tuples have different and unique data values for all other attributes. As long as there is some violated $\sigma \in \Sigma$ we apply the chase rule.

We will see how this algorithm can be generalized to work on trees in Chapter 11.2.

Incomplete Data

Traditionally, database theory was studied with the assumption that all data is complete. However in practice, data is often unknown or even non existent at all. Take for an example an address database, where one can store contact information about persons like postal addresses, phone numbers and e-mail addresses. Usually one should expect, that it is possible to store phone numbers for people, where the postal address is unknown. Still it is desirable to be able to specify constraints for the data.

In the relational world, incomplete data is usually represented by the means of null values. A null value in some relation says the data is unknown or not existent. In contrast, in the XML world, there is usually no need for null values, as incomplete data can be represented by non-existent subtrees. Accordingly, we will translate non-existent subtrees to null values, when we describe our framework in the next chapter.

When dealing with relations which may contain null values, it can be helpful to use the more powerful fictitious functional dependencies instead of the usual functional dependencies. A *fictitious functional dependency* $\rho = Y \xrightarrow{Z} B$ consists of two sets of attributes Y, Z and a single attribute B. It is satisfied by a relation R, if all tuples, which are non-null on all the attributes of Y and Z and agree on the attributes in Y also agree on B. We note, that it is allowed that the attribute B is null in both tuples if B is not contained in Z.

Another useful kind of constraints in the presence of null values are non-null constraints. A *non-null constraint* $\rho = \mathrm{NN}(Y, Z)$ consists of two sets of attributes Y and Z. It holds in a relation R, if all tuples, which are non-null in all attributes of Y are non-null in all attributes of Z.

We refer to the set of fictitious functional dependencies by FFD and to the set of non-null constraints by NN. Implication of fictitious functional dependencies and non-null constraints on relational databases has been investigated in [AM86].

9.2 Integrity Constraints on Trees

Now we focus on the challenges for specifying XML integrity constraints. On the theoretical side, an XML integrity constraint language should be expressive and have low algorithmic complexities for problems like the consistency and the implication problem. Naturally, both goals are contradictory, therefore we need good compromises.

On the practical side, an XML integrity constraint language should have a user-friendly syntax and intuitive semantics. We will restrict to studying the theoretical aspects of XML integrity constraint languages, by providing a framework for reasoning about XML integrity constraints in the next chapter and analyzing the complexity of the model checking and implication problem for some instantiations of the framework in Chapter 11.

As in the relational world, we can specify constraints by means of first order sentences. For example, we can express our two example dependencies for our CMS database using the following formulas.

$$\begin{aligned}
\Psi_{\text{unique}} &= \forall v_p, v_u, w_p, w_u.\quad \text{PERSON}(v_p) \wedge \text{USER-ID}(v_u) \wedge \text{PERSON}(w_p) \wedge \\
&\qquad \text{USER-ID}(w_u) \wedge E(v_p, v_u) \wedge E(w_p, w_u) \wedge v_u \sim w_u \;\rightarrow\; v_p = w_p \\
\Psi_{\text{uid-exists}} &= \forall v_d, v_u.\quad \text{DOCUMENT}(v_d) \wedge \text{USER-ID}(v_u) \wedge E(v_d, v_u) \;\rightarrow \\
&\qquad \exists w_p, w_u.\quad \text{PERSON}(w_p) \wedge \text{USER-ID}(w_u) \wedge E(w_p, w_u) \wedge v_u \sim w_u
\end{aligned}$$

Here E denotes the edge relation of the tree (the child axis of the XML document) and PERSON, USER-ID and DOCUMENT denote unary relations containing all nodes with the labels `person`, `user-id` and `document` respectively. To express more complicated constraints, especially on trees whose depth is not bounded by a constant, it might be necessary to additionally use the descendant relation E^+.

There are obvious similarities to the relational constraints presented in the previous section. The obvious difference is that the formulas are longer than the relational ones, as we need to explicitly test for the labels of the nodes, where the attribute names in relational databases are uniquely specified by their position in the relation. Note that the presented constraints have been kept simple on purpose. For example one might want to change the constraints in such a way that only person nodes in the user database part of the tree are considered. This might be necessary if person nodes could occur in different parts of the tree.

Specifying integrity constraints as first order sentences has the same downsides in the XML world, as in the relational world. The consistency problem is undecidable and they

are even more user-unfriendly due to the increased verbosity resulting from explicitly specifying labels of nodes.

Different to the relational world, we do not need null values to reason over XML databases with incomplete data, as incomplete data in XML trees can simply be represented by missing subtrees.

10 A Framework for XML Integrity Constraints

In this chapter, we will have a look on a framework for specifying XML integrity constraints. We start with introducing and defining our framework in a very general way in Section 10.1. Afterwards, we introduce tree patterns in Section 10.2 and specify some instantiations of the framework based on tree patterns in Section 10.3. Finally, in Section 10.4, we will show how previous approaches for XML integrity constraints and official standards integrate into our framework. The analysis of algorithmic problems, especially the implication problem concerning these constraints, is postponed to the following chapter.

10.1 XML-to-Relational Constraints

In general, an XML-to-relational constraint (X2R-constraint for short) (m, ρ) consists of two parts: a mapping m that maps trees to relations and a relational constraint ρ that refers to the relations yielded by m. To keep our framework flexible, we allow the mapping m to return null values $\perp$. To simplify notation, we denote the set $S \cup \{\perp\}$ by $S_\perp$ for every set S.

Informally, we require that the mapping is independent of actual data values in the sense that any (not necessarily injective) renaming of data values commutes with the mapping.

More formally, a *XML2Relational-Mapping (X2R-mapping)* is a function

$$m : \mathcal{T} \to 2^{\mathcal{V}_\perp^\ell \times \mathcal{D}_\perp^n},$$

for some ℓ and n, such that

- $m(t) \subseteq V_\perp^\ell \times \mathrm{dv}(t)_\perp^n$; and
- for every t and every mapping[1] $\delta : \mathcal{D} \to \mathcal{D}$ it holds $m(\delta(t)) = \delta(m(t))$, where $\delta(t)$ results from t by renaming all data values according to δ.

A tree t is *valid* with respect to an X2R-constraint $\sigma = (m, \rho)$ if $m(t) \models \rho$. In that case, we write $t \models \sigma$ and also say that t satisfies σ.

[1] This includes non-injective mappings.

A constraint instance Σ is a set of constraints. We write $\Sigma \models \tau$, if for every tree t for which $t \models \Sigma$ holds, also $t \models \tau$ holds. We write $\Sigma \models_D \tau$, where D is some schema[2], if it holds that $t \models \tau$ whenever $t \models \Sigma$ and $t \models D$.

10.2 Tree Patterns and Tree Pattern Mappings

While the framework in general can be instantiated with arbitrary mapping languages, we will mainly concentrate on constraints, where the mappings are induced by tree patterns and the relational constraints are functional dependencies. In the following, we give the necessary definitions and fix notation.

A *tree pattern* $p = (X, A, \text{lab})$ consists of

- a set X of *variables*,
- an edge relation $A = A_/ \cup A_{//}$ on variables, and
- a labeling function $\text{lab} : X \to \mathcal{L} \cup \{*\}$,

such that (X, A) is a directed tree with a unique root, denoted $\text{root}(p)$, such that all edges are directed away from $\text{root}(p)$. In the remainder, we will often use the synonyms *tree* for XML tree and *pattern* for tree pattern, respectively.

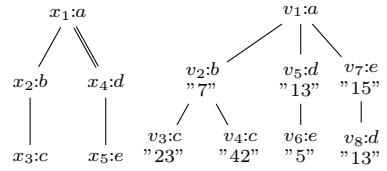
We call edges in $A_/$ *short edges* and edges in $A_{//}$ *long edges* (depicted as double lines in figures). Intuitively, they correspond to the child axis and the descendant axis in the sense of XPath. The *wildcard* symbol $*$ is intended to match every label.

Tree pattern mappings are defined via embeddings. The intuitive idea is that every possible embedding of a tree pattern in a tree gives a tuple in the relation. We will define two different variants of embeddings/mappings. While full embeddings always embed the whole pattern, partial embeddings can also embed only parts of the pattern. This is useful for specifying constraints over incomplete data.

For a pattern $p = (X, A, \text{lab}_p)$ and a tree $t = (V, E, \text{lab}, \text{dv})$ a (partial) function $\pi : X \rightharpoonup V$ is a *partial embedding* of p in t if it fulfills the following conditions, for every $x, y \in X$:

1. if $(x, y) \in A$ and $\pi(y) \neq \bot$ then $\pi(x) \neq \bot$;
2. if $\text{lab}_p(x) \neq *$ and $\pi(x) \neq \bot$ then $\text{lab}_p(x) = \text{lab}(\pi(x))$
3. if $(x, y) \in A_/$ and $\pi(y) \neq \bot$ then $\pi(x)$ is the parent of $\pi(y)$ in t;
4. if $(x, y) \in A_{//}$ and $\pi(y) \neq \bot$ then $\pi(y)$ is a descendant of $\pi(x)$ in t;
5. $\pi(\text{root}(p)) = \text{root}(t)$.

[2]The precise kinds of schemas that we consider will be defined later on.



(a) Pattern p (b) Tree t

x_1	$x_1.@$	x_2	$x_2.@$	x_3	$x_3.@$	x_4	$x_4.@$	x_5	$x_5.@$
v_1	1	v_2	7	v_3	23	v_5	13	v_6	5
v_1	1	v_2	7	v_4	42	v_5	13	v_6	5
v_1	1	v_2	7	v_3	23	v_8	13	$\perp$	$\perp$
v_1	1	v_2	7	v_4	42	v_8	13	$\perp$	$\perp$

(c) mapping $m_p(t)$

Figure 10.1: Abstract example for a tree pattern mapping

Here, we write $\pi(x) = \perp$ to denote that $\pi(x)$ is undefined. A *maximal partial embedding* is a partial embedding π, such that there does not exist a partial embedding π' with $\pi'(x) = \pi(x)$ for all x with $\pi(x) \neq \perp$ and $\pi'(x) \neq \perp$ for some x with $\pi(x) = \perp$. A *full embedding* is an embedding π, such that $\pi^{-1}(\perp) = \emptyset$.

In the presence of an sDTD D, we can represent an embedding π of a pattern p into the expansion $[t]_D$ of a tree t, by specifying nodes in $[t]_D$ as pairs (u, w), where u is a node of t and a label sequence w as defined in Section 12.1. We say that such an embedding uses *relative node addresses*.

We will use compact XPath notation to denote tree patterns. For example the pattern in Figure 10.1(a) can be abbreviated as $/a[/b/c]//d$.

Variables x in a tree pattern refer to nodes in trees, therefore we also call them *node terms*. To refer to the data value of a node, we use *data terms* of the form $x.@$. A *variable term* B is a node term or a data term, its *underlying variable* is denoted by $\mathrm{var}(B)$. That is, $\mathrm{var}(x) = x$ and $\mathrm{var}(x.@) = x$. We denote the set of all data terms for a variable set X by $X^@ =_{\mathrm{def}} \{x.@ \mid x \in X\}$.

If π is an embedding of a tree pattern p in a tree t, we use the abbreviation $\pi(x.@) =_{\mathrm{def}} \mathrm{dv}(\pi(x))$.

With a tree pattern p one can associate an X2R-mapping in a straightforward fashion: every variable x of p can give rise to two attributes in the resulting relation, one for the node v matching x and one for its data value $\mathrm{dv}(v)$. However, in the interest of more flexibility and, often, smaller relations, we allow that the target relation consists of a subset of all attributes.

A *tree pattern mapping* $\mu = (p, W)$ consists of a tree pattern $p = (X, A, \mathrm{lab}_p)$ and a set $W \subseteq X \cup X^@$. With an embedding π of p in a tree $t = (V, E, \mathrm{lab}, \mathrm{dv}, <_c)$ we associate the tuple $\theta_{\pi,\mu}$ defined as $\theta_{\pi,\mu}(x) =_{\mathrm{def}} \pi(x)$, for every $x \in W$.

For a tree pattern mapping μ and a tree t we let

- $\mu(t) =_{\mathrm{def}} \{\theta_{\pi,\tau} \mid \pi \text{ is a full embedding of } p \text{ in } t\}$, and
- $\mu_\perp(t) =_{\mathrm{def}} \{\theta_{\pi,\tau} \mid \pi \text{ is a maximal partial embedding of } p \text{ in } t\}$

In other words, for every possible full embedding π of p in T, the relation $\mu(t)$ has one tuple corresponding to π. The relation $\mu_\perp(t)$ has additional tuples for maximal partial embeddings that are no full embeddings. Figure 10.1 gives an example mapping for a

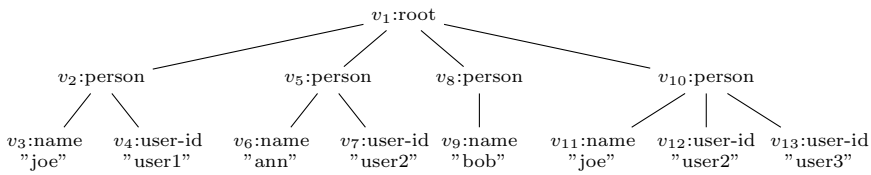


Figure 10.2: Fragment of the CMS Example

pattern p and a tree t. The last two tuples are only included in $\mu_\perp(t)$. Intuitively $\mu_\perp$ maps non-existing sub-trees to null values, where μ just ignores such tuples altogether.

We denote the set of all mappings that can be specified in this way by TP (or by TP$[/, //, *]$, if we want to stress the availability of the axes and the wildcard symbol). We denote fragments of TP by TP$[/, *]$, TP$[/, //]$ and TP$[/]$, with the obvious meaning. Even if mappings with full embeddings are incomparable to mappings with partial embeddings, it is justified to refer to mappings with partial embeddings as the more general notion in the sense, that the classes of dependencies specified using partial embeddings are more general.

10.3 Tree Pattern Based X2R-Constraints

We will mainly study tree-pattern based X2R-constraints in this and the following chapter. A tree-pattern based X2R-constraint $\sigma = (m, \rho)$ consists of a mapping $m = (p, Y)$ and a (possibly fictitious) functional dependency ρ.

We make use of the following (hopefully) intuitive notation. We specify p by an XPath expression in simplified syntax. The pattern positions that correspond to (node or data) variables in Y are succeeded by a variable name in brackets. The set Y contains both the node and the data variable for every variable name occurring in the expression. For readability, we drop the set notation from functional dependencies in the (very common) case of singleton sets.

Example 10.1 *We use the tree of Figure 10.2 as a small example document, which contains user names of persons. A possible constraint that one might want to require is, that each user-id uniquely identifies a person, i.e. there are no two persons with the same user-id. In our framework, we can express this constraint as*

$$\sigma_{user} = (//person\langle x_p\rangle/user\text{-}id\langle x_u\rangle, x_u.@\rightarrow x_p).$$

The pattern selects all pairs (v_1, v_2), where v_2 is a child with label user-id of a node v_1 with label person. Figure 10.3 shows the relevant part of result of the mapping. The constraint is not satisfied due to tuples 2 and 3.

	x_p	x_u	$x_u.@$
1:	v_2	v_4	user1
2:	v_5	v_7	user2
3:	v_{10}	v_{12}	user2
4:	v_{10}	v_{13}	user3

Figure 10.3: Mapping result of σ_{user}

As we have seen, there are two different natural definitions for tree pattern based mappings: mappings based on full embeddings and mappings based on maximal partial embeddings.

At this point we want to make a design choice: Should we evaluate relational constraints in tree pattern based mappings with respect to full embeddings or with respect to maximal partial embeddings. For fictitious functional dependencies and non-null constraints the obvious answer is, that we should evaluate with respect to maximal partial embeddings. As these constraints explicitly deal with null values, there is no reason to use them on relations, which cannot have null values. Non-null constraints would be trivially satisfied and FFDs would be equal to FDs.

For the case of (non-fictitious) functional dependencies both possibilities would make sense. We decide to define satisfaction with respect to full embeddings. Note that this is a choice, which we take to provide a technically simpler framework in the case where one has not to deal with incomplete data. We suggest to stick to the more powerful fictitious dependencies when dealing with incomplete data. When researching the complexity of the implication problem in Chapter 11, we show all lower bounds for non-fictitious dependencies and all upper bounds for fictitious dependencies. Note that we did not find any complexity theoretic difference between fictitious and non-fictitious dependencies. However the algorithms for the model checking and implication problem and the upper bound proofs become more technical, when null values need to be considered.

Formally: An FD ρ is satisfied by a tree t if $\mu(t) \models \rho$ and an FFD or NN ρ is satisfied by a tree t if $\mu_\perp(t) \models \rho$.

We note that an FD $\sigma = (p, Y \rightarrow B)$ is equivalent to the FFD $\sigma = (m, Y \xrightarrow{Z} B)$, where Z constraints all variable terms of p. Thus we can evaluate FDs, FFDs and NNs together by converting all FDs to FFDs.

We call an FD $\sigma = (m, Y \rightarrow B)$, in which B is a node variable a *XML-key functional dependency*[3] *(XKFD)*.

For an X2R-mapping language $\mathcal{M}$ and a relational constraint language $\mathcal{C}$ we denote the resulting set of X2R-constraints by $\text{XC}(\mathcal{M}, \mathcal{C})$. For example, $\text{XC}(\text{TP}, \text{FD})$ stands for the class of constraints, yielded by tree patterns and functional dependencies.

[3]The name stems from the fact that these FDs very closely correspond to XML key constraints.

Inclusion Constraints for XML

We want to shortly sketch, how inclusion constraints can be defined in our framework. There are two natural ways of defining inclusion constraints. The first one is by defining two mappings m_1 and m_2 and require inclusion of the resulting relations. That is a tree t satisfies an inclusion constraint $\subseteq_{m_1,m_2}$, if and only if $m_1(t) \subseteq m_2(t)$. We note that the resulting relations need to be compatible, that is they should have the same set of attributes. This definition is not strictly an X2R-constraint following our definition, as it uses two mappings.

Another possibility, which uses our general definition of an X2R-constraint, is to define only one mapping m and request inclusion between different attributes (sets of attributes) of the resulting relation using a relational inclusion constraint ρ. We note that both definitions are equally expressive. For one direction, m can be defined as $m_1 \times m_2$ for the other direction m_1 and m_2 can be defined as projections of m.

Tree pattern and relational inclusion constraints give a natural instantiation of our framework. The example constraint, that all user ids referenced from documents should exist can be enforced by the constraint

$$\sigma_{\text{uid-exists}} = (/\text{root}[//\text{document/owner}\langle x_o\rangle]//\text{person/user-id}\langle x_u\rangle, x_o \subseteq x_u).$$

10.4 Comparing the X2R-Framework with Existing Work

The framework of X2R-constraints can be instantiated with an arbitrary X2R-mapping language $\mathcal{M}$ and an arbitrary relational constraint language $\mathcal{C}$. In the remainder of this section, we are going to sketch ways in which XML constraint languages that are used in practice or were proposed in the literature can be viewed as particular instantiations of the X2R-mapping based framework. We will mostly focus on constraints that have been proposed as key constraints and functional dependencies for XML.

Hierarchical Constraints

Hierarchical key constraints have been proposed in early work of Buneman et al. [BDF$^+$02, BDF$^+$03]. In the works, a framework is proposed that builds on top of path languages, where a path language may be any language that selects paths out of an XML tree. Especially the path languages may depend on nodes outside of the selected paths, as for example XPath with node tests.

Before, we discuss, which type of path languages are relevant, we want to define hierarchical key constraints.

Definition 10.2 [[BDF$^+$02]] A key specification $(Q, \{P_1, \ldots, P_k\}$ consists of a target path expression Q and key path expressions $\{P_1, \ldots, P_k\}$.

A tree t satisfies a key specification $(Q, \{P_1, \ldots, P_k\})$, if for any v_1, v_2 in

$$\{v \mid [\text{root}(t), v)] \in L(Q)\}$$

it holds that if there exist nodes $w_1^1, \ldots, w_k^1, w_1^2, \ldots, w_k^2$, such that

$$w_i^j \in \{v \mid [v_j, v] \in P_i\} \text{ for } i \in \{1, \ldots, k\} \text{ and } j \in \{1, 2\}$$

and $w_i^1 \sim w_i^2$ for $i \in \{1, \ldots, k\}$ then $v_1 = v_2$.

The whole concept of hierarchical constraints integrates nicely with the X2R-framework. Let $(Q, \{P_1, \ldots, P_k\})$ be a key specification. We can formulate this specification equivalently in our framework as $(m, \{P_1.@, \ldots, P_k.@\} \rightarrow Q)$, where m is defined as

$$m = \{(v, v_1, \ldots, v_k) \mid [\text{root}(t), v] \in L(Q) \wedge [v, v_i] \in L(P_i)\}$$

and the attribute names in $m(t)$ from left to right are $Q, P_1, \ldots, P_n$.

In the hierarchical framework, the path expressions can be of arbitrary type. Note that one might even choose language classes of different expressiveness for the target path expression and for the key path expressions.

If all used path expressions can be written as tree patterns, then the complete key specification can be rewritten as tree pattern based X2R-constraint.

Relative Key Constraints

Relative key constraints are key constraints that do not need to hold on complete trees, but only on subtrees.

Consider for example a company with several establishments. In this case user-ids might be local to establishments, i.e. persons from different establishments are allowed to use the same user-id. This constraint can be written (using our syntax) as

$$\sigma = (/\text{establishment}\langle x_e \rangle / \text{person}\langle x_p \rangle / \text{user-id}\langle x_u \rangle, \{x_e, x_u.@\} \rightarrow x_p).$$

Using the node variable x on the left side changes the constraint to be local to establishments, as tuples referring to different establishments cannot conflict any more. In general, relative key constraints can be expressed as XC(TP[/], XKFD) constraints in our framework.

Relative key constraints have been considered by several people. Buneman et al. considered relative key constraints in their hierarchical key constraints framework [BDF$^+$02]. In their framework, a relative key specification $(R, Q, \{P_1, \ldots, P_k\})$ consists of a key specification $(Q, \{P_1, \ldots, P_k\})$ and a path expression R identifying the subtrees in which the key specification should hold. A relative key specification $(R, Q, \{P_1, \ldots, P_k\})$ is satisfied in a tree t, if for every node v with $[\text{root}(t), v] \in L(R)$ it holds that the key specification $(Q, \{P_1, \ldots, P_k\})$ is satisfied on the subtree t_v.

This approach integrates again nicely with our framework, in much the same way, as described wrt. hierarchical constraints above. A relative key specification can be rewritten as $(m, \{R, P_1.@, \ldots, P_k.@\} \rightarrow Q)$, where m is defined appropriate.

Arenas, Fan and Libkin investigated a variant of relative key constraints with respect to XFDs as they are described below in [AFL08].

```
 1: <xs:element name="root">
 2:   [...]
 3:   <xs:key name="uid">
 4:     <xs:selector xpath="./person"/>
 5:     <xs:field xpath="user-id"/>
 6:   </xs:key>
 7:   <xs:keyref name="files" refer="uid">
 8:     <xs:selector xpath=".//file"/>
 9:     <xs:field xpath="user-id"/>
10:   </xs:keyref>
11: </xs:element>
```

Figure 10.4: XML Schema Key and Foreign Key Constraint

XML Schema Integrity Constraints

To compare XML Schema integrity constraints with our framework, we need to introduce some terminology.

For every tree t valid wrt. to an XSD X, X assigns a type to every node v of t. For every possible type α of an XSD X, the set of nodes matched by α can be described by a regular language L_α over ancestor strings [MNSB06]. A node v belongs to the type α, if and only if the ancestor string of v is in L_α.

XML Schema [GSMT$^+$12] describes three kinds of integrity constraints: *unique constraints*, *key constraints* and *foreign key constraints*. Every XML Schema integrity constraint is specified relative to an element definition, that is XML Schema integrity constraints are relative constraints, like the constraints investigated in [AFL08].

Figure 10.4 gives an example for an XML key constraint roughly equivalent to σ_{user} from Example 10.1 in XML Schema notation. We have skipped the declaration of the content model of the element.

Line 1 starts an element declaration for elements named root. We leave out the structural part of the type definition. Line 3 starts the definition of the key constraint and specifies a name for it, which is relevant, e.g., for foreign key constraints. Line 4 specifies the selector path (./person), which is a restricted XPath-expression that is evaluated relative to nodes matched by the element declaration: in this example it is evaluated relative to nodes of label root. Note that the element declaration not necessarily matches all elements of label root. Line 5 specifies the field of the constraint (user-id in the example). This XPath-expression is evaluated relative to nodes matched by the expression from Line 4. In general, there may be arbitrarily many field expressions $F_1, \ldots, F_n$.

We only give a simplified description of integrity constraints in XML Schema, as they are quite complex in general. For a tree t, to satisfy a key constraint, the following conditions have to be met by every node v matched by the surrounding element declaration:

(1) for every node v', that is matched by the selector path, it holds that every field expression F_i matches exactly one node v_i, and

(2) for every two nodes v_{t_1} and v_{t_2} matched by the selector path, the vector of data values of the nodes matched by the field specifications are not identical.

Let us assume for the moment, that the element declaration in Figure 10.4 only matches the root node. From Conditions (1) and (2) we then get, that the key constraint from Figure 10.4 corresponds to two constraints in our framework. From (1) we get the constraints $(/\text{root}/\text{person}\langle y\rangle/\text{user-id}\langle z\rangle, y \to z)$ and $(/\text{root}/\text{person}\langle y\rangle/\text{user-id}\langle z\rangle, \text{NN}(y, z))$, which say that every person-node (directly below the root) should have at most one (respectively at least one) user-id node as child.

From (2) we get our intended constraint

$$(/\text{root}\langle x\rangle/\text{person}\langle y\rangle/\text{user-id}\langle z\rangle, \{x, z.@\} \to y),$$

which is equivalent to σ_{user}. We note that constraints relative to the root node are equivalent to absolute constraints.

This looks like key constraints could be described by a subset of XC(TP[/,//],XKFD). This is true, if the structural part of (the relevant part of) the XML Schema can be described by a DTD. However in general, the element declaration could be enclosed inside a complex type declaration. In this case we have to ensure, that an XML Schema integrity constraint definition is only applied to nodes matched by the element declaration.

There are two straightforward ways to accomplish this. First, we could use tree patterns which can talk about regular paths, second, we could allow tree patterns to match nodes according to their type.

XML integrity constraints, which are defined over regular paths have been investigated in [AFL08]. However, these constraints do not fully cover XML Schema integrity constraints, as the field expressions are restricted to paths of length one.

Let L_{root} be the regular language describing all possible ancestor strings for elements matched by the element declaration and R be a regular expression with $L(R) = L_{\text{root}}$. Note that in our example $L(R) = \{\text{root}\}$. Then the constraints can be described using the tree pattern $p = /R/\text{person}/\text{user-id}$.

The second approach has the advantage, that we get the types of nodes for free, when a tree is validated against a schema, as in the validation process the types have to be computed anyway. These types can then be used to match nodes of a tree pattern using existing algorithms.

XML Schema unique constraints have the same syntax as XML Schema key constraints, only the semantic differs. Unique constraints do not enforce that every field matches at least one node, i.e. it could match zero nodes. Accordingly, (2) is modified, that it only enforces the vector of data values to be different, when all fields match one node. In our framework, the difference is, that unique constraints do not enforce non-null constraints.

Foreign key constraints again use a very similar syntax. An example is given in Figure 10.4 Lines 7 to 11. The only difference in syntax is, that foreign keys reference a key constraint, in the example the uid constraint from above. The example foreign key specifies, that the user-id of files (described somewhere in the XML tree) should exist, i.e. there should be a person with this user-id.

For space reasons, we do not describe the semantics here, but just note that foreign key constraints can be expressed by inclusion constraints (over tree pattern mappings) in our framework.

XML functional dependencies (XFDs)

The literature has several different definitions of functional dependencies for XML data, e.g., [AL04, KW07, HL03, VLL04, LLL02].

We concentrate here on XFDs as introduced by Arenas and Libkin [AL04] and further examined by Kot and White [KW07]. An XFD $\sigma = Y \rightarrow Z$ consists of two sets of paths specifying the attributes of the functional dependency. As shown[4] in [KW07], XFDs can be canonically expressed using $\mathrm{XC}(\mathrm{TP}_{\perp}[/], \mathrm{FFD})$, where the tree pattern p is the (unique) smallest tree pattern (with respect to the number of nodes) that contains all paths from Y and Z. However, tree patterns for XFDs need to be duplicate free[5], that is they do not contain two edges (x, y) and (x, z) with $\mathrm{lab}(x) = \mathrm{lab}(z)$ and $y \neq z$. Thus, XFDs have the same expressiveness as functional dependencies over duplicate-free tree patterns.

It is immediately clear, that the restriction to the child axis limits the expressivity, as constraints over recursive parts of schemas cannot be expressed, the restriction to duplicate free patterns is more subtle. Note that the dependency

$$(/r/a\langle x_a\rangle[/b/c\langle x_c\rangle]/b/d\langle x_d\rangle, \{x_c, x_d\}\rightarrow x_a)$$

cannot be expressed with duplicate free patterns. Especially it is different from

$$(/r/a\langle x_a\rangle/b[/c\langle x_c\rangle]/d\langle x_d\rangle, \{x_c, x_d\}\rightarrow x_a).$$

Kot and White give a complete axiomatization of XFDs [KW07]. The axiomatization includes FFDs and NNs. They also present a chase-algorithm to decide the implication problem in polynomial time.

Another (more general) definition of XFDs was proposed by Hartmann and Link [HL03], allowing XFDs to compare complete subtrees. For example, they can specify the dependency that there are no two a-labeled nodes that have equivalent (meaning isomorphic) subtrees. Dependencies of this kind cannot be expressed in our framework as they are second order constraints, i.e. they can compare sets of nodes.

Relational and XML Data Exchange

Arenas et al. have investigated the data exchange problem for relational and XML databases [ABLM10]. To specify source-to-target dependencies for XML databases they use a framework very similar to ours. Their source-to-target dependencies can be formulated

[4] We note that Kot and White define the mapping of a tree pattern using unfolding of nested relations. The definition is equivalent to our definition using embeddings.

[5] Duplicate free tree patterns have been considered in [MS04]

using inclusion dependencies between two tree pattern mappings, one for the source and one for the target database.

Structural Constraints

Structural constraints — as we have investigated in Part I of this thesis — are usually given by schemas. Popular schema languages for XML include XML Schema and Document Type Definitions (DTDs). Now, we will have a brief look on the interaction of integrity constraints with simple DTDs as defined in Chapter 3.3. Simple DTDs are an important subclass of DTDs.

It has been observed before (e.g. [KW07]), that simple DTDs imply certain integrity constraints as follows. Let D be a simple DTD. We define $\Sigma_D \subseteq \mathrm{TP}_\perp(/,//,\mathrm{FD}_s,\mathrm{NN}_s)$ as $\Sigma_D =_{\mathrm{def}}$

$$\left\{ (//a\langle x_a\rangle/b\langle x_b\rangle, x_a \to x_b) \;\middle|\; \begin{array}{l} a \rightsquigarrow \gamma b\gamma' \in D \vee \\ a \rightsquigarrow \gamma b?\gamma' \in D \end{array} \right\} \bigcup$$
$$\left\{ (//a\langle x_a\rangle/b\langle x_b\rangle, \mathrm{NN}((,x)_a, x_b)) \;\middle|\; \begin{array}{l} a \rightsquigarrow \gamma b\gamma' \in D \vee \\ a \rightsquigarrow \gamma b^+\gamma' \in D \end{array} \right\}$$

The first row contains all functional dependencies enforced by D, as there is at most one child with a particular label. The second row contains not null constraints enforced by D, as there is at least one child with a particular label.

Lemma 10.3 ([KW07]) *For every simple DTD D, every tree t with $t \models D$ satisfies Σ_D.*

In [KW07] it is stated, that for implication of functional dependencies under a given simple DTD D, the D can be replaced by the set Σ_D of dependencies (Theorem 4 in [KW07]). However, this is not entirely correct, as for example functional dependencies using labels not present in D are satisfied trivially under D.

11 Implication of XML-to-Relational Constraints

In this chapter, we investigate the complexity of the implication problem for XML-to-relational constraints.

For a set Σ of X2R-constraints and a single X2R-constraint τ we write $\Sigma \models \tau$ if for every tree t, $t \models \Sigma$ implies $t \models \tau$. If D is a schema, we write $\Sigma \models_D \tau$ if for every tree t with $t \models D$, $t \models \Sigma$ implies $t \models \tau$.

Of course, the complexity may depend on the actual choice of the allowed kinds of X2R-mappings, relational constraints and schema languages, therefore the implication problem has three parameters, $\mathcal{M}$, $\mathcal{C}$, and $\mathcal{S}$.

XCS-IMP$(\mathcal{M},\mathcal{C},\mathcal{S})$	
Given:	A set Σ of constraints and a single constraint σ from XC$(\mathcal{M},\mathcal{C})$, and a schema D from schema language $\mathcal{S}$.
Question:	Does $\Sigma \models_D \tau$?

We will also consider the implication problem (that is, whether $\Sigma \models \tau$) in which no schema is given. We denote it by XC-IMP$(\mathcal{M},\mathcal{C})$.

We will restrict to implication problems, where the relational constraints are functional dependencies and we also study the special case of XKFDs. We start with general upper and lower bounds, using first-order logic (FO) and monadic second-order logic (MSO) as the mapping language and the regular tree languages $\mathcal{S}$ as schemas. We consider MSO logic over a signature with the edge relation E, the children order $<_c$, and a unary relation P_a, for every symbol a. For FO logic we assume also the binary descendant relation.

An MSO formula Ψ over trees with free variables $x_1, \ldots, x_n$ defines a mapping

$$m_\Psi(t) = \{(x_1, x_1.@, \ldots, x_n, x_n.@) \mid t \models \Psi(x_1, \ldots, x_n)\}.$$

By our choice of signature for MSO-formulas, we ensure, that MSO-defined mappings do not depend on the data values, i.e., MSO-defined mappings are X2R-mappings.

Theorem 11.1

(a) XCS-IMP*(MSO,XKFD,Reg) is decidable.*

(b) XC-IMP*(FO,FD) is undecidable.*

	TP[/]		TP[/, ∗]	
	XKFD	FD	XKFD	FD
without DTD	in PTIME	in PTIME	in PTIME	**in PTIME**
simple DTD	in PTIME	**in PTIME**	**coNP**	coNP-hard **in EXPTIME**

	TP[/, //]		TP[/, //, ∗]	
	XKFD	FD	XKFD	FD
without DTD	**coNP**	**coNP**	**coNP**	coNP-hard
simple DTD	**coNP**	coNP-hard	**PSPACE**	**undecidable**

Table 11.1: Complexity results for the implication problem. Highlighted complexities are main results. The other results are by restriction/generalization.

A proof is given at the end of this chapter.

Theorem 11.1 shows that the restriction to XKFDs yields a decidable implication problem, even for very powerful mapping languages like MSO. However, the complexity of XCS-Imp(MSO, XKFD, Reg) is non-elementary, as this already holds for the satisfiability problem for first-order logic on strings [Sto74].

In the remainder of this chapter, we restrict our attention to more tractable instances of the implication problem, based on tree pattern mappings, that is, we investigate the complexity of XC-Imp(TP, FD, sDTD) and XC-Imp(TP, XKFD, sDTD) as well as of implication problems based on more restricted tree patterns and/or without schemas.

More precisely, we show the complexities in Table 11.1. All lower bounds (including the undecidability result) in the presence of schemas already hold for esDTDs and FDs.

Theorem 11.2

(a) XCS-Imp(*TP*[/], *FD*, sDTD) *and* XC-Imp(*TP*[/, ∗], *FD*) *are in* PTIME.

(b) The following implication problems are complete for coNP*:*

- XC-Imp(*TP*[/, //], *FD*),
- XC-Imp(*TP*[/, //], *XKFD*),
- XCS-Imp(*TP*[/, //], *XKFD*, sDTD),
- XCS-Imp(*TP*[/, ∗], *XKFD*, sDTD), *and*
- XC-Imp(*TP*, *XKFD*).

(c) XCS-Imp(*TP*[/, ∗], *FD*, sDTD) *is in* EXPTIME.

(d) XCS-Imp(*TP*, *XKFD*, sDTD) *is* PSPACE*-complete.*

(e) XCS-Imp(*TP*, *FD*, sDTD) *is undecidable.*

term	symbol	definition	usage
implication instance	$I = (\Sigma, \tau, D)$	tuple consisting of a set of constraints Σ, a target dependency τ and optionally a DTD D	
witness pair	(π_1, π_2)	pair of 2 embeddings proving that an X2R constraint σ is violated in a tree t	
z-witness pair	(π_1, π_2)	partial witness pair for the subpattern rooted at z	
initial tree	t_τ	most general counterexample to a target dependency τ	11.2
tree homomorphism	Θ, $t_1 \preceq_\Theta t_2$	function mapping nodes and data values of a tree t_1 to nodes and data values of a tree t_2 that is compatible with the edge relation, labels and data values of t_1	11.2

Table 11.2: Terms used in this chapter, together with the usually used symbols, a brief definition and optionally the section, where they are used.

A *counter-example* for an instance (Σ, τ, D) of the implication problem is a finite tree t with $t \models \Sigma$, $t \not\models \tau$, and $t \models D$. All upper bounds depicted in Table 11.1 are based on counter-examples — in some cases counter-examples are computed by chase algorithms (PTIME upper bounds and EXPTIME upper bound), in others they are non-deterministically guessed and the bound follows by a "small or simple" counter-example property. We prove the upper bounds based on chase algorithms in Section 11.2 and those based on small counter-examples in Section 11.3. The PSPACE upper bound is based on more complex counter-example properties and shown in Section 11.4. The lower bounds are shown in Section 11.5 and 11.6. As a tool for all kinds of upper bounds we introduce the notion of witness pairs in Section 11.1 and show that they can be computed in polynomial time.

In Table 11.2, we give brief definitions of terms used in this chapter. Detailed definitions are given where needed.

11.1 Witness Pairs and Model Checking

Informally, a witness pair (π_1, π_2) for a tree t and a pattern-based X2R-constraint $\sigma = (p, Y \xrightarrow{Z} B)$ is a pair of embeddings of p into t that shows that σ does not hold in t.

Additionally to witness pairs for complete dependencies, we define z-witness pairs for sub-patterns P_z of p, where p_z denotes the sub-pattern rooted at at some variable z of p. These z-witness pairs will mainly be used in the dynamic programming algorithm for model checking. Abusing notation slightly, we will write $Y \cap p_z$ for a set of attributes Y, to denote the set of all attributes (node and data terms) that occur in Y and p_z.

Let $\sigma = (p, Y \xrightarrow{Z} B)$ be an X2R-constraint, z a node of p, and t a tree. Let π_1, π_2 be two partial embeddings of p_z in t. We call (π_1, π_2) a *z-witness pair for σ in t* if

- for every $C \in (Y \cup Z) \cap p_z$ it holds $\pi_1(C) \neq \bot$ and $\pi_2(C) \neq \bot$;
- for every $C \in Y \cap p_z$ it holds $\pi_1(C) = \pi_2(C)$;
- for every $x \in p_z$ and every $i \in \{1, 2\}$ it holds that if $\pi_i(x) = \bot$ and $\pi_i(\text{parent}(x)) \neq \bot$, then $\pi_i(\text{parent}(x))$ has

  - no children if $\text{lab}(x) = *$
  - no $\text{lab}(x)$ labeled child if $\text{lab}(x) \neq *$ and $(\text{parent}(x), x)$ is a child edge
  - no $\text{lab}(x)$ labeled descendant if $\text{lab}(x) \neq *$ and $(\text{parent}(x), x)$ is a descendant edge; [1] and

- if $B \in p_z$, then $\pi_1(B) \neq \pi_2(B)$ and $\pi_1(B) \neq \perp$.[2]

A *witness pair for σ in t* is a root(p)-witness pair for σ in t. We note that in a z-witness pair for subpatterns not containing B, both embeddings of the subpattern may be identical.

In the proof of the correctness of the chase algorithm, we will further assume that π_1 always is a full embedding and that the only nodes mapped to $\perp$ in π_2 are on the root-path of B. This can be enforced by restricting the pattern p to nodes in $Y \cup Z \cup B$ and their ancestors. Note that restricting p to these nodes does not change the semantics of the dependency.

The significance of witness pairs is illustrated by the following lemma which is straightforward to show.

Lemma 11.3 *For a tree t and an X2R-constraint σ it holds $t \models \sigma$ if and only if there does* not *exist any witness pair for σ in t.*

In the presence of an sDTD D, witness pairs for a tree of the form $[t]_D$, for some tree t, are specified by embeddings with relative node addresses.

The following lemma will be useful both for chase-based as well as for counter-example based algorithms. It shows that even for the most general kind of X2R-constraints considered, (1) it can be checked in polynomial time whether a constraint holds in a given tree, and (2) if the constraint does *not* hold, a witness pair can be computed in polynomial time.

Lemma 11.4

(a) There is a polynomial time algorithm that tests whether $t \models \sigma$ for trees t and constraints $\sigma = (p, Y \xrightarrow{Z} B) \in XC(TP, FFD)$ and computes a witness pair (π_1, π_2) if $t \not\models \sigma$.

(b) Given a sDTD *D, the algorithm tests in polynomial time whether $[t]_D \models \sigma$ and computes a witness pair if $[t]_D \not\models \sigma$.*

Proof. We start with (a). The algorithm is an adaptation of the algorithm in [MS04], which computes whether a tree pattern can be embedded in a tree t and follows a simple dynamic programming approach. It computes, in a bottom-up fashion, a ternary relation W that contains all triples (u, v, z) of nodes u, v of t and a node z of p, for which there

[1] This constraint ensures that all embeddings are maximal, as required in the definition of tree pattern based mappings.

[2] The restriction $\pi_1(B) \neq \perp$ is not strictly necessary, but it will simplify some proofs. Note that at least one of $\pi_1(B)$ and $\pi_2(B)$ has to be different from $\perp$ in any case and we can exchange π_1 and π_2.

exists a z-witness pair (π_1, π_2) such that $\pi_1(z) = u$ and $\pi_2(z) = v$. Note that we allow u and/or v to be $\perp$, if σ is an FFDs.

We explain, how $(u, v, z) \in W$ can be decided, once W is computed for all triples (u', v', z') with nodes u' below u, v' below v and pattern nodes z' below z. We distinguish 3 cases, that depend on whether u and v are null values.

The first case is that u and v are both non-null. The tuple (u, v, z) is added to W, if all the following conditions hold.

- $\text{lab}(z) = *$ or $\text{lab}(u) = \text{lab}(v) = \text{lab}(z)$.
- If Y contains z then $u = v$.
- If Y contains $z.@$ then $u \sim v$.
- If B is z then u and v are different nodes.
- If B is $z.@$ then u and v carry different data values, that is $u \not\sim v$.
- For every $A_/$-child z' of z, there is a child u' of u and a child v' of v such that $(u', v', z') \in W$. Instead of being children of u and v, one or both of u' and v' can be $\perp$. We note that $\{z', z'.@\} \cap (Y \cup Z) = \emptyset$ is checked when adding triples containing z'.
- For every $A_{//}$-child z' of z, there is a node u' strictly below u and a node v' strictly below v such that $(u', v', z') \in W$. Again u' and/or v' can be $\perp$.

The second case is that exactly one of u or v is $\perp$. We assume that v is $\perp$. The other case is completely symmetric. The tuple $(u, \perp, z)$ is added to W, if all the following conditions hold.

- $z \notin Y \cup Z$.
- $\text{lab}(z) = *$ or $\text{lab}(u) = \text{lab}(z)$.
- For every $A_/$-child z' of z, there is a child u' of u such that $(u', \perp, z') \in W$. Instead of being a child of u, u' can be $\perp$.
- For every $A_{//}$-child z' of z, there is a node u' strictly below u such that $(u', \perp, z') \in W$. Again u' can be $\perp$.

The last case is $u = v = \perp$. The tuple $(\perp, \perp, z)$ is added to W, if all the following conditions hold.

- $z \notin Y \cup Z \cup B$.
- For every $A_/$-child or $A_{//}$-child z' of z, it holds that $(\perp, \perp, z') \in W$.

It is easy to prove by induction on the depth of subpatterns that the final relation W exactly contains those triples (u, v, z) of nodes u, v of t and a node z of p, for which there exists a z-witness pair (π_1, π_2) such that $\pi_1(z) = u$ and $\pi_2(z) = v$.

This algorithm can be performed in $\mathcal{O}(|t|^4|p|)$ steps and thus in polynomial time.

Therefore, by Lemma 11.3, $t \not\models \sigma$ holds, if and only if $(\text{root}(t), \text{root}(t), \text{root}(p)) \in W$. It is straightforward to construct a witness pair (π_1, π_2) in a top down fashion from W if $t \not\models \sigma$.

We now sketch the proof of (b). We extend the relation W to include triples, where the first two components can be labels from a given sDTD D instead of nodes from t. The intended meaning of (a, b, z) is that there exists a z-witness pair (π_1, π_2), where $\pi_1(z) = \text{root}(t_a)$ and $\pi_2(z) = \text{root}(t_b)$, where t_a and t_b are subtrees of $[t]_D$ that are mandatory by D and not included in t. The meaning of mixed triples, where only one of the first two components is a label, is analogous.

The algorithm first computes all triples, where both first components are labels, starting with labels, where D does not specify any mandatory child nodes. In the case that both labels are equal, the algorithm always needs to distinguish, whether they refer to the same node or to different nodes with the same label.

The running time of the algorithm changes to $\mathcal{O}((|t| + |D|)^4|p|)$, due to the increased size of W. □

We note that the run time of the above algorithm can be improved to $\mathcal{O}(|t|^2|p|)$ steps by computing another relation W' containing all triples (u, v, z) for which there exists a z-witness pair (π_1, π_2) such that $\pi_1(z) = u'$ and $\pi_2(z) = v'$, for some nodes u' below u and v' below v.

The following lemma will be often used in proofs. We call a tree t *π-diverse*, for a witness pair $\pi = (\pi_1, \pi_2)$ for some σ in t if all nodes outside the range of π carry pairwise distinct data values that are different from the data values of the nodes in the range of π.

Lemma 11.5 *Let t be a counter example tree for some instance (Σ, τ, D) of* XC-IMP(*TP, XKFD,* sDTD) *and π be a witness pair with respect to τ. Then, by changing data values in t, a π-diverse counter-example t' for (Σ, τ, D) can be obtained.*

Proof sketch. Let t' be an arbitrary π-diverse tree obtained from t by changing data values outside the range of π. As π is not changed it remains a witness pair for τ in t'. On the other hand, as no new equalities between data values are introduced, all XKFDs from Σ still hold in t'. □

It should be noted that the Lemma 11.5 does not hold for arbitrary (fictitious) functional dependencies because references to data terms can occur on the right-hand side. However, the lemma can easily be generalized with respect to other mapping languages.

11.2 Chasing on Trees

The outline of this section is as follows: We first describe a rather direct application of the relational chase on X2R-mappings and discuss some difficulties with this approach.

Afterwards, we describe a chase algorithm working directly on trees including an example run. However, we will restrict to (non-fictitious) functional dependencies, as the chase gets quite technical in the presence of null values. We continue by proving the correctness of the chase algorithm and deriving some upper bounds using the chase. We close this section with an extension of the chase algorithm to fictitious functional dependencies and the corresponding correctness proof.

Applying the Relational Chase to X2R-mappings

As already said, we now sketch an (exponential time) chase, which works on the produced relation(s) instead of the tree.

For simplicity, we assume, that we only have to deal with one relation R, because all functional dependencies use the same tree pattern p. Without proof, we note that this can be enforced by converting the FDs to FFDs and merging all patterns to one "universal" pattern.

The chase based on R needs to incorporate the following constraints, which are implicit, due to the tree structure of our data model:

- every tree has a unique root
- every node (except the root) has a unique parent
- every node has a unique data value
- join dependencies corresponding to branchings in the pattern[3]
- inclusion dependencies corresponding to inclusion of sub-patterns

The first 3 constraints can be described by relational functional dependencies. For details see [KW07]. For the other constraints, we just give two examples. Let p be the tree pattern $/a\langle x\rangle[/b\langle y\rangle]/c\langle z\rangle$. Due to the branching structure of trees, the join dependency $\sigma_{\bowtie} = \{x, y\} \bowtie \{x, z\}$ holds for all trees. Let now p be the pattern $/a[/b\langle x\rangle]/b\langle y\rangle/c$. Due to the inclusion of sub-patterns of p, the inclusion dependency $\sigma_{\subseteq} = y \subseteq x$ holds for all trees.

The standard chase algorithm has an exponential worst-case running time when these constraints are added. The reason is, that join and inclusion dependencies can enforce an exponential size relation. In [KW07] it is shown how the join dependencies can be incorporated into the chase by additional chase rules that avoid an exponential blow-up. This way, it is no longer necessary to compute all tuples resulting from join dependencies explicitly. However, the incorporation of the implicit inclusion dependencies seems to be harder. This is one reason, why we now show, how the chase can directly be applied to trees. Another reason is, that we believe. that a tree based chase has application outside of the implication problem. For example the chase can be applied to an existing database to enforce constraints.

[3]We will not introduce join and inclusion dependencies formally.

Algorithm 6 Tree Chase

```
1: function CHASE(t, Σ)
2:     while ∃σ = (p, Y→B) ∈ Σ. t ⊭ σ do
3:         (π1, π2) := witness-pair(t, σ)
4:         merge(t, π1(B), π2(B))
```

Algorithm 7 Merge two nodes

```
1: function MERGE(t, v1, v2)
2:     if v1 = v2 then return
3:     if v1 = root(t) ∨ v2 = root(t) then fail
4:     replace all occurrences of v2.@ by v1.@
5:     if lab(v1) ≠ lab(v2) then
6:         if lab(v1) = # then lab(v1) := lab(v2)
7:         else if lab(v2) = # then lab(v2) := lab(v1)
8:         else fail
9:     replace all occurrences of v2 with v1
10:    merge(t, parent(v1), parent(v2))
```

Description of the Tree Based Chase

The tree chase algorithm works similarly as the relational chase. Starting from a tree that does not satisfy τ, it applies chase steps as long as there exists a dependency $\sigma = Y \rightarrow B \in \Sigma$, that is not satisfied by the current tree. Whether σ is satisfied in the current tree is tested by the algorithm of Lemma 11.4. A single tree chase step either merges two nodes or identifies two data values, depending on whether B is a node or a data term, and based on the witness pair yielded by the test algorithm.

To be able to chase on trees, we need three ingredients:

(1) An algorithm which computes an initial tree, which is a minimal counter-example to the target dependency.

(2) The actual tree chase.

(3) A subprocedure for the propagation of the merge of two nodes, which is used in (2).

As the definition of the initial tree can be easily described with the help of (3), we first describe (2) and (3).

In the following, let $I = (\Sigma, \tau)$ be an instance of XC-IMP(TP[/, ∗], FD) with $\tau = (p, Y \rightarrow B)$. We already note that the initial tree, t_τ for the basic tree chase might have nodes labeled by # that indicate that the label of that node has not yet been fixed by the algorithm and still may match any (but only one) label.

Algorithm 6 implements (2) and uses the merge algorithm given as Algorithm 7 for (3). Algorithm 6 has as input the tree t and the set Σ of dependencies used for chasing.

Algorithm 8 Algorithm for XC-IMP(TP[/, ∗], FFD)

1: Compute initial tree t_τ
2: **if** chase(t, Σ) fails **then** Output "Yes"
3: $t :=$ chase(t, Σ)
4: **if** $t \models \tau$ **then** Output "Yes" **else** Output "No"

The identification of two data values d_i and d_j is simply done by replacing all occurrences of d_j by d_i, and it does not matter which is replaced by which.[4] The merge of two different nodes v_1 and v_2 requires a bit more care: First of all, it is only possible if the labels of v_1 and v_2 are compatible, which is the case if they are equal or one of them is the wildcard label #. If the labels are compatible, the nodes can be combined into one node which gets all children of v_1 and v_2. However, unless v_1 and v_2 have the same parent, their parents have to be merged recursively. Otherwise the structure would no longer be a tree. This is exactly the point, where the tree chase differs from the relational chase. It should be noted that, as we apply the tree chase only in the context of tree patterns without descendant axis, only nodes of the same depth need to be merged.

Next, we define the initial tree t_τ for the basic tree chase for a given instance $I = (\Sigma, \tau)$ with $\tau = (p, Y \to B)$. Intuitively, it is minimal with the property $t_\tau \not\models \tau$. To this end, let t_1 and t_2 be two copies of p (which use node ids from $\mathcal{V}$ instead of variables from X) in which all data values are distinct (every data value occurs at most once in $t_1 \cup t_2$) and π_1 and π_2 be the canonical embeddings of p in t_1 and t_2, respectively. All nodes in t_1 and t_2 whose corresponding node in p has a wildcard label $*$ are labeled by #.[5] The tree t_τ results by merging the roots of t_1 and t_2 and all pairs $(\pi_1(z), \pi_2(z))$, for which z occurs as a node term in Y and it identifies all pairs of data values $(\pi_1(z).@, \pi_2(z).@)$, for which $z.@$ is a data term in Y. By applying the node merges the embeddings π_1 and π_2 yield two embeddings π_1' and π_2' such that (π_1', π_2') is a witness pair for t_τ and τ.

The decision algorithm for XC-IMP(TP[/, ∗], FD) is given as Algorithm 8.

Example 11.6 *An example run of the chase algorithm is depicted in Figure 11.3. Starting from the initial tree, the run corresponds to testing the implication of* $\{\sigma_1, \sigma_2\} \models \tau$*, where all dependencies use the tree pattern*

$$p = /root/person\langle x_p\rangle[/name\langle x_n\rangle]/user\text{-}id\langle x_u\rangle,$$

$\sigma_1 = (p, x_u.@ \to x_u)$, $\sigma_2 = (p, x_p \to x_n)$ *and* $\tau = (p, x_u.@ \to x_n.@)$*. Intuitively,* σ_1 *expresses that user IDs are unique,* σ_2 *that every person only has one name, and* τ *whether every*

[4] We note without proof, that one can achieve that the resulting tree is independent of the order in which violated dependencies are corrected, by introducing a total order on the data values and always replace the larger data value with the smaller one. Analogously a total order on the nodes of the tree needs to be introduced.

[5] We choose a different label because the meanings of $*$ and # are slightly different. The label $*$ is a true wildcard that can match any symbol. The label # is chosen for a node, which has some unique label, but we do not know yet which label this should be. The label $*$ only occurs in patterns while the label # only occurs in trees.

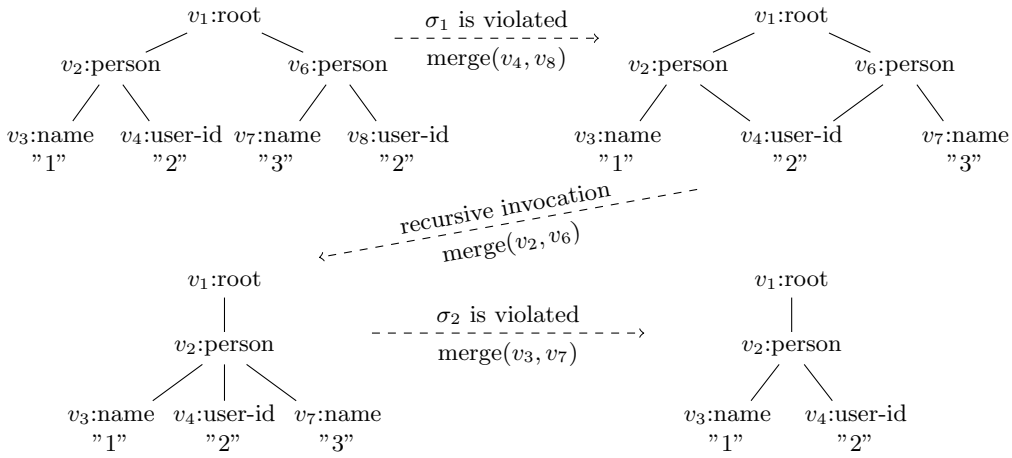


Figure 11.3: Example run of the chase algorithm.

user ID has exactly one associated name. Note that the constraints only apply to person, name and user-id nodes, where the person has at least one name and at least one user ID, as other nodes are not contained in the mapping.

The chase merges v_4 and v_8, as σ_1 enforces them to be equal. The recursive call in Line 11 of the merge function unifies v_2 and v_6 to restore the tree structure. Finally v_3 and v_7 and their data values are identified, as σ_2 is now violated. We note while in this case there exists only one possible run of the chase algorithm, in general there can be many runs, which differ in the order in which the rules are applied. In the resulting tree, $\{\sigma_1, \sigma_2\}$ is satisfied, as well as τ. We will see in the proof of Proposition 11.7 that this implies $\{\sigma_1, \sigma_2\} \models \tau$.

This concludes our description of the chase algorithm and we can continue with the correctness proof.

Correctness of the Tree Based Chase

Before we state the complexity result for XC-Imp(TP[/, ∗], FD), we first show the correctness of Algorithm 8.

Proposition 11.7 *For every instance $I = (\Sigma, \tau)$ of* XC-Imp(*TP*[/, ∗], *FD*)*, Algorithm 8 terminates and answers "Yes" if and only if $\Sigma \models \tau$.*

Proof. Let $I = (\Sigma, \tau)$ be an instance of XC-Imp(TP[/, ∗], FD) with $\tau = (p_\tau, Y_\tau \to B_\tau)$.[6] Clearly, if Algorithm 8 terminates and yields a tree t, no constraint from Σ is violated in t. Thus, if the output of Algorithm 8 is "No" (and thus $t \not\models \tau$), t is a counter-example

[6] We use the index τ to distinguish references to components of the target dependency from references to dependencies from Σ.

for $\Sigma \models \tau$ and thus, the answer "No" is always correct. The proof that "Yes"-answers are also correct uses the following notion of tree homomorphisms. Tree homomorphisms map nodes to nodes and data values to data values.

More formally, a *tree homomorphism* θ from a tree $t_1 = (V_1, E_1 \operatorname{lab}_1, \operatorname{dv}_1) \in \mathcal{T}$ to a tree $t_2 = (V_2, E_2 \operatorname{lab}_2, \operatorname{dv}_2) \in \mathcal{T}$ is a function $\theta : \mathcal{V} \cup \mathcal{D} \to \mathcal{V} \cup \mathcal{D}$, such that

- $\theta(t_1)$ is a valid tree,
- $\theta(V_1) \subseteq V_2$, $\theta(E_1) \subseteq E_2$, and
- for all $v \in V_1$, it holds that $\operatorname{dv}_2(\theta(v)) = \theta(\operatorname{dv}_1(v))$ and $\operatorname{lab}_1(v) \neq \#$ implies $\operatorname{lab}_2(\theta(v)) = \operatorname{lab}_1(v)$.

In this case, we write $t_1 \preceq_\theta t_2$.

Let t_i be the tree after i chase steps where a chase step is a call of merge in Algorithm 6. We note that recursive invocations of merge (in Algorithm 7) are no separate chase steps. For convenience t_0 is the initial tree.

Claim 11.8 *If there is a counter-example tree t' for $\Sigma \models \tau$ with witness pair $(\rho^1_{t'}, \rho^2_{t'})$ for τ and t', then the tree chase on input (Σ, τ) does not fail and for every chase step i it holds that*

(i) there exist a tree homomorphism $t_i \preceq_{\theta_i} t'$;

(ii) there exist a witness pair (ρ^1_i, ρ^2_i) for $t_i \not\models \tau$; and

(iii) $\theta_i(\rho^j_i(x)) = \rho^j_{t'}(x)$ for $j \in \{1, 2\}$ and all terms x of p_τ.

Applying this claim to the final tree t immediately yields the correctness of the algorithm: if there is a counter example tree for $\Sigma \models \tau$, the tree chase does not fail, and thus $t \models \Sigma$, Furthermore, by (ii) $t \not\models \tau$. Thus the algorithm answers "No", as desired. We note that t might well contain nodes labeled by #. In the end, they do not have any special meaning. However, it remains true that they might only match wildcard nodes of a pattern, not any other nodes.

The proof of Claim 11.8 is by induction on the number of chase steps. For the induction base we show that the initial tree $t_0 = t_\tau$ fulfills the condition of the claim. Let t' be a counter-example to $\Sigma \models \tau$ and let $(\rho^1_{t'}, \rho^2_{t'})$ be a witness pair for $t' \not\models \tau$. As $\rho^1_{t'}$ and $\rho^2_{t'}$ need to coincide on nodes from Y_τ and as nodes u, v in t_τ have different data values unless value equality is enforced by Y_τ, there are partial homomorphisms from $\pi'_1(p)$ to t' and from $\pi'_2(p)$ to t' which can be combined to a homomorphism θ from t_τ to t' fulfilling (i) and (iii).

For the induction step, we always assume that $\sigma = (p, Y \to B) \in \Sigma$ does not hold in t_i and that (χ_1, χ_2) is a witness pair for t_i and σ. If there would be no such dependency σ, the chase would terminate and there would be nothing to show.

As χ_1 and χ_2 are embeddings of p in t_i, $\theta_i \circ \chi_1$ and $\theta_i \circ \chi_2$ are embeddings of p in t'. As (χ_1, χ_2) is a witness pair for t_i and σ, $\chi_1(C) = \chi_2(C)$, for every $C \in Y$, and thus

$\theta_i(\chi_1(C)) = \theta_i(\chi_2(C))$, for every $C \in Y$. As $t' \models \sigma$, we can conclude that

$$\theta_i(\chi_1(B)) = \theta_i(\chi_2(B)). \tag{11.1}$$

We distinguish two cases, depending on whether B is a data term or a node term.

If B is a data term, the tree structure is not changed by the call to merge, as t_{i+1} results from t_i by replacing the data value $\chi_1(B)$ with the data value $\chi_2(B)$. We define $\theta_{i+1} = \theta_i$, $\rho^1_{i+1} = \rho^1_i$ and $\rho^2_{i+1} = \rho^2_i$. From Equation 11.1 and $t_i \preceq_{\theta_i} t'$ we get, that θ_{i+1} is a valid tree homomorphism from t_{i+1} to t' and thus (i) is satisfied. Furthermore $(\rho^1_{i+1}, \rho^2_{i+1})$ is a witness pair for $t_{i+1} \not\models \tau$ and $\theta_{i+1}(\rho^j_i(x)) = \rho^j_{t'}(x)$ still holds for $j \in \{1, 2\}$ and all x, as the structure of t_i has not changed and $\theta_i(\rho^1_i(B_\tau)) \neq \theta_i(\rho^2_i(B_\tau))$. Thus θ_{i+1} also satisfies (ii) and (iii).

If B is a node term, the labels of $\chi_1(B)$ and $\chi_2(B)$ are compatible, thanks to Equation 11.1. The same holds for the ancestors of $\chi_1(B)$ and $\chi_2(B)$. Thus, the next tree t_{i+1} is merge$(t_i, \chi_1(B), \chi_2(B))$ and the chase does not fail. We define θ_{i+1} as the function resulting from θ_i by restricting the domain to (the nodes of) t_{i+1}. It is easy to verify, that $t_{i+1} \preceq_{\theta_{i+1}} t'$ and thus (i) holds. Towards (ii) and (iii), we define $\rho^j_{i+1} = \nu \circ \rho^j_i$ for $j \in \{1, 2\}$, where ν is the tree homomorphism from t_i to t_{i+1} induced by the merge operation.[7] As ν is a tree homomorphism from t_i to t_{i+1} and ρ^1_i and ρ^2_i are embeddings of p in t_i, ρ^1_{i+1} and ρ^2_{i+1} are embeddings of p in t_{i+1}.

From Equation 11.1 and the definition of ν it follows that $(\theta_i \circ \nu^{-1} \circ \nu)(y) = \theta_i(y)$, for every term y of t_i. Therefore, we get

$$\begin{aligned} (\theta_{i+1} \circ \rho^j_{i+1})(y) &= (\theta_i \circ \nu^{-1} \circ \nu \circ \rho^j_i)(y) \\ &= (\theta_i \circ \rho^j_i)(y) \end{aligned} \tag{11.2}$$

for every term y of t_i and $j \in \{1, 2\}$. We can conclude that (ii) and (iii) still hold by applying (11.2) to the induction hypothesis.

This concludes the proof of the claim and thus the proof of the proposition. □

Complexity Results Based on the Chase

We get the following easy corollary.

Corollary 11.9 XC-Imp($TP[/, *], FD$) *can be solved in polynomial time.*

Proof. As the algorithm is correct and witness pairs can be computed in polynomial time (Lemma 11.4) and there are at most linearly many merge steps the algorithm always terminates and only needs polynomial time. □

The tree chase can be extended in the presence of sDTDs, however the definition of the initial tree has to be adapted[8], as the initial tree should be consistent with D. This

[7] That is ν maps $\chi_2(B)$ and its root path to $\chi_1(B)$ and its root path and is the identity on all other nodes.

[8] And we will see soon that there is more than one initial tree.

modification might involve replacing leave nodes with a label ℓ by trees t_ℓ but also the insertion of additional trees of the form t_ℓ below inner nodes of t_τ and the merge of two sibling nodes if the sDTD only allows one child with their label. Another difference to the schema-free case is that we apply the chase to a set T of trees that results from the initial tree by replacing #-labels in all possible ways. If the modified initialization is successful then during the tree chase only D-valid trees will be constructed and the correctness proof is similar to the one of Proposition 11.7.

Proposition 11.10 *For every instance* $I = (\Sigma, \tau, D)$ *of* XC-Imp($TP[/, *], FD,$ sDTD)*, Algorithm 8 with modified initialization terminates for some tree in* T *and answers "Yes", if and only if* $\Sigma \models_D \tau$*.*

Proof. Let $I = (\Sigma, \tau, D)$ be an instance of XCS-Imp(TP$[/, *]$, FD, sDTD) with $\tau = (p, Y \xrightarrow{Z} B)$ and sDTD D. Let t_τ be defined as in the proof of Proposition 11.7. Let $t_{\tau,D}$ be the tree resulting from t_τ by adding a new node with label ℓ and a new data value as a child of each inner vertex v which (according to D) requires an ℓ-labeled child. Let $T_{\tau,D}$ denote the set of all trees that can be obtained from $t_{\tau,D}$ by replacing each #-label in $t_{\tau,D}$ with some other label allowed and consistent with D. If necessary, further children trees for nodes whose label changes might be added to accommodate D, analogous as before.

Claim 11.11 *If there is a counter-example tree* t' *for* $\Sigma \models \tau$*, then there is a tree* $t_0 \in T_{\tau,D}$ *such that the tree chase on input* (Σ, τ) *starting from* t_0 *does not fail and for every tree* t *that occurs in the tree chase before or after a call of* merge *the following conditions hold.*

(i) There exists a witness pair (ρ_1, ρ_2) *for* τ *in* t*, and*

(ii) $t \preceq_\theta t'$*, for some tree homomorphism with* $\theta(\rho_1(B)) \neq \theta(\rho_2(B))$*.*

The claim can be shown by induction on the number of chase steps similar to the proof of Claim 11.8 and it yields the proposition. □

However, as T might consist of an exponential number of trees of exponential size (in the size of D and τ), Proposition 11.10 does not immediately yield a polynomial time algorithm. However, we get the following result.

Proposition 11.12 XC-Imp($TP[/, *], FD,$ sDTD) *can be solved in exponential time.*

Proof. For the exponential time bound we observe that

(1) the algorithm only uses trees whose depth is bounded by the number of symbols in D plus the depth of p;

(2) the algorithm only uses trees in which the number of children per node is at most twice the number of symbols in D; and

(3) the number of # symbols in $[t']_D$ is at most $2|\tau|$ and thus $|T_{\tau,D}|$ is at most $|D|^{2|\tau|}$.

By (1) and (2), all trees in $T_{\tau,D}$ are of at most exponential size in $|p|$ and $|D|$. Therefore, it is possible in exponential time, to apply the chase algorithm to all (at most) exponentially many trees in $T_{\tau,D}$, thus yielding the desired result. □

For XC-Imp(TP[/], FD, sDTD) we can do better by using a condensed representation of trees that avoids the exponential blowup that might be caused by the sDTD.

Proposition 11.13 XC-Imp($TP[/], FD$, sDTD) *can be solved in polynomial time.*

Proof. Let $I = (\Sigma, \tau, D)$ be an instance of XC-Imp(TP[/], FD) with $\tau = (p, Y \to B)$ and simple sDTD D. We basically use the algorithm from the proof of Proposition 11.12. However, thanks to the fact that tree patterns do not contain wildcard symbols here, there are no #-symbols in $t_{\tau,D}$ and therefore, $T_{\tau,D} = \{t_{\tau,D}\}$.

We modify the algorithm of Proposition 11.12 in that it never explicitly applies the extension $[t]_D$ to any tree but rather works with t. The modification of the algorithm of Proposition 11.12 is as follows. Whenever a witness pair (π_1, π_2) occurs, for which $\pi_1(y)$ or $\pi_2(y)$ is in $[t]_D - t$ then t is extended by this node (or both) and its ancestors which are not yet in t. This guarantees the invariant that nodes in $[t]_D - t$ have pairwise distinct data values, different from those in t. Otherwise, the algorithm remains unchanged.

The polynomial upper bound follows from the observation that for each node v in t and each path in some pattern (without wildcard and descendant axis) there can be at most one node in $[t]_D$ below v. Therefore, the total number of tree extending steps is bounded by $2|\tau||D||\Sigma|$. Here, $2|\tau||D|$ bounds the size of $t_{\tau,D}$ and $|\Sigma|$ bounds the number of extension steps that are possible below one node of $t_{\tau,D}$.

As the number of chase steps is polynomially bounded in the size of the occurring trees and this size is bounded in $|\tau||D|$ we obtain a polynomial time bound. □

Chasing with Fictitious Functional Dependencies

Now, we extend the chase algorithm, such that it works in the presence of fictitious functional dependencies. To extend the chase algorithm, we have to deal with two separate issues. First, the dependencies in Σ can be fictitious. In this case, we have the problem that $\pi_2(B)$ might be null in Line 4 of Algorithm 6.[9] We note that Algorithm 7 cannot handle null values. On the other hand, the target dependency τ can be fictitious.

To address both issues, we change the definition of the initial tree and extend Algorithm 6, as can be seen in Algorithm 9. The red parts (Lines 6–11 and additional function parameters) are added to deal with a fictitious target dependency and the blue part (Lines 4–5) is added to deal with fictitious dependencies in Σ. We note that removing the red and blue parts in Algorithm 9 gives exactly Algorithm 6. Algorithm 10 is identical to Algorithm 8, except that it computes a witness pair for $t_\tau \not\models \tau$ and provides the additional parameters to the chase function.

We first describe how we deal with fictitious dependencies in Σ, especially with the case that $\pi_2(B)$ is null in Line 4. We address this problem, by adding a function that

[9] Remember, that we assume w.l.o.g. that $\pi_1(B) \neq \perp$.

Algorithm 9 Tree Chase with FFDs

```
1:  function CHASE(t, Σ, τ = (p_τ, Y_τ --Z_τ--> B_τ), (ρ1, ρ2))
2:      while ∃σ = (p, Y --Z--> B) ∈ Σ. t ⊭ σ do
3:          (π1, π2) := witness-pair(t, σ)
4:          if π2(B) = ⊥ then
5:              (t, π2) := remove-null(t, p, π2, B)
6:          if ρ2(B_τ) = ⊥ and B is a node term then
7:              y := highest ancestor of B_τ with ρ2(y) = ⊥
8:              x := parent(parent(x))
9:              if π1(B) and π2(B) are descendants of ρ1(x) and ρ2(x) then
10:                 (t, ρ2) := remove-null(t, p, ρ2, y)
11:                 goto 6:
12:         merge(t, π1(B), π2(B))
```

Lines 4–5 deal with fictitious dependencies in Σ
Lines 6–11 deal with fictitious target dependencies.

Algorithm 10 Algorithm for XC-IMP(TP[/, ∗], FFD)

```
1: Compute initial tree t_τ and a witness pair (ρ1, ρ2) for t_τ ⊭ τ
2: if chase(t, Σ, τ, (ρ1, ρ2)) fails then Output "Yes"
3: t := chase(t, Σ, τ, (ρ1, ρ2))
4: if t ⊨ τ then Output "Yes" else Output "No"
```

removes null values from embeddings by adding nodes to the tree, such that the null value is replaced by a node or data value. We therefore define the function remove-null that takes as input a tree t, a pattern p, a partial embedding π of p in t and a node x of p such that $\pi(x) = \bot$.

The function remove-null computes and returns a tree t' and a (partial) embedding π' of p in t' as follows. Let y be the lowest ancestor of x with $\pi(y) \neq \bot$, z be the child of y on the path $[y, z]$ and v be $\pi(y)$. The tree t' is derived from t by adding a copy of the path $[z, x]$ to t below $\pi(y)$, where each added node gets a fresh node identifier from $\mathcal{V}$ and a fresh data value from $\mathcal{D}$ and wildcard symbols are replaced by #. The embedding π' is derived from π by embedding the path $[z, x]$ to the newly inserted path in t'.

The intuitive idea behind this function is that $\pi_1(B)$ must be identified with $\pi_2(B)$ to satisfy σ. Therefore either $\pi_1(B)$ must become $\bot$ (meaning $\pi_2(y)$ has to be removed from the tree) or $\pi_2(B)$ most become equal to $\pi_1(B)$, which especially means that it must become different to $\bot$. We just note that removing nodes from the tree (contrary to merging them) is a bad idea, because the initial tree is an — in some sense — minimal counterexample. Therefore we add nodes to remove the nulls. For correctness we refer to the formal proof given below.

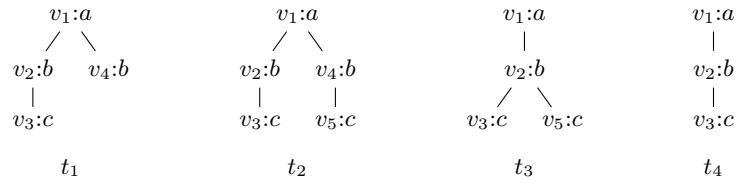


Figure 11.4: Example run of the chase algorithm for fictitious dependencies.

Note that in the case that $B = y$, where y is a node, the chase will merge the newly added node with $\pi_1(B)$ afterwards, i.e., the effect of remove-null followed by the merge is equivalent of merging $\pi_1(y)$ with $\pi_2(y)$, where y is defined as the lowest ancestor of B, which is not mapped to $\perp$.

We now describe how to deal with the case, where the target dependency τ is fictitious. To understand the underlying problem, we give an abstract example.

Example 11.14 *We consider the mapping induced by the pattern* $p = /a\langle x_a\rangle/b\langle x_b\rangle/c\langle x_c\rangle$ *and the dependencies* $\tau = (p, x_a \xrightarrow{\emptyset} x_c)$, $\sigma_1 = (p, x_a \xrightarrow{x_c} x_c)$, *and* $\sigma_2 = (p, x_a \xrightarrow{\emptyset} x_b)$. *We note that* σ_1 *is not strictly fictitious, as it only applies to embeddings where no node is mapped to* $\perp$. *It can be easily seen that* $\sigma_1 \not\models \tau$ *and* $\sigma_2 \not\models \tau$, *as the tree* t_1 *in Figure 11.4 is a counter-example to* $\sigma_1 \models \tau$ *and* t_3 *is a counter-example to* $\sigma_2 \models \tau$. *Later, we will see that* $\{\sigma_1, \sigma_2\} \models \tau$.

Starting with $\Sigma = \{\sigma_1\}$ and τ, Algorithm 8 will incorrectly report that $\sigma_1 \models \tau$, as it starts with the initial tree t_2, merges nodes v_3 and v_5 to satisfy σ_1 and recursively merges nodes v_2 and v_4 to restore the tree structure. The resulting tree t_4 satisfies σ_1 and τ and therefore Algorithm 8 erroneously reports that $\sigma_1 \models \tau$.

The intuitive reason for the incorrect behavior is that Algorithm 8 does not consider trees, where the witness-pair for τ involves null values, i.e., τ is treated as a non-fictitious dependency.

It is easy to see that starting with t_1 as initial tree would allow Algorithm 6 to correctly decide that $\sigma_1 \not\models \tau$. However, simply adapting the initial tree will not work, as can be seen by the dependency σ_2. Starting with initial tree t_1 and $\Sigma = \{\sigma_2\}$, Algorithm 6 will merge nodes v_2 and v_4 to satisfy σ_2. The resulting tree t_4 again satisfies τ, which leads to the (again incorrect) result $\sigma_2 \models \tau$. The intuitive reason now is, that with t_1 as initial tree, Algorithm 6 no longer considers trees where the witness-pair for τ does not use null values.

A possible solution would be to run Algorithm 6 starting from both initial trees and output "yes", if and only if both runs output "yes". We note without proof that this solution could be generalized to arbitrary FFDs resulting in at most linearly many initial trees in the depth of p, each with a different number of nodes from p mapped to null in the witness-pair. Instead of this approach, we take a more elegant solution in using an initial tree, where the witness-pair has as many as possible null values and extend Algorithm 6 such that it adds more nodes to the tree when it becomes apparent that the chosen initial tree will result in a final tree satisfying τ.

Therefore, we define the initial tree of a fictitious functional dependency $\tau = (p, Y \xrightarrow{Z} B)$ as follows: Let t_1 and t_2 be copies of p, where t_1 only contains the nodes referenced in Y, Z and B together with their ancestors and t_2 contains only the nodes referenced by Y and Z together with their ancestors. Let again π_1 and π_2 be the canonical embeddings of p in t_1 and t_2, respectively. As before, t_1 and t_2 contain node ids from $\mathcal{V}$ instead of variables from X, all data values in t_1 and t_2 are distinct and wildcards $*$ are replaced by $\#$. The tree t_τ again results by merging the roots of t_1 and t_2 and all pairs $(\pi_1(z), \pi_2(z))$, for which z occurs as a node term in Y and it identifies all pairs of data values $(\pi_1(z).@, \pi_2(z).@)$, for which $z.@$ is a data term in Y. By applying the node merges the embeddings π_1 and π_2 yield two embeddings π_1' and π_2' such that (π_1', π_2') is a witness pair for t_τ and τ.

Furthermore, we add the red parts to Algorithm 9, which take care of adding additional nodes to the tree when necessary, i.e., when a merge occurs that would result in ρ_2 not being a maximal embedding. In this case x (as computed in Line 8) cannot be mapped to $\perp$ any more and the function remove-null is invoked to add a new node for mapping x. We loop using the goto statement in Line 11 as it might be necessary to remove further nulls.[10]

Coming back to Example 11.14, we want to give the chase sequence for $\Sigma = \{\sigma_1, \sigma_2\}$ and τ. Algorithm 10 starts with computing the initial tree t_1 (in Figure 11.4). As σ_2 is not satisfied, the nodes v_2 and v_4 need to be merged. Prior to this merge, remove-null is called in Line 10, resulting in tree t_2. The chase continues with merging v_2 and v_4 resulting in tree t_3.[11] Now σ_1 is violated resulting in a merge of v_3 and v_5 and the final tree t_4. As $t_4 \models \tau$, Algorithm 10 reports that $\{\sigma_1, \sigma_2\} \models \tau$. As we will see in Proposition 11.15, this result is correct.

Proposition 11.15 *For every instance $I = (\Sigma, \tau)$ of* XC-Imp($TP[/, *], FFD$)*, Algorithm 10 terminates and answers "Yes" if and only if $\Sigma \models \tau$.*

Proof. The proof follows a very similar outline to the proof of Proposition 11.7. The basic difference is, that we will consider calls to remove-null in lines 5 and 10 as separate merge steps in our induction.

Let $I = (\Sigma, \tau)$ be an instance of XC-Imp(TP$[/, *]$, FFD) with $\tau = (p_\tau, Y_\tau \xrightarrow{Z_\tau} B_\tau)$. Clearly, if Algorithm 10 terminates and yields a tree t, no constraint from Σ is violated in t. Thus, if the output of Algorithm 10 is "No" (and thus $t \not\models \tau$), t is a counter-example for $\Sigma \models \tau$ and thus, the answer "No" is always correct. The proof that "Yes"-answers are also correct again uses tree homomorphisms as defined in the proof of Proposition 11.7.

Let t_i be the tree after i chase steps where a chase step is either a call to remove-null or a call to merge in Algorithm 9. This differs to the proof of Proposition 11.7, where only calls to merge were considered chase steps, as there where no calls to remove-null.

Claim 11.16 is identical to Claim 11.8, except that it holds for fictitious dependencies and uses the updated definition of a chase step.

[10] We use this algorithm to simplify the correctness-proof. It is possible to directly compute the correct node of p such that one invocation of remove-null suffices.

[11] If remove-null would not have been called, the resulting tree would have been t_4 implying that τ already follows from σ_2.

Claim 11.16 *If there is a counter-example tree t' for $\Sigma \models \tau$ with witness pair $(\rho^1_{t'}, \rho^2_{t'})$ for τ and t', then the tree chase on input (Σ, τ) does not fail and for every chase step i it holds that*

(i) there exist a tree homomorphism $t_i \preceq_{\theta_i} t'$;

(ii) there exist a witness pair (ρ^1_i, ρ^2_i) for $t_i \not\models \tau$; and

(iii) $\theta_i(\rho^j_i(x)) = \rho^j_{t'}(x)$ for $j \in \{1, 2\}$ and all terms x of p_τ with $\rho^j_i(x) \neq \perp$.

Again, the claim immediately yields the correctness of the algorithm.

The proof of Claim 11.16 is by induction on the number of chase steps. We distinguish 3 cases for the induction step depending on the type of the chase step: Calls to remove-null in Line 5, calls to remove-null in Line 10 and calls to merge (in Line 12). We note that we can show these cases in any order.

We want to remember that we can safely assume that for each witness pair (π_1, π_2) considered in this proof it holds that π_1 is a full embedding and the only nodes mapped to $\perp$ in π_2 are on the path from the root to B, where B is the node or data term on the right-hand side of the corresponding dependency.

The induction base for $t_0 = \tau$ can be shown exactly as in the proof of Proposition 11.7. The same holds true for the induction step in the case of calls to merge. It should be noted that in calls to merge, ρ_2 always is a full embedding. This is ensured by the call to remove-null in Line 5 that precedes the call to merge in Algorithm 9. It remains to show the induction step in the case of calls to remove-null.

We first show the case, where remove-null is called in Line 10 of Algorithm 9. Let v be the node added in remove-null and x_τ be the corresponding term in the pattern p_τ. We define θ_{i+1} to map v to $\rho^2_{t'}(x_\tau)$ and to be equal to θ_i on all other nodes. By definition of embeddings, $\rho_{t'}(x_\tau)$ has to be $\perp$ or a child of $\rho_{t'}(\text{parent}(x_\tau)) = \Theta_i(\rho^2_i(\text{parent}(x_\tau)))$. However, $\rho_{t'}(x_\tau)$ cannot be $\perp$, as we can conclude from Equation 11.1 that $\Theta_i(\rho^1_i(\text{parent}(x_\tau))) = \Theta_i(\rho^2_i(\text{parent}(x_\tau)))$. Observe that B is a node term and $\rho^1_i(\text{parent}(x_\tau))$ and $\rho^2_i(\text{parent}(x_\tau))$ are ancestors (in the same level of the tree) of $\chi^1_i(B)$ and $\chi^2_i(B)$. This shows that Θ_{i+1} is a valid tree homomorphism (and therefore (i) is satisfied). We define $\rho^1_{i+1} = \rho^1_i$ and ρ^2_{i+1} to map x_τ to v and to be equal to ρ^2_i on all other variables. It is straightforward to show that $(\rho^1_{i+1}, \rho^2_{i+1})$ is a witness pair for $t_{i+1} \not\models \tau$, satisfying (ii). Especially ρ^2_{i+1} is a maximal partial embedding as ρ^2_i is a maximal partial embedding and all nodes that are mapped to $\perp$ in ρ^2_{i+1} are below x_τ, which is mapped to a leaf of the tree. From the definition of Θ_{i+1} and $(\rho^1_{i+1}, \rho^2_{i+1})$, it follows that (iii) still holds, as the only node added to the image of $(\rho^1_{i+1}, \rho^2_{i+1})$ is mapped accordingly in Θ_{i+1}.

At last we discuss the case where remove-null is called in Line 5 of Algorithm 9. Let $[v, w]$ be the path added to t in remove-null and $[x, y]$ be the corresponding path in the pattern p. Let y_τ be a variable in p_τ with the same ancestor string as y. Such a variable exists, as $\chi_1(y)$ is in the image of ρ^1_i. We define x_τ, such that $[x_\tau, y_\tau]$ has the same length and labels as $[x, y]$. Again, we can conclude from Equation 11.1 that $\rho_{t'}(y_\tau)$ cannot be $\perp$, as $\rho^2_{t'}(\text{parent}(x_\tau)) = \Theta_i(\chi^2_i(\text{parent}(x)))$. We define θ_{i+1} such that it is equal to θ_i for all nodes of t_i and that it maps the nodes $[v, w]$ to $[\rho^2_{t'}(x_\tau), \rho^2_{t'}(y_\tau)]$. Similarly to the last

case, it is easy to verify that θ_{i+1} is a valid tree homomorphism and therefore we can conclude (i). We define $\rho^1_{i+1} = \rho^1_i$ and ρ^2_{i+1} to map the path $[x, y]$ to the path $[v, w]$ and to be equal to ρ^2_i on all other nodes. Again, it is straightforward to show that $(\rho^1_{i+1}, \rho^2_{i+1})$ is a witness pair (showing (ii)) and that Θ_{i+1} is defined in a way, such that (iii) holds. □

Using Proposition 11.15, the following results can be shown in the same way, as we have shown the results from Corollary 11.9, Proposition 11.12, and Proposition 11.13. In particular, the extension of the chase in the presence of sDTDs, which we have shown to work in Proposition 11.10 does not interfere with the addition of null values.

Corollary 11.17

(a) XC-Imp($TP[/, *], FFD$) *can be solved in polynomial time.*

(b) XC-Imp($TP[/, *], FFD$, sDTD) *can be solved in exponential time.*

(c) XC-Imp($TP[/], FFD$, sDTD) *can be solved in polynomial time.*

11.3 Upper Bounds Based on Small Counter Examples

For counterexample based proofs the following two lemmas are useful. The first lemma establishes small counterexample properties for various kinds of constraints when no sDTD is present. The second lemma does the same in the presence of sDTDs. By leaves(t), we denote the set of leaves of a tree t.

Lemma 11.18 *Let $\Sigma \subseteq XC(TP, FFD)$ be a set of constraints and $\tau = (p, Y \xrightarrow{Z} B)$ be a constraint. If there is a tree t with $t \models \Sigma$ and $t \not\models \tau$ then there is a tree t' with*

(1) $t' \models \Sigma$ *and* $t' \not\models \tau$*;*

(2) $|\operatorname{leaves}(t')| \leq 2|\operatorname{leaves}(p)|$*;*

(3a) if all tree patterns are from $TP[/, *]$ then $\operatorname{depth}(t') \leq \operatorname{depth}(p)$*;*

(3b) if all tree patterns are from $TP[/, //]$ then $\operatorname{depth}(t') \leq 8 \operatorname{depth}(p)$*;*

(3c) if all FFDs are XKFDs then $\operatorname{depth}(t') \leq 8m \operatorname{depth}(p)$*, where m is the maximal depth of all patterns in Σ.*

Proof. Let t be a tree with $t \models \Sigma$ and $t \not\models \tau$. Then there is some witness pair (π_1, π_2) for $t \not\models \tau$. Let P be the set of nodes of t to which some pattern node is mapped via π_1 or π_2.

We first describe the construction of a tree t'_1 fulfilling (1) and (2) and, if all patterns are from TP$[/, *]$, also (3a).

Let t'_1 be the tree obtained from t by removing all nodes that are not in P and not ancestors of nodes in P. It is straightforward that $t'_1 \models \Sigma$, $t'_1 \not\models \tau$ and $|\operatorname{leaves}(t'_1)| \leq 2|\operatorname{leaves}(p)|$. Furthermore, if all tree patterns are from TP$[/, *]$, then $\operatorname{depth}(t') \leq \operatorname{depth}(p)$. Thus, t'_1 fulfills (1), (2) and (3a). The construction of t'_1 is not affected if an esDTD has

to be respected, as it does not change the set of labels of the tree (unlike the following two constructions).

If all patterns are from TP[/, //] we can construct another tree t'_2 from t'_1 as follows. Let P' contain all nodes from P and all nodes of t that are lowest common ancestors of at least two nodes of P. Clearly $|P'| \leq 2|P| \leq 4|p|$. To obtain t'_2, we replace in t'_1 all maximal paths of nodes that are not in P' by a path of length 2 whose single intermediate node carries a new label # that does not occur in any pattern of Σ. By construction, $|\operatorname{leaves}(t'_2)| \leq |\operatorname{leaves}(t'_1)| \leq 2|\operatorname{leaves}(p)|$ and $t'_2 \not\models \tau$. On the other hand, if all patterns in Σ are from TP[/, //], every embedding of a pattern in t'_2 is also an embedding in t and therefore $t'_2 \models \Sigma$. This is, because an embedding of a pattern without wildcards can only "bridge" the gaps introduced by the new symbols # with the help of descendant edges. Finally, the depth of t'_2 is at most twice the depth of P' and thus $\operatorname{depth}(t'_2) \leq 8 \operatorname{depth}(p)$. Thus, t'_2 fulfills (1), (2) and (3b).

Let t'_3 be the tree obtained from t'_1 by replacing every maximal path of length $> m$ of nodes that are not in P' by a path of length m in which every node gets a separate new data value and is labeled with a new label # that does not occur in any pattern of Σ. It is easy to see that this transformation does not introduce any violations of any XKFDs from Σ (as the new paths do not match any subpatterns that were not matched before by the replaced path), and thus, t'_3 is a counter-example tree of depth $\leq 8m \operatorname{depth}(p)$. □ □

Lemma 11.19 *Let $I = (\Sigma, \tau, S)$ be a constrain instance with a set $\Sigma \subseteq XC(TP, FFD)$ of constraints, a constraint $\tau = (p, Y \xrightarrow{Z} B)$ and an* sDTD *S. If there is a tree t with $t \models \Sigma$ and $t \not\models \tau$ then there is a tree t' with*

(1) $[t']_D \models S$, $[t']_D \models \Sigma$ *and* $t' \not\models \tau$;

(2) $|\operatorname{leaves}(t')| \leq 2|\operatorname{leaves}(p)|$;

(3a) if p is from TP[/, ∗] then $\operatorname{depth}(t') \leq \operatorname{depth}(p)$;

(3b) if all tree patterns are from TP[/, //] and all FFDs are XKFDs, then $\operatorname{depth}(t')$ *is bounded polynomially in $|I|$;*

Proof. Let t be a tree with $t \models S$, $t \models \Sigma$ and $t \not\models \tau$ witnessed by some pair (π_1, π_2) for $t \not\models \tau$. Let P be the set of nodes of t to which some pattern node is mapped via π_1 or π_2.

We construct a tree t' fulfilling (1), (2) and (3a), exactly as in the proof of Lemma 11.18. We note, that if $t \models S$, then for any tree t' derived from t by removing nodes it holds that $[t']_D \models S$.

Towards (3b), let L be the set of strings v such that there exists a path r in some pattern p, such that r only consists of /-edges and the sequence of labels of r equals v. Clearly $|L|$ is bounded quadratically in $|I|$. We construct the DFA $\mathcal{A}$ that matches patterns in L, i.e., the current state q of the automaton always corresponds to the longest string in L that can be matched on the last read symbols. The size of $\mathcal{A}$ is bounded by the size of L, as L is closed under prefixes.

Let now r be a path in t' such that

- $|r| > 2m + |\mathcal{A}|$, where m is the maximum of the depths of all patterns; and
- r does not contain any node v such that $v \in P$ or v is the lowest common ancestor of two nodes of P.

If no such path exists, the depth of t' is clearly bounded by $|P|(2m + |\mathcal{A}|)$ and thus polynomially bounded.

Let r' be the infix of r without the m topmost and bottom-most nodes. Let u and v be nodes of r' such that $\mathcal{A}$ is in the same state after reading u and v. As $|r'| > |\mathcal{A}|$ such nodes exist.

We now do some pumping by removing all nodes between u and v and the node v, making the (by construction unique) child v' of v a child of u.

The resulting tree t'' still satisfies $t'' \not\models \tau$, as we did not remove any node from P. Furthermore, $[t'']_D$ still satisfies S, as u and v have the same label. At last, we show that t'' still satisfies $[t'']_D \models \Sigma$ by showing that any witness pair (ρ_1, ρ_2) for $\sigma \in \Sigma$ and $[t'']_D$ can be transformed to a witness pair (ρ_1', ρ_2') for σ and $[t']_D$, contradicting $[t']_D \models \sigma$.

Let x_1, x_2 be variables from p and $i \in \{1, 2\}$ be such that x_1 and x_2 are connected by a child edge, $\rho_i(x_1) = u$, and $\rho_i(x_2) = v'$. Let y be the highest ancestor of x_2 that is reachable only using child-edges in p. We change ρ_i such that it embeds $p_y \setminus p_{x_2}$ in the neighborhood of v (in $[t']_D$) just as it is embedded in the neighborhood of u. By construction of $[t']_D$ and $\mathcal{A}$, we know that this is possible, as the affected parts of $[t']_D$ are isomorphic, either because of our pumping or (for nodes not on the main path) because of S.

We iteratively apply the pumping procedure to all long edges to obtain a tree, whose depth is bounded by $|P|(2m + |\mathcal{A}|)$. □ □

By combining Lemmas 11.4, 11.18 and 11.19 we get the following upper bounds.

Theorem 11.20 *The following implication problems are in co*NP.

(a) XC-IMP(*TP*[/, //], *FFD*)

(b) XC-IMP(*TP*, *XKFD*)

(c) XCS-IMP(*TP*[/, ∗], *FFD*, sDTD)

(d) XCS-IMP(*TP*[/, //], *XKFD*, sDTD)

Proof. Let in the following always $I = (\Sigma, \tau)$ or $I = (\Sigma, \tau, S)$ be an instance of the implication problem at hand with $\tau = (p, Y \xrightarrow{Z} B)$ and S an sDTD, in case of (c) and (d). Lemma 11.18 guarantees for the cases (a) and (b), and Lemma 11.19 guarantees for the cases (c) and (d)[12] that, if there is a counter-example tree t to I at all, there is one of depth polynomial in Σ, $|\tau|$ and (if given) $|S|$ and with a number of leaves in $\mathcal{O}(|\tau|)$. This yields immediate NP-algorithms for the complement of each of the three implication problems: guess a tree t that obeys the depth and width bounds of Lemma 11.18 and verify whether it is a counter-example to I using the algorithm of Lemma 11.4. Thus, the coNP upper bound follows in all four cases. □ □

[12] In cases (c) and (d), actually $[t]_D$ is the actual counter-example.

11.4 Polynomial Space Upper Bound Based on Skeletons

In the previous section, we considered counter-example trees of the form $[t]_D$ that could be of exponential size in the implication instance I but whose "backbone" t had only polynomial size. For the remaining PSPACE upper bound for XCS-IMP(TP[/, //, ∗], XKFD, SDTD), we need to use an even more compact representation of counter-examples. We do not only leave out nodes that are enforced by S as in the step from $[t]_D$, but we further leave out nodes in t that are not needed to verify that Σ is satisfied but τ not.

To this end, we use trees in which some paths of the tree are represented by path edges that do not contain any information about node labels along that path. More precisely, a *skeleton tree* s is just an XML tree with two kinds of edges, *child edges* and *path edges*, where path edges represent vertical paths of length at least two, similarly as wildcard symbols represent labels in a pattern. Path edges are additionally marked by the label of the highest node of the path. The semantics of pattern-based X2R-constraints with respect to skeleton trees is just defined as for normal trees, with the understanding that path edges match descendant edges of patterns, but do not match child edges.

A set U of nodes of a tree t is *suitable* if the following two conditions hold.

(1) U contains the root of t, all leaves of t and all inner nodes with more than one child, and

(2) if U contains nodes u and v of t of distance 2 then it also contains their intermediate node.[13]

By $s_U(t)$ we denote the skeleton tree that results from t by replacing all (maximal) paths of nodes that are not in U by path edges (and marking all path edges by the label of the highest node of the path). We call a set U Σ-*preserving* if for every $\sigma \in \Sigma$ it holds that $[t]_D \models \sigma$ if and only if $[s_U(t)]_D \models \sigma$. Here, we denote by $[s]_D$ a skeleton tree that is induced by S from a skeleton tree s in a canonical way, similarly[14] as $[t]_D$ is induced from t. In particular, $[s]_D$ has the same path edges as s.

We show that for each instance $I = (\Sigma, \tau, S)$ of XCS-IMP(TP, XKFD, SDTD) and each π-diverse tree t of I with a witness pair π for $t \not\models \tau$, there is a suitable set U of t of polynomial size in $|I|$ that is Σ-preserving and such that $[s_U(t)]_D \not\models \tau$. Furthermore, it can be tested in polynomial space whether a given skeleton tree s is of the form $s_U(t)$ for some tree t $[t]_D \models S$ and a suitable, Σ-preserving set U. [15]

Let, in the following, $I = (\Sigma, \tau, S)$ denote a fixed instance of XCS-IMP(TP, XKFD, SDTD) with $\tau = (p_\tau, Y_\tau \to B_\tau)$, let t be a π-diverse tree and $\pi = (\pi_1, \pi_2)$ a witness-pair showing that $[t]_D \not\models \tau$. Let m be the maximal depth of all patterns in Σ and τ. With $\sigma = (p, Y \to B)$ we always denote some constraint from Σ.

In order to show that sets U of polynomial size suffice for our purposes, we prove that if there are witness pairs for constraints in Σ then there are such witness pairs of a particularly simple form. To this end, we need to introduce some additional notation.

[13] In other words: all paths missing in U consist of at least two nodes.

[14] Since the label of a path edge below a node u indicates the label of the child of u on that path, the extension does not need to add another child of u with that label.

[15] In the final algorithm, U has to satisfy some additional conditions.

term	symbol	definition
cluster	c	connected component of a tree pattern when removing //-edges
connector node		node in a cluster with outgoing descendant edges
special nodes		the set of nodes used in a (fixed) witness pair for a target dependency τ closed under least common ancestors
skeleton (tree)	$s_U(t)$	condensed representation of a tree t containing only the nodes from U, uses path edges to represent the skipped nodes

Table 11.5: Notation used in Section 11.4.

For two nodes v and w of a tree t we write

- $v \to w$, if v is an ancestor of w;
- $v \xrightarrow{n} w$, $v \xrightarrow{\leq n} w$, $v \xrightarrow{\geq n} w$ if $v \to w$ and the distance between v and w is n, $\leq n$, $\geq n$ for $n \in \mathbb{N}$;
- $v \leftrightarrow w$, if $v = w$ or $v \to w$ or $w \to v$;
- $v \nleftrightarrow w$, if $v \leftrightarrow w$ does not hold.

A *cluster* of a pattern $p \in \mathrm{TP}$ is a maximal sub-pattern in which all edges are child edges.[16] The *cluster tree* $\mathrm{CT}(p)$ of a pattern p has as nodes the clusters of p and as edges the descendant edges of p. We use the usual tree notation, both for the cluster tree and for individual clusters, e.g., $\mathrm{root}(\mathrm{CT}(p))$ denotes the cluster containing $\mathrm{root}(p)$ and for any cluster c, $\mathrm{root}(c)$ denotes the topmost node of the cluster. We note, that the depth of each cluster is bounded by m.

The *connector nodes* of a cluster c are the nodes of c with outgoing descendant edges (in p). We say x is a connector node *towards* c', if x is a connector node and $\mathrm{root}(c')$ is a descendant of x.

Let c be a cluster from p. With p_c we denote the sub-pattern $p_{\mathrm{root}(c)}$, i.e., the sub-pattern consisting of the cluster c and all its descendant clusters. We say that a node of y of p is *data-sensitive* if $y.@ \in Y$, and a cluster c is *data-sensitive*, if p_c contains a data-sensitive node y.

We write $\rho(c) \to \rho'(c')$ if $\rho(\mathrm{root}(c)) \to \rho'(\mathrm{root}(c'))$ and likewise $\rho(c) \leftrightarrow \rho'(c')$, $\rho(c) \nleftrightarrow \rho'(c')$ and $\rho \xrightarrow{n,\leq n,\geq n} \rho'$.

A witness pair (ρ_1, ρ_2) for t and σ is *normal*, if there exists at most one cluster c such that

(n1) $\rho_1(c) \leftrightarrow \rho_2(c)$ and $\rho_1(c) \neq \rho_2(c)$,

and if $\rho_1(c) \to \rho_2(c)$ for that cluster.

A cluster c is called *ρ-critical*, if it satisfies (n1) or simply *critical*, when ρ is clear from the context. By definition, in a normal witness pair, there can be at most one critical cluster.

We will use the following lemma to bound the search space for witness pairs.

[16] Stated otherwise, a cluster is a connected component of p after removing all descendant edges.

Lemma 11.21 *Let t be a tree such that $t \not\models \sigma$. Then there exists a normal witness pair for σ.*

Proof. To establish the lemma we first show that for every witness pair with the minimal number of critical clusters, each cluster additionally satisfies:

(n2) $B \in p_c$; and

(n3) if $B \notin c$, then $\rho_1(x_B^c) \nleftrightarrow \rho_2(x_B^c)$.

Let thus $\rho = (\rho_1, \rho_2)$ be a witness pair for an XML tree t and some constraint σ with the minimal number of critical clusters. Towards a contradiction, we assume that some critical cluster c does not satisfy both (n2) and (n3). Without loss of generality, we assume that $\rho_1(c) \to \rho_2(c)$.

$t \not\models \sigma$ and every witness pair $\rho = (\rho_1, \rho_2)$ for t and σ is not normal. Let (ρ_1, ρ_2) be any witness pair with the minimal number of ρ-critical clusters and let c be a ρ-critical cluster.

If c does not satisfy (n2), we define ρ_1' to be equal to ρ_2 for all variables in p_c and equal to ρ_1 on all other variables of p. It is easy to see that (ρ_1', ρ_2) is a witness pair for t and σ, as $\rho_1'(\text{root}(c))$ is a descendant of $\rho_1(\text{root}(c))$ and $B \notin p_c$. Furthermore, c is not critical with respect to (ρ_1', ρ_2) and thus (ρ_1', ρ_2) contains fewer ρ-critical clusters than (ρ_1, ρ_2) which is a contradiction to our assumption.

The other case is that c satisfies (n2) but not (n3). That is, $B \notin c$ but $\rho_1(x_B^c) \leftrightarrow \rho_2(x_B^c)$. Since $\rho_1(c) \to \rho_2(c)$ we have $\rho_1(x_B^c) \to \rho_2(x_B^c)$. Let c' be the child cluster of c with $B \in p_{c'}$. Let ρ_2' be defined as ρ_2 for $p_{c'}$ and as ρ_1 for all other clusters. Again, it is easy to verify that (ρ_1, ρ_2') is a witness pair for t and σ with fewer ρ-critical clusters than (ρ_1, ρ_2).

We can thus assume in the rest of the proof that all critical clusters satisfy (n2) and (n3) and therefore all ρ-critical clusters are on the path from the root to B in the cluster tree. Let c_1 and c_2 be the topmost two ρ-critical clusters with c_1 being an ancestor of c_2. We assume without loss of generality that $\rho_1(c_2) \to \rho_2(c_2)$. We define ρ_2' such that it embeds all clusters from p_{c_2} as in ρ_2 and all other clusters as in ρ_1. Again it is easy to verify that (ρ_1, ρ_2') is a witness pair for σ and that t has fewer ρ-critical clusters than (ρ_1, ρ_2). Observe that B is embedded differently in both embeddings, as it is contained in p_{c_2}. □

In the remainder of this section, we will only consider normal witness pairs and therefore usually omit the attribute "normal".

In the following we will reason about partial witness pairs for a constraint σ, that are induced by one cluster c of the pattern p underlying σ. Obviously, for a witness pair $\rho = (\rho_1, \rho_2)$ and each cluster c of p, one of the following statements holds.

(1) $\rho_1(\text{root}(c)) = \rho_2(\text{root}(c))$;

(2) $\rho_1(c) \nleftrightarrow \rho_2(c)$;

(3) $\rho_1(\text{root}(c)) \neq \rho_2(\text{root}(c))$ and $\rho_1(c) \leftrightarrow \rho_2(c)$;

It follows immediately, from Lemma 11.21 that, if ρ is normal, statement (3) can hold for at most one cluster c, the ρ-critical cluster.

For an embedding ρ of a pattern p into a tree t and a cluster c of p, the *c-embedding* ρ^c is just the restriction of ρ to p_c. We refer to c as the *top cluster* of ρ^c.

For each witness pair $\rho = (\rho_1, \rho_2)$ and each cluster c of the underlying pattern p, we define the *c-witness pair*[17] $\rho^c = (\rho_1^c, \rho_2^c)$. We note that a c-witness pair is just a z-witness pair for $z = \text{root}(c)$. Therefore, if $z = \text{root}(c)$, we refer by c-witness pair also to z-witness pairs, even if they can not be extended to full witness pairs.

We say that a c-witness pair is of type (k) if statement (k) holds for c. By definition, each (full) witness pair is of type (1).

To achieve our goal to guarantee the existence of a suitable set U of polynomial size, we basically consider only witness pairs which embed p in a lowest possible way into t. However, the details require some care and the following definitions of top-minimal c-embeddings, top-minimal c-witness pairs and minimal witness pairs are a bit more complicated than one might expect.

We say a c-embedding is *safe*, if each data-sensitive node is embedded on a special node. For two safe embeddings ρ_1 and ρ_2, we write $\rho_1 \sim \rho_2$ if $\rho_1(y) \sim \rho_2(y)$ for each data-sensitive node. Clearly, the two c-embeddings of every type (2) c-witness pair are safe.

We call a c-embedding ρ *top-minimal* if

- there is no c-embedding ρ' with $\rho(c) \to \rho'(c)$; or
- c is safe and there is no c-embedding ρ' with $\rho(c) \to \rho'(c)$ and $\rho \sim \rho'$.

If the first condition holds, we say that ρ is *perfectly top-minimal*.

We call a c-witness pair $\rho = (\rho_1, \rho_2)$ *top-minimal* if one of the following conditions holds.

- ρ is of type (1) and ρ_1 and ρ_2 are perfectly top-minimal.
- ρ is of type (2) and ρ_1 and ρ_2 are top-minimal.
- ρ is of type (3) and there is no type (3) c-witness pair (ρ_1', ρ_2') with
  - $\rho_1(c) \to \rho_1'(c)$ or
  - $\rho_1(\text{root}(c)) = \rho_1'(\text{root}(c))$ and $\rho_2(c) \to \rho_2'(c)$.

We call a c-embedding ρ *minimal* if for each cluster c' of p_c, $\rho^{c'}$ is top-minimal. Likewise, we call a c-witness pair ρ *minimal* if for each cluster c' of p_c,$\rho^{c'}$ is top-minimal. A witness pair is called *minimal* if ρ^c is top-minimal for each cluster c that is not the root cluster. We can now improve Lemma 11.21.

We write $\rho' \leq \rho$ for two c-embeddings ρ, ρ' if $\rho'(\text{root}(c)) = \rho(\text{root}(c))$ or $\rho'(c) \to \rho(c)$, and $\rho < \rho'$ if $\rho(c) \to \rho(c)$. We write $\rho' \leq \rho$, for two c-witness pairs $\rho = (\rho_1, \rho_2)$ and $\rho' = (\rho_1', \rho_2')$ if $\rho_1' \leq \rho_1$ and $\rho_2' \leq \rho_2$, and $\rho' < \rho$ if one of these inequalities is strict.

[17] We note that an induced witness pair is not necessarily a witness pair but rather a sub-witness pair.

Lemma 11.22 *(a) For each c-embedding ρ there is a minimal c-embedding ρ' such that $\rho' \leq \rho$ and $\rho' \sim \rho$.*

(b) For each c-witness pair $\rho =$ there is a minimal c-witness pair ρ' such that $\rho' \leq \rho$.

(c) If $[t]_D \not\models \sigma$, then $[t]_D$ has a minimal normal witness pair.

Proof. We prove all three statement simultaneously by induction on the depth of the cluster tree of pattern p or p_c, respectively.

We first observe that from the definition it follows immediately, that for each c-embedding ρ there exists a top-minimal c-embedding ρ' with $\rho' \leq \rho$ and $\rho' \sim \rho$. Likewise, for every normal c-witness pair ρ, there is a top-minimal c-witness pair ρ' with $\rho' \leq \rho$. This yields the base case of the induction.

For the inductive step for (a), let ρ be an arbitrary normal c-embedding. Again, there must be a top-minimal c-embedding $\rho' \leq \rho$ and $\rho' \sim \rho$. By induction, for every child cluster c' of c, the c'-embedding $\rho^{c'}$ can be replaced by a minimal c'-embedding $\rho''_{c'}$ with $\rho''_{c'} \leq \rho^{c'}$ and these can be combined with $\rho'(c)$ to a minimal c-embedding. For (b) and (c) the argumentation is almost identical. □

We call a node $u \in t$ Σ-*useful* if there is a cluster c and a minimal c-witness pair $\rho = (\rho_1, \rho_2)$ or a minimal c-embedding ρ_1 in $[t]_D$ such that $u = \rho_1(\text{root}(c))$ or $u = \rho_2(\text{root}(c))$.

We need to establish two results about useful nodes: first, that there is always a counter-example for which the set U of useful nodes is of polynomial size (which allows us to guess a polynomial-size skeleton tree $s_U(t)$) and, second, that it is possible to test in polynomial space whether for a skeleton tree s, there is a tree t and a set U containing all useful nodes of t such that $s = s_U(t)$. For the latter, we will actually restrict the set of useful nodes a bit further, below.

Lemma 11.23 *If $\Sigma \not\models_S \tau$ for a pattern-based instance $I = (\Sigma, \tau, S)$ then there is a tree t with polynomially many useful nodes in $|I|$ such that $[t]_D$ is a counter example for I.*

Proof. By Lemma 11.18, we can restrict to trees with only linearly many leaves, in the size of I.

By definition of minimal c-witness pairs, for each path from the root of t to a leave, there is at most one useful node from c-witness pairs of type 1 and type 3. Thus, the number of useful nodes for top-minimal c-witness pairs of type 1 and type 3 is linear in the number of leaves of t and therefore polynomial in $|I|$. As useful nodes for minimal c-witness pair of type 2 are subsumed by useful nodes for minimal c-embeddings, it only remains to establish a bound on the number of nodes induced by minimal c-embeddings. Therefore, for each cluster c, we let U_c denote the set of nodes v, such that $\rho(\text{root}(c)) = v$ for some minimal c-embedding ρ.

Let now c be a fixed cluster. For each child cluster c' of c, we denote the distance between root(c) and the connector node towards c' in c by $d_{c'}$.

We show that for each node $u \in U_c$ one of the following is true:

(a) $u \xrightarrow{\leq m} v$ for some special node v;

(b) u is a lowest node with $u \xrightarrow{>m} v$ for some special node v; or

(c) there exists a data-sensitive cluster $c' \in \text{childs}(c)$ and a node $w \in U_{c'}$ such that $u \xrightarrow{>d_{c'}} w$, but for every node $v \in U_c$ with $u \to v$ it does *not* hold $v \xrightarrow{>d_{c'}} w$.

Towards a contradiction, we assume that $u \in U_c$ and (a-c) are not satisfied. Let ρ be a minimal c-embedding with $\rho(\text{root}(c)) = u$ and let $v \in U_c$ be the next lower node (below u). Since (a-b) do not hold for u, this node is unique, as t can not branch between u and a next lower node in U_c.

Since ρ is minimal and therefore $\text{root}(c')$ is useful, and since (c) does not hold for u, we can conclude that for every data-sensitive cluster $c' \in \text{childs}(c)$, there is a node $v_{c'} \in U_c$ with $v = v_{c'}$ or $v \to v_{c'}$ and $v_{c'} \xrightarrow{>d_{c'}} \rho(\text{root}(c'))$.

Since (b) does not hold for u, we know that $v \xrightarrow{>m} w$ for each special node w. Therefore, and by $v \in U_c$, there exists a c-embedding ρ'' with $\rho''(\text{root}(c)) = v$ such that all connector nodes towards data-sensitive clusters are embedded on the same path of t. Therefore, and because c cannot contain any data-sensitive nodes, by

$$\rho'(c') = \begin{cases} \rho''(c') & \text{if } c' = c \text{ or } c' \text{ is data-insensitive} \\ \rho(c') & \text{if } c' \text{ is data-sensitive} \end{cases}$$

a c-embedding is defined with $\rho' < \rho$, contradicting the minimality of ρ. Thus, every node in U_c must fulfil one of (a-c).

It now only remains to show that, for each c, U_c only contains polynomially many nodes. Indeed, we show that that

$$|U_c| \leq (m+1)A + \sum_{c' \in \text{childs}(c)} |U_{c'}| \,,$$

where A is the number of special nodes, mA accounts for all ancestors of distance up to m from special nodes, and the remainder of the sum accounts for nodes due to condition (c).

By an easy induction over p, we can conclude that $|U_c| \leq (m+1)A \cdot |p_c|$ for each cluster c. □

The following lemma enables us to decide XCS-Imp(TP, XKFD, sDTD) by guessing a skeleton tree and verifying that it conforms to Σ by guessing paths for each path edge, separately.

Lemma 11.24 *Let σ be a constraint with pattern p. Let t be an XML-tree and U be a suitable set of nodes of t that contains all nodes of t with distance up to m to special nodes.*

- *If there is a minimal c-witness pair $\rho = (\rho_1, \rho_2)$ for some cluster c of p, such that $\rho_i(c)$ is not in $[U]_D$ for some $i \in \{1, 2\}$, then*

  - *there exists a minimal c'-witness pair $\rho' = (\rho'_1, \rho'_2)$ and $i \in \{1,2\}$ such that $\rho'_i(c')$ is not in $[U]_D$, but otherwise the range of ρ' is in $[U]_D$;*
  - *there exists a minimal c'-witness pair $\rho' = (\rho'_1, \rho'_2)$ such that $\rho_1(c) \xrightarrow{\leq m} \rho_2(c)$ and $\rho_j(p_c \setminus c) \subseteq [U]_D$ for $j \in \{1,2\}$; or*
  - *there exists a minimal c'-embedding ρ' such that $\rho'(c')$ is not in $[U]_D$, but otherwise the range of ρ' is in $[U]_D$.*

- *If there is a minimal c-embedding ρ for some cluster c of p, such that $\rho(c)$ is not in $[U]_D$, then there exists a minimal c'-embedding ρ' such that $\rho'(c')$ is not in $[U]_D$, but otherwise the range of ρ' is in $[U]_D$.*

Proof. We first show the statement for embeddings. Let ρ be a minimal c-embedding such that $\rho(c)$ is not in $[U]_D$. We let c' be a lowest cluster, such that the range of $\rho^{c'}$ is not in $[U]_D$. Is is easy to verify that $\rho^{c'}$ satisfies the condition.

Let now $\rho = (\rho_1, \rho_2)$ be a minimal witness pair. Let c' be a lowest cluster such that $\rho_i(c')$ is not in U. Restriction to $\rho^{c'}$ again gives us, that all clusters below c' are embedded in $[U]_D$.

Towards the lemma statement, we distinguish four cases:

- $\rho_1(c') \xrightarrow{\leq m} v$ for some special node v: By assumption on U, $\rho_1(c)$ is embedded on $[U]_D$.
- $\rho_1(c') \xrightarrow{\leq m} \rho_2(c')$: nothing to show
- $\rho_1(c') \xrightarrow{>m} v$ for every special node v and $\rho_1(c') \xrightarrow{>m} \rho_2(c')$: In this case, all connector nodes in c' towards data-sensitive clusters are embedded on t and above $\rho_2(c')$ by ρ_1. Therefore, $\rho_2^{c'}$ is a minimal c'-embedding. We note that all nodes $y \in p_{c'}$ with $y.@ \in Y$ are embedded identically in ρ_1 and ρ_2.
- $\rho_1(c') \nleftrightarrow \rho_2(c')$: the embedding ρ_i^c satisfies the lemma □

We now define the extended skeleton tree by defining a suitable set of nodes, that not only includes the special nodes (and their neighbourhood), but also the useful nodes.

The set $U(I, \pi, t)$ contains

- the root;
- all π-special nodes of t;
- all useful nodes;
- all nodes of distance m of the nodes specified above;
- for each label a, all lowest nodes with label a.

The extended skeleton tree is the tree $s_{U(I,\pi,t)}(t)$.

We can conclude from Lemma 11.23, that the size of $U(I, \pi, t)$ and thus the size of $s_{U(I,\pi,t)}(t)$ is polynomial in $|I|$ and $|\operatorname{leaves}(t)|$.

Theorem 11.25 XCS-IMP($TP, XKFD,$ sDTD) *is in* PSPACE.

Proof. We provide a nondeterministic polynomial space algorithm for the complement of XCS-IMP(TP, XKFD, sDTD). The algorithm works as follows:

1. Guess a skeleton s such that $s = s_U(t)$ for some tree t and a suitable set U of nodes of t.
2. If $s \models \tau$ reject.
3. If $[s]_D \not\models \Sigma$ reject.
4. For each path edge e of s, guess a path r_e bottom-up, remembering always the m most recent labels.
   - If the labels do not conform to S then reject.
   - If for some label a, some lowest node with label a is not in U then reject.
   - If there is some node v on r_e such that there exists a minimal c-witness pair (ρ_1, ρ_2) or a minimal c-embedding ρ_1 such that $\rho_1(c) = v$ or $\rho_2(c) = v$, but otherwise the range of ρ_1, ρ_2 is included in $[U]_D$ then reject.
5. accept

We first prove the correctness of the algorithm. Afterwards, we show that the algorithm runs in polynomial space and especially, that the condition in the last if-statement of step 4 can be checked in polynomial space.

If $\Sigma \not\models_S \tau$, then there exists a tree t such that there is a π-diverse counter-example to $\Sigma \models_S \tau$ for some witness-pair $\pi = (\pi_1, \pi_2)$. The algorithm can guess $s = s_{U(I,\pi,t)}(t)$ in step 1. Furthermore it can guess for each path edge e of s the label sequence of the corresponding path in t. It is not possible that $[s']_D \not\models \Sigma$, as $[t]_D \models \Sigma$, and s contains a subset of the nodes from t. As t is a counter example, the label sequences have to be consistent with S and π witnesses that $t \not\models \tau$. Furthermore, $s_{I,\pi}$ contains all necessary nodes. Therefore, the algorithm accepts.

On the other hand, if $\Sigma \models_S \tau$, there can be no tree t such that $[t]_D \models \Sigma$, $t \not\models \tau$ and $[t]_D$ is valid wrt. S. Assume in contradiction that the algorithm accepts. Then let t be the tree guessed by the algorithm, i.e., t contains all nodes from s and for each path edge e of s it contains a sequence of nodes labeled as guessed by the algorithm. As the algorithm accepts, we know that $s \not\models \tau$ and t is consistent with S. Furthermore, there can be no c-witness pair (ρ_1, ρ_2), such that $\rho_i(c))$ is not in $[U]_D$, but otherwise the range of ρ_i is included in $[U]_D$. We note, that if there is a node v on some path edge with some minimal c-embedding ρ, such that $\rho \subseteq [v]_D$, but $\rho(\text{root}(c)) \neq v$, then there is an isomorphic minimal c-embedding in $[w]_D$, where w is some lowest node with label $\text{lab}(v)$.

Using the Lemmas 11.22 and 11.24 we can conclude that $\Sigma \not\models_S \tau$, which is a contradiction to our assumption.

As s is of polynomial size, Step 1 is doable in nondeterministic polynomial time. By Lemma 11.4, Steps 2 and 3 can be done in polynomial time. For Step 4, we only need to remember the last m nodes V_{last} of the path and then check whether there is some c-witness pair or c-embedding in U_{last} using the algorithm of Lemma 11.4. □

11.5 Lower Bounds by Reductions from 3SAT

The coNP lower bounds in the following proposition are all by reduction from SAT to the complement of the respective implication problem.

Proposition 11.26 *The following implication problems are co*NP*-hard.*

(a) XCS-IMP($TP[/,*], XKFD, \text{esDTD}$)

(b) XC-IMP($TP[/,//], XKFD$)

Proof. Both proofs are by reductions from 3-SAT to the complement of the implication problem. The algorithmic problem 3-SAT asks whether a given propositional formula in 3-CNF is satisfiable. A propositional formula in 3-CNF is a conjunction $\varphi = C_1 \wedge \cdots \wedge C_m$ of clauses, over some variables $y_1, \ldots, y_n$, where each clause $C_i = \ell_{i1} \vee \ell_{i2} \vee \ell_{i3}$ is a disjunction of three literals.

Let a 3-CNF formula $\varphi = C_1 \wedge \cdots \wedge C_m$ with variables $y_1, \ldots, y_n$ and clauses of the form $C_i = \ell_{i1} \vee \ell_{i2} \vee \ell_{i3}$ be given. An implication instance (D, Σ, τ) for the reduction for (a) is constructed from φ as follows. The idea for the reduction is to associate truth assignments θ with 0-1-labeled paths such that $\theta(y_i) = 1$ if and only if the i-th symbol is 1. Thus, first of all, the esDTD D enforces the alphabet $\{0, 1\}$.

For every clause C_i, we add a XKFD σ_i with pattern p_i to Σ that states that the last node of a path of length n is non-branching if the truth assignment of that path *fails* to satisfy C_i. That is, if there is a path of length n that does not match any pattern p_i (and thus its corresponding truth assignment satisfies all constraints) then a counterexample to $\Sigma \models \tau$ can be constructed by branching at its n-th node. The overall effect is that τ is implied by Σ if and only if there is no satisfying truth assignment for φ.

We now describe the construction in more detail.

The target dependency τ is defined as the XKFD

$$\tau =_{\text{def}} (/*/*/\ldots/*\langle x\rangle/*\langle y\rangle, x \rightarrow y)$$

with $n+1$ consecutive $*$ positions, stating that a node at depth n can have only one child node. For every i, let σ_i be the XKFD

$$\sigma_i = (/\alpha_{ij}/\ldots/\alpha_{in}\langle x\rangle/*\langle y\rangle, x \rightarrow y),$$

where α_{ij} is 0 if y_j occurs in C_i, 1 if $\neg y_j$ occurs in C_i and $*$, otherwise.

The reduction can be carried out in polynomial time. It remains to prove that φ is satisfiable, if and only if $\Sigma \not\models_D \tau$.

(if): Let us assume $\Sigma \not\models_D \tau$. By the proof of Lemma 11.18 and as D allows that 0-labeled and 1-labeled nodes can be leaves, there is a tree $t = (V, E, \text{lab}, \text{dv}, <_C)$ with $V = \{r, v_1, \ldots, v_n, w_1, w_2\}$ and $E = \{(r, v_1), (v_n, w_1), (v_n, w_2)\} \cup \{(v_i, v_{i+1}) \mid i \in \{1, \ldots, n-1\}\}$ such that $t \models D$ and $t \models \Sigma$. By definition of t it holds that $t \not\models \tau$.

Thanks to $t \models D$, all nodes in t are labeled 0 or 1. Let θ be the truth assignment induced from t, that is, $\theta(y_j)$ is the label of v_j, for every j.

Towards a contradiction, let us assume that, for some $i \leq m$, $\theta \not\models C_i$. Then the pattern p_i of σ_i matches the two paths of t of length $n+1$ and thus σ_i does not hold. This is a contradiction from which we can conclude that $\theta \models \varphi$ and that, in particular, φ is satisfiable.

(only if): Let us assume that φ is satisfiable via some truth assignment θ. Let t be the tree with the same set V of vertices and set E of edges as the tree in the (if)-part and let node v_j carry label $\theta(y_j)$, for every j. Let w_1, w_2 be labeled with 1, for concreteness. As v_n has two children, $t \not\models \tau$. On the other hand, as $\theta \models C_i$, for every i, none of the patterns p_i of the constraints in Σ matches t and thus $t \models \Sigma$, as desired. Therefore, t is a counter-example for $\Sigma \models_D \tau$.

The proof of (b) is also by a reduction from 3-SAT to the complement of the implication problem and follows a similar approach, but the encoding of truth assignments is different: For every $i \leq n$, there are two symbols, a_i and b_i, both of which have to occur in any path matching τ. The corresponding truth assignment θ is defined by $\theta(y_i) = 1$ if a_i is a descendant of b_i and $\theta(y_i) = 0$ otherwise.

We now describe the construction in more detail. Let again $\varphi = C_1 \wedge \cdots \wedge C_m$ be a 3-SAT formula with variables $y_1, \ldots, y_n$ and clauses of the form $C_i = \ell_{i1} \vee \ell_{i2} \vee \ell_{i3}$. An implication instance (D, Σ, τ) is constructed from φ as follows.

We let

$$\tau =_{\text{def}} ([//a_1// \ldots //a_n/a]//b_1// \ldots //b_n//a\langle x\rangle/b\langle y\rangle, x{\rightarrow}y)$$

and, for every i,

$$\sigma_i =_{\text{def}} [//\alpha_1//\beta_1//a][//\alpha_2//\beta_2//a]//\alpha_3//\beta_3//a\langle x\rangle/b\langle y\rangle, x{\rightarrow}y),$$

where $\alpha_k = a_j$ and $\beta_k = b_j$ if $\ell_{ik} = y_j$, and $\alpha_k = b_j$ and $\beta_k = a_j$ if $\ell_{ik} = \neg y_j$. The constraint set Σ contains all constraints σ_i and the additional constraint $\sigma_a = (//a\langle x\rangle, \emptyset{\rightarrow}x)$. We note that σ_a expresses that at most one a-labeled node can occur in the tree.

Again, the reduction can be easily carried out in polynomial time and it thus only remains to prove that φ is satisfiable, if and only if $\Sigma \not\models \tau$.

(if): Let us assume $\Sigma \not\models \tau$ and let t be a tree with $t \models \Sigma$ and $t \not\models \tau$. As $t \models \sigma_a$, there can be at most one a-labeled node in t. On the other hand, as $t \not\models \tau$, the pattern of τ needs to match t and therefore, there must be a path ϱ from root(t) to a unique a-labeled node v_a, which contains nodes with the labels $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$. As $t \not\models \tau$, v_a has two b-labeled children.

We define a truth assignment θ as follows: $\theta(y_i) =_{\text{def}} 1$ if there is a b_i-labeled node with an a_i-labeled descendant node in ϱ, and $\theta(y_i) =_{\text{def}} 0$, otherwise.

Towards a contradiction, let us assume that, for some $i \leq m$, $\theta \not\models C_i$. Then the pattern p_i of σ_i matches the two paths of t through v_a to a b-labeled child of v_a and thus σ_i does not hold. Again, this is a contradiction from which we can conclude that $\theta \models \varphi$ and that φ is satisfiable.

(only if): Let us assume that φ is satisfiable via some truth assignment θ. We construct a tree t as follows: it consists of a path $r, v_0, \ldots, v_{2n-1}, v$ and two further b-labeled nodes w_1 and w_2 that are children of the a-labeled node v. For every $i \leq n$, if $\theta(y_i) = 1$ then v_{2i} is a_i-labeled and v_{2i+1} is b_i-labeled, otherwise v_{2i} is b_i-labeled and v_{2i+1} is a_i-labeled.

Clearly, $t \not\models \tau$ but $t \models \sigma_a$. As $\theta \models C_i$, for every i, none of the patterns p_i of the constraints in Σ finds a match in t and thus $t \models \Sigma$, as desired. Therefore, t is a counter-example for $\Sigma \models \tau$. □

11.6 Lower Bounds by Reductions from Tiling Problems

The two remaining lower bounds are both by reduction from tiling problems. In both proofs, tilings are encoded by unary trees, that is, trees without branching. That only unary trees have to be considered can be enforced by putting the constraint $\sigma_{\text{unary}} = (//*\langle x\rangle/*\langle y\rangle, x{\to}y)$ into Σ.

We also use constraints that (essentially) forbid certain patterns in trees. To this end, we define, for each pattern p, the constraint $\sigma_\neg(p) =_{\text{def}} ([p]//*\langle x\rangle, \emptyset{\to}x)$, that is violated in all trees that match p and contain at least two nodes.

Theorem 11.27

(a) XCS-IMP(*TP*, *XKFD*, ESDTD) *is* PSPACE*-hard.*

(b) XC-IMP(*TP*, *FD*, ESDTD) *is undecidable.*

Proof. We note that the proof of (a) does not refer to data values at all, instead it only uses structural constraints. In the proof of (b), data values are used to uniquely identify positions in a tiling of arbitrary size.

We start with (a). As already announced, the reduction is from the PSPACE-complete corridor tiling problem to the complement of XCS-IMP(TP, XKFD, ESDTD) (which is sufficient as PSPACE is closed under complementation).

Let thus $\mathcal{U} = (U, H, V, u_0, u_F, 1^n)$ be a corridor tiling instance.

For simplicity, we will represent unary trees that encode tilings by strings (without data values). A string encoding of a valid tiling (with at least two rows) will thus match the pattern $u_0U^{n-1}\$(U^n\$)^*U^{n-1}u_F$. For a more concise notation, we write the child axis in patterns just as concatenation, e.g., we write $//a*c//d*$ for the pattern $//a/*/c//d/*$.

Each row λ_i of a corridor tiling is represented by the string $s_i =_{\text{def}} \lambda(i,1)\cdots\lambda(i,n)$ and the whole tiling is represented by the string $s_1\$s_2\$\cdots\$s_m$, where $\$ \notin U$ and m is the height of the tiling.

The idea behind our construction is that D enforces that only valid tiles (and a row separator) are used as tree labels, τ enforces that the tiling starts with the initial tile and ends with the final tile and Σ enforces that the tiling encoded by the tree obeys the constraints.

The ESDTD D allows only labels from $U \cup \{\$\}$. The target constraint is

$$\tau = \sigma_\neg(/u_0/\,*^{n-1}\,\$//u_F\$),$$

stating that a tree does not have a path starting with u_0, ending with $u_F\$$ and having \$ at position $n+1$.

The constraint set Σ contains the following constraints whose intention is to ensure that a string is an encoding of a (not necessarily valid) tiling.

- σ_{unary} (the tree is unary);
- $\sigma_\neg(//\$ *^n u)$, for every $u \in U$ (the $(n+1)$st position after a \$ can only be a \$);
- $\sigma_\neg(//\$ *^i \$)$, for every $i < n$ (no two \$-labeled positions within distance i).

It is not hard to see that every string conforming to these constraints and matching the pattern of τ has a prefix that encodes a (not necessarily valid) tiling of width n whose first row begins with u_0 and whose last row ends with u_F.

The remaining constraints of Σ deal with the validity of the tiling. For each pair (u,v) of tiles such that $(u,v) \notin H$, Σ contains a constraint $\sigma_{H,u,v} =_{\text{def}} \sigma_\neg(uv)$ that is violated by every tree with a u-labeled node that has a v-labeled child. Thus, every encoding of a tiling that does not respect H violates some $\sigma_{H,u,v}$. For each pair (u,v) of tiles such that $(u,v) \notin V$, Σ contains a constraint $\sigma_{V,u,v} =_{\text{def}} \sigma_\neg(u *^n v)$ that is violated by every tree with a u-labeled node that has a v-labeled node as a descendant $(n+1)$ levels below. Thus, every encoding of a tiling that does not respect V violates some $\sigma_{V,u,v}$.

It is not hard to figure out that the reduction can be computed in polynomial time and that $\Sigma \models \tau$ holds if and only if $\mathcal{U}$ does *not* have a valid tiling of width n.

This concludes the proof of (a). We continue with the proof of (b). In this case, the proof is by a reduction from the undecidable unbounded tiling problem to the complement of XCS-IMP(TP, FD, ESDTD).

Let thus $\mathcal{U} = (U, H, V, u_0, u_F)$ with $U = \{u_1, \ldots, u_n\}$ be a tiling instance. Without loss of generality we assume that $\mathcal{U}$ has no valid tiling of width less than 4. As always in our lower bound proofs, the ESDTD D only fixes the alphabet $\Gamma = U \cup \{\$, \&, c, r, x, a, b\}$ but does not restrain content models any further.

The encoding of tilings for this reduction is more complex than the encoding used in the proof of (a). First we describe how tilings are represented as trees. Afterwards, we describe the construction of Σ and τ. The row number i and the column number j of each position (i,j) of a tiling are represented by two data values.

Furthermore each tile $\lambda(i,j)$ is preceded by a sequence that encodes the set of tiles that are *not* vertically compatible with the tile $\lambda(i,j)$, that is, which should not occur at position $(i-1,j)$. Each such "forbidden" tile comes with the two data values encoding $(i-1,j)$ and therefore a simple key constraint can rule out that $\lambda(i-1,j)$ is a forbidden tile.

Now we describe the representation in more detail. We use natural numbers as (intended) data values but the actual choice of data values does not affect the proof.

A position (i,j) of the tiling carrying $u =_{\text{def}} \lambda(i,j)$ is encoded by the string

$$\underbrace{\underbrace{(cr\alpha_{u,1}) \cdots (cr\alpha_{u,n})}_{\text{disallowed tiles}}}_{\text{first part}} \;\&\; \underbrace{\underbrace{(cru)}_{\text{tile}}\underbrace{(crx)^{n-1}}_{\text{padding}}}_{\text{second part}},$$

where $c, r, x, \&$ are symbols from $\Gamma \setminus U$, $\alpha_{u,j} = u_j$ if $(u_j, u) \notin V$ and $\alpha_{u,j} = x$ otherwise.

That is, the representation of a tile consists of two parts separated by &, where each part consists of n repetitions of $cr\alpha$, $\alpha \in U_x$, where $U_x =_{\text{def}} U \cup \{x\}$. The intention is that all c-nodes carry the column number j as data value.

The first part encodes which tiles are not allowed in the row below. The intention is that all r-nodes in the first part carry the row number $i-1$ of the previous row as data value (which is 0 for the first row).

The second part encodes the actual tile u. The $n-1$ repetitions of crx are for technical reasons related to the concatenation of rows that will be explained below. All r-nodes of the last n positions shall carry the row number i as data value.

We note that we do not use any separator symbols between (encodings of) tiling positions. However, rows are separated by $3n+1$ \$-signs, again for technical reasons.

The complete tiling is prefixed with the string $s_{\text{pre}} =_{\text{def}} (xbx)^n(axx)^n\&(abx)^n\$^{3n+1}$, for technical reasons. We denote the length of s_{pre} by $k = 12n+2$.

Altogether, we represent tilings by unary trees, whose path from top to bottom has a prefix conforming to the pattern language of

$$R = \underbrace{s_{\text{pre}}}_{\text{prefix}} \underbrace{\Big[\big(\underbrace{(crU_x)^n\&(crU)(crx)^{n-1}}_{\text{one tile}} \big)^* \$^{3n+1} \Big]^*}_{\text{one row}}.$$

We call a tree t *well formed*, if it fulfills the following conditions:

(i) t is unary (hence we assume that t is a path in the following conditions);

(ii) the node labels of the maximal path t' of t that ends with the pattern $\$^{3n+1}$ (to which we refer as the *main path* of t) conform to R;

(iii) for each encoding of a tile, the set of disallowed tiles is correctly encoded, i.e., matches the vertical constraints;

(iv) for every encoding of a tile position, all c-nodes carry the same data value (we refer to this value as the *column number* of the respective tile);

(v) in every row, exactly the same column numbers occur, in the same order;

(vi) for tiles in the same row, all r-nodes in the second part of tiles have the same data value (we refer to this value as the *row number* of that row); and

(vii) for tiles in the same row, all r-nodes in the first part of tiles have the same data value which happens to be (in non-first rows) the row number of the previous row.

Let tree be the function, which maps tilings to their tree encodings and let tiling be the inverse function, which maps trees to the tiling corresponding to their main path. The function tree is defined for all tilings, while the function tiling is only defined for well formed trees.

The following claim will be shown below.

Claim 11.28 *For every tiling instance $\mathcal{U}$, a constraint set Σ_{wf} can be computed in polynomial time, such that*

- *every tree t that contains the pattern $\$^{3n+1}$ at least once and fulfills $t \models \Sigma_{wf}$ is well formed; and*
- *for every valid tiling λ, it holds that* tree(λ) *fulfills Σ_{wf}.*

We next define sets Σ_H and Σ_V that are supposed to enforce that the tiling encoded by a well formed tree respects the horizontal and vertical constraints of $\mathcal{U}$, respectively. The horizontal constraints can be easily enforced by disallowing the forbidden patterns. To this end, Σ_H contains, for every pair $(u_i, u_j) \notin H$, the constraint $\sigma_\neg(//\&cru_i(***)^{2n-1}\&cru_j)$. In the framework established by Σ_{wf}, the vertical constraints can be enforced as follows. The set Σ_V contains, for every i, the constraint[18]

$$(//c\langle x_c\rangle/r\langle x_r\rangle/u_i\langle x_u\rangle, \{x_c.@, x_r.@\}\rightarrow x_u).$$

Finally, we let

- $\Sigma =_{\text{def}} \Sigma_{\text{wf}} \cup \Sigma_H \cup \Sigma_V$ and
- $\tau =_{\text{def}} \sigma_\neg(/*^{k+3n+3}u_0//u_F***^{n-1}\$^{3n+1})$.

We note that τ can only be violated by trees t containing the pattern $\$^{3n+1}$ at least once.

We show now that $\Sigma \models_D \tau$ holds if and only if there is no valid tiling for $\mathcal{U}$. We sketch the proof argument for this.

(if): Let us assume that there is no valid tiling for $\mathcal{U}$. Thus, every tree t is either not well formed or its tiling is not valid. In the first case, $t \not\models \Sigma_{\text{wf}}$, in the second case $t \not\models \Sigma_H \cup \Sigma_V$ or t does not match the pattern forbidden by τ. In both cases, t is not a counter-example for $\Sigma \models_D \tau$, thus there is no such counter-example and $\Sigma \models_D \tau$ holds.

(only if): Let us assume $\Sigma \models_D \tau$ holds. Towards a contradiction let us assume further that $\mathcal{U}$ has a valid tiling λ. Let t be tree(λ). Thus, $t \models \Sigma$ and t matches the pattern of τ, therefore $t \not\models_D \tau$, the desired contradiction.

It thus only remains to prove Claim 11.28. The conditions in the definition of well-formedness can be enforced as follows by constraints.

(i) σ_{unary}.

(ii) That t' begins with the string s_{pre} can be enforced by constraints that disallow, for every position, all other labels. As an example, $\sigma_\neg(/*a)$ forbids the second symbol to be an a (instead of the required b). The number of such constraints is bounded by the length of s_{pre} times the size of the alphabet and is therefore polynomial. Similarly, it can be enforced that the first $6n + 1$ positions after each \$-block conform to the tile encoding pattern and that, after each tile encoding, there is another tile encoding or a \$-block.

[18] We note that this is a key constraint, but also the only kind of FD with a binary set of attributes in Σ.

(iii) That the list of forbidden tiles is correct, for each tile, can similarly be expressed by suitable $\sigma_{\neg}$-constraints.

(iv) First, the constraint $(//a\langle x\rangle, \emptyset \to x.@)$ ensures that all a-nodes in the prefix have the same data value. Next, the constraint $(//a\langle x_a\rangle(***)^{3n} * c\langle x_c\rangle, x_a.@ \to x_c.@)$ enforces that all c-positions of the first tile have the same data value. The constraint

$$\sigma_{c+} =_{\text{def}} (//c\langle x_1\rangle(***)^{2n} c\langle x_2\rangle, x_1.@ \to x_2.@)$$

ensures that, in the first row, in every tile, all c-positions have the same data value (because it is already guaranteed that all c-nodes of the first tile of the first row have the same data value). The constraint

$$\sigma_{c1} =_{\text{def}} (//c\langle x_1\rangle(***)^{2n+1} c\langle x_2\rangle, x_1.@ \to x_2.@)$$

then enforces that in the first tile of the second row all c-positions have the same data value. Together, σ_{c+} and σ_{c1} guarantee that within the encoding of every tile, the c-positions carry the same data value.

(v) By $(//\$/c\langle x\rangle, \emptyset \to x.@)$ it can be ensured that the data value of the c-nodes of the first tile is the same in all rows. Likewise, $(//c\langle x\rangle/**\$, \emptyset \to \langle x\rangle)$ ensures that the data value of the c-nodes of the last tile is the same in all rows. That the column number of one tile determines the column number of the next tile is already enforced by σ_{c+}. Altogether these condition enforce (v).

(vi) The constraint $(//b\langle x\rangle, \emptyset \to x.@)$ ensures that all b-nodes have the same data value (which might or might not be different from the value of the a-nodes[19]). The constraint $(//b\langle x_1\rangle(***)^{5n} ** r\langle x_2\rangle, x_1.@ \to x_2.@)$ enforces that all r-positions in the second parts of the first two tiles of the first row have the same data value. The constraint

$$\sigma_{r+} =_{\text{def}} (//r\langle x_1\rangle(***)^{2n} r\langle x_2\rangle, x_1.@ \to x_2.@)$$

then ensures that, in the first row, *all* r-positions of second parts have the same data value. This is because, the r-values of (the right part of) the second tile are determined by those of the first tile, but all these values are already guaranteed to be equal. Therefore, all further implied values of this kind, within the row, must also take this value. However, this constraint (intentionally[20]) has no impact across rows as there is no pair of r-nodes from different rows with distance $6n + 1$. The constraint

$$\sigma_{r2} =_{\text{def}} (//r\langle x_1\rangle(***)^{6n} ** r\langle x_2\rangle, x_1.@ \to x_2.@)$$

[19] We note that it does not matter, whether the set of row numbers is disjoint from the set of column numbers as they never interact with each other.

[20] This condition partially explains the complication of the encoding.

then guarantees that, in the second row, all r-positions in the second part of the first two tiles of the first row have the same data value[21]. Together, σ_{r+} and σ_{r2} enforce that, in every row, all r-positions of second parts of tiles have the same data value.

(vii) The pattern of the constraint

$$\sigma_{2,1} =_{\text{def}} (//r\langle x_1\rangle(***)^{4n}**r\langle x_2\rangle, x_1.@\rightarrow x_2.@)$$

matches corresponding r-nodes in second (and also in first) parts of tiles of (tile) distance two in the same row. As the other constraints already enforce that all second-part r-nodes in the same row have the same data value it always matches inside a row with the same value for $x_1.@$ and $x_2.@$. Due to the length of the \$-block at the end of a row, it also matches between the r-nodes in the second part of the last tile of a row and the r-nodes in the first part of the second tile of the next row, as well as between the r-nodes in the second part of the last but one tile of a row and the r-nodes in the first part of the first tile of the next row. Therefore $\sigma_{2,1}$ guarantees that, for every row, the r-nodes in the first parts of the first two tiles have the same value as the r-nodes in the second parts of the last two tiles in the previous row.[22] Hence, $\sigma_{2,1}$ and σ_{r+} together ensure that the data value of all r-nodes in first parts of tiles are just the row numbers of the previous row (if there is a previous row). □

It still remains to prove Theorem 11.1.

Statement of Theorem 11.1

(a) XCS-Imp(MSO,XKFD,Reg) is decidable.

(b) XC-Imp(FO,FD) is undecidable.

Proof. Statement (b) can be shown by a reduction from XCS-Imp(TP, FD, esDTD), which is undecidable by Theorem 11.27. From a given tree-pattern based instance (Σ, τ, D) it constructs an instance (Σ', τ') as follows: the patterns from Σ and τ are simply translated into FO formulas. We enforce that all considered trees are valid for D, by some additional constraint $\sigma_D = (m_\varphi, \emptyset\rightarrow x)$, where m_φ is the mapping induced by the FO formula $\varphi(x)$ that selects all nodes if t contains a label not allowed by D and no node otherwise. The constraint σ_D enforces that either D is satisfied or t has at most one node.

Towards (a), let (Σ, τ, S) be an instance of XCS-Imp(MSO,XKFD,Reg), where $S \in$ Reg is a regular tree language.

Let n be the number of free variables of the MSO-formula defining the mapping of τ. Let us assume that t is a counter-example for Σ and τ and that $\pi = (\pi_1, \pi_2)$ is a

[21] We note that this constraint connects the last tile of a row with the second of the next row and the last tile but one of one row with the first tile of the next row.

[22] Here we use the assumption that valid tilings have width ≥ 4.

witness pair[23] for $t \not\models \tau$ and, by Lemma 11.5, that t is π-diverse. We can conclude that if $\Sigma \not\models_S \tau$, then there is a counter-example in which at most n data values (in the range of π_1 and π_2) may occur twice, all other data values occur exactly once.

Let Γ be the set of labels used by S and $\Gamma' = \Gamma \times \{d_1, \ldots, d_n, \bot\}$. We now look at trees over the label set Γ', with the intended meaning that a node labeled by (a, d_i) has label $a \in \Gamma$ and data value d_i and a node labeled by $(a, \bot)$ has a unique data value. It is now straightforward to construct an MSO formula that, given a tree t over the label set Γ', tests that $\mathrm{de}(t) \models D$, where $\mathrm{de}(t)$ projects all labels to Γ and tests whether Σ and τ holds in the resulting tree.

The decidability of XCS-Imp(MSO,XKFD,Reg) thus follows from the decidability of the finite satisfiability problem for MSO logic on trees [TW68]. □

11.7 Conclusions and Further Research on X2R-constraints

A big part of this thesis is dedicated to analyze XML-to-relational constraints. After giving some basics in Chapter 9, we defined our framework of X2R-constraints in Chapter 10. We especially focused on X2R-constraints based on tree patterns as mapping language and functional dependencies as relational constraint language. We showed that existing research towards XML integrity constraints is compatible with our framework, i.e., earlier work can be defined by the means of our framework. Furthermore, in the current chapter, we showed that we can use this framework to reason about integrity constraints. In particular we looked at the complexity of the implication problem for XML integrity constraints. There are two central insights that should be pointed out:

- Restricting equality generating dependencies (like functional dependencies) to enforce equality only on nodes can lead to a much lower complexity of the implication problem. Restriction to those constraints is also motivated by existing work on XML normal forms [AL04].

- Navigational properties of the used constraint mechanism should be separated from semantic features that can compare data values. In Chapter 12, we will present an approach that uses FO logic for navigation and for specifying constraints. We will see that allowing the logic to do both navigation and data comparisons leads to high complexity.

  Separating the navigational aspects (mappings) and the semantic aspects (relational integrity constraints) allowed us a much easier access to the problem and has led to some classes of integrity constraints with a tractable implication problem.

In the area of integrity constraints, there are many possible directions for further research. In our studies we focused on functional dependencies and an XML variant

[23] We assume a more general notion of a witness pair here. It is clear that Lemma 11.5 can be generalized for witness pairs, where the mapping is specified by an FO formula.

of key constraints (XKFDs). Looking at relational databases, a canonical next step would be to add foreign key constraints to the picture. This obviously has to include to find a good specification for foreign key constraints in the XML context. Directly using relational foreign key constraints will not work, as these constraints assume the existence of relational key constraints, which are different from XKFDs.

A different direction of research would be to combine the research towards XML functional dependencies with ongoing research in XML data exchange. Arenas et al. have studied relational and XML data exchange [ABLM10]. Their framework for studying source-to-target constraints in XML databases is very similar to our framework we used to study functional dependencies and key constraints. An obvious direction of further research is the combination of both frameworks to study source-to-target dependencies in combination with target functional dependencies or key constraints (XKFDs).

It should be noted that contrary to Part I, here we have concentrated on the underlying mechanisms for representing and inferring constraints, i.e., we did nod develop or describe languages that can be used by database administrators to directly describe constraints. One way to put this research into productive use is to use existing mechanisms for specifying constraints (like the mechanisms defined in XML Schema), convert them into our framework (as we have done in Chapter 10.4) and then use our knowledge to, e.g., compute the inference relation. However, we have seen, that XML Schema constraints cannot be represented in a one-to-one fashion in our framework, as they involve a functional dependency (XKFD) and a non-null constraint. Towards XRMS's, which are aimed at large amounts of data instead of single documents, it might be worth the effort to design a new constraint specification language. Such a language could be more transparent to the user in which exact constraints are enforced on the data, allowing a database programmer to more precisely specify the needed constraints. Towards such a language there should be some more research to identify tractable instantiations of our framework. To give a more concrete example, we have discovered, that allowing descendant edges in tree patterns rises the complexity. However, without descendant edges, it is not possible to describe constraints for documents of arbitrary depth. It would be nice to identify tractable fragments that allow limited use of descendant edges to combine low complexity with high expressivity.

12 Two Variable First Order Logic and Key Constraints

As already pointed out in Chapter 9, integrity constraints (on relations) are heavily based on first order logic. Furthermore, it is well known that two-variable first order logic has interesting connections to the foundations of XML. For instance, Core XPath 1.0 corresponds exactly to two-variable logic on unranked, ordered trees [Mar05] and the regular tree languages, which capture the structural part of existing schema languages, exactly correspond to existential monadic second order logic with two first-order variables (see, e.g., [BMSS09]).

The main results of [BMSS09, BMS$^+$06] are that

- satisfiability of two-variable logic over data words is decidable even if formulas can use the linear order and the successor relations on positions [BMS$^+$06], and that
- satisfiability of two-variable logic over data trees is decidable if only the parent-child relationship and the direct sibling relationship between the children nodes of a parent are available [BMSS09].

The former problem has unknown (but probably huge) complexity, the latter can be solved in 3-NEXPTIME.

Even though two-variable logic can express a lot of interesting properties of XML documents, their ability to express integrity constraints is limited. More precisely, they can express in general key, foreign key and inclusion constraints only if they are unary.

As an example that integrity constraints can indeed be expressed by two variable logic, we once more come back to our running example and give two variable formulas for our example constraints, i.e., user ids are unique and for each document, the document owner exists. For readability, we use the same variable names, as in Section 10.1. However, we add upper indices x and y to denote whether the variable should be the first or the second variable in a two variable formula, i.e., to obtain an equivalent two variable formula, all variable with upper index x should be replaced by x and all variables with upper index y should be replaced by y.

$$\begin{aligned}\Psi_{\text{uid-unique}} \quad = \quad & \forall v_u^x, w_u^y.\ \Big[\text{USER-ID}(v_u^x) \wedge \text{USER-ID}(w_u^y) \wedge v_u^x \sim w_u^y \wedge \\ & \big(\exists v_p^y.\ \text{PERSON}(v_p^y) \wedge E(v_u^x, v_p^y)\big) \wedge \\ & \big(\exists w_p^x.\ \text{PERSON}(w_p^x) \wedge E(w_u^y, w_p^x)\big)\Big] \quad \rightarrow \quad v_u^x = w_u^y\end{aligned}$$

$$\Psi_{\text{uid-exists}} = \forall v_d^x, v_u^y.\ \ \text{DOCUMENT}(v_d^x) \wedge \text{USER-ID}(v_u^y) \wedge E(v_d^x, v_u^y) \ \rightarrow \Big(\exists w_u^x.\ \text{USER-ID}(w_u^x) \wedge v_u^y \sim w_u^x \wedge \big(\exists w_p^y.\ \text{PERSON}(w_p^y) \wedge E(w_p^y, w_u^x)\big)\Big)$$

In this chapter, we aim to shed some light on the problem to decide whether, for a given formula of two-variable logic and a set of (not necessarily unary) key constraints, there is an XML document that fulfills the formula and the constraints. However, as the problem has turned out to be quite complex, we restrict to study it only for data *words*.

After some definitions, we will show in Section 12.2 that the complexity of satisfiability of two-variable logic with equality on data value and successor relation drops to NEXPTIME when considering strings instead of trees. On top of this result we show our main result in Section 12.3, where we show that the problem remains decidable when adding k-ary key constraints. Note that we are not able to give an elementary upper bound for the problem. Whether the corresponding problem for trees is decidable is still open.

12.1 Definitions

A *data word with symbols* over an alphabet Σ and a data domain[1] dom is a finite, non-empty sequence of pairs $w = (\sigma_1, d_1) \cdots (\sigma_n, d_n)$ where each $\sigma_i \in \Sigma$ and $d_i \in \text{dom}$. A *data word with propositions* over a finite set $\mathcal{P}$ of propositions and a data domain dom is a finite sequence $w = (P_1, d_1) \cdots (P_n, d_n)$ where each $P_i \subseteq \mathcal{P}$ and $d_i \in \text{dom}$. For data words, we denote the *length* n of w by $|w|$. We call the string $\text{str}(w) =_{\text{def}} \sigma_1 \cdots \sigma_n$ the *string projection* of w, likewise for data words with propositions. The set of data values occurring in w is denoted by $\text{dom}(w)$. For each $d \in \text{dom}(w)$, the *class of* d *in* w is the set $\text{Class}_d(w)$ of positions with value d. A *zone* of a data word is a maximal substring in which all positions carry the same data value.

The *Parikh image* $\text{parikh}(w)$ of a Σ-word (or data word over Σ) is the function that maps every symbol in Σ to the number of its occurrences in w. The Parikh image $\text{parikh}(L)$ of a language is just the set $\{\text{parikh}(w) \mid w \in L\}$.

An *atomic $\mathcal{P}$-type* is a set of propositions and negated propositions from $\mathcal{P}$. The *full atomic $\mathcal{P}$-type* of a position i in a data word is the formula $\alpha_P(x) = \bigwedge_{p \in P_i} p(x) \wedge \bigwedge_{p \in \mathcal{P} - P_i} \neg p(x)$. The set of full atomic types over $\mathcal{P}$ is denoted by $\mathcal{T}(\mathcal{P})$. Clearly, there is a simple relationship between subsets $P \subseteq \mathcal{P}$ and full atomic $\mathcal{P}$-types α. Therefore we sometimes represent a full atomic type α by the set P of its positive propositions and can identify $\mathcal{T}(\mathcal{P})$ with $2^{\mathcal{P}}$.

In this chapter, we deal mainly with data words over propositions, however we will frequently make use of the fact, that data words over a set $\mathcal{P}$ of propositions can be considered as data words with symbols over the alphabet $\mathcal{T}(\mathcal{P})$. Likewise, we will define automata for words with propositions as (usual) automata over $\mathcal{T}(\mathcal{P})$.

Note that upper bounds for data words over propositions (as Theorem 12.2) translate to data words with symbols but not necessarily vice versa.

[1] In this chapter we use dom instead of $\mathcal{D}$, as $\mathcal{D}$ can too easily be confused with dog sets, which we will introduce later.

Definition 12.1 A *key constraint* κ for words with symbols is a sequence of entries from $(2^\Sigma \times \{\bullet, \circ\})$. For a key constraint $\kappa = (K_1, \otimes_1) \cdots (K_k, \otimes_k)$ we call k the *length* of κ.

We say that a key constraint $\kappa = (K_1, \otimes_1) \cdots (K_l, \otimes_l)$ *matches a position* i in a data word $w = (a_1, d_1) \cdots (a_n, d_n)$ over an alphabet Σ if for every $j \in [1, k]$, $a_{i+j-1} \in K_j$.

A key constraint is *violated* in w if it matches two different positions $i_1 \neq i_2$ and, for every $j \in \{1, \ldots, l\}$, if $\otimes_j = \bullet$ then $d_{i_1+j-1} = d_{i_2+j-1}$. Otherwise it is *fulfilled*. We write $w \models \kappa$ if κ is fulfilled in w and, for a set $\mathcal{K}$ of key constraints, $w \models \mathcal{K}$ if w fulfills every key constraint in $\mathcal{K}$. For data words with propositions the definition is analogous with $\mathcal{T}(\mathcal{P})$ in place of Σ and full types α_i in place of symbols a_i.

For a set $\mathcal{K}$ we denote the maximum length of a key in $\mathcal{K}$ by $k(\mathcal{K})$.

In this chapter, a data word $w = (P_1, d_1) \cdots (P_n, d_n)$ with propositions from $\mathcal{P}$ is represented by a logical structure with universe $\{1, \ldots, n\}$, a successor relation $+1$, an equivalence relation $\sim$ that holds for two positions if they carry the same data value, and one unary relation p for every $p \in P$.

The logic $\mathrm{FO}^2(\sim, +1)$ is just first-order logic over such structures, restricted to the use of variables x and y. Thus, quantifiers range over positions of a data word, the formula $x + 1 = y$ expresses that y is the right neighbor of x and $p(x)$ indicates that at position x proposition p holds. A formula $x \sim y$ expresses that at x and y the same data value occurs.

$\mathrm{FO}^2(\sim, +1)$ over words with symbols is defined analogously with atomic formulas $\sigma(x)$ for symbols $\sigma \in \Sigma$.

In the following we consider functions from I to $\mathbb{N}$, for various sets I. Each such function can be considered as a vector with $|I|$ entries from $\mathbb{N}$. A *linear set* Lin over I is a set of functions $I \to \mathbb{N}$ that can be represented as $\{\mathrm{f} + \sum_i j_i \mathrm{f}_i \mid j_i \in \mathbb{N}\}$, where f and all f_i are functions $I \to \mathbb{N}$. A *semi-linear set* SLin over I is the union of finitely many linear sets.

A *counting language* $L(\mathcal{A}, \mathrm{SLin})$ is defined by a finite automaton $\mathcal{A}$ over Σ and a semi-linear set SLin over Σ. A string v is in $L(\mathcal{A}, I)$, if and only if $v \in L(\mathcal{A})$ and $\mathrm{parikh}(v) \in I$. The term counting language has already been used in [CDFI12].

12.2 $\mathrm{FO}^2(\sim, +1)$ without Key Constraints

The main theorem of this section is

Theorem 12.2 *The satisfiability problem for $FO^2(\sim, +1)$ formulas over data words with propositions is decidable in* NEXPTIME.

Note that the theorem is formulated for data words with propositions but it also holds for data words with symbols.

Our approach is an adaptation of the techniques of [BMSS09] from trees to strings. The same approach has already been used in [NS11]. We improve upon the results in [NS11] by providing a better upper bound (NEXPTIME instead of 2-NEXPTIME).

We will see, that we can divide the constraints enforced on a string by an $\mathrm{FO}^2(\sim, +1)$ formula φ in two classes: local constraints (e.g. every position labeled a has a neighbor

Algorithm 11 Test satisfiability of $\mathrm{FO}^2(\sim, +1)$-formulas

```
1: function SAT(φ)
2:     Compute φ' in Scott normal form s.t. φ' is satisfiable iff φ is satisfiable
3:     Compute φ'' in data normal form s.t. φ'' is satisfiable iff φ' is satisfiable
4:     Guess string profile T
5:     if T is not compatible with φ'' then reject
6:     Guess border profile T_b
7:     if T_b is not compatible with T then reject
8:     Compute counting language L(A, SLin) s.t. L(A, SLin) ≠ ∅ iff T_b is satisfiable
9:     if L(A, SLin) is satisfiable then accept
10:    reject
```

labeled b with a different data value) and global constraints (e.g. there are no two a positions with the same data value). Intuitively, local constraints stem from the use of the successor relation and global constraints stem from subformulas not speaking about successors.

The basic approach of [BMSS09] and [NS11] is to construct a string automaton $\mathcal{A}$ that checks the local constraints and a (semi-)linear set Lin containing Parikh images of string projections that encode the global constraints. A formula φ is satisfiable, if and only if the counting language $L(\mathcal{A}, \mathrm{Lin})$ is nonempty. As an intermediate representation (string) profiles are defined, which allow an easier separation of the global and local constraints imposed by a formula.

We will define profiles in much the same way as [BMSS09, NS11], but our approach will involve an additional intermediate representation, which we denote by border profiles to achieve a better upper bound.

We depict the high-level algorithm for deciding satisfiability of $\mathrm{FO}^2(\sim, +1)$-formulas as Algorithm 11. We introduce the necessary formalisms (Scott normal form, data normal form, string profile, border profile), where they are needed.

The intuitive idea behind string profiles and border profiles is to capture the constraints of the formula with syntactically very restricted formalisms. The global constraints can then be catched by a semi-linear set and the local constraints by a string automaton. Compatibility will be defined in such a way, that a string profile T is compatible with a formula φ, if and only if any solution of T yields a solution for φ. The same holds for compatibility of string profiles and border profiles.

As the proof uses many definitions, we list the important terms in Table 12.1 together with brief definitions. Formal definitions are given in the proof at the position they are needed.

term	symbol	definition
class	c	maximal subsequence with the same data value
zone	z	maximal substring with the same data value
$\mathcal{P}_2$ type	α	full atomic formula over $\mathcal{P}_2$
set of all $\mathcal{P}_2$ types	$\mathcal{T}(\mathcal{P}_2)$	powerset of $\mathcal{P}_2$
dog	D	(set of) $\mathcal{P}_2$ type(s) occurring exactly once in a class or zone
sheep	S	(set of) $\mathcal{P}_2$ type(s) occurring arbitrarily often in a class or zone
dog zone		zone containing at least one dog type
sheep zone		zone consisting only of sheep types
class type	τ	set of all $\mathcal{P}_2$ types of a class, which may occur exactly once (dogs D) or arbitrarily often (sheep S)
zone type	τ	the same as class type, but refers to a single zone
border type	β	specifies the leftmost and rightmost border $\mathcal{P}_2$ type of a zone and whether the zone is a sheep or dog zone
set of all border types	B	$B = \mathcal{T}(\mathcal{P}_2) \times \mathcal{T}(\mathcal{P}_2) \times \{\eta_D, \eta_S\}$
border string		string over the alphabet $B \times (\text{dom} \cup \{\bot\})$ containing all border types of a string together with their (possibly unknown) data value
explicit data value		data value assigned in the border string
class border type	τ_b	set of border types occurring in a class
profile	T	set of class types occurring once/twice/more than twice in a string
border profile	T_b	set of class border types occurring in a string
	$T(w)$	unique profile of the string w
	$T_b(w)$	unique border profile of the string w

Table 12.1: Terms used in the proof of Theorem 12.2, together with the usually used symbols and a brief definition.

12.2.1 Normal Form

In this section, we will bring formulas in data normal form, which allows us to separate global and local constraints enforced by a formula. In a first step, we will bring a formula into Scott normal form.

An FO^2-formula is in *Scott normal form (SNF)* if it is of the form

$$\psi = \big(\forall x \forall y\ \chi \wedge \bigwedge_i \forall x \exists y\ \chi_i\big),$$

where χ and each χ_i are quantifier-free $\text{FO}^2(\sim, +1)$ formulas (see [GO99] for a reference).

In a standard fashion, any $\text{FO}^2(\sim, +1)$ formula can be translated into a formula in Scott normal form that is equivalent with respect to satisfiability, as stated in the following lemma[2].

Lemma 12.3 *For each $FO^2(\sim, +1)$ formula φ a $FO^2(\sim, +1)$ formula φ' in Scott normal form can be computed in polynomial time such that φ is satisfiable over data words if and only if φ' is satisfiable over data words.*

[2]If stated for data words with symbols this lemma would come with an exponential blow-up of the number of symbols due to the need to encode all possible combinations of the R_i relations.

Proof. From φ one can compute an existential second-order formula

$$\psi = \exists R_1 \cdots R_m \big(\forall x \forall y\ \chi \wedge \bigwedge_i \forall x \exists y\ \chi_i\big),$$

that is equivalent to φ and where the relation symbols R_i are unary. Let $\mathcal{P}' = \mathcal{P} \cup \{p_1, \ldots, p_m\}$, where the propositions p_i are new. Let φ' be the formula obtained from ψ by removing the quantification of the relations R_j and replacing each atom $R_j(x)$ by $p_j(x)$ (and likewise each $R_j(y)$ by $p_j(y)$). Clearly, φ' is satisfied by some data word over $\mathcal{P}'$ if and only if φ is satisfied by some data word over $\mathcal{P}$. □

Thus, we can assume henceforth that the $\mathrm{FO}^2(\sim, +1)$-formula φ that shall be tested for satisfiability is in Scott normal form.

In the following, we will annotate word positions by propositions that reflect the propositions of the adjacent positions and whether the own data value equals the data values of the adjacent positions. To this end we use additional propositions of the form p^{-1} and p^{+1}, for every $p \in \mathcal{P}$. We define $\mathcal{P}^{-1} = \{p^{-1} \mid p \in \mathcal{P}\}$ and $\mathcal{P}^{+1} = \{p^{+1} \mid p \in \mathcal{P}\}$. Furthermore, we use the additional propositions $p_=^{-1}$ and $p_=^{+1}$ to indicate data equalities and $p_\triangleright$ and $p_\triangleleft$ to mark border positions. Finally, the propositions p_1 and p_2 are used to mark up to two occurrences of a type in a class. By $\mathcal{P}_1$ we denote $\mathcal{P} \cup \mathcal{P}^{-1} \cup \mathcal{P}^{+1} \cup \{p_=^{-1} p_=^{+1}, p_\triangleright, p_\triangleleft\}$ and by $\mathcal{P}_2 = \mathcal{P}_1 \cup \{p_1, p_2\}$. Clearly, $|\mathcal{P}_2| = \mathcal{O}(|\mathcal{P}|)$.

Before we continue, let us clarify the relationship between full $\mathcal{P}$-types and full $\mathcal{P}_2$-types. Each $\mathcal{P}_1$-type basically consists of the $\mathcal{P}$-type of a position, the $\mathcal{P}$-types of its left and right neighbor and the information whether the left and right neighbor have the same data value as the position. Thus, we can view a $\mathcal{P}_1$-type as a tuple $(\alpha, \alpha^{-1}, \alpha^{+1}, p_=^{-1}, p_=^{+1})$ of three full $\mathcal{P}$-types and two atomic propositions. In $\mathcal{P}_2$-types, the propositions p_1 and p_2 additionally mark up to two occurrences of every $\mathcal{P}_1$-type α in a class.

We will use the additional propositions to rewrite a given formula in a way, that it does not need the successor relation. This way, we separate the global constraints, encoded in the rewritten formula, from the local constraints, encoded by validity of data words, where a data word is valid if the additional propositions are consistent, as described below.

A data word $w = (P_1, d_1) \cdots (P_n, d_n)$ over $\mathcal{P}_2$ is *valid* if it fulfills the following conditions.

(i) P_1 contains $p_\triangleright$ but not $p_=^{-1}$ and no proposition of the form p^{-1}.

(ii) P_n contains $p_\triangleleft$ but not $p_=^{+1}$ and no proposition of the form p^{+1}.

(iii) If $i < n$ then P_i contains a proposition p^{+1} if and only if P_{i+1} contains p. Furthermore, it contains $p_=^{+1}$ if and only if $d_i = d_{i+1}$.

(iv) If $i > 1$ then P_i contains a proposition p^{-1} if and only if P_{i-1} contains p. Furthermore, it contains $p_=^{-1}$ if and only if $d_i = d_{i-1}$.

(v) If a class contains at least one position with a $\mathcal{P}_1$-type α then it contains exactly one such position with proposition p_1 (and at this position p_2 does not hold).

(vi) If a class contains at least two positions with a $\mathcal{P}_1$-type α then it contains exactly one such position with proposition p_2 (and at this position p_1 does not hold).

We call a data word over $\mathcal{P}_1$ valid if it fulfills all but the last two conditions. A data word over $\mathcal{P}_1$ cannot fulfill the last two conditions, as $\mathcal{P}_1$ does not contain the propositions p_1 and p_2. By $\mathrm{de}(w)$ we denote the data word over $\mathcal{P}$ that is obtained from a data word w over $\mathcal{P}_2$ by dropping all other propositions. Clearly, for every data word w over $\mathcal{P}$ there is a unique valid data word w' over $\mathcal{P}_1$ with $\mathrm{de}(w') = w$. But there can be more than one such w' over $\mathcal{P}_2$.

We refer to $\mathcal{P}_2$-types which have p_1 or p_2 as dog types and to other $\mathcal{P}_2$-types as sheep types.

Definition 12.4 We say an $\mathrm{FO}^2(\sim)$ formula over the proposition set $\mathcal{P}_2$ is in data normal form, if it is a conjunction of formulas of the following forms, where α and β always denote dog types and $\delta(x, y)$ is always one of $x \sim y$ and $x \not\sim y$:

(i) $\neg\big(\exists x\ \alpha(x)\big)$

(ii) $\neg\big(\exists x \exists y\ \alpha(x)\ \wedge\ \beta(y)\ \wedge\ \delta(x, y)\big)$

(iii) $\forall x \exists y\ \alpha(x) \rightarrow \bigvee_i \beta_i(y)\ \wedge\ \delta_i(x, y)$

It is important to note that formulas in data normal form do not use the successor relation explicitly. However, the definition of valid data words uses the successor relation.

The following proposition says, that we can rewrite every $\mathrm{FO}^2(\sim, +1)$ formula to data normal form. The proposition only talks about equivalence up to satisfiability, however in fact, the satisfying models are equal on the common set of propositions and only differ in the additional propositions introduced in $\mathcal{P}_2$.

Proposition 12.5 *Let φ be an $FO^2(\sim, +1)$ formula over proposition set $\mathcal{P}$. Then there exists an $FO^2(\sim)$ formula φ' over proposition set $\mathcal{P}_2$ in data normal form, such that*

- *for every valid data word v it holds $\mathrm{de}(v) \models \varphi$ if and only if $v \models \varphi'$*
- *φ' is of at most exponential size; and*
- *φ' can be computed from φ in exponential time.*

Proof. Let φ be a formula of the form

$$\big(\forall x \forall y\ \chi \wedge \bigwedge_i \forall x \exists y\ \chi_i\big).$$

It is straightforward to bring χ into CNF, and to rewrite $\forall x \forall y\ \chi$ as a conjunction of (exponentially many) formulas of the form:

$$\psi = \forall x \forall y \neg\big((\alpha(x)\ \wedge\ \beta(y)\ \wedge\ \delta(x, y)\ \wedge\ \gamma(x, y)\big),$$

where α and β are full atomic $\mathcal{P}$-types, $\delta(x,y)$ is either $x{\sim}y$ or $x{\not\sim}y$, and $\gamma(x,y)$ is one of $x=y$, $x=y+1$, $x=y-1$ and $x \notin [y-1,y+1]$. The latter expression is an abbreviation for the formula $x \neq y-1 \wedge x \neq y \wedge x \neq y+1$.

Note that ψ can be rewritten to

$$\psi = \neg\big(\exists x \exists y\ \alpha(x)\ \wedge\ \beta(y)\ \wedge\ \delta(x,y)\ \wedge\ \gamma(x,y)\big).$$

While this form is obviously equivalent to the one above, it gives a better intuition about the meaning of ψ, namely that every of the formulas ψ just disallows certain patterns to appear in the string.

We show now, how a formula ψ can be rewritten to use the additional propositions of $\mathcal{P}_2$ in place of the successor relation. We distinguish cases based on $\gamma(x,y)$.

In the first case, $\gamma(x,y)$ is $x=y$. Note that in this case ψ is trivially true, if $\delta(x)$ is $x \not\sim y$ or if $\alpha(x)$ and $\beta(y)$ are contradictory. Therefore, we assume that $\alpha(x)$ and $\beta(y)$ are the same full atomic $\mathcal{P}$-type and $\delta(x,y)$ is $x \sim y$. Then ψ can be rewritten as $\neg\exists x\ \alpha(x)$.

The next two cases where $\gamma(x,y)$ is one of $x=y+1$ and $x=y-1$ are completely symmetric. In these cases, we can rewrite ψ as a conjunction of formulas of the form $\neg\exists x\ \alpha'(x)$, where α' is a full atomic $\mathcal{P}_2$-type that is compatible with the $\mathcal{P}$-type α. Remember that any $\mathcal{P}_2$-type encodes the $\mathcal{P}$-types of its left and right neighbor and whether the neighbors have the same data value.

The last case is $\gamma(x,y)$ is $x \notin [y-1,y+1]$, that is x and y are not neighbored. It can be shown, that ψ can be rewritten as a conjunction of formulas of the form $\neg\exists x \exists y\ \alpha'(x)\ \wedge\ \beta'(y)\ \wedge\ \delta(x,y)$, where α' and β' are full atomic $\mathcal{P}_2$-types and δ is as before. The proof is a straightforward but tedious case analysis, as special attention needs to be taken, when a $\mathcal{P}_2$-type α' indicates, that α and β occur next to each other. In this case the additional propositions p_1 and p_2 can be used to detect two occurrences of α-β-neighbors and forbid these. If α and β do not occur as neighbours, it is easier, as we can forbid types α' and β' to occur at the same time.

We now turn to formulas of the form $\forall x \exists y\ \chi_i$. The formula χ_i can be transformed into a conjunction of (possibly exponentially many) formulas of the form

$$\alpha(x) \rightarrow \bigvee_i \beta_i(y) \wedge \delta_i(x,y) \wedge \gamma_i(x,y)$$

where α is a full atomic $\mathcal{P}_2$-type, each of the β_i is a full atomic $\mathcal{P}$-type and δ_i and γ_i are as before. Note that we require α to be a $\mathcal{P}_2$-type instead of a $\mathcal{P}$-type. This can easily be accomplished by adding one subformula for each $\mathcal{P}_2$-type compatible with a given $\mathcal{P}$-type.

We will concentrate on the subformulas of the form $\beta_i\ \wedge\ \delta_i\ \wedge\ \gamma_i$. Note that such a subformula can be rewritten to either true or false in the case that γ_i is one of $x=y$, $x=y+1$ and $x=y-1$, as both, the $\mathcal{P}$-type of y and the logical value of $\delta_i(x,y)$ can be inferred from the $\mathcal{P}_2$-type of x. We therefore assume that all γ_i are $x \notin [y-1,y+1]$.

It can again be shown by a case analysis, that every subformula $\beta_i(y)\ \wedge\ \delta_i(x,y)\ \wedge\ x \notin [y-1,y+1]$ can be rewritten as a disjunction of formulas of the form $\beta'(y)\ \wedge\ \delta_i(x,y)$,

where β' is a full atomic $\mathcal{P}_2$-type. Again the propositions p_1 and p_2 can be used to identify a second occurrence of a $\mathcal{P}_2$-type, in case there are occurrences of α and β as neighbors. □

12.2.2 String Profiles

In the following, we define class type functions for $\mathcal{P}_2$-words. The intention is that the class type function of a valid $\mathcal{P}_2$-word w contains all relevant information to decide whether $w \models \varphi$, where φ is a formula in data normal form.

If w is a valid data word over $\mathcal{P}_2$ and c a class of w, the *class type* τ of c is the pair (D, S), where

- D is the set of all full types α in c that contain p_1 or p_2; and
- S is the set of all full types α such that
  - α does not contain p_1 or p_2; and
  - $(\alpha \setminus \{\neg p2\}) \cup \{p_2\}$ occurs in c.

We call the types in D *dog types* of τ and the types in S *sheep types*. Note that dog types occur exactly once in a class, whereas sheep types can occur arbitrarily often (including not occurring at all). Note further that if w is valid then each class type of every class in w fulfills that

- if it contains a $\mathcal{P}_1$-type α at all then D contains $\alpha \cup \{p_1, \neg p_2\}$, and
- if it contains a $\mathcal{P}_2$-type α with $\{\neg p_1, \neg p_2\} \subseteq \alpha$, then $\alpha \in S$.

Furthermore, no full types containing p_1 and p_2 occur in a class type. We call a class type *valid* if it fulfills these conditions.

Note that the set of sheep types of a class can easily be computed from the set of dog types. However, we explicitly denote the sheep types, because the sheep types will become important when talking about individual zones, where it will not be possible to compute the sheep types from the dog types, as the corresponding dog types can occur in different zones.

By $\mathrm{CT}(\mathcal{P}_2)$ we denote the set of all valid class types. The *class type function* $\mathrm{ctf}_w : \mathrm{CT}(\mathcal{P}_2) \to \mathbb{N}_0$ of a data word w over $\mathcal{P}_2$ maps every class type to the number of classes of w with this class type.

Intuitively, a *profile* is an abstraction of a class type function, where we are only interested, if a class type occurs exactly once, more than once or not at all. Formally, a *profile* is a function $T : \mathrm{CT}(\mathcal{P}_2) \to \{0, 1, *\}$. A class type $\tau = (S, D)$ *occurs* in T if $T(\tau) \neq 0$. We say that a class type function ctf is *compatible* with a profile T (short: $\mathrm{ctf} \models T$) if, for every class type τ either $T(\tau) = \mathrm{ctf}(\tau)$ or $T(\tau) = *$ and $\mathrm{ctf}(\tau) > 1$.

A valid data word w is a *solution* for a profile T (short: $w \models T$) if its class type function ctf_w is compatible with T. A profile that has a solution is called *satisfiable*.

The size of a profile T, denoted by $|T|$, is the number of occurring class types, that is $|T| = |\{\tau \mid T(\tau) \neq 0\}$.

We show in the following proposition that profiles contain all necessary information to decide whether a valid data word satisfies an $\mathrm{FO}^2(\sim)$ formula in data normal form. In particular, we show that either for all solutions w of a profile T it holds $\mathrm{de}(w) \models \varphi$ or for none. In the former case we call T *φ-compatible*. Statement (c) will be one ingredient for our small model property. It says that removing all but exponentially many (in the size of φ) class types from a φ-compatible profile T, still yields a φ-compatible profile. However, (c) is only the first step towards a small model property. In particular it does not guarantee, that the smaller profiles are still satisfiable.

Proposition 12.6

(a) Let φ be a $FO^2(\sim)$-formula in data normal form. For each profile T either for all solutions w to T it holds $w \models \varphi$ or for all solutions w to T it holds $w \not\models \varphi$.

(b) There is an algorithm that on input φ and a satisfiable profile T, decides whether T is φ-compatible in time that is polynomial in $|\varphi|$ and $|T|$.

(c) Let T be a φ-compatible profile. Then there exists a set of class types $\tilde{T}$, such that $|\tilde{T}| \leq 2|\mathcal{T}(\mathcal{P}_2)|$ and every profile T', which fulfills the following conditions is φ-compatible.

- *For all $\tau \in \tilde{T}$ it holds that $T'(\tau) = T(\tau)$.*
- *For all τ with $T(\tau) = 0$ it holds $T'(\tau) = 0$.*

Proof. Let T be a profile and $w \models T$ be a data string compatible with T. It is straightforward to verify, that $w \models \varphi$, if and only if all of the following are true:[3]

(1) For every subformula $\chi = \neg\big(\exists x\ \alpha(x)\big)$ of type (i) in φ it holds that there does not exist a class type $\tau = (D, S)$, such that $\alpha \in D$ and $T(\tau) \neq 0$.

(2) For every subformula $\chi = \neg\big(\exists x \exists y\ \alpha(x)\ \wedge\ \beta(y)\ \wedge\ x \sim y\big)$ of type (ii) in φ it holds that there does not exist a class type $\tau = (D, S)$, such that $\{\alpha, \beta\} \subseteq D$ and $T(\tau) \neq 0$.

(3) For every subformula $\chi = \neg\big(\exists x \exists y\ \alpha(x)\ \wedge\ \beta(y)\ \wedge\ x \not\sim y\big)$ of type (ii) in φ it holds that there does not exist class types $\tau = (D, S)$ and $\tau' = (D', S')$, such that $\alpha \in D$, $\beta \in D'$, $T(\tau) \neq 0$, $T(\tau') \neq 0$ and if $\tau = \tau'$ then $T(\tau) = *$.

(4) For every subformula $\chi = \forall x \exists y\ \alpha(x) \rightarrow \bigvee_i \beta_i(y)\ \wedge\ \delta_i(x, y)$ of type (iii) in φ and every class type $\tau = (D, S)$ with $\alpha \in D$ and $T(\tau) \neq 0$, one of the following is true:

(4a) There exists an i, such that $\delta_i(x, y)$ is $x \sim y$ and $\beta_i \in D$.

(4b) There exists a class type $\tau' = (D', S')$ and an i, such that $\delta_i(x, y)$ is $x \not\sim y$, $\beta_i \in D'$, $T(\tau') \neq 0$, and if $\tau = \tau'$ then $T(\tau) = *$.

[3] The formula types are from Definition 12.4.

Statement (a) follows, as all these conditions only depend on T and not on the concrete string w, which is compatible to T. Statement (b) follows from the fact that all these conditions can be checked in polynomial time.

Towards (c), we choose $\tilde{T}$ in such a way that for every $\mathcal{P}_2$-type α that occurs in T

- there are two class types τ_1 and τ_2 in $\tilde{T}$ such that $T(\tau_1) \neq 0$, $T(\tau_2) \neq 0$, $\alpha \in \tau_1$, and $\alpha \in \tau_2$; or
- there exists only one class type τ such that $T(\tau) \neq 0$ and $\alpha \in \tau$. This τ is in $\tilde{T}$.

It is easy to see, that the size of $\tilde{T}$ is at most $2|\mathcal{T}(\mathcal{P}_2)|$.

To analyze the impact of changes of a profile T with respect to the conditions (1) to (4), we distinguish three different (atomic) changes. A class type $\tau \notin \tilde{T}$ can be *removed* from T, that is $T(\tau)$ can be changed to 0, it can be *decreased*, that is $\mathcal{T}(\tau)$ is changed from $*$ to 1, and it can be *increased*, that is $T(\tau)$ is changed from 1 to $*$. Note that we do not allow to add new class types to a profile.

We now show, that conditions (1) to (4) cannot become false by changing a class type that is not in $\tilde{T}$. It is easy to observe, that the conditions (1) to (3) cannot become false by removing or decreasing a class type. Furthermore the conditions (1), (2) and (4) cannot become false by increasing a class type.

We still have to show, that

(A) Condition (4) cannot become false by removing or decreasing a class type $\tau \notin \tilde{T}$; and

(B) Condition (3) cannot become false by increasing a class type $\tau \notin \tilde{T}$.

We start with (A). Let χ be a subformula and τ be a class type as in (4). Then either (4a) has to be true or (4b) has to be true. If (4a) is true, it cannot become false by removing or decreasing some class type. Therefore, we assume that (4b) is true. However, (4b) cannot become false by removing/decreasing a class type not in $\tilde{T}$, as for each β_i (from (4b)) there are at least two class types τ with $\beta_i \in \tau$ in $\tilde{T}$ (or there is only one, but this is the only such class type in T). We note, that we might need two such class types in $\tilde{T}$, if the first class type τ is exactly the type from (4) and $T(\tau) = 1$.

We continue with (B). We first observe, that Condition (3) can only become false, if we increase a class type τ that has a $\mathcal{P}_2$-type β not occurring in any other class type of T. However, all class types τ with a $\mathcal{P}_2$-type β that does not occur in any other $\mathcal{P}_2$-type of T are contained in $\tilde{T}$. This concludes the proof. □

Proposition 12.6 (a) and (b) almost yield a decision algorithm for $\mathrm{FO}^2(\sim, +1)$. This algorithm could guess a profile T and test whether it is φ-compatible. However, it could happen that T does not have a solution. In that case, Proposition 12.6 does not guarantee a correct answer. Thus, we need an additional algorithm that tests satisfiability of profiles. Then, we can decide satisfiability of φ by guessing T and testing that it is satisfiable and φ-compatible.

12.2.3 Border Strings and Border Profiles

Up to now, we have profiles, which represent the constraints of the formula φ. To decide, whether a profile T is satisfiable, we need to find a data word w that not only fulfills the constraints of T, but also is valid. In particular, data words have two different definitions of zones. First, zones are defined by data values and second, $\mathcal{P}_2$-types encode, where one zone ends and another zone begins. To find a solution for a profile T, we need to find a valuation of some string over $\mathcal{T}(\mathcal{P}_2)$ with data values, such that the zone information inside the $\mathcal{P}_2$-types is consistent.

Towards this goal, we will introduce border strings, where one position in the border string represents a complete zone. A position in the border string has a data value and a label describing the borders of the represented zones. This label will be called the border type.

Just as we have introduce class types, that encode which cP_2-types occur in a class, we will introduce class border types, that encode which border types occur in a class. And just as we have introduced profiles, that encode which class types occur in a data string, we will introduce border profiles that encode which class border types occur in a class.

Let w be a valid data word over $\mathcal{P}_2$. Recall that a zone of w is a maximal substring in which all positions carry the same data value. Let c be a class in w and let $\tau = (S, D)$ be its class type. Towards the definition of border types and border strings, we first look at validity of individual zones.

For each zone z of c, the *zone type* $\tau = (D_z, S)$ of z is the set S of sheep types of c and the set of dog types D_z that occur in z. A zone z is called a *dog zone* if $D_z \neq \emptyset$ otherwise a *sheep zone*. It should be emphasized that the set D_z only includes the dog types of the zone itself, while the set S of sheep types includes all sheep types of the entire class. We write $z \models \tau$ if a zone z satisfies a zone type τ, that is all sheep types occurring in z are from S and z contains exactly the dog types of D_z.

Clearly, all sets D_z of a class are pairwise disjoint and together they contain all types from D, that is, they induce a partition of D. We note also that each class c can have at most $|\mathcal{T}(\mathcal{P})|$ dog zones.

In the following, we describe how to test whether a profile is satisfiable. We therefore define validity of zones to enforce local consistency inside zones and border strings, which will be used to enforce local consistency at zone borders.

When talking about (strings of) a single zone, we usually omit the data values of the positions, as they have to be equal anyway. Note that the following conditions for valid zones are directly derived from the definition of a valid data string.

A zone $z = \alpha_1 \dots \alpha_n$ is *valid*, if the following conditions are met.

- α_1 does not contain $p_=^{-1}$ and α_n does not contain $p_=^{+1}$;
- for $i < n$
  - α_i contains $p_=^{+1}$ and does not contain $p_\lhd$; and
  - α_i contains a proposition p^{+1}, if and only if α_{i+1} contains p;
- for $i > 1$

- α_i contains $p_=^{-1}$ and does not contain $p_\triangleright$; and
- α_i contains a proposition p^{-1}, if and only if α_{i-1} contains p;

The *border type* $\beta = (\alpha_1, \alpha_n, \eta) \in B = \mathcal{T}(\mathcal{P}_2) \times \mathcal{T}(\mathcal{P}_2) \times \{\eta_D, \eta_S\}$ of a zone $z = \alpha_1 \dots \alpha_n$ consists of its leftmost and rightmost $\mathcal{P}_2$ type and the information whether the class contains any dog type (denoted by η_D) or only sheep types (denoted by η_S). We denote the set of all border types by B and the subsets of all dog and sheep border types by B_D and B_S respectively. We write $z \models \beta$, if a zone satisfies a border type.

A border type β is compatible with a zone type τ, if there exists a valid zone, such that $z \models \beta$ and $z \models \tau$.

Proposition 12.7 *Let τ be a zone type and β be a border type. It can be checked nondeterministically in time exponential in $|\mathcal{P}|$, whether τ is compatible with β.*

Proof. Let $\mathcal{A}_\beta$ be the minimal DFA that accepts all valid zone strings z, which conform to the border type β. It is easy to see, that $\mathcal{A}_\beta$ has at most exponential size in $|\mathcal{P}|$, as it only needs to remember the last read symbol.

With e_α we denote the indicator function for $\alpha \in \mathcal{T}(\mathcal{P})$. We define the linear set Lin_τ according to the zone type $\tau = (D, S)$.

$$\mathrm{Lin}_\tau = \left\{ \sum_{\alpha \in D} e_\alpha + \sum_{\alpha \in S} i_\alpha e_\alpha \,\middle|\, i_\alpha \in \mathbb{N}_0 \right\}.$$

The first sum encodes, that every dog type of τ should occur exactly once and the second sum encodes that every sheep type of τ may occur arbitrarily often.

It is easy to see, that τ is compatible with β, if and only if $L(\mathcal{A}_\beta, \mathrm{Lin}_\tau) \neq \emptyset$.

As the automaton $\mathcal{A}_\beta$ is of at most exponential size in $|\mathcal{P}_2|$, the Parikh image of $L(\mathcal{A}_\beta)$ has coefficients of at most exponential size (see, e.g., [To10, Proposition 4.3]). Thus, it can be tested non-deterministically in exponential time (in $|\mathcal{P}_2|$) whether there is a zone string $v \in L(\mathcal{A}_\beta, \mathrm{Lin}_\tau)$. □

A *border string* is a string over the alphabet $\Gamma = B \times (\mathrm{dom} \cup \{\perp\})$, where each position has a border type and might have a data value (from dom). However, we allow positions to be labeled with $\perp$ to indicate that they have an unknown data value.

A border string $v = (\alpha_{1\triangleright}, \alpha_{\triangleleft 1}, \eta_1, d_1) \cdots (\alpha_{n\triangleright}, \alpha_{\triangleleft n}, \eta_n, d_n)$ is *valid*, if the following conditions hold.

- $\alpha_{1\triangleright}$ contains $p_\triangleright$ and $\alpha_{\triangleleft n}$ contains $p_\triangleleft$; and
- For every pair of adjacent symbols $(\alpha_{i\triangleright}, \alpha_{\triangleleft i}, \eta_i, d_i)$ and $(\alpha_{i+1\triangleright}, \alpha_{\triangleleft i+1}, \eta_{i+1}, d_{i+1})$
  - $\alpha_{\triangleleft i}$ contains a proposition p^{+1} if and only if $\alpha_{i+1\triangleright}$ contains p; and
  - $\alpha_{\triangleleft i}$ contains a proposition p if and only if $\alpha_{i+1\triangleright}$ contains p^{-1}
  - either $d_i \neq d_{i+1}$ or $d_i = d_{i+1} = \perp$.

Again, the constraints are directly derived from validity of data strings, i.e., if a data string is valid, then the corresponding border string (as defined below) is also valid.

With a data word $w = (z_1, d_1) \dots (z_m, d_n)$ over $\mathcal{P}_2$, where $z_1, \dots, z_n$ are the zones of w and $d_1, \dots, d_n$ are their data values, we associate the border string

$$\Gamma(w) = \Big(\text{first}(z_1), \text{last}(z_1), \eta_1, d_1\Big) \cdots \Big(\text{first}(z_n), \text{last}(z_n), \eta_n, d_n\Big),$$

where $\text{first}(z_i)$ and $\text{last}(z_i)$ are the first and last $\mathcal{P}_2$ type of a zone and $\eta_i = \eta_D$, if and only if the zone z_i contains any dog type.

Using these definitions, we now define class border types. As class types describe how often some $\mathcal{P}_2$-type α occurs in a class c, class border types describe how often some border type β occurs in c.

A *class border type* $\tau_b : B \to \{0, 1, \dots, |\mathcal{T}(\mathcal{P}_2)|, *\}$ maps each border type to a number of occurrences, where $*$ means arbitrarily often. A class border type is *valid*, if all sheep border types are either mapped to 0 or $*$ and no dog border type is mapped to $*$. We denote the set of all valid class border types with $\text{BCT}(\mathcal{P}_2)$.

Note that class types cannot be mapped to class border types in a one to one fashion. This is due to the fact, that there may be different border types for the same zone type (and vice versa).

A valid class border type τ_b is compatible with a class type $\tau = (D, S)$, if there exist zone types $\tau_1 = (D_1, S), \dots, \tau_n = (D_n, S)$ and border types $\beta_1, \dots, \beta_n$, such that

- $D = D_1 \cup \dots \cup D_n$;
- $D_i \cap D_j = \emptyset$ for $i \neq j$;
- β_i is compatible with τ_i for $i \in [1, n]$; and
- the frequencies of border types in the sequence $\beta_1, \dots, \beta_n$ match the frequencies required by τ_b

We also define border profiles analogously to profiles. Intuitively, the differences between profiles and border profiles are:

- a border profile talks about class border types instead of class types; and
- a border profile has more detailed information about the frequency of class border types.

The latter is necessary, as there might be different class types, which all have to occur exactly once and are compatible with the same class border type.

Formally, a *border profile* is a function $T_b : \text{BCT}(\mathcal{P}_2) \to \{0, 1, \dots, 3|\mathcal{T}(\mathcal{P}_2)|, *\}$. A class border type τ_b occurs in T_b if $T_b(\tau_b) \neq 0$. A valid border string v, such that v has only explicit data values (i.e. no $\perp$ markings), is a *solution* to T_b, if for every class border type τ_b such that $T_b(\tau_b) \neq *$, it holds that there are exactly $T_b(\tau_b)$ classes with class border type τ_b in v and for every class border type τ_b such that $T_b(\tau_b) = *$, there are more than $3|\mathcal{T}(\mathcal{P}_2)|$ many classes with class border type τ_b in v. A data string w is a solution to

T_b, if $\Gamma(w)$ is a solution to T_b. A border profile T_b is *compatible* with a profile T, if for each valid border string v, such that v has only explicit data values and v is a solution to T_b, there exists a valid data string w, such that $\Gamma(w) = v$ and w is a solution to T. We denote the unique border profile of a data string w with $T_b(w)$.

The following proposition looks at the relationship between border profiles (of border strings) and profiles (of data strings). To convert (valid) border strings without unknown data values to (valid) data strings, we replace border types with zone strings. A border type β with data value d in a border string v is replaced by a zone string $\alpha_1 \dots \alpha_n \in \mathcal{T}(\mathcal{P}_2)^*$ by replacing (β, d) in v with $(\alpha_1, d) \dots (\alpha_n, d)$. Replacing all border types in v with compatible zone strings yields a data string w. The data string w is valid, if v is valid and all zone strings are valid.

Proposition 12.8

(a) For each profile T and each solution w to T, the border profile $T_b(w)$ is compatible with T.

(b) There exists a non-deterministic algorithm that given a profile T and a satisfiable border profile T_b decides in time polynomial in the size of T, T_b, and $\mathcal{T}(\mathcal{P}_2)$ whether T and T_b are compatible.

Proof. Towards (a), let v be the border string of w and u be a valid border string (without $\perp$ markings) that is a solution for the border profile $T_b = T_b(w)$.

We will use the following fact throughout the proof:

(F) Let c be a class of u, τ_b be the border type of c and τ be a class type compatible with τ_b, then the border types of c can be replaced by zone strings, such that the class type of c is τ.

We note that (F) follows from the definition of compatibility of class border types with class types. Let $\beta_1, \dots, \beta_n$ be the border types in c and τ be a class type compatible with τ_b. Then, by definition of compatibility, there exist zone types $\tau_1, \dots, \tau_n$ such that β_i is compatible with τ_i and the class type of a class consisting exactly of n zones with zone types $\tau_1, \dots, \tau_n$ is τ.

We let C_u be all classes of u and C_v be all classes of v. For each border type τ_b that occurs in T_b (and therefore in v and u), $C_u^{\tau_b}$ are the classes in u with border type τ_b and $C_v^{\tau_b}$ are the classes in v with border type τ_b. We note, that the sets $C_u^{\tau_b}$ are a partition of C_u and the sets $C_v^{\tau_b}$ are a partition of C_v.

For each class border type τ_b that occurs in T_b, we let $\mathrm{f}_{\tau_b} : C_v^{\tau_b} \to C_u^{\tau_b}$ be a partial injective function, such that

- for each class type τ occurring exactly once in $C_v^{\tau_b}$ (for some τ_b), the class $c \in C_v^{\tau_b}$ with class type τ is mapped to some class c' in $C_u^{\tau_b}$; and
- for each class type τ occurring at least twice in $C_v^{\tau_b}$ (for some τ_b), at least two classes with class type τ are mapped to classes in $C_u^{\tau_b}$.

We note that such such functions always exist. In the case that $|C_u^{\tau_b}| = |C_v^{\tau_b}|$, we can use any bijection between $C_u^{\tau_b}$ and $C_v^{\tau_b}$. In the other case $|C_u^{\tau_b}| > 2|\mathcal{T}(\mathcal{P}_2)|$ and $|C_v^{\tau_b}| > 2|\mathcal{T}(\mathcal{P}_2)|$ hold by the definition of border profiles. We note that $T_b(\tau_b) = *$ in this case, as there are two solutions to T_b, where the count of classes with class border type τ_b differs.

Furthermore, we can conclude that there is a class type τ compatible with τ_b such that $T(\tau) = *$ in the case that $|C_u^{\tau_b}| \neq |C_v^{\tau_b}|$. As there are only $|\mathcal{T}(\mathcal{P}_2)|$ many different class types, there has to be a class type τ occurring more than once in $|C_v^{\tau_b}|$, as $|C_v^{\tau_b}| > 2|\mathcal{T}(\mathcal{P}_2)|$.

We let $\mathrm{f} : C_v \to C_u$ be the partial injective function that results from combining all functions f_{τ_b}. For any class c in the image of f, we replace border types with zone strings, such that the class type of c equals the class type of $\mathrm{f}^{-1}(c)$. For all classes c of border type τ_b not in the image of f (in the case $|C_u^{\tau_b}| \neq |C_v^{\tau_b}|$) we replace border types with zone strings, such that the class type of c is τ, where τ is a class type compatible with τ_b, such that $T(\tau) = *$. According to (F) such a replacement is possible, as by definition of f the border types of a class c with border type τ_b shall always be replaced with zone strings such that the resulting class type τ is compatible with τ_b.

The resulting data string w' is valid, as u is valid. Furthermore it is a solution to T, as each class type, that appears exactly once in w also appears exactly once in w' and each class type that appears at least twice in w also appears at least twice in w' by the definition of f. Furthermore all classes not in the image of f have a class type τ with $T(\tau) = *$.

This concludes the proof of (a). We want to emphasize, that we did not use the fact that v is a valid border string, i.e., the validity of w follows already from the validity of u and the validity of all used zone strings. In the following proof of (b), we need that the argumentation also works, if v contains neighboring positions with identical data values.

It remains to prove (b). We show that the following nondeterministic algorithm correctly decides compatibility in polynomial time:

1. Compute a border string $v = (\beta_1, d_1) \dots (\beta_n, d_n)$ over $B \times \mathrm{dom}$, such that v satisfies the border profile T_b according to frequencies of class border types.

2. For each position i of v guess a zone string z_i of the border type β_i such that z_i conforms to β_i.

3. Accept if w satisfies T.

We note that v has no $\perp$ markings, i.e., the classes of v are well defined. However, v is not necessarily valid, as neighboring positions can have the same data value.

It is easy to see, that the algorithm works in the given time constraints. Computing a string v as described above can be easily done deterministically in polynomial time. According to Proposition 12.7, compatibility of border types with zone types can be checked in polynomial time, as well as testing whether w is a solution to T.

We have to show that the algorithm correctly decides compatibility of profiles T with satisfiable border profiles T_b. First we show, that if the algorithm accepts, then the guessed profile T is compatible with T_b and therefore the answer is correct.

Let u be a valid border string with $u \models T_b$ and v and w be the border string and data string of the algorithm. With the argumentation of (a), it follows, that there exists a valid data string w' such that $\Gamma(w') = u$ and $w' \models T$. We remember, that the argumentation of (a) does not require that v is valid. We can conclude that T is compatible with T_b.

For the other direction, we have to show, that if T is compatible to T_b and T_b is satisfiable, then the algorithm has an accepting run. By definition of compatibility, for any border string $v = \beta_1 \dots \beta_n$ that satisfies T_b, there exists zone strings $z_1, \dots, z_n$, such that the string $w = (z_1, d_1) \dots (z_n, d_n)$ is a valid data string satisfying T. The algorithm can guess $z_1, \dots, z_n$ and thus accept. □

12.2.4 Capturing Border Profiles with Semi-Linear Sets

We next describe the linear constraints that we derive from a border profile T_b. For technical reasons that become apparent later, we need to take care of up to ℓ classes, which have to get an *explicit data value*, where ℓ is some natural number.

Let therefore ℓ be a natural number and Γ_ℓ be the alphabet $B \times ([1, \ell] \cup \{\bot\})$, where we assume $[1, \ell]$ to be a subset of dom. With other words, Γ_ℓ is an alphabet for border strings, where up to ℓ classes can have explicit class types.

For a data string w and some natural number ℓ, let $\Gamma_\ell(w)$ of w be the border string of w, where each data value outside $[1, \ell]$ is mapped to $\bot$.

The border automaton $\mathcal{A}_\ell = (\Gamma_\ell, Q, \delta, q_0, Q_F)$ accepts a border string v over Γ_ℓ, if v is valid. It is easy to see, that $\mathcal{A}_\ell$ has at most $|\Gamma_\ell|$ many states, as it only needs to remember the last seen symbol. Note that Γ_ℓ is of exponential size in $\mathcal{P}_2$ and linear size in ℓ.

We will first describe semi-linear sets, that capture one class of some class border type τ_b. Intuitively these sets can be seen as capturing a profile with only one class border type, which is occurring exactly once. Using these semi-linear sets, we define a semi-linear set for a border profile T_b as a linear combination, capturing the multiplicities of class types as denoted by T_b.

For each border type β and each $i \in [1, \ell] \cup \{\bot\}$, let $e_{\beta,i}$ be the indicator function for $(\beta, i) \in \Gamma_\ell$, that is, it maps (β, i) to 1 and all other symbols to 0.

For a class border type τ_b and a data value $i \in [1, \ell] \cup \{\bot\}$ we define the linear set

$$\mathrm{Lin}(\tau_b, i) = \left\{ \sum_{\beta \in B_D} \tau_b(\beta) \cdot e_{\beta,i} + \sum_{\beta \in \tau_b^{-1}(*)} k_\beta \cdot e_{\beta,i} \;\middle|\; k_\beta \in \mathbb{N}_0 \text{ for } \beta \in \tau_b^{-1}(*) \right\},$$

where τ_b^{-1} is the inverse of τ_b. Especially $\tau_b^{-1}(*)$ is the set of all sheep types occurring in τ_b.

We note that the first sum in the definition of $\mathrm{Lin}(\tau_b, i)$ evaluates to one vector, which describes exactly the dog border types of a class, whereas the second sum described the sheep border types. Remember that sheep border types may occur arbitrarily often.

On top of the sets $\mathrm{Lin}(\tau_b, i)$, we will define the semi-linear set $\mathrm{SLin}_\ell(T_b)$ for a border profile T_b. First we will define a linear set $\mathrm{Lin}(T_b)$, which does not respect the classes

with explicit data value. It will not be used in the remainder of the proof, but it helps clarifying things a bit.

$$\mathrm{Lin}(T_b) = \left\{ \sum_{\tau_b \notin T_b^{-1}(*)} T_b(\tau_b) \cdot \mathrm{Lin}(\tau_b, \bot) \;+ \sum_{\tau_b \in T_b^{-1}(*)} k_{\tau_b} \cdot \mathrm{Lin}(\tau_b, \bot) \;\middle|\; k_{\tau_b} > 3|\mathcal{T}(\mathcal{P}_2)| \right\}$$

The first sum applies to class border types occurring only a "few" times, whereas the second sum applies to class border types occurring frequently.

It follows directly from the definition of $\mathrm{Lin}(T_b)$, that to every border string v with $\mathrm{parikh}(v) \in \mathrm{Lin}(T_b)$, we can assign data values in such a way, that the resulting fully valued border string v' conforms to the border profile T_b. Note that we do not require v' to be a valid border string here. The border string v' can be inconsistent in two different ways:

(1) The border string might not be consistent with respect to $\mathcal{P}_2$ types.

(2) The border string might have two neighboring positions with the same data value.

We deal with the first type of inconsistencies, by intersecting the language induced by the linear set with the language of $\mathcal{A}_\ell$. The automaton ensures, that no such inconsistencies occur.

For the second type of inconsistencies, we will show in Proposition 12.9, that we can always repair these inconsistencies by exchanging data values of positions with the same border type for all border types that occur frequently. However for seldom border types, this does not work. Therefore, we allow some (up to ℓ) classes to have a pre-determined data value. The automaton $\mathcal{A}_\ell$ already ensures, that there are no inconsistencies of the second type for these pre-determined data values. It remains to allow these pre-determined data values in the (semi-)linear set.

Let therefore $T_\ell = \tau_1, \ldots, \tau_\ell$ be a sequence of ℓ class border types, such that for each τ_i it holds that $T_b(\tau_i) \neq *$. We note that T_ℓ does not need to be a set, as a class border type can occur multiple times in T_ℓ. We define $\mathrm{SLin}(T_b, T_\ell) =$

$$\left\{ \sum_{i=1}^{\ell} \mathrm{Lin}(\tau_i, i) + \sum_{\tau_b \in T_b^{-1}(*)} k_{\tau_b} \cdot \mathrm{Lin}(\tau_b, \bot) \;\middle|\; \begin{array}{ll} k_{\tau_b} > 3|\mathcal{T}(\mathcal{P}_2)| & T_b(\tau_b) = * \\ k_{\tau_b} + |\{i \mid \tau_i = \tau_b\}| = T_b(\tau_b) & T_b(\tau_b) \neq * \end{array} \right\}$$

We note that in this case, the first sum takes care of all classes with explicit data values and the second sum is for all other classes. The semi-linear set $\mathrm{SLin}_\ell(T_b)$ is the union over all sets $\mathrm{SLin}(T_b, T_\ell)$ with $T_\ell = \tau_1, \ldots, \tau_\ell$ for class border types $\tau_1, \ldots, \tau_\ell$, such that for each border type β it holds that either

- β does not occur in T_b, i.e., $T_b(\tau_b) = 0$ for each τ_b with $\tau_b(\beta) > 0$; or
- β occurs with at least $3|\mathcal{T}(\mathcal{P}_2)|$ different data values in T_ℓ, i.e., $|\{\tau_i \mid \tau_i(\beta) \neq 0\}| \geq 3|\mathcal{T}(\mathcal{P}_2)|$.

We note that this definition ensures that each border type with only a few data values occurs only with explicit data values. The following proposition shows, that the semi-linear sets are sound if ℓ is chosen big enough, i.e., large enough that every such border type can get explicit data values.

Proposition 12.9 *Given a border profile T_b, $L(\mathcal{A}_\ell, SLin_\ell(T_b)) \neq \emptyset$ if and only if T_b is satisfiable, where $\ell = 3|\mathcal{T}(\mathcal{P}_2)|^2$.*

Proof. We start with the direction $L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_b)) \neq \emptyset$ implies T_b is satisfiable. Let therefore T_b be a border profile, ℓ be as defined in the proposition statement, and $v = (\beta_1, j_1) \cdots (\beta_n, j_n)$ be a border string such that $v \in L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_b))$ and for every border type β it holds that

- $(\beta, \perp)$ does not occur in v;
- β is a dog border type and $(\beta, \perp)$ occurs more than $3|\mathcal{T}(\mathcal{P}_2)|$ times in v; or
- β is a sheep border type and T_b ensures, that β occurs in at least 3 different classes.

By definition of $\mathrm{SLin}(T_b, T_\ell)$, such a string always exists, as $\mathrm{SLin}(T_b, T_\ell)$ enforces that each border type β that occurs with less than $3|\mathcal{T}(\mathcal{P}_2)|$ different data values only has explicit data values.

To show that there exists a solution to T_b, we will assign data values to positions marked with $\perp$ in v. The assignment of data values, will be done in 3 steps:

- First, we do a provisional assignment of data values to all dog positions of the border string. This assignment should fulfill all frequency constraints of T_b.
- Second, among the dog positions, we exchange data values until there are no neighboring dog positions left with the same data value.
- Third, we assign data values to the sheep positions.

We assign data values (from $\mathrm{dom} \setminus [1, \ell]$) to all dog positions of v marked with $\perp$, such that we get a border string u which might have inconsistencies of type (2), but is otherwise compatible with T_b. As already noted, this is always possible, due to the definition of $\mathrm{SLin}_\ell(T_b)$ and $\mathcal{A}_\ell$.

Observe, that not assigning data values to sheep positions cannot make an (otherwise compatible) border string incompatible to T_b as, by definition of sheep, the corresponding border types are not required to appear in the classes.

Now, we will ensure that there are no neighboring dog positions with the same data value. Note that all neighboring positions which have the same data value must be dog positions, as sheep positions either have no data value yet or $\mathcal{A}_\ell$ ensures that their data values are consistent.

We correct data values of dog border types inductively from left to right. To this end, let i be the first position, such that i has the same data value d as $i - 1$.

Let $j \neq i$ be a position with border type $\beta_j = \beta_i$ which has a data value $d' \neq d$, such that none of the neighbor positions of j has the value d. As each class has at most

$|\mathcal{T}(\mathcal{P}_2)|$ dog border types altogether, there can be at most $|\mathcal{T}(\mathcal{P}_2)| - 1$ other positions with data value d. Remember that no sheep position can have data value d, yet. Each of these positions can be a neighbor to at most 2 other positions. As there are at least $3|\mathcal{T}(\mathcal{P}_2)| - 1$ candidate positions to choose from, it follows that one of the candidates neither has data value d nor has a d-valued neighbor. We exchange the data values of j and i.

Finally, it is easy to assign data values to sheep positions, as there are at least 3 different data values available for each sheep position. As each sheep position has at most 2 neighbors, we can always assign a data value different from the neighboring data values.

We continue with the proof of the other direction, i.e., T_b is satisfiable implies that $L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_b)) \neq \emptyset$. Let v be a border string that is a solution to T_b. We permute the data values of v such that every border type β such that β has less than $3|\mathcal{T}(\mathcal{P}_2)|$ different data values only occurs with data values, which are at most ℓ and every other border type β occurs with at least $3|\mathcal{T}(\mathcal{P}_2)|$ different data values $\leq \ell$. We have chosen ℓ large enough that such a permutation is always possible. We note, that permuting the data values cannot destroy the validity of v.

We choose $T_\ell = \tau_1, \ldots, \tau_\ell$, where τ_i is the class border type of the class with data value i. We let v_ℓ be the border string v, where each data value larger than ℓ is replaced by $\perp$. It is not hard to verify, that v_ℓ is in $\mathrm{SLin}(T_b, T_\ell)$ and therefore in $\mathrm{SLin}_\ell(T_b)$. As v is a valid border string, v_ℓ is in $L(\mathcal{A}_\ell)$. We can conclude that $v \in L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_b))$. □

12.2.5 Small Model Property

Now, we could show that satisfiability of $\mathrm{FO}^2(\sim, +1)$ is decidable in 2-NEXPTIME. To show that satisfiability is also decidable in NEXPTIME, we need the following small model property, which basically says, that if there is a satisfying string, than there always is such a string with at most exponentially many different class types.

Proposition 12.10 *If there exists a φ compatible profile T, a T compatible border profile T_b and a border string $v \in L(\mathcal{A}_\ell, SLin_\ell(T_b))$ with $\ell = 3|\mathcal{T}(\mathcal{P}_2)|^2$ then there also exists a φ compatible profile T' and a T' compatible border profile T'_b such that $v \in L(\mathcal{A}_\ell, SLin_\ell(T'_b))$, $|T'| \leq 2^{4|\mathcal{P}_2|+1}$, and $|T'_b| \leq 2^{4|\mathcal{P}_2|+1}$.*

Proof. We first give an intuitive idea of the proof. The question whether there is a string v, such that $v \in L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_b))$ reduces to the question whether $\mathrm{parikh}(\mathcal{A}_\ell) \cap \mathrm{SLin}_\ell(T_b) \neq \emptyset$, which reduces to satisfiability of linear equation systems.

The corresponding linear equation system has roughly one equation for each possible border type and one variable for each class border type occurring in the profile. Remember that the number of possible border types (and therefore the number of equations) is exponential in $|\mathcal{P}_2|$, whereas the number of used class border types (and therefore the number of variables) can be doubly exponential in $|\mathcal{P}_2|$. Therefore, if there are many different class border types, the equation system is heavily under-specified. We will show, that we can reduce the number of variables/class border types to be exponential.

Now we continue with the proof. Let T be a φ-compatible profile, T_b be a compatible border profile, $\mathrm{Lin}(T_b, T_\ell)$ be a linear set from $\mathrm{SLin}_\ell(T_b)$ and $v \in L(\mathcal{A}_\ell, \mathrm{Lin}(T_b, T_\ell))$ be a valid border string.

Let $\tilde{T}$ be the set of class types as in Proposition 12.6 (c), i.e., $|\tilde{T}| \leq 2|\mathcal{T}(\mathcal{P}_2)|$ and every profile T', which fulfills the following conditions is φ-compatible.

- For all $\tau \in \tilde{T}$ it holds that $T'(\tau) = T(\tau)$.
- For all τ with $T(\tau) = 0$ it holds $T'(\tau) = 0$.

We assume that there are at most $3|B_S|$ many classes in w that have sheep zones, due to the following observation: Let z_1 and z_2 be two zones of w such that

- z_1 and z_2 have the same sheep border type but different data value; and
- the neighbor zones of z_2 both have a different data value than z_1.

Then the data string derived from w by replacing z_2 with a copy of z_1 and changing the data value of z_2 to that of z_1 is valid and satisfies φ. Remember that sheep are allowed to occur arbitrary often in a class.

We now divide w into two subsequences w_1 and w_2. The intuitive idea is, that w_1 is the subsequence containing all the classes we do not want to tamper with and w_2 is the subsequence of w, where we are allowed to change the frequencies of class types without "leaving" the profile T according to Proposition 12.6 (c). We will use this freedom to find a profile with not too many class types, which allows the same border string v.

The subsequence w_1 contains classes of w as follows:

- every class with a data value from $[1, \ell]$;
- every class containing sheep zones;
- every class with a class type τ, such that $\tau \in \tilde{T}$ and $T(\tau) = 1$; and
- two classes of every class type τ, such that $\tau \in \tilde{T}$ and $T(\tau) = *$.

The subsequence w_2 contains all other classes. Let v_1 and v_2 be the corresponding subsequences of the border string v. We note that by definition of w_2, in v_2 there are no explicit data values.

It is easy to see, that $\mathrm{parikh}(v_2)$ can be written as

$$\mathrm{parikh}(v_2) = \sum_{i=1}^{m} k_i \, \mathrm{f}_i,$$

where

- m and every k_i is a natural number
- every f_i is a function $\mathrm{f}_i : \Gamma \to [0, |2^{\mathcal{P}_2}|]$, which maps dog border types marked with $\perp$ to a number of occurrences

With $\mathcal{F}$, we denote the set $\{\mathrm{f}_1, \ldots, \mathrm{f}_m\}$.

We note that each function $\mathrm{f} \in \mathcal{F}$ corresponds exactly to the first sum in the definition of $\mathrm{Lin}(\tau_b, \perp)$ for some class border type τ_b. The second sum in the definition of $\mathrm{Lin}(\tau_b, \perp)$ only refers to sheep border types, which are not present in v_2. We can restrict the range of the functions $\mathcal{F}$ to $[0, |\mathcal{T}(\mathcal{P}_2)|]$, as each class has at most $|\mathcal{T}(\mathcal{P}_2)|$ dog zones. With other words, f_i encodes exactly the dog border types of j_i classes, each of these classes has the same class border type τ_b^i and some class type τ. We can conclude that for every $i \in [1, m]$, there exists a class type τ_i such that $T(\tau_i) \neq 0$, $\tau_i \notin \tilde{T}$, and τ_i is compatible to τ_b^i.

Let $T_b^1(\tau_b)$ denote the number of classes in v_1 with class border type τ_b and

$$T_b^2(\tau_b) = \begin{cases} k_i & \text{if } \mathrm{f}_i \in \mathcal{F} \text{ s.t. } \mathrm{f}_i \text{ corresponds to the first sum in the definition of } \tau_b \\ 0 & \text{otherwise} \end{cases}$$

We define the border profile T_b' as

$$T_b'(\tau_b) = \begin{cases} * & \text{if } T_b^1(\tau_b) + T_b^2(\tau_b) > 3|\mathcal{T}(\mathcal{P}_2)| \\ T_b^1(\tau_b) + T_b^2(\tau_b) & \text{otherwise} \end{cases}$$

It is easy to see that the number of class border types in T_b' is bounded by $3|B_S| + |\tilde{T}| + \ell + m$. By construction of T_b', we know that $v \in L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_b'))$. Therefore T_b is satisfiable.

Let $\Lambda : \mathbb{N}_0 \to \{0, 1, *\}$ be the function that maps every number greater than one to $*$ and $\tau(w_1)$ be the number of occurrences of τ in w_1 for each class type τ. We define T' as

$$T'(\tau) = \begin{cases} \Lambda(\tau(w_1)) & \text{if } \tau \notin \{\tau_1, \ldots, \tau_m\} \\ \Lambda(\tau(w_1) + k_i & \text{if } \tau = \tau_i \text{ with } i \in \{1, \ldots, m\} \end{cases}$$

It is easy to see that $|T'|$ is bounded by $3|B_S| + |\tilde{T}| + \ell + m$. Furthermore, by construction T' is compatible to T_b'. We note that class types from $\tilde{T}$ only occur in w_1 and therefore $T'(\tau) = T(\tau)$ for every class type $\tau \in \tilde{T}$. Furthermore $T'(\tau) = 0$ if $T(\tau) = 0$ by choice of $\{\tau_1, \ldots, \tau_m\}$.

We will show that $|\mathcal{F}|$ can be bounded by $2^{4|\mathcal{P}_2|}$, by showing that $\mathrm{parikh}(v_2)$ can always be defined as a linear combination of at most $2^{4|\mathcal{P}_2|}$ different functions from $\mathcal{F}$. This shows the proposition statement, as

$$3|B_S| + |\tilde{T}| + \ell + 2^{4|\mathcal{P}_2} \leq 3 \cdot 2^{2|\mathcal{P}_2|} + 3 \cdot 2^{|\mathcal{P}_2|} + 3 \cdot 2^{2|\mathcal{P}_2|} + 2^{4|\mathcal{P}_2|} \leq 2^{4|\mathcal{P}_2|+1}.$$

Towards contradiction, we assume that $m > 2^{4|\mathcal{P}_2|}$, none of the j_i is zero and $\mathrm{parikh}(v_2)$ cannot be written as a linear combination of functions from $\mathcal{F}$, where one coefficient is zero.

For every subset F of $\mathcal{F}$, we compute the function $\mathrm{f}_F = \sum_{\mathrm{f} \in F} \mathrm{f}$. Note that the set $\{\mathrm{f}_F \mid F \subseteq \mathcal{F}\}$ contains at most $(m \cdot |2^{\mathcal{P}_2}| + 1)^{|2^{\mathcal{P}_2} \times 2^{\mathcal{P}_2}|} < 2^m$ different functions, due to

Algorithm 12 Test satisfiability of $\mathrm{FO}^2(\sim, +1)$-formulas

```
1: function SAT(φ)
2:     Compute φ′ in Scott normal form s.t. φ′ is satisfiable iff φ is satisfiable
3:     Compute φ″ in data normal form s.t. φ″ is satisfiable iff φ′ is satisfiable
4:     Guess string profile T with at most 2^{4|P_2|+1} class types
5:     if T is not compatible with φ then reject
6:     Guess border profile T_b with at most 2^{4|P_2|+1} class border types
7:     if T_b is not compatible with T then reject
8:     Let ℓ be 3|T(P_2)|^2
9:     Guess linear set Lin from SLin_ℓ(T_b)
10:    if L(A_ℓ, Lin) is nonempty then accept
11:    reject
```

the restricted range of the functions in $\mathcal{F}$. However there are 2^m subsets of $\mathcal{F}$. By the pigeonhole principle, we can conclude that there are two different subsets F_1 and F_2 of $\mathcal{F}$, such that $\mathrm{f}_{F_1} = \mathrm{f}_{F_2}$. We may safely assume that F_1 and F_2 are disjoint, as else we could reason over $F_1 \setminus F_2$ and $F_2 \setminus F_1$.

Let $k = \min\{k_i \mid \mathrm{f}_i \in F_1 \cup F_2\}$ be the smallest coefficient of functions in F_1 and F_2 and i be the corresponding index. W.l.o.g. we assume, that $\mathrm{f}_i \in F_1$. As $\mathrm{f}_{F_1} = \mathrm{f}_{F_2}$, we can conclude that

$$\mathrm{parikh}(v_2) = \sum_{\mathrm{f}_j \in \mathcal{F} \setminus (F_1 \cup F_2)} k_j \, \mathrm{f}_j + \sum_{\mathrm{f}_j \in F_1} (k_j - k) \, \mathrm{f}_j + \sum_{\mathrm{f}_j \in F_2} (k_j + k) \, \mathrm{f}_j \, .$$

Note that f_i has the coefficient 0 and all other coefficients are greater or equal to zero. This is a contradiction to our assumption. We can conclude that we can always define $\mathrm{parikh}(v_2)$ as a linear combinations of at most $2^{4|\mathcal{P}_2|}$ different functions.

This concludes the proof. □

12.2.6 Putting Everything Together

Now we can continue with the proof of Theorem 12.2.

Statement of Theorem 12.2 The satisfiability problem for $\mathrm{FO}^2(\sim, +1)$ formulas over data words with propositions is decidable in NEXPTIME.

Proof. For convenience, we depict Algorithm 11 again as Algorithm 12. The only differences are the added size restrictions for the profiles according to Proposition 12.10.

It is easy to verify that Algorithm 12 runs in nondeterministic exponential time, as the size of all representations (Scott normal form, data normal form, profile, border profile) of the formula are bounded exponentially in φ and compatibility can be checked in polynomial time in the size of the representations (Propositions 12.6 and 12.8).

It remains to show that Algorithm 12 is correct. Let therefore w be a solution to φ and w' be a valid data word over $\mathcal{T}(\mathcal{P}_2)$ such that $\mathrm{de}(w') = w$. Let T be the profile of w' and

T_b be the border profile of w'. By definition T is compatible to φ and by Proposition 12.8a, T_b is compatible to T. By Proposition 12.9, we get that $L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_B))$ is nonempty and therefore, there exists a linear set Lin in $\mathrm{SLin}_\ell(T_b)$ such that $L(\mathcal{A}_\ell, \mathrm{Lin})$ is nonempty. If T and T_b respect the size bounds given in the algorithm, we know that there exist an accepting run (guess T, T_b and Lin). Otherwise, Proposition 12.10 assures us that there exist a profile T' compatible to φ and a border profile T'_b compatible to T' that respect the given size bounds, such that T'_b is satisfiable. The algorithm can accept by guessing T', T'_b and Lin' such that $\mathrm{Lin}' \in \mathrm{SLin}_\ell(T_b)$ and $L(\mathcal{A}_\ell, \mathrm{Lin})$ is nonempty.

For the other direction, we show that if the algorithm accepts, then φ is satisfiable. Let v be a border string in $L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_b))$. By Proposition 12.9, we know that there exists a solution v' for T_b. As T_b is compatible to T, there is a solution w to T and as T is compatible to φ'', there is a solution to φ''. Therefore, φ' and φ are satisfiable. □

We have shown that satisfiability of $\mathrm{FO}^2(\sim, +1)$ over data words is complete for NEXPTIME. The previously known upper bound was 2-NEXPTIME from [NS11]. The best known upper bound for data trees instead of data words is 3-NEXPTIME from [BMSS09]. The improvement over data trees has two reasons. The first reason is, that in the proof for data trees, there is an exponential blowup for limiting the number of zones with many neighbors, whereas in data strings each zone trivially has at most two neighbors. The second reason is, that we introduce an additional intermediate step using border profiles, whereas [BMSS09] and [NS11] use an intermediate profile description that is more closely connected to the formula and gives another exponential blowup. It might be possible to use an intermediate representation similar to our border profiles to improve the upper bound for data trees to 2-NEXPTIME. However, this is speculation and needs further investigation.

12.3 $\mathrm{FO}^2(\sim, +1)$ with Key Constraints

This section is devoted to the proof of the main theorem of this chapter.

Theorem 12.11 *It is decidable whether for a given $FO^2(\sim, +1)$ formula φ and a set $\mathcal{K}$ of key constraints there is a data word w such that $w \models \varphi$ and $w \models \mathcal{K}$.*

The result holds for data words with symbols as well as for data words with propositions.

Before we give the details of the proof we first discuss its general strategy and the underlying ideas. Even though the basic idea of the construction is as in Section 12.2, there are some differences. Especially the definition of border types and border strings is different, to allow to reason about the key constraints.

It turns out that key constraints of different "arities" (number of •-entries) can be handled by different strategies. If a key has arity zero it basically states that certain string patterns should not occur more than once. If on the other hand, a key has arity one then it basically states that in the same class the set of symbols of the •-position can occur at most once within the context given by the key. Both conditions be translated into conditions on the occurring class types. The second conditions is indeed very related to dog types.

Thus, it mainly remains to deal with key constraints of arity ≥ 2. To illustrate the idea let us consider a simple key of the form $(\{a\}, \bullet), (\{b\}, \bullet)$. If one of the symbols a or b is a dog in its class type, that is, it is allowed to occur only once in each class, then the key cannot be violated (if the class type restrictions are met). Thus, we can assume that a and b are sheep[4]. If we know that the class types of a and b occur frequently in the solutions of A and $\mathcal{F}$ that we consider, say $\omega(n)$ many times where n is the length of the data word, then we have a lot of freedom to assign data values to occurrences of a and b. More precisely, there is a quadratic number of possible assignments of pairs of data values to substrings ab but, of course, there can be only a linear number of occurrences of such substrings in any data string. Thus, it will turn out that in this case the key constraint can be fulfilled.

In general it will be slightly more complicated, as the same key constraint can match inside a zone and crossing zone borders. In the first case there is only one relevant data value available, even if there are several $\bullet$-positions. Therefore when looking at the arity, we will look at the arity of individual matches. Furthermore, when determining arity (of matches), we will not count $\bullet$-positions, that are matched on positions marked with an explicit data value, that is an data value from the set $[1, \ell]$ as already used in Section 12.2. The reason is, that for these positions, there might not be enough data values available to exchange data values in a way to satisfy the constraints. As in Section 12.2, we will consider the data values of these positions as explicit and not change them any more.

In the following, we will first describe the extended notation do deal with key constraints, i.e., we will give revised definitions of border types and border strings (Section 12.3.1). Afterwards, we define k-ary matches of key constraints and describe how we can already see from the border string, whether a key constraint is violated by nullary or unary matches (Section 12.3.2). In Section 12.3.3, we define (semi-)linear sets, as in Section 12.2.4 with the difference, that here the sets should also take account of nullary and unary matches of key constraints. A sufficient condition for the existence of a satisfying data word is given in Section 12.3.4. In Section 12.3.5 we put everything together and conclude the proof.

We note that as we anyway cannot give an elementary upper bound for the overall algorithm we do not always aim to represent conditions on solutions in the most efficient way. Instead we favor simplicity over algorithmic efficiency.

12.3.1 Extending Border Strings

The main difference of the border strings here compared to the case without key constraints is, that here the border string have to store more information to be able to conclude where key constraints can possibly match.

Therefore, a border type does not only store the two border $\mathcal{P}_2$-types, but for each border the m $\mathcal{P}_2$-types next to the border, where $m = k(\mathcal{K})$ is the maximal length of some key constraint. In case of short zones, the border type simply stores the whole zone. Additionally, a border type stores, which key constraints can be matched inside the zone.

[4]However, in general it is not *that* simple due to the fact that keys have sets of symbols at a position.

We define the set of all border types as

$$B = \Big(\underbrace{(\mathcal{T}(\mathcal{P}_2))^{2m}}_{\text{long zones}} \cup \underbrace{\bigcup_{i=1}^{2m} (\mathcal{T}(\mathcal{P}_2))^i}_{\text{short zones}} \Big) \times \underbrace{2^{\mathcal{K}}}_{keys} \times \{\eta_D, \eta_S\}.$$

We also need to change the definition of border strings. Additionally to exchanging the old definition of border type with the new one, we also have to store some more information about the neighbors. In Section 12.2, the relevant information about the neighbors was already stored in the border $\mathcal{P}_2$-types. Here, we need additional information about up to m neighboring positions (not zones) to each side of the zone, to be able to see (from one symbol of the border string), whether a key constraint can be matched across zone borders.

Therefore, we define the alphabet of *border strings* to be

$$\Gamma = B \times \underbrace{(\text{dom})}_{\text{data value}} \times \underbrace{\Big(\mathcal{T}(\mathcal{P}_2) \times (\text{dom})\Big)^{2m}}_{\text{neighbors}}.$$

Every symbol consists of a border type, a data value and information ($\mathcal{P}_2$-type and data value) about the m neighboring positions to the left and right.

Remark *Near the borders of a string, we need slightly different symbols, to encode that there are fewer then m positions to the left or right of a zone. This can be done using a slightly different definition of $\mathcal{P}_2$, which includes a dummy proposition saying "this position does not exists".*

We say a border string is valid, if neighboring positions are compatible with respect to their border type and information about their neighbors. As in Section 12.2, we define a restricted alphabet Γ_ℓ, where dom is replaced by $[1, \ell] \cup \{\perp\}$, i.e.,

$$\Gamma_\ell = B \times ([1, \ell] \cup \{\perp\}) \times \Big(\mathcal{T}(\mathcal{P}_2) \times ([1, \ell] \cup \{\perp\})\Big)^{2m}.$$

It is easy to see, that Γ_ℓ is of exponential size in $\mathcal{P}_2$, m and $\mathcal{K}$ and of linear size in ℓ.

Again is it easy to see, that there exists an automaton $\mathcal{A}_\ell$ of size at most $|\Gamma_\ell|$ that tests validity of extended border strings, as it is always enough to remember the last seen symbol.

We define

- compatibility of border types with zone types,
- border class types,
- compatibility of border class types with class types,
- border profiles, and

- compatibility of border profiles with profiles

analogously to Section 12.2.

In detail, we use the following definition for compatibility of border types with zone types.

A border type $\beta = (\ldots, K, \delta)$ is compatible with a zone type $\tau = (S, D)$, if there exists a valid zone z, such that

- the leftmost and rightmost m $\mathcal{P}_2$ types of z are compatible with the information in the border type;
- the set of sheep types of z is a subset of S;
- the set of dog types of z is D;
- $\delta = \delta_K$, if and only if $D \neq \emptyset$; and
- every key constraint $\kappa \in K$ matches exactly once in z.

Note that if a key constraint κ matches twice in the same zone, κ is definitely violated. Therefore, we can restrict to encode at most one inner match per zone and key constraint.

All other mentioned definitions are exactly, as in Section 12.2, with the only difference that they use the updated definitions for border types and compatibility of border types with zone types.

12.3.2 Unary and Nullary Matches of Key Constraints

As already said, the strategies for satisfying key constraint will depend on how many data values contribute to the key constraint. More precisely, for each individual match of the key constraint, we are interested in how many data values contribute to a match. Furthermore we are only interested in non explicit data values, i.e. those data values which are not pre-determined by the border string.

We say a match of a key constraint κ is *k-ary*, if there are exactly k zones $z_1, \ldots, z_k$, such that for each $i \in [1, k]$

- z_i has at least one position matched by a $\bullet$-position of κ; and
- z_i is marked with $\perp$ in the border string, i.e. has no explicit data value.

Note that a match will be nullary if all zones of the match have explicit data values according to the border string.

A symbol γ from Γ is a *witness for a match* of a key constraint κ, if one of the following is true

- κ can be matched inside the zone (indicated by the border type);
- κ can be matched across zone borders (indicated by the border type and the information about the neighboring positions)

We note that due to the information about (the data values of) the neighboring positions, we can always infer how many (non explicit) data values contribute to the match.

A symbol γ from Γ is a *witness for a nullary match* of κ if γ is a witness for a match of κ, the match is nullary (i.e., every $\bullet$-position of κ is matched inside a zone with an explicit data value) and γ is the first position in the border string involved in the match. The latter is the case

- if κ is matched inside the zone, as then γ represents the only zone involved in the match; or
- if the match of κ uses some of the rightmost m $\mathcal{P}_2$ types and some of the right neighbor positions of γ.

We denote the set of all symbols that are witnesses for a nullary match of κ with $\Gamma_\kappa^{\text{nullary}}$.

A symbol γ from Γ is a *witness for a unary match* of κ, if γ is a witness for a match of κ, the match is unary and γ represents the only zone involved in the match, which is matched by a $\bullet$-position and has no explicit data value. We denote the set of all symbols that are witnesses for a unary match of κ with $\Gamma_\kappa^{\text{unary}}$.

We note that for nullary matches, we always use the first zone as witness, whereas for unary matches we always use the only zone with unspecified data value. The choice for the first zone in the case of nullary matches is just to have one unique witness for every nullary match, whereas for unary matches it will turn out to be important to use the zone with the unspecified data value.

It is easy to see, that every violation of a key constraint by nullary matches can already be seen from the border string w, as a key is violated by nullary matches if and only if there are at least two positions in w, which witness a nullary match of κ, that is there are no two positions in w that are labeled with symbols from $\Gamma_\kappa^{\text{nullary}}$.

In the same way, we can deduct violations of key constraints by unary matches, by looking at individual classes, as a key κ is violated by unary matches, if and only if there exists a class c, such that there are at least two positions in c, which witness a unary match of κ.

The difference (looking at the complete string versus looking at one class) reflects in the strategy used to avoid violations by nullary and unary matches. To avoid violations by nullary matches, we disallow the appearance of two witnesses for nullary matches of the same key constraint in the complete extended border string, whereas for unary matches we disallow two witnesses for unary matches of the same key constraint in the same class. Both restrictions will be implemented by defining the linear sets appropriately.

12.3.3 Semi-linear Sets with Key Constraints

As in Section 12.2, we now give a definition of a semi-linear set, depending on a border profile and — different to the definition in Section 12.2 — also depending on the key constraints. The set $\text{SLin}_\ell(T_b, \mathcal{K})$, will additionally ensure, that data values can be

assigned in such a way, that no key constraint is violated by nullary and/or unary matches.

Let τ_b be a border class type and $i \in \text{dom}$ be a data value indicator. As in Section 12.2, we define $\text{Lin}(\tau_b, i)$ to be

$$\text{Lin}(\tau_b, i) = \left\{ \sum_{\beta \in B_D} \tau_b(\beta) \cdot e_{\beta,i} + \sum_{\beta \in \tau_b^{-1}(*)} k_\beta \cdot e_{\beta,i} \;\middle|\; k_\beta \in \mathbb{N}_0 \text{ for } \beta \in \tau_b^{-1}(*) \right\},$$

To make sure, that key constraints are not violated by unary matches, we define a semi-linear set $\text{SLin}_\mathcal{K}^{\text{unary}}$, such that for each string $v \in \Gamma^*$ with $\text{parikh}(v) \in \text{SLin}_\mathcal{K}$ and each key constraint $\kappa \in \mathcal{K}$, it holds that there is at most one position labeled with a symbol from $\Gamma_\kappa^{\text{unary}}$. Analogously, we define a set $\text{SLin}_\mathcal{K}^{\text{nullary}}$.

Remark *It can be verified that $SLin_\mathcal{K}^{unary}$ is of at most exponential size. However for our results it suffices that $SLin_\mathcal{K}^{unary}$ is computable.*

We define $\text{SLin}(\tau_b, \mathcal{K}, i)$ to be $\text{Lin}(\tau_b, i) \cap \text{SLin}(\mathcal{K}, i)$ for $i \in [1, \ell] \cup \{\bot\}$. Note that semi-linear sets are closed under intersection. The set $\text{SLin}(\tau_b, \mathcal{K}, i)$ is intended to capture exactly one class of the resulting string. Therefore, this definition ensures that there are no key constraint violation by unary matches, as in each class for each key constraint, there is at most one unary match.

As in Section 12.2, we define a semi-linear set $\text{SLin}_\ell(T_b, T_\ell, \mathcal{K})$ from a given border profile T_b and a given sequence of ℓ border class types. The difference is, that we use the sets $\text{SLin}(\tau_b, \mathcal{K}, i)$ instead of $\text{Lin}(\tau_b, i)$, to ensure that there are no key constraint violations within a class.

Let therefore be $T_\ell = \tau_1, \ldots, \tau_\ell$ be a sequence of ℓ border class types. We then define $\text{SLin}(T_b, T_\ell, \mathcal{K}) =$

$$\left\{ \sum_{i=1}^{\ell} \text{SLin}(\tau_i, \mathcal{K}, i) + \sum_{\tau_b} k_{\tau_b} \cdot \text{SLin}(\tau_b, \mathcal{K}, *) \;\middle|\; \begin{array}{ll} k_{\tau_b} + |\{i \mid \tau_i = \tau_b\}| = T_b(\tau_b) & T_b(\tau_b) \neq * \\ k_{\tau_b} > 3|\mathcal{T}(\mathcal{P}_2)| & T_b(\tau_b) = * \end{array} \right\}$$

The set $\text{SLin}_\ell(T_b, \mathcal{K})$ is defined as

$$\left(\bigcup_{T_\ell} \text{SLin}(T_b, T_\ell, \mathcal{K}) \right) \cap \text{SLin}_\mathcal{K}^{\text{nullary}}.$$

We intersect with $\text{SLin}_\mathcal{K}^{\text{nullary}}$ to ensure that no constraint is violated by nullary matches.

12.3.4 Fulfilling the Key Constraints

In this section we give a sufficient condition for the existence of a solution for a constraint instance $I = (T, \mathcal{K})$.

The following proposition is the analog to Proposition 12.9. It basically states that if the constraints of the semi-linear set are fulfilled and additionally, the border string is valid (i.e., accepted by $\mathcal{A}_\ell$), then we can consistently assign data values to all positions labeled with $\perp$. Different to Proposition 12.9, we require that there is an unbounded linear set, as defined below. This allows us to obtain a string $u \in L(\mathcal{A}_\ell, \mathrm{SLin}_\ell(T_b, \mathcal{K})$, where each class border type that occurs without explicit data value is frequent.

A border type β is bounded in some linear set $\mathrm{Lin} = \{\mathrm{f} + \sum_j i_j \, \mathrm{f}_j \mid i_j \in \mathbb{N}\}$ over Γ_ℓ, if either

- $\mathrm{f}(\gamma) = 0$ for each symbol $\gamma \in \{\beta\} \times \{\perp\} \times (\mathcal{T}(\mathcal{P}_2) \times ([1, \ell] \cup \{\perp\}))^{2m}$; or
- $\mathrm{f}_j(\gamma) \neq 0$ for some symbol $\gamma \in \{\beta\} \times \{\perp\} \times (\mathcal{T}(\mathcal{P}_2) \times ([1, \ell] \cup \{\perp\}))^{2m}$.

A linear set Lin over Γ_ℓ is *unbounded*, if each border type $\beta \in B$ is unbounded in Lin.

Proposition 12.12 *Let T_b be a border profile and $\mathcal{K}$ be a set of key constraints such that there is an unbounded linear set in* $\mathrm{parikh}(L(\mathcal{A}_\ell)) \cap \mathit{SLin}_\ell(T_b, \mathcal{K})$. *Then there exists a border string u without $\perp$ markings, such that u satisfies $\mathcal{K}$ and u is a solution to T_b.*

Proof. The proof is similar to the proof of Proposition 12.9. However, we have to take key constraints into account.

Let $\mathrm{Lin}_0 = \{\mathrm{f} + \sum_j i_j \, \mathrm{f}_j \mid i_j \in \mathbb{N}\}$ be an unbounded linear set in $\mathrm{parikh}(L(\mathcal{A}_\ell)) \cap \mathrm{SLin}_\ell(T, \mathcal{K})$. Let

$$c = \sum_{\gamma \in \Gamma_\ell} \sum_{\mathrm{h} \in \{\mathrm{f}, \mathrm{f}_1, \ldots, \mathrm{f}_n\}} \mathrm{h}(\gamma_\ell)$$

be the sum of all coefficients occurring in Lin_0. Let further $m = k(\mathcal{K})$ and $M = 4 + 16m^2(c + |\mathcal{T}(\mathcal{P}_2)|^2)$. Let g be the function obtained from L_0 by setting $i_j := M$, for all j, and let $v \in L(\mathcal{A}_\ell)$ be a border string with $\mathrm{parikh}(v) = \mathrm{g}$.

The assignment of data values will (as in Proposition 12.9) be done in three steps. First we assign data values to dog zones, then we change these data values such that no two neighboring zones have identical data values, and finally, we assign data values to sheep zones.

Let u be a border string, such that u is identical to v, except that all $\perp$ markings of dog positions are replaced by data values from $\mathrm{dom} \setminus [1, \ell]$, such that

- u fulfills the frequency constraints of T_b, and
- u has no violations of key constraints by unary matchings.

Note that u cannot have violations by nullary matchings, as v has no such violations and the data values are completely irrelevant to nullary matchings. Furthermore u cannot have any violations by unary key constraints, as these are forbidden by $\mathrm{SLin}_\ell(T_b, \mathcal{K})$.

The argumentation that such a string exists is identical to the argumentation in Proposition 12.9.

However, u might have inconsistencies of type (2), that is neighboring positions with the same data value. Furthermore u might have violations of key constraints by k-ary matchings with $k \geq 2$. We denote key constraint violations as type (3) inconsistencies.

We correct data values for dog zones inductively from left to right. Let therefore be i the first position, that

- has the same data value as its left neighbor; or
- is matched by a $\bullet$-position of some matching of a key constraint κ. such that to the left of i there is another matching of κ with the same data values at all $\bullet$-positions of κ.

If there is some position $j > i$ to the right of i, such that $\beta_j = \beta_i$, d_j is different from the data value at position $i - 1$ and assigning d_j to position i will not violate a key constraint, then we exchange the data values of i and j.

Otherwise, we have to exchange the data value of i with the data value of some position to the left of i.

Let therefore j be a position, such that

(1) the data value d' of j is different from d

(2) j has no d-valued neighbor;

(3) assigning d to j would not cause j to violate a key constraint; and

(4) assigning d' to i would not cause i to violate a key constraint.

We exchange the data values of i and j.

As each position only has 2 neighbors and there are at most $|\mathcal{T}(\mathcal{P}_2)|$ dog positions with data value d, the first two conditions rule out at most $3|\mathcal{T}(\mathcal{P}_2)|$ many (dog) positions.

The third condition can rule out a position j, only if there exists a dog position j', such that

- j' has data value d
- j has a nearby (distance at most m) position k and j' has a nearby position k', such that k and k' have the same data value.

As there are at most $|\mathcal{T}(\mathcal{P}_2)|$ positions with data value d, there are at most $2m|\mathcal{T}(\mathcal{P}_2)|$ positions near positions with data value d. These positions have at most $2m|\mathcal{T}(\mathcal{P}_2)|$ different data values D. With the same argumentation we get, that for each data value d' of D there are at most $2m|\mathcal{T}(\mathcal{P}_2)|$ positions near to some occurrence of d' and therefore the third condition can rule out at most $(2m|\mathcal{T}(\mathcal{P}_2)|)^2$ many positions.

The forth condition is symmetric and can therefore rule out at most $(2m|\mathcal{T}(\mathcal{P}_2)|)^2$ many positions, too.

Altogether, there are at most $M' = 3|\mathcal{T}(\mathcal{P}_2)| + 8m^2|\mathcal{T}(\mathcal{P}_2)|^2$ ruled out positions. As there are at least $M > M'$ candidate positions available, we can always find a position j, such that we can exchange the data values of i and j.

It remains to assign data values to sheep positions. By the choice of v, we know that for each sheep border type β (occurring with a $\perp$ marker), there are at least M different

data values available. For each sheep border type β we denote the set of data values available to β with Δ_β.

We have to assign data values from Δ_β to positions labeled with β, such that the key constraints are satisfied and no neighboring positions get the same data value. We will keep as invariant, that each data value from Δ_β is assigned to at most $X = 2c$ positions.

A data value d from Δ_β could be ruled out for a position i labeled with β for one of the following reasons:

(i) a neighbor of i carries d;

(ii) assigning d to i would yield a key violation involving i; or

(iii) d has already been assign to X positions.

Clearly, (i) rules out at most 2 values from Δ_β. A data value d can only fulfill condition (ii) if for some value d' occurring at distance $< m$ from i, d has an occurrence at distance $< m$ from some position with data value d'. This rules out at most $4m^2X$ data values. Finally, at most $\frac{|v|}{X}$ data values can be ruled out by condition (iii). It is obvious that $|v| \leq Mc$ by definition of v.

However, it holds

$$\begin{aligned} 2 + 4m^2X &= 2 + 8m^2c < \frac{M}{2}, \\ \frac{|v|}{X} &\leq \frac{Mc}{2c} = \frac{M}{2}. \end{aligned}$$

Thus, fewer than M data values are ruled out. As $|\Delta_\beta| \geq M$, it is possible to find a data value $d \in \Delta_\beta$ that does not violate any constraints and maintains the invariant.

We can conclude that it is possible to assign data values in such a way that the resulting border string is valid. We note that due to the definition of border types, the key constraints are always satisfied, regardless, how the border types are replaced by compatible zone strings. This concludes the proof. □

12.3.5 Separating Frequent from Infrequent Class Types

To proof the main result we need one more step. In Section 12.2 it was sufficient to choose ℓ as large as $3|\mathcal{T}(\mathcal{P}_2)| \times |T_b|$ to show that we can always find an assignment of data values. Unfortunately, Proposition 12.12 can only be applied if there is an unbounded set in $\mathrm{SLin}_\ell(T_b, \mathcal{K})$.

Our general strategy is to increase ℓ until all bounded border types can get explicit data values. Alongside we forbid bounded border types to occur without an explicit data value. Note however, that such a step (increasing ℓ and disallowing bounded types without explicit data value) can switch some border types from unbounded to bounded and thus it might be necessary to repeat this for each border type.

Statement of Theorem 12.11 It is decidable whether for a given $\mathrm{FO}^2(\sim, +1)$ formula φ and a set $\mathcal{K}$ of key constraints there is a data word w such that $w \models \varphi$ and $w \models \mathcal{K}$.

Algorithm 13 Satisfiability of Border Profiles with Key Constraints

```
1:  function SAT(T_b, K)
2:      ℓ := 0
3:      Γ' := Γ_ℓ
4:      repeat
5:          Lin :=  guess linear set from SLin_ℓ(T_b, K) ∩ parikh(Γ'*)
6:          if Lin is unbounded then accept
7:          B_seldom := {β | β is bounded in Lin}
8:          ℓ := maximal number of classes using B_seldom in Lin
9:          Γ' := Γ_ℓ \ (B_seldom × {⊥})
10:     until |Lin| < ∞
11:     if ∃v ∈ (B × [1, ℓ])*. parikh(v) ∈ Lin ∧ v ⊨ K then accept else reject
```

Proof. The overall decision algorithm is similar to Algorithm 12. First we compute some formula φ' in data normal form from φ. Afterwards, we guess a profile T and a border profile T_b such that T is compatible with φ' and T_b is compatible with T. We note, that the proof for Proposition 12.8 works without changes for the modified definition of border types, we use here. Altogether, we can conclude that there exists a data string w such that $w \models \varphi$ and $w \models \mathcal{K}$ if and only if there exists a solution to T_b that satisfies the key constraints. We remember, that all information needed to test whether the key constraints are satisfied are encoded in the border string v, i.e., every to v compatible data string w will satisfy the key constraints, when they are satisfied in v.

We prove the Theorem statement by showing that Algorithm 13 decides satisfiability of border profiles with key constraints.

First, we describe Algorithm 13. The variable ℓ always holds the current number of explicit data values used for seldom border types. It is initialized with 0, as at the beginning the set B_{seldom} of seldom border types is empty. The variable Γ' always holds the currently allowed alphabet. This alphabet is equivalent to Γ_ℓ, except that seldom border types are not allowed to occur without an explicit data value.

The repeat loop is the central part of the algorithm. In Line 5 the algorithm guesses a linear set from the semi-linear set $\text{SLin}_\ell(T_b, \mathcal{K})$, such that no seldom border type occurs without an explicit data value. The latter condition is ensured by intersecting with the Parikh image of Γ'^*.

If the linear set is unbounded the algorithm accepts. Otherwise it recomputes the set of seldom border types, increases ℓ, such that all classes using some seldom border type can get an explicit data value, and recomputes Γ'.

The algorithm either terminates when an unbounded set is found (Line 6) or when the linear set is finite. In the latter case, the algorithm can enumerate all possible border strings, such that the Parikh image is in Lin and accept, if it finds a solution.

The algorithm always terminates, as B_{seldom} is strictly increasing. Once $B_{\text{seldom}} = B$, all border types are bounded and therefore Lin has to be finite.

We have to prove correctness. First we show, that if the algorithm accepts, then T_b and $\mathcal{K}$ are satisfiable at the same time. If the algorithm accepts in Line 6, it has found an unbounded linear set in $\mathrm{SLin}_\ell(T_b, \mathcal{K})$, which is a sufficient condition according to Proposition 12.12. In the case the algorithm accepts in Line 11, it has found a solution by the condition in the if statement. As there is no other possibility for the algorithm to accept, there can be no false positives.

Now we show, that if T_b and $\mathcal{K}$ are satisfiable at the same time, then there is an accepting run of the algorithm. Let therefore v be a solution, that is v is a border string without $\perp$ markers, such that $v \models T_b$ and $v \models \mathcal{K}$.

Let for some permutation π of the data values, $\pi(v)$ denote the data string derived from v by applying π to the data values of v. Note that we will use π only to decide which data values of v are explicit data values, that is data value from $[1, \ell]$.

We do an induction over the number of iterations of the loop, to show that there always is a linear set Lin in $\mathrm{SLin}_\ell(T_b, \mathcal{K}) \cap \mathrm{parikh}(\Gamma'^*)$ and a permutation π on the data values, such that $\mathrm{parikh}(\Gamma_\ell(\pi(v))) \in \mathrm{Lin}$. The correctness follows then from the fact, that the algorithm can only reject, if $\mathrm{parikh}(\Gamma_\ell(\pi(v))) \notin \mathrm{Lin}$.

The statement is true for the induction base case, as $\mathrm{parikh}(v) \in \mathrm{SLin}_0(T_b, \mathcal{K})$ and $\Gamma' = \Gamma_0$ during the first iteration.

For the induction step let Lin be a linear set from $\mathrm{SLin}_\ell(T_b, \mathcal{K}) \cap \mathrm{parikh}(\Gamma'^*)$ that contains $\mathrm{parikh}(\Gamma_\ell(\pi(v)))$. By the induction hypothesis such a set exists. Let B_{seldom} be as in Line 7. By definition of "maximal", the new number ℓ' computed in Line 8 is at least as big as the number of classes using border types from B_{seldom} in v. Therefore there exists a permutation π' which maps all data values used with seldom border types in v to the set $[1, \ell']$. It follows that $\mathrm{parikh}(\Gamma_\ell(\pi'(v))) \in \mathrm{SLin}_{\ell'}(T_b, \mathcal{K}) \cap \mathrm{parikh}(\Gamma'^*)$

This concludes the induction and the proof of the theorem. □

12.4 Conclusion on $\mathrm{FO}^2(\sim, +1)$

In this chapter, we took a look at constraints specified using first order sentences with two variables. This research was motivated by the fact that first order sentences are the unifying constraint language for relational databases. Basically all means for specifying integrity constraints on relational databases build on-top of first order logic. Furthermore, we know from literature that we can also specify some structural properties using this logic. We analyzed the complexity of the consistency problem of a first order sentence with two variables together with a set of key constraints. However, we restricted our research to strings which can be interpreted as very simple trees. Towards this goal we first looked at the consistency problem of the $\mathrm{FO}^2(+1, \sim)$ logic over data strings. We improved the known upper bound from 3-NEXPTIME (upper bound for trees from [BMSS09]) to NEXPTIME. A more straightforward application of the methods from [BMSS09] gives a 2-NEXPTIME upper bound as shown in [NS11]. Extending the methods used for this analysis, we could show that the problem still is decidable when key constraints of arbitrary arity are added. However, we have no idea of the complexity of this problem (aside from a more or less trivial NEXPTIME lower bound) and we have no idea whether

the same problem over data trees (with a reasonable definition of key constraints on trees) is decidable. For practical purposes it seems appropriate to separate structural properties from semantic properties, like we have done in our framework depicted in Chapter 10. In fact, the research on X2R-constraints started chronologically after the results of the current chapter to get some better bounds for practically relevant classes of integrity constraints.

Part III

Prototype

13 FoXLib

In this chapter, we present FoXLib, which is a collection of prototype implementations of some algorithms in the context of XML described in various papers.

Roughly, FoXLib consists of three big parts: the Formal Language Toolkit (FLT), the Schema Toolkit and the BonXai Editor Plugin. Additionally there are a few smaller parts.

13.1 Formal Language Toolkit

The Formal Language Toolkit is a collection of tools to represent and manipulate formal languages. It has been developed by various members of Hasselt University and is divided into several modules. WE only list the most important ones.

FLT-core This is the central part of the Formal Language Toolkit. It provides the means to store and manipulate finite automata and regular expressions. Among others, it contains algorithms for converting regular expression into finite automata and vice versa and algorithms for determinizing and minimizing finite automata.

The automaton library is capable of storing DFA-based XSDs, as we have introduced them in Chapter 5.

This part of the library allows for an easy integration of existing automaton based algorithms. All other parts of FoXLib use directly or indirectly the functionality of FLT-core.

FLT-learning This part contains regular expression and XSD inference algorithms, as they have been described in [BNSV10]. The algorithms learn automata models of the input data. To actually output schemas, these automata have to be converted to models of the Schema Toolkit, which is described below.

FLT-disambiguate This part contains an algorithm for repairing the unique particle attribution constraint of XML Schema which has been proposed in [BGMN09]. The algorithms computes approximations of the input language (given by a finite automaton).

13.2 Schema Toolkit

The Schema Toolkit constitutes the heart of the system. It provides modules for the representation, import and export, and the conversion between DTD, XSD, and BonXai.

The Schema Toolkit originated from a student project in Dortmund [DGG+09]. The students

Import and Export For all schema languages we have import and export modules, which can work on streams, files or java strings, whichever is best suited for the user of the library.

Object Models Schemas can be represented in an abstract way as DFA-based XSDs. To facilitate manipulation of schemas, each class of schemas additionally has its own object model. These object models store additional information, such as key, foreign key and uniqueness constraints, identifiers used for namespaces, typenames, etc.

XML Validator The XML validator validates XML documents against DFA-based XSDs and can thus be used to validate XML documents against BonXai schemas, DTDs, as well as XSDs.

Conversion Routines As all conversions pass through the DFA-based XSD representation, there are six conversion routines. The translation to and from XSDs and DTDs is rather direct. The computation of a DFA-based XSD from a BonXai schema is discussed in detail in Chapter 6. It basically reduces to the construction of a product automaton encompassing all regular contexts in the schema. The converse direction requires to compute regular contexts for every state of the DFA-based XSD. In addition, the conversion routine creates mappings between automaton states and the corresponding BonXai rules or XML schema types. This information together with the mappings between XML nodes and automaton states produced by the XML validator, is used by the BonXai Editor Plugin to highlight matching nodes/rules in the editor. Information about constraints and namespace identifiers is directly converted between the object models.

13.3 BonXai Editor

On top of the Formal Language Toolkit and the Schema Toolkit we have developed a BonXai Editor which provides a graphical user interface to the features of the two toolkits. The BonXai Editor has been presented as a VLDB demo in 2012 [MNNS12]. Figure 13.1 presents an overview of the BonXai Editor.

The GUI of our current implementation is provided through a plugin for the open source editor JEdit [JEd]. JEdit provides basic text editing functionalities, syntax highlighting and a flexible plugin interface. Through the GUI, the user can directly develop BonXai schemas if desired. The BonXai-Plugin provides the connection between the Formal Language Toolkit, the Schema Toolkit and JEdit.

GUI Features The GUI aids to understand the correspondence between BonXai rules and the generated complex types in the transformed XSD. Advanced functionalities

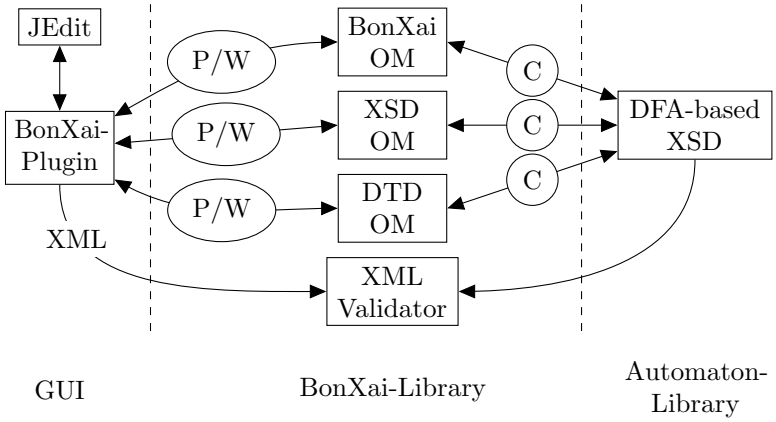


Figure 13.1: Schematic overview of the BonXai Editor.
P/W: Parser/Writer, C: Converter, OM: Object Model.

of our GUI facilitate schema development and -debugging. In particular, we support the analysis of the relationships between an XML document, a BonXai schema, and a corresponding XSD as follows:

- Highlighting of XML elements matched by a certain BonXai rule or by an XSD complex type.
- Highlighting of the rule/type that matches an element in an XML tree.
- Highlighting the BonXai rule corresponding to an XSD complex type and vice versa.
- Finding nodes in an XML tree violating the schema.
- Finding nodes in an XML tree which are unconstrained by the schema, i.e., for which the schema allows arbitrary content.

Figure 13.2 shows a screenshot of the GUI, which demonstrates the highlighting capabilities of our plugin based on our running example. Inside the screenshot, the `section` element is marked orange, as its content model is not correct. The `boldd` element is marked red as it is not allowed to appear at this position in the document (note the spelling error). The turquoise `font` element has been selected by the user in the XML document. The plugin shows where the element is declared in the BonXai schema and XML schema respectively by highlighting the element declaration in purple. Furthermore it shows where the content model of the selected element is declared by marking its ancestor path (BonXai) and type declaration (XML Schema) in green.

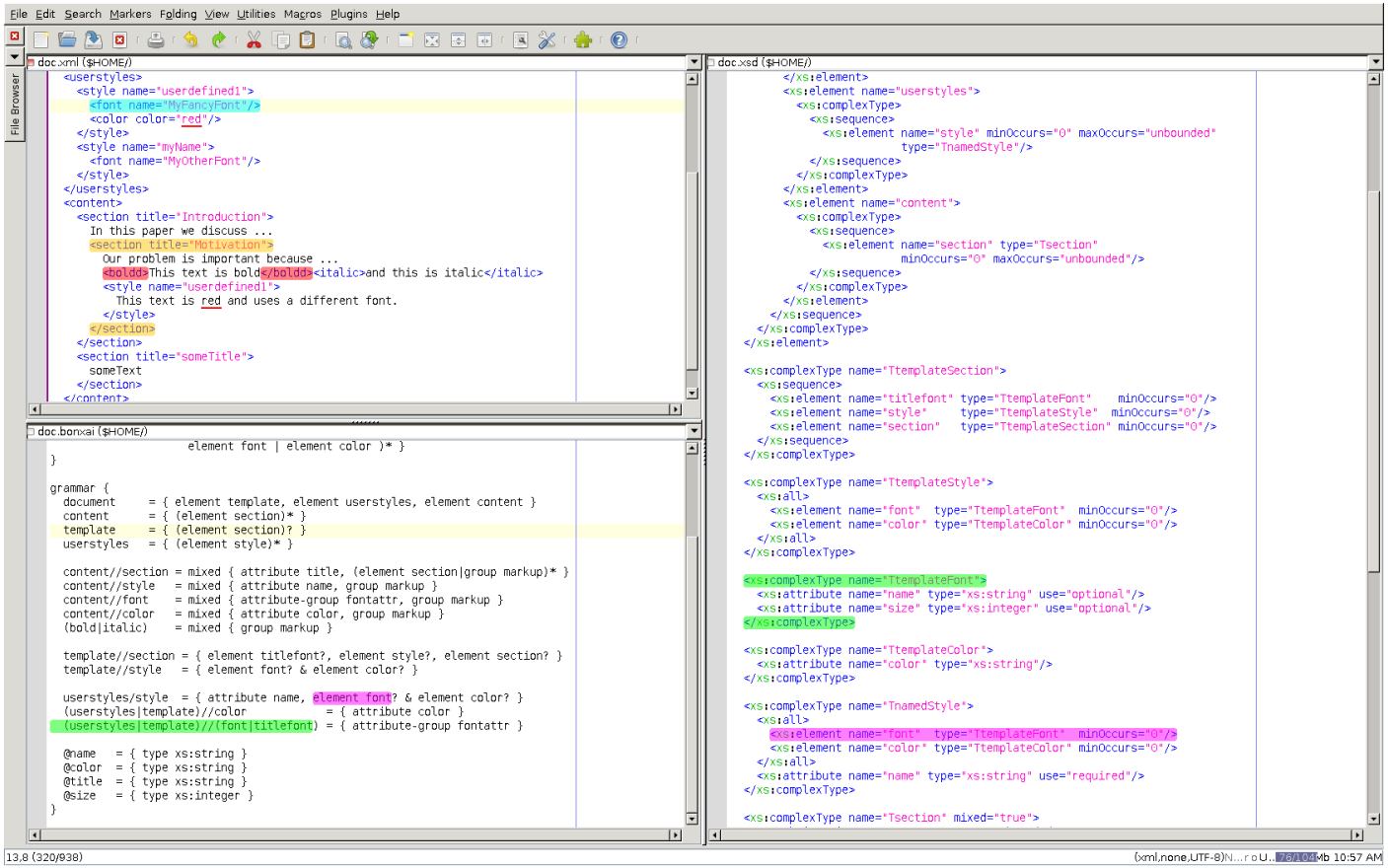


Figure 13.2: Screenshot of the jedit-plugin with three editor panes: Top-Left: XML document from Figure 4.2, Bottom-Left: BonXai schema from Figure 5.2, Right: XML Schema from Figures 4.5–4.7.

13.4 History of FoXLib

The Formal Language Toolkit is provided by Hasselt University. The initial draft of the Schema Toolkit was developed by a student group at TU Dortmund University. It consisted of the BonXai and XML Schema object models, conversion routines between them and parsers for both schema languages. The parser for the BonXai language was not functional and the conversion routines used a self-implemented automaton library which was faulty, i.e., which produced erroneous results.

On top of this work, two master students out of this student group implemented conversion routines between XML Schema, DTDs and RelaxNG [Sch10] and algorithms for computing (approximations of) intersections and unions of XML Schemas [Wol10]. Both works suffered from the faulty implementation of the automaton library.

The author of this thesis integrated the Formal Language Toolkit and the Schema Toolkit. Especially, the faulty automaton library was replaced by an automaton library from the Formal Language Toolkit. This integration forced a rewrite of much of the existing code inside Schema Toolkit. Due to the rewriting the parts related to RelaxNG are not functional any more.

The integration of both libraries into one has additional benefits. It is now possible to use the inference algorithms of the FLT to learn actual schemas instead of only automaton models. In the other direction it is possible to parse actual schemas and derive automaton models from them that can be further analyzed using the means of the FLT.

The BonXai editor plugin was implemented on top of the other parts of the library by the author of this thesis. This included writing a working BonXai parser and a new XML Schema parser. The original XML Schema parser was build on top of the DOM object model, which does not provide any information about the location of elements in the XML file. This information is needed to allow for the highlighting features of the BonXai editor.

13.5 Future of FoXLib

Up to now, the library has only been used by a few academic researchers, mainly persons from Dortmund and Hasselt. Towards the goal of promoting BonXai, the library has to be deployed to a wider audience. Therefore, we have registered the domain bonxai.org to bundle all available documentation about FoXLib and BonXai and to provide access to the library.

Conclusion & Bibliography

14 Conclusions and Directions for Further Research

We started this thesis by looking at features of database management systems in Chapter 2 to identify where XML repository management systems differ from their relational counterparts. We recognized that there are at least some differences in almost all areas. However, the main part of this thesis only covers one aspect of these system, the data definition language, which is an important part of the interface between the system and the programmer or administrator. Additional to that, the research on implication of integrity constraints possibly has some applications in the area of query optimization which is part of the query evaluation engine. To actually build XML Repository Management Systems, further research in almost all aspects of these systems is necessary. In the following conclusion, we will again focus on the aspects, we actually analyzed. This should in no way imply that the other aspects are of minor importance or do not need further research. We already have given hints on further research at the individual topics, which we do not want to repeat. Instead, we want to have some broader conclusion here, a bit focusing on the goal of designing and implementing an XML Repository Management System.

In the Chapters 5 and 6, we introduced and analyzed the BonXai schema language. While it is certainly possible, to build an XRMS that only stores the XML data, without looking at semantics, it is desirable, if the system knows (and checks) semantic constraints. On one hand, the knowledge about these constraints can be used to optimize access to data. On the other hand, the system can actively help to recognize (and possibly repair) invalid data. Therefore, the system should be schema-aware. Managing large amounts of data usually means, that there is more need to present the increased amount of schema information in a user-understandable way. This should give enough motivation for further research in the area of pattern based schema languages and possibly extending the BonXai schema language.

After studying the BonXai schema language, we summarized known results about deterministic regular expressions, which are used in various XML schema languages including BonXai. While further research on DREs is interesting from a formal language point of view, it is of minor importance for XRMS's. Existing schema languages are using DREs. However, there is no compelling reason to use DREs in the core of some XRMS. Most algorithm will work on some automata representation anyway and if there is some need to export schema information to existing schema languages, there is always the pragmatical approach to use existing approximations.

In Chapter 8 we made an excursion into distributed XML repository management systems. While it is obvious that data gets distributed more and more, a first XRMS implementation probably will focus on local management of data. However, XRMS's

should be designed with distribution in mind, meaning that it should be possible to add support for distribution at a later time without the need of a big redesign of the system. On the longer perspective, managing distributed data, will be absolutely necessary and therefore the research on distributed XML schema design should be continued.

After the excursion into distributed documents, we changed our focus completely from purely structural descriptions to semantic constraints in Part II of this thesis. Contrary to Part I, in Part II, we actually looked at the data instead at only focusing on its structure. We start our retrospect of this part with Chapter 12, whose results chronologically precedes the research in Chapters 9 to 11.

The research on first order logic with two variables is certainly interesting from a theoretical point of view. While we have succeeded to show that $\mathrm{FO}^2(\sim, +1)$ on data words is decidable in combination with key constraints, there is no direct application to XRMS's, as we only managed to show satisfiability on data words (and not on trees). Even if the results could be transferred to trees (which we have no clue about), the approach is too complex to be deployed to actual XRMS's. Towards a working XRMS, the approach of X2R-constraints, which we analyzed in Chapters 9 to 11 is the better alternative. This corresponds to one central insight, which we want to repeat here: In XML integrity constraints, navigational properties of the used constraint mechanism should be separated from semantic features that can compare data values. $\mathrm{FO}^2(\sim, +1)$ mixes navigational properties with data comparisons, as it has direct access to data values.

We spent a big part of this thesis to analyze XML-to-relational constraints. As already pointed out, there are many possible research directions in the area of XML integrity constraints. Integrity constraints have always been an important aspect of relational databases and relational DBMS's. There is no evidence that this changes fundamentally for XML databases. While usage of key constraints in small documents might not have been widely used, once it comes to managing big amounts of data (as in XRMS's), the need for integrity constraints will rise. Therefore, there should be ongoing research in this area. We have depicted a very general framework. Towards the design of some actual system, it is necessary to identify tractable yet expressive fragments. In Chapter 11, we have studied the implication problem of some sort of functional dependencies and key constraints. Ongoing research can look at some different fragments to identify the border of tractability.

We want to emphasize again, that in the second part, we have concentrated on designing the underlying theoretical concepts instead of designing languages aimed at users, i.e., database administrators and programmers. Towards a usage of our concepts in actual XRMS's, there is still some way to go, as indicated at the end of Chapter 11.

At the end of this thesis, we took a look on the FoXLib library, that implements some algorithms designed for XML schema handling and the prototype implementation of a plugin based on the jedit platform. This library is in particular meant to promote the BonXai schema language. It allows conversions between BonXai and XML Schema and thus allows people who want to try BonXai to stay compatible with existing systems. The library and implementation is not meant to be an XRMS. Especially it is not designed to handle big amounts of data, as the focus is on schema design and schema analysis.

Nevertheless, the implemented algorithms can of course be useful tools for a database administrator helping to design and manage XML databases inside some XRMS.

In summary, we can conclude that there is still a lot of work to be done on the way towards a working XRMS.

Bibliography

[ABLM10] M. Arenas, P. Barcelo, L. Libkin, and F. Murlak. Relational and xml data exchange. *Synthesis Lectures on Data Management*, 2(1):1–112, 2010.

[AF05] S. Avgustinovich and A. Frid. A unique decomposition theorem for factorial languages. *International Journal of Algebra and Computation*, 15:149–160, 2005.

[AFL08] M. Arenas, W. Fan, and L. Libkin. On the complexity of verifying consistency of XML specifications. *SIAM Journal on Computing*, 38(3):841–880, 2008.

[AGM09] S. Abiteboul, G. Gottlob, and M. Manna. Distributed XML design. In *International Symposium on Principles of Database Systems (PODS)*, pages 247–258, 2009.

[AHV94] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison Wesley, 1994.

[AL04] M. Arenas and L. Libkin. A normal form for XML documents. *ACM Transactions on Database Systems*, 29:195–232, 2004.

[AM86] P. Atzeni and N. M. Morfuni. Functional dependencies and constraints on null values in database relations. In *Information and Control*, volume 70(1), pages 1–31, 1986.

[Bal04] S. Bala. Regular language matching and other decidable cases of the satisfiability problem for constraints between regular open terms. In *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 596–607, 2004.

[BDF+02] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML. *Computer Networks*, 39(5):473–487, 2002.

[BDF+03] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. *Information Systems*, 28(8):1037–1063, 2003.

[BGMN09] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML Schema: effortless handling of nondeterministic regular expressions. In *International Symposium on Management of Data (SIGMOD)*, pages 731–744, New York, NY, USA, 2009. ACM.

[BK92] A. Brüggemann-Klein. Regular expressions into finite automata. volume 583 of *Lecture Notes in Computer Science*, pages 87–98. Springer Berlin Heidelberg, 1992.

[BKW98] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[BMS+06] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 7–16, 2006.

[BMSS09] M. Bojanczyk, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and XML reasoning. *Journal of the ACM*, 56(3), 2009.

[BNSV10] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems*, 35(2):11:1–11:47, 2010.

[BPSM+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language XML 1.0 (fifth edition). Technical report, World Wide Web Consortium (W3C), November 2008. W3C Recommendation, http://www.w3.org/TR/2008/REC-xml-20081126/.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[BV84] C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[CDFI12] C. Cui, Z. Dang, T. R. Fischer, and O. H. Ibarra. Information rate of some classes of non-regular languages: An automata-theoretic approach. submitted, 2012.

[CDLM13] W. Czerwiński, C. David, K. Losemann, and W. Martens. Deciding definability by deterministic regular expressions. In *Foundations of Software Science and Computation Structures*, pages 289–304, 2013.

[CFPR03] J. Czyzowicz, W. Fraczak, A. Pelc, and W. Rytter. Linear-time prime decompositions of regular prefix codes. *International Journal of Foundations of Computer Science*, 14:1019–1031, 2003.

[CGLV02] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi. Rewriting of regular expressions and regular path queries. *Journal of Computer and System Sciences*, 64(3):443–465, 2002.

[CGM88] U. S. Chakravarthy, J. Grant, and J. Minker. *Foundations of semantic query optimization for deductive databases*. Morgan Kaufmann Publishers Inc., 1988.

[CHM11] P. Caron, Y. Han, and L. Mignot. Generalized one-unambiguity. In *International Conference on Developments in Language Theory (DLT)*, pages 129–140, 2011.

[CM01] J. Clark and M. Murata. Relax NG specification. http://www.relaxng.org/spec-20011203.html, December 2001.

[CMM13] W. Czerwinski, W. Martens, and T. Masopust. Efficient separability of regular languages by subsequences and suffixes. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 150–161, 2013.

[CMV04] C. S. Coen, P. Marinelli, and F. Vitali. Schemapath, a minimal extension to xml schema for conditional constraints. In *WWW*, pages 164–174, 2004.

[Cod72] E. F. Codd. *Relational completeness of data base sublanguages.* IBM Corporation, 1972.

[Con71] J. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, 1971.

[CV85] A. K. Chandra and M. Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[DGG+09] N. Douib, O. Garbe, D. Günther, D. Oliana, J. Kroniger, F. Lücke, T. Melikoglu, K. Nordmann, G. Özen, T. Schlitt, L. Schmidt, J. Westhoff, and D. Wolff. PG 530 — pattern based schema languages. Technical report, TU Dortmund, 2009.

[DSD02] DSD. Document structure description (DSD). `http://www.brics.dk/DSD/`, 2002.

[EKSW04] K. Ellul, B. Krawetz, J. Shallit, and M. Wang. Regular expressions: new results and open problems. *Journal of Automata, Languages, and Combinatorics*, pages 233–256, 2004.

[EZ76] A. Ehrenfeucht and H. P. Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12(2):134–146, 1976.

[FGMV04] D. Fiorello, N. Gessa, P. Marinelli, and F. Vitali. Dtd++ 2.0: Adding support for co-constraints. In *Extreme Markup Languages*, 2004.

[Fig10] D. Figueira. *Reasoning on Words and Trees with Data.* Phd thesis, Laboratoire Spécification et Vérification, ENS Cachan, France, December 2010.

[GGM12] W. Gelade, M. Gyssens, and W. Martens. Regular expressions with counting: Weak versus strong determinism. *SIAM Journal on Computing*, 41(1):160–190, 2012.

[GJ08] H. Gruber and J. Johannsen. Optimal lower bounds on regular expression size using communication complexity. In *Foundations of Software Science and Computation Structures*, pages 273–286, 2008.

[GM78] H. Gallaire and J. Minker, editors. *Logic and Data Bases*. Perseus Publishing, 1978.

[GMUW02] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems – The Complete Book*. Prentice Hall, 2002.

[GN08] W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. In *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 325–336, 2008.

[GN11] W. Gelade and F. Neven. Succinctness of pattern-based schema languages for XML. *Journal of Computer and System Sciences*, 77(3):505–519, 2011.

[GO99] E. Grädel and M. Otto. On logics with two variables. *Theoretical Computer Science*, 224(1-2):73–113, 1999.

[GSMT+12] S. Gao, C. Sperberg-McQueen, H. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema definition language (XSD) 1.1 part 1: Structures. `http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/`, April 2012.

[HL03] S. Hartmann and S. Link. More functional dependencies for XML. In L. Kalinichenko, R. Manthey, B. Thalheim, and U. Wloka, editors, *Advances in Databases and Information Systems*, volume 2798 of *LNCS*, pages 355–369. Springer Berlin / Heidelberg, 2003.

[HSW06] Y.-S. Han, K. Salomaa, and D. Wood. Prime decompositions of regular languages. In *International Conference on Developments in Language Theory (DLT)*, pages 145–155, 2006.

[JEd] jEdit programmer's text editor. `www.jedit.org`.

[JR93] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.

[Kin81] J. J. King. *Query Optimization by Semantic Reasoning*. PhD thesis, Stanford University, 1981.

[KS07] G. Kasneci and T. Schwentick. The complexity of reasoning about pattern-based XML schemas. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 155–164, 2007.

[Kun07] M. Kunc. What do we know about language equations? In *International Conference on Developments in Language Theory (DLT)*, pages 23–27, 2007.

[KW80] C. Kintala and D. Wotschke. Amounts of nondeterminism in finite automata. *Acta Informatica*, 13:199–204, 1980.

[KW07] L. Kot and W. M. White. Characterization of the interaction of XML functional dependencies with DTDs. In *International Conference Database Theory (ICDT)*, pages 119–133, 2007.

[LBC14] P. Lu, J. Bremer, and H. Chen. Deciding determinism of regular languages. draft, 2014.

[LLL02] M. Lee, T. Ling, and W. Low. Designing functional dependencies for XML. In *International Conference on Extending Database Technology (EDBT)*, pages 145–158. 2002.

[LMN12] K. Losemann, W. Martens, and M. Niewerth. Descriptional complexity of deterministic regular expressions. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 643–654, 2012.

[Los10] K. Losemann. Boolesche Operationen auf deterministischen regulären Ausdrücken. Master's thesis, TU Dortmund, October 2010.

[Mar05] M. Marx. First order paths in ordered trees. In *International Conference Database Theory (ICDT)*, pages 114–128, 2005.

[Mei10] M. Meier. *On the termination of the chase algorithm.* PhD thesis, University of Freiburg, 2010.

[MLMK05] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[MMN+14] W. Martens, V. Mattick, M. Niewerth, S. Agarwal, N. Douib, O. Garbe, D. Günther, D. Oliana, J. Kroniger, F. Lücke, T. Melikoglu, K. Nordmann, G. Özen, T. Schlitt, L. Schmidt, J. Westhoff, and D. Wolff. Design of the BonXai schema language. Available at `http://ls1-www.cs.tu-dortmund.de/cms/bonxai/`, Draft 2014.

[MMS79] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):455–469, 1979.

[MN07] W. Martens and J. Niehren. On the minimization of XML Schemas and tree automata for unranked trees. *Journal of Computer and System Sciences*, 73(4):550–583, 2007.

[MNNS12] W. Martens, F. Neven, M. Niewerth, and T. Schwentick. Developing and analyzing XSDs through BonXai. *International Conference on Very Large Data Bases (VLDB)*, 5(12):1994–1997, 2012.

[MNS07] W. Martens, F. Neven, and T. Schwentick. Simple off the shelf abstractions of XML Schema. *SIGMOD Record*, 36(3):15–22, 2007.

[MNS09] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing*, 39(4):1486–1530, 2009.

[MNS10] W. Martens, M. Niewerth, and T. Schwentick. Schema design for XML repositories: Complexity and tractability. In *International Symposium on Principles of Database Systems (PODS)*, pages 239–250, 2010.

[MNSB06] W. Martens, F. Neven, T. Schwentick, and G. Bex. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, 2006.

[MS04] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[MS06] A. Møller and M. Schwartzbach. *An introduction to XML and web technologies*. Addison-Wesley, 2006.

[NS11] M. Niewerth and T. Schwentick. Two-variable logic and key constraints on data words. In *International Conference Database Theory (ICDT)*, pages 138–149, 2011.

[NS14] M. Niewerth and T. Schwentick. Reasoning about XML constraints based on XML-to-relational mappings. In *International Conference Database Theory (ICDT)*, pages 72–83, 2014.

[NS15] M. Niewerth and T. Schwentick. Reasoning about XML constraints based on XML-to-relational mappings. *Theory of Computing Systems*, 2015. submitted.

[PGM+12] D. Peterson, S. Gao, A. Malhotra, C. Sperberg-McQueen, H. Thompson, and P. Biron. W3C XML Schema definition language (XSD) 1.1 part 2: Datatypes. `http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/`, April 2012.

[PV00] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *International Symposium on Principles of Database Systems (PODS)*, pages 35–46, 2000.

[RG03] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[Sal08] K. Salomaa. Language decompositions, primality, and trajectory-based operations. In *International Conference on Implementation and Application of Automata (CIAA)*, pages 17–22, 2008.

[Sch99] Schematron. Schematron. `http://www.schematron.com/`, 1999.

[Sch10] L. Schmidt. Konvertierung zwischen RELAX NG, XML Schema und Document Type Definition. Master's thesis, TU Dortmund, 2010.

[SRM05] H. Su, E. A. Rundensteiner, and M. Mani. Semantic query optimization for xquery over xml streams. In *International Symposium on Principles of Database Systems (PODS)*, pages 277–288, 2005.

[SSY08] A. Salomaa, K. Salomaa, and S. Yu. Length codes, products of languages and primality. In *International Conference on Language and Automata Theory and Applications (LATA)*, pages 476–486, 2008.

[Sto74] L. Stockmeyer. The complexity of decision problems in automata and logic, 1974. Ph.D. Thesis, MIT, 1974.

[SY99] A. Salomaa and S. Yu. On the decomposition of finite languages. In *International Conference on Developments in Language Theory (DLT)*, pages 22–31, 1999.

[To10] A. W. To. Parikh images of regular languages: Complexity and applications. *Computing Research Repository*, abs/1002.1464, 2010.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[van97] P. van Emde Boas. The convenience of tilings. In A. Sorbi, editor, *Complexity, Logic and Recursion Theory*, volume 187 of *Lecture Notes in Pure and Applied Mathematics*, pages 331–363. Marcel Dekker Inc., 1997.

[Var95] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop (BANFF)*, pages 238–266, 1995.

[VLL04] M. W. Vincent, J. Liu, and C. Liu. Strong functional dependencies and their application to normal forms in XML. *ACM Transactions on Database Systems*, 29(3):445–462, September 2004.

[Wie09] W. Wieczorek. An algorithm for the decomposition of finite languages. *Logic Journal of the IGPL*, 2009. Appeared on-line August 8, 2009.

[WLA+00] L. Wood, A. Le Hors, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor, and C. Wilson. Document Object Model (DOM) level 1 specification (second edition). `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`, September 2000.

[Wol10] D. Wolff. Untersuchung und Implementierung von Durchschnitt, Vereinigung und Differenz für XML Schema. Master's thesis, TU Dortmund, 2010.

[Yu97] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 2. Springer, 1997.