FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

XML Data Integration

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

im Rahmen des Erasmus-Mundus-Studiums

Computational Logic

eingereicht von

Ilina Stoilkovska
Matrikelnummer 1328320

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Projektass. Dr.techn. MSc Vadim Savenkov

Wien, August 27, 2014

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

XML Data Integration

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science (M.Sc.)

in

Computational Logic

by

Ilina Stoilkovska
Registration Number 1328320

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Assistance: Projektass. Dr.techn. MSc Vadim Savenkov

Vienna, August 27, 2014 _____ _____

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Ilina Stoilkovska
Herndlgasse 20/10, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

I would like to express my gratitude to the people without whom this thesis would not have been possible.

First, I would like to thank my supervisor, Reinhard Pichler, for his support, encouragement and helpful comments in the process of writing this thesis. Many thanks to Vadim Savenkov, for the guidance, advice and countless discussions on the topic throughout the development of the work. Their contribution to this thesis is invaluable.

I thank the Joint Commission of EMCL for giving me the opportunity to be a part of this master program and for supporting me financially during my studies. I am grateful to all the people involved in the organization of this program, especially for making the transition from one university to the other as smooth as possible.

To my family, relatives and friends who are always believing in me and encouraging me no matter how far apart we are. Thank you for your love and support. To all my classmates in Dresden, Bolzano, Vienna and Lisbon - thanks for all the fun, traveling, SLUB-ing and all the nice memories we made together during the past two years.

Abstract

Since the publication of Codd's paper, the relational databases have dominated the database world and are still in wide use nowadays. With the advance of the Web technologies, the database research community has oriented its focus on bridging the gap between the traditional ways of storing data using relational databases and the novel techniques of transferring data on the Web. XML has emerged as a standard for data transmission on the Web, by setting its main goal to be providing a simple and efficient way of storing and transferring data. Problems such as data integration, data exchange and answering queries using views have become a topic of interest in recent years, and their formalization in an XML setting has received a significant amount of attention. In this thesis, we propose a unified framework for analyzing and comparing the features of a multitude of works that analyze these problems in the context of XML. We introduce a query language for XML trees, called extended tree patterns, which allows us to define XML mapping assertions that successfully capture the expressive power of the mapping assertions used in a large subset of the works that we overview. We classify different approaches based on the expressive power of their mapping assertions and point out the similarities and differences along several criteria. Finally, we give an overview of the problems that have been addressed so far, and identify which specific variants of the respective problems have not been tackled yet.

Kurzfassung

Seit der Veröffentlichung von Codd's Artikel haben relationale Datenbanken die Welt der Datenbanken dominiert und ihre Verwendung ist auch heute noch weit verbreitet. Mit dem Aufkommen von Internet-Technologien hat die Datenbankforschung ihren Fokus darauf gelegt, die Lücke zwischen traditionellen Arten der Datenspeicherung mittels relationaler Datenbanken und neuen Techniken des Datentransfers über das Internet zu überbrücken. XML hat sich hierbei als ein Standard zur Übermittlung von Daten über das Internet etabliert, da das Hauptziel von XML in der Bereitstellung einfacher und effizienter Methoden der Datenspeicherung und -übertragung liegt. Probleme wie die Integration von Daten, der Datenaustausch und die Beantwortung von Abfragen mit Hilfe von Sichten haben in den vergangenen Jahren großes Interesse erweckt, und die Formalisierung dieser Probleme in XML hat viel Aufmerksamkeit erlangt. In dieser Masterarbeit schlagen wir ein vereinheitlichtes Gerüst zur Analyse und zum Vergleich von Eigenschaften einer Vielzahl von Werken, welche diese Probleme im Kontext von XML analysieren, vor. Wir führen eine Abfrage-Sprache für XML-Bäume ein, sogenannte 'extended tree patterns', welche es uns erlauben 'xml mapping assertions' zu definieren, die wiederum erfolgreich die Ausdruckskraft der 'mapping assertions' charakterisieren, welche in einem Großteil der Arbeiten, die wir aufführen, verwendet werden. Wir klassifizieren verschiedene Herangehensweisen, basierend auf der Ausdruckskraft ihrer 'mapping assertions', und zeigen Ähnlichkeiten und Unterschiede in Bezug auf eine Reihe von Kriterien auf. Letztendlich geben wir einen Überblick über die bisher gelösten Probleme und heben hervor, welche spezifischen Varianten der jeweiligen Probleme bisher noch nicht gelöst worden sind.

Contents

CHAPTER 1

Introduction

We live in a world that is based on data. Our everyday activities involve tasks that rely on data transmission, even if sometimes we are not aware of it. Whether we read our emails, take money out from a cash machine or apply for a job or a university, our actions are driven by the data we provide to the systems we are interacting with, and the system's way of handling these data. Over the years, a lot of research has been focused on finding good and efficient methods for data management. As the contemporary information systems heavily depend on the data they get as input, data management is a crucial ingredient in their design.

The data in the information systems are usually structured in databases. Databases are collections of data organized in some particular way, such that the organization reflects a particular state of the world. The data stored in a database is manipulated through a database management system (DBMS). In the 1960s, a three-level architecture for database design was developed, and it separates a database into three levels: a physical, a logical and an external level [3]. The physical level is concerned with how the data is stored on the physical devices. The logical level contains definitions of the logical structure of the data. The external level consists of views over the data which allow the data in the database to be seen from a different angle. The logical level is of particular interest in the database theory research. Over the years, different logical data models have been developed, such as the hierarchical, network, relational, object oriented, and post-relational. One of the most widely used and heavily researched is the relational model, introduced by Codd in 1970 [17]. The relational model requires that the data are organized in tuples, which are stored in relations. The relations can also be seen as tables, and their columns as attributes. Each relation and attribute have their own names. It is also useful to distinguish between the definition of the relations and their attributes (called a database schema) and the actual data (called a database instance).

1.1 Motivation

In recent years, much attention has been oriented towards the post-relational data models. One of the many models that are referred to as post-relational is the XML data model [48]. In this

model, the data are stored in XML documents, and can be queried, accessed and exported in a desired format. XML has been widely used recently because it was accepted as a standard for data transmission on the Web. This has motivated the research community to revisit many problems which are already well defined and researched for the relational databases and adapt them in order to find suitable solutions in the XML setting. Such problems include, among others, XML data integration, XML data exchange and answering XML queries using XML views.

Given a setting with a large number of databases, it is not unnatural to think of a scenario where some of them in fact contain information that refers to the same state of the world. Think of a university that stores data about its students enrolled at different faculties. If the data are stored at the faculty level, the university information system will need to manage several number of different databases, depending on the number of faculties in the university. Also, it is not excluded that these databases might have different schemas. Therefore, in order to obtain information for a particular student on the university level, one would need to write queries specific for the database of the faculty the student is enrolled at.

Here is where data integration intervenes, and makes this querying process more convenient for the end user. With a data integration system, the query for the student is posed over a global schema, which is connected to the sources by means of mappings. In this way, the global schema creates a virtual view over the data from the different sources (i.e. the faculty databases). The mappings that are used to connect the sources to the global schema are in fact schema mappings from the source schemas to the global schema. The main task of the data integration system is to use these mappings to translate the query posed over the global schema into several queries over the source schemas, extract the answers from the corresponding source databases and finally combine these answers into a single answer of the query posed by the user. In other words, if the university in our example decides to implement a data integration system and someone from its administration asks for a student enrolled at some faculty, then the user just formulates the desired query in the vocabulary of the global schema. The data integration system does all the work; it rewrites the query into queries over the source schemas using the mappings, it finds the particular student and it returns it as an answer to the user.

A special setting of data integration corresponds to another problem, namely the problem of query answering using views [29]. The problem of query answering using views can be applied in a setting where the data from the sources is not available explicitly, but through a set of views defined over the sources. Given a set of views and a query, both defined in a certain query language, the goal of query answering using views is to find a rewriting of the original query such that the rewriting refers only to the views in the predefined set and returns the same answers as the original query.

Another data management task that uses schema mappings in its core is data exchange. Given two schemas called a source and a target schema, the task of data exchange is to restructure the data stored in source databases that satisfy the source schema, such that they conform to the target schema. The query posed over a data exchange setting is answered using the materialized target instance. In the university domain described above, data exchange may come into play in a scenario where a faculty needs to change the structure of the data it stores. This means that the data already stored under the old database schema needs to be translated into data that conforms

to the new schema. The queries posed to the faculty database after the translation has been made would also return answers that were previously stored under the old database schema.

All of these problems have been already researched extensively in the context of relational databases. With the expansion of the XML data model, a lot of interest has been directed towards formally defining these problems and providing solutions for them in an XML context. Throughout the literature, there are numerous works that have addressed these problems by trying to adapt the existing formalisms for relational databases into formalisms for XML. Although these works are related in their perspective and understanding of the respective problems, they differ greatly in the way they represent and study them. Namely, different works use different languages that have different expressive powers for querying XML documents and expressing schema mapping assertions. Also, the formal definitions of XML documents vary from one approach to another.

1.2 Goals and Methodology

To the best of our knowledge, a survey on data management in an XML setting has not been written yet. The goal of this work is to provide a thorough overview of the works that address problems in XML data management and to make a first step towards a comprehensive survey on the topic. We aim at bringing together the different approaches and provide a uniform view over their formalisms. We focus primarily on understanding the formalisms introduced in the works that study several XML data management problems, such as XML data exchange, XML data integration and XML query answering using XML views. Along this, another goal of this work is proposing a framework for uniformly representing different works that revolve around similar topics in XML data management.

In this thesis, we intend to compare different approaches on XML data management by using a newly defined framework. This framework includes a definition of a query and a mapping language for XML documents. By defining such a framework that unifies different approaches, we point out the similarities and differences between them. Principally, our analysis is aiming at showing which approaches are comparable w.r.t. the expressive power of the query and mapping languages they use. Moreover, we plan to provide comparisons of the works along several other dimensions, such as the number of data sources (i.e. XML documents) considered, the type of query answering, the interpretation of the schema mappings as well as the presence of constraints. We expect to gain a further insight into the problems that have already been tackled for specific query and mapping languages, and into those that remain as open questions. This is a valuable result, as it facilitates the understanding of the problems that have been solved so far, and furthermore, since it provides directions for future research.

We analyze the approaches and compare them by means of two new formalisms that we introduce. First, we introduce a query language for XML, called extended tree patterns. The expressive power of this language matches the expressive powers of a large set of query languages used throughout the literature. We provide means to translate queries written in different query languages into the uniform language of extended tree patterns. Based on the extended tree patterns, we define the second formalism, called XML mapping assertions. Using this formalism, we are able to formally and uniformly represent a multitude of approaches for XML data

management. However, not all of the approaches considered in this thesis are expressible using the XML mapping assertions that we define. We identify the works whose approaches are not expressible using our formalisms and give explanations why this is the case.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows. In Chapter 2 we give more details on the problems that have been very well researched in the relational database setting, and to which we focus our attention in the XML setting. Such problems include databases with incomplete information, data integration, data exchange and query answering using views. In Chapter 3 we introduce formal definitions of XML documents as well as XML query languages that can be encountered in the literature. In Chapter 4 we introduce the framework which we use to compare the different approaches, namely the unifying query language and the XML mapping assertions. Chapter 5 contains the main contributions of this thesis. In this section, we provide the comparisons of the different approaches using the previously defined framework. We point out three different classes of approaches, we identify which of them are expressible using the formalisms we define and we identify which problems still remain as open in the context of XML data management. Finally, Chapter 6 concludes this thesis.

CHAPTER 2

Query Answering Under Schema Mappings in Relational Databases

Schema mappings are logical expressions that are used to specify high-level relationships between two database schemas [32]. Many data management problems are based on schema mappings, such as data integration, data exchange, database management, peer-to-peer database systems as well as metadata management. In this thesis, we mainly focus on schema mappings used in data integration and data exchange, along with the closely related problems of databases with incomplete information and query answering using views.

Given two database schemas, usually called a source and target schema (in the context of data exchange), or a source and global schema (in the context of data integration), a schema mapping is a set of assertions that specify the correspondence between the two schemas. Data exchange focuses on transforming the data residing at a database that satisfies the source schema into data that satisfies the target schema and the schema mapping assertions, which in a data exchange setting are called dependencies. Very often, several target instances can be obtained from a single source instance. Such target instances are called solutions. On the other hand, in data integration, the goal is to uniformly view and query heterogeneous sources using a global schema, which plays the role of a virtual database. The global database is not materialized, and it is used as a unified vocabulary for posing queries. The schema mapping assertions specify which data stored at the sources corresponds to which global schema element. More details on data integration and data exchange can be found in Section 2.2 and Section 2.4 respectively.

Query answering in data exchange and data integration is performed in two different ways. Namely, since in data exchange materialized target instances are present, the queries are answered using the data stored in these instances. In data integration however, the global database is virtual, hence the query posed over the global schema needs to be translated into one or more queries over the sources. The process of translating the query is called query rewriting or query reformulation. Although the two approaches of query answering are different, in both cases the same semantics is adopted. This is because in both data integration and data exchange, the query is not posed over a single database, but rather over a virtual global database that refers

to multiple source databases in the former, and over multiple solutions in the latter. Thus, as semantics for query answering, a concept that emerged from databases with incomplete information, called certain answers, is used. The notion of *certain answers* is defined as a set of tuples that is an intersection of the tuples occurring in the answer of the query over all available databases. Intuitively, the certain answers are those tuples that are always returned as answers to the query, regardless of the database used to extract them. The problem of finding the certain answers given a set of databases $\mathbf{D}$ can be formulated as the following decision problem:

PROBLEM:	CERTAINANSWER$_{\mathbf{D}}(q, \bar{t})$
shorthand:	CA$_{\mathbf{D}}(q, \bar{t})$
INPUT:	A query q and a tuple $\bar{t}$ of the same arity as q
QUESTION:	Is $\bar{t}$ a certain answer of q?

In data integration, the data is usually accessible through materialized views defined over the sources. This is due to the form of the mapping assertions used in data integration, which will be discussed in more detail in Section 2.2. Answering a query using a set of materialized views is referred to as view based query processing or query answering using views. When producing a rewriting of a query using a set of views, the goal is to obtain a rewriting that returns exactly those answers that the original query would return. Such a rewriting is called an *equivalent rewriting*. However, this may not always be the case, as an equivalent rewriting may not exist. Hence, a computation of another rewriting that approximates best the original query is of interest. A *maximally contained rewriting* is an expression that captures the original query the best by returning the maximal number of tuples that are contained in the answer of the original query. More details on the problem of answering queries using views can be found in Section 2.3. The computation of a rewriting can be expressed as a decision problem, and in fact we consider the two different flavors of rewriting. When defining the decision problem, it is important to differentiate between the query language $\mathcal{L}_1$ in which the original query is expressed, and the query language $\mathcal{L}_2$ of the rewritten query. Also, we need to take into account a set of mapping assertions $\mathcal{M}$, which correspond to the mapping assertions in data integration or the set of view definitions in the case of answering queries using views. The set of mapping assertions can either be fixed or be considered as part of the input to the problem. Thus, if the mapping assertions are not considered as part of the input, we define the following decision problems:

PROBLEM:	EQUIVALENTREWRITING$_{\mathcal{M}}^{\mathcal{L}_1,\mathcal{L}_2}(q)$
shorthand:	ER$_{\mathcal{M}}^{\mathcal{L}_1,\mathcal{L}_2}(q)$
INPUT:	A query q expressed in $\mathcal{L}_1$
QUESTION:	Does there exist an equivalent rewriting r expressed in $\mathcal{L}_2$ of q using $\mathcal{M}$?

PROBLEM:	MAXIMALLYCONTAINEDREWRITING$_{\mathcal{M}}^{\mathcal{L}_1,\mathcal{L}_2}(q)$
shorthand:	MCR$_{\mathcal{M}}^{\mathcal{L}_1,\mathcal{L}_2}(q)$
INPUT:	A query q expressed in $\mathcal{L}_1$
QUESTION:	Does there exist a maximally contained rewriting r expressed in $\mathcal{L}_2$ of q using $\mathcal{M}$?

In the case when the mapping assertions are part of the input, we have the following decision problems:

PROBLEM:	EQUIVALENTREWRITING$^{\mathcal{L}_1,\mathcal{L}_2}(q, \mathcal{M})$
shorthand:	ER$^{\mathcal{L}_1,\mathcal{L}_2}(q, \mathcal{M})$
INPUT:	A query q expressed in $\mathcal{L}_1$ and a set of mapping assertions $\mathcal{M}$
QUESTION:	Does there exist an equivalent rewriting r expressed in $\mathcal{L}_2$ of q using $\mathcal{M}$?

PROBLEM:	MAXIMALLYCONTAINEDREWRITING$^{\mathcal{L}_1,\mathcal{L}_2}(q, \mathcal{M})$
shorthand:	MCR$^{\mathcal{L}_1,\mathcal{L}_2}(q, \mathcal{M})$
INPUT:	A query q expressed in $\mathcal{L}_1$ and a set of mapping assertions $\mathcal{M}$
QUESTION:	Does there exist a maximally contained rewriting r expressed in $\mathcal{L}_2$ of q using $\mathcal{M}$?

In the rest of the section, we give an introduction to some of the problems that use schema mappings in a relational setting, which we are interested to analyze in an XML setting. Such problems include databases with incomplete information, data integration, data exchange and answering queries using views.

2.1 Incomplete Information

Very often, the data stored in a database might have missing values, and hence it does not provide a complete description of the application domain. The incompleteness of information is an important problem that requires special attention and has motivated a high amount of research in the database community. The research is mainly focused on finding ways of representing the incomplete databases, as well as on answering queries over such databases. Incomplete information in databases appears due to absence, irrelevance or fuzziness of the information. The most common reason for incompleteness is the absence of information. In terms of relational databases, absence of information is simply a missing value for an attribute in a record of some relation. These missing values are referred to as *null values* (or *nulls*). The paper by Imeliński and Lipski [30] provides the theoretical foundations of handling nulls in relational databases, and formalizes a representation system for such incomplete relational databases. Other works that address the problem of handling incomplete information in relational databases are [4, 45]. When handling database tables with nulls, a new semantics should be specified, since the nulls are treated as variables, and therefore can obtain different values under different valuations. The semantics of an incomplete database table can be defined w.r.t. the closed world assumption (CWA) or open world assumption (OWA). The existence of multiple valuations of the null values in an incomplete database implies the existence of multiple ground instances of a single database. When answering queries over such databases, it is important to consider only those tuples that appear in the answer of the query in every database instance. Hence, the certain answer semantics is taken into account when answering queries over incomplete databases. In fact, certain answers are a notion that has been introduced first in query answering over incomplete

databases, and later reused in query answering w.r.t. schema mappings. More precisely, the certain answers extract the maximum knowledge from a collection of ground database instances. Thus, we can see that there is a tight relationship between handling incomplete information on one side, and data integration and data exchange on the other. More on data integration and data exchange can be found in Section 2.2 and Section 2.4.

2.2 Data Integration

As briefly mentioned in the introduction, data integration is the problem of creating a uniform query interface over data from multiple heterogenous sources, that conform to different schemas [29]. This query interface allows the data stored in the different sources to be uniformly accessible through a virtual database that conforms to a schema, called global or mediated schema. In a data integration system, the user poses queries that use symbols from the global schema. In order to obtain data from the sources, the user query over the global schema is transformed into a set of queries over the sources. Once the answers are obtained, they are combined and returned to the user. The data integration system facilitates the query answering for the user. When posing queries over the global schema, the user does not need to know the vocabularies of the source schemas. It is also not of the user's concern how the data is stored in the sources, and which data model is used to structure the data at each source. The data integration system acts like a black box and provides an answer to a query, which should be equal to the union of the answers of the corresponding queries posed over the available sources.

A logical framework and a formal definition of a data integration system is given in [34]. This definition is a general one, i.e. it does not depend on the technique used in the design of the data sources. Formally, a data integration system $\mathcal{I}$ is a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, such that:

- $\mathcal{G}$ is the global or mediated schema, which provides a unified view over the data from the sources;
- $\mathcal{S}$ is the source schema, which describes the sources where the actual data is stored;
- $\mathcal{M}$ is the mapping between the source and the global schema. It consists of mapping assertions of the form $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$ and $q_{\mathcal{G}} \rightsquigarrow q_{\mathcal{S}}$, where $q_{\mathcal{S}}$ is a query over the source schema $\mathcal{S}$ and $q_{\mathcal{G}}$ is a query over the global schema $\mathcal{G}$. The mapping assertions describe the correspondence between the source and the global schema.

The semantics of a data integration system $\mathcal{I}$ is defined as follows. Let $\mathcal{D}$ be a source database, i.e. a database that conforms to the source schema $\mathcal{S}$ and satisfies the constraints implied by it. A global database is any database that satisfies the global schema $\mathcal{G}$. A global database $\mathcal{B}$ is *legal w.r.t.* $\mathcal{D}$ if $\mathcal{B}$ satisfies all the constraints imposed by $\mathcal{G}$ and $\mathcal{B}$ satisfies the mapping $\mathcal{M}$ w.r.t. $\mathcal{D}$. Hence, there might exist multiple legal global databases w.r.t. $\mathcal{D}$ and the need of a definition of certain answers of a query posed to $\mathcal{I}$ emerges. Intuitively, certain answers are those tuples that are answers to the query and appear in every legal global database. Suppose $\mathcal{D}$ is a source database (i.e. an instance of the source schema) for the data integration system $\mathcal{I}$, and q is a query over $\mathcal{I}$. The set of tuples t that are in the answer of q for every legal global database $\mathcal{B}$ is called the set of *certain answers* of q over $\mathcal{I}$ w.r.t. $\mathcal{D}$ and is written as $q^{\mathcal{I},\mathcal{D}}$.

Whether a global database $\mathcal{B}$ is legal w.r.t. a source database $\mathcal{D}$ depends on the interpretation of the mapping assertions in $\mathcal{M}$. There are three kinds of mapping assertions based on the expressive power of $q_{\mathcal{S}}$ and $q_{\mathcal{G}}$:

1. local as view (LAV) - In the LAV setting, each element of the source schema is associated with a query (i.e. a view) over the global schema. Hence, the mapping assertions are of the form $s \rightsquigarrow q_{\mathcal{G}}$, where s is an element from $\mathcal{S}$;

2. global as view (GAV) - On the other hand, the GAV mapping assertions describe the elements of the global schema using a view over the sources. Thus, the mapping assertions in a GAV setting have the following layout: $g \rightsquigarrow q_{\mathcal{S}}$;

3. global-local as view (GLAV) - The GLAV setting is a generalization of both LAV and GAV, where a view over the source schema is assigned a view over the global schema, and therefore, the mapping assertions are of the form $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$. As a language for description of the sources, GLAV is more expressive than LAV and GAV combined [28].

The three types of mapping assertions differ in the way they perform query processing. In a LAV setting, query processing is a rather involved task. Since the sources in a LAV data integration system are represented as views over the global schema, the query processing amounts to computing answers to the query based on the views. Two types of query processing using views can be considered: query rewriting using views and query answering using views. The former considers a query and a set of view definitions and as a result reformulates the original query such that it refers only to the views. The latter is given the query, the view definitions and the view extensions as input, and its goal is to compute answers to the query using the data stored in the view extensions, regardless of the means used to process the query and extract the answers. GAV query processing is slightly more straightforward than LAV query processing. In this setting, each member of the global schema is described by a query (i.e. a view) over the source schema. Hence, query processing in a GAV data integration system consists of replacing each occurrence of an element of the global schema by the source query that defines it. Finally, in GLAV query processing, one can perform query processing by splitting a GLAV mapping assertion into two mapping assertions, one of which is a GAV and another one which is a LAV mapping assertion [16, 28] in the following way:

- for each GLAV mapping assertion $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$, introduce an intermediate view symbol v that has the same arity as $q_{\mathcal{S}}$ and $q_{\mathcal{G}}$ and conforms to an intermediate schema;

- split each GLAV mapping assertion $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$ into a GAV mapping assertion $q_{\mathcal{S}} \rightsquigarrow v$ and a LAV mapping assertion $v \rightsquigarrow q_{\mathcal{G}}$;

- materialize the intermediate view v for each GAV mapping assertion $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$;

- perform LAV query processing using the LAV mapping assertion and the result of the materialized intermediate view v.

In order to define the relationship between the source and the global schema more precisely, each mapping assertion needs to be given a specification which further defines how the mapping

assertion is interpreted. Thus, a mapping assertion $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$ is assigned one of the following specifications:

- sound, interpreted as $\forall \overline{x}\ (q_{\mathcal{S}}(\overline{x}) \rightarrow q_{\mathcal{G}}(\overline{x}))$;
- complete, interpreted as $\forall \overline{x}\ (q_{\mathcal{S}}(\overline{x}) \leftarrow q_{\mathcal{G}}(\overline{x}))$;
- exact, interpreted as $\forall \overline{x}\ (q_{\mathcal{S}}(\overline{x}) \leftrightarrow q_{\mathcal{G}}(\overline{x}))$.

Sound and exact mapping assertions are the most commonly encountered ones in data integration systems.

2.3 Query Answering Using Views

From another perspective, data integration can be seen as the problem of answering queries over materialized views [29]. This particularly corresponds to the one of the approaches of query processing in a LAV data integration system, described above. The problem of query answering using views, as well as its applications, is studied in [29], and is defined as follows. Let Q be a query and $\mathcal{V} = \{V_1, \ldots, V_n\}$ a set of view definitions. A query Q' that mentions only the views $V_1, \ldots, V_n$ is called a rewriting of Q using the views. When computing the rewritings, one is interested in finding a rewriting that returns answers that approximate the original query the best. In order to determine the relationship between the original query and a rewriting, the concepts of query containment and equivalence are used. A query Q_1 is contained in a query Q_2 if for all database instances D it holds that the set of tuples corresponding to the answer of Q_1 over D is a subset of the tuples in the answer of Q_2 over D. Given two queries Q_1 and Q_2, Q_1 is equivalent to Q_2 if Q_1 is contained in Q_2 and Q_2 is contained in Q_1.

Depending on the answers obtained by the rewritten query, two different types of rewritings can be distinguished:

- equivalent rewritings. A rewriting Q' is said to be an equivalent rewriting of Q using the views $V_1, \ldots, V_n$, if Q' refers only to the views $V_1, \ldots, V_n$ and is equivalent to Q, after the view definitions have been unfolded in the rewriting;
- maximally-contained rewritings. It is not always possible to obtain a rewriting that is equivalent to the original query. In such cases, it is of interest to obtain maximally-contained rewritings. A rewriting Q' is said to be a maximally-contained rewriting of Q w.r.t. the views $V_1, \ldots, V_n$ if Q' mentions only the views, it is contained in Q after the unfolding of the view definitions, and there is no other rewriting Q'' such that Q' is contained in Q'', Q'' is contained in Q and Q'' is not equivalent to Q.

As the purpose of queries is to extract answers, an important question is how to obtain all the answers of a query given a set of views. A natural idea is to find a rewriting and to evaluate this rewriting over the views. This evaluation will generate all the answers if the rewriting is equivalent, but this is not always the case. The notion of certain answers describes what it means to obtain all the answers of a query given a set of views and their extensions. Depending on the contents of the view extensions (i.e. whether they are complete or partial), there are two

different definitions for the certain answers. Suppose Q is a query and $\mathcal{V} = \{V_1, \ldots, V_n\}$ is a set of views s.t. the sets of tuples $v_1, \ldots, v_n$ are the extensions of the views $\{V_1, \ldots, V_n\}$ respectively. The following two cases can be distinguished:

1. a tuple $\bar{t}$ is a certain answer of Q under the closed world assumption (CWA) given $v_1, \ldots, v_n$ if for all database instances D, $\bar{t}$ is in the answer of Q over D, and additionally the tuples contained in each extension v_i are exactly those tuples obtained by evaluating each view V_i over D, for $1 \leq i \leq n$. In this case, we can also say that the views are defined using the exact semantics, i.e. using exact LAV mapping assertions;

2. a tuple $\bar{t}$ is a certain answer of Q under the open world assumption (OWA) given $v_1, \ldots, v_n$ if for all database instances D, $\bar{t}$ is in the answer of Q over D, and additionally the tuples contained in each extension v_i are contained in the tuples obtained by evaluating each view V_i over D, for $1 \leq i \leq n$. The OWA setting coincides with the definition of a LAV data integration system, where the mapping assertions are interpreted as sound LAV mappings.

The number of possible rewritings that can be generated for a query and a set of views is exponential in the size of the query [29]. However, usually, not all of the exponentially many rewritings are relevant for obtaining the answers, as most of them are contained in or equivalent to rewritings that are considered relevant (i.e. those that are equivalent or maximally-contained rewritings). Many algorithms have been developed to efficiently compute rewritings of a query using views in the context of data integration, such as the bucket algorithm [35] as well as its improvement, the MiniCon algorithm [42], and the inverse rules algorithm [22].

Some issues that deserve attention in the theory of answering queries using views are: the completeness of the query rewriting algorithms and the extraction of certain answers. A query rewriting algorithm is said to be complete if it finds a rewriting of a query Q in a given language using views $\mathcal{V}$ (often defined in the same language as the query) if one exists. The task of extracting certain answers depends on whether the views are interpreted under the exact or sound semantics (CWA vs. OWA) and whether the rewritings are equivalent or maximally contained. Also, another very important question that needs to be answered when generating a rewriting of a query using views is which views are relevant and should be considered in the rewriting process.

2.4 Data Exchange

Data exchange is another problem which has been revisited recently and has been a topic of interest in the research community, in both relational and XML settings. Given two schemas called a source and a target schema, the task of data exchange is to restructure the data stored in sources that satisfy the source schema, such that they conform to the target schema.

A formal framework for data exchange has been presented in [23, 24]. In the core of data exchange again lie schema mappings, which are necessary to express the relationship between the source and the target schema. A relational data exchange setting is represented by a triple

$\langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$, where $\mathbf{S}, \mathbf{T}$ are the source and the target schema respectively, and Σ is a set of dependencies, which can be either source-to-target or target dependencies. The source-to-target dependencies are of the form $\forall \overline{x}\ (\exists \overline{y}\ \varphi_{\mathbf{S}}(\overline{x}, \overline{y}) \rightarrow \exists \overline{z}\ \psi_{\mathbf{T}}(\overline{x}, \overline{z}))$, where $\varphi_{\mathbf{S}}$ and $\psi_{\mathbf{T}}$ are conjunctive queries over the source and target schema respectively. Their intuitive meaning is that whenever the query $\varphi_{\mathbf{S}}$ is satisfied in the source instance, the query $\psi_{\mathbf{T}}$ has to be satisfied in the target instance. The target dependencies are constraints imposed on the target schema. They impose restrictions on the target data, in the sense that the newly obtained target instance needs to satisfy each target dependency.

Suppose a data exchange setting $\langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$ is fixed. The data exchange problem is defined as follows. Given an instance I of the source schema $\mathbf{S}$, materialize an instance J of the target schema $\mathbf{T}$ such that I and J together satisfy the source-to-target dependencies and J satisfies the target dependencies. If it exists, such materialized target instance J is called a solution for I. Since in general there might exist more solutions for I, when answering queries over the target schema w.r.t. the source instance, it is useful to define *certain answers*. The set of certain answers of a query q is the intersection of the answers of q over each solution J for the source instance I.

The difference between data exchange and data integration is that in data exchange the target schema is materialized, i.e. the queries are posed over a materialized target instance, while in data integration the global schema serves only as a unified vocabulary used for formulating queries over multiple sources. Moreover, a data exchange setting $\langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$ can be seen as a data integration system $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where $\mathbf{S}$ corresponds to the source schema $\mathcal{S}$, $\mathbf{T}$ corresponds to the global schema $\mathcal{G}$ and the source-to-target dependencies from Σ correspond to the set of mapping assertions $\mathcal{M}$. Given the form of the source-to-target dependencies, the mapping assertions in the data integration system are interpreted as sound GLAV mappings.

CHAPTER 3

XML Data Model and Query Languages

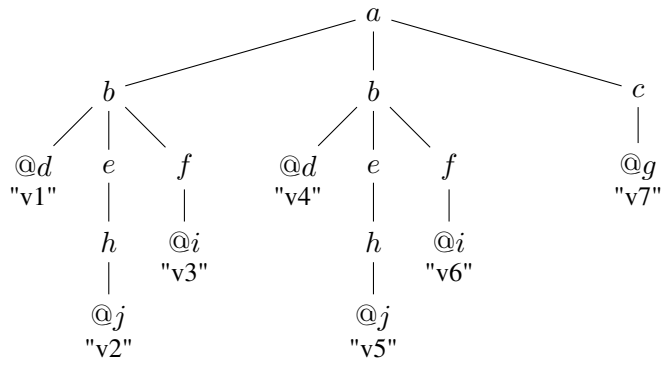
With the advance in the Web technologies, a need for a standard for publishing data on the Web has emerged. In its basic form, data publishing on the Web consists of several steps [2]. First, a user creates a file that contains some data. Then, this file is published by sharing its URL with other users. Finally, by accessing its URL, any user can reach and retrieve this file, at any time. The type of files usually transmitted on the Web are HTML files, which have a predefined structure that enables rendering of the contents of the file visually in a Web browser. On the contrary, the task of data retrieval from a relational database follows different steps. When the data is structured using a relational schema, one needs to pose queries in a given query language designed for querying relational databases in order to extract the data. In order to bring the power of database and Web technologies together, a new technology that would bridge the gap between the two was needed.

XML (eXtensible Markup Language) is a markup language that is used for transferring data on the Web, that has been accepted as a W3C Recommendation in 1998. Unlike HTML documents, whose main purpose is to provide rendering of a text content in a Web browser, XML documents are used to store data on the Web in a structured way. Their content is structured in nested tags, each of which has to be opened and closed. The tags are defined by the user. Furthermore, an additional document that specifies the allowed tags and their respective structure in the XML document might be attached to it. The objective of the tags is to describe the meaning of the part of the document that is enclosed within an opening and closing tag, rather than being concerned with its visual display. The pair of opening and closing tag with the same name, along with the embedded content, is called an *element*. The opening tag of the element can contain a set of name-value pairs, called *attributes*. An example of an XML document can be seen in Figure 3.1a.

In the rest of this section, we provide definitions of the basic notions that will be used throughout this thesis. We will formally define an XML document as an XML tree and we will give a definition of a DTD, which is used to specify the desired structure of the underlying

```
<a>
  <b @d="v1">
    <e>
      <h @j="v2" />
    </e>
    <f @i="v3" />
  </b>
  <b @d="v4">
    <e>
      <h @j="v5" />
    </e>
    <f @i="v6" />
  </b>
  <c @g="v7" />
</a>
```

(a) *XML document*



(b) *XML tree*

Figure 3.1: *An XML document and its representation as an XML tree*

XML document. We will also introduce and analyze different query languages for XML that have been encountered in the literature.

3.1 XML Documents as Trees

XML documents have a nested structure, which is characterized by elements containing nested subelements. Each element is assigned a label from a countably infinite set of element type names. Additionally, element nodes may maintain a list of attribute-value pairs. Following the formalization of XML documents in [9, 12], we assume that all the values (i.e. the data) in the XML documents are actually stored in the attributes. This is not an unreasonable assumption, since every document can be transformed into an equivalent one where all the values are stored in the attributes. For example, if an XML document contains a text node, it is transformed into an empty node with an attribute $@value$ whose value is set to the value of the text node.

In its simplest form, an XML document can be seen as a labeled ordered tree [46]. In accordance with the definitions in [9, 12], we proceed with defining XML trees as follows. Consider a countably infinite set $\mathcal{E}$ of element type names, a countably infinite set $\mathcal{A}$ of attribute names, a set S of strings, which is the domain of the attribute values, and a set ID of unique identifiers. Let $El \subset \mathcal{E}$ and $Att \subset \mathcal{A}$ be finite proper subsets of $\mathcal{E}$ and $\mathcal{A}$ respectively.

Definition 3.1.1 (XML tree)**.** An *XML tree* T over the finite sets El of elements and Att of attributes is a finite ordered labeled tree $(N, E, \downarrow, \rightarrow, label, value_{@a}, id, root)$ where

- N is the set of nodes, $N = N_{El} \cup N_{Att}$, where N_{El} is the set of element nodes and N_{Att} is the set of attribute nodes;
- E is the edge relation, $E \subseteq N_{El} \times (N_{El} \cup N_{Att})$;

- $\downarrow$ is the child relation, which is a subset of the edge relation. For two nodes $n_1, n_2 \in N_{El}$, $n_1 \downarrow n_2$ if n_2 appears immediately below n_1 in the labeled ordered tree. Moreover, in this case, n_1 is said to be the parent of n_2. The root of the tree is the only node that does not have a parent, while the nodes with no children are called leaf nodes;

- $\rightarrow$ is the next sibling relation, which imposes an order on the children of a node. For two nodes $n_1, n_2 \in N_{El}$ it holds that $n_1 \rightarrow n_2$ if n_1 and n_2 have the same parent (i.e. there exists a node $n \in N_{El}$ such that $n \downarrow n_1$ and $n \downarrow n_2$) and n_2 is the next node after n_1 in the children order of their parent;

- $label$ is a labeling function that assigns a label to each node, $label : N \rightarrow El \cup Att$, i.e. for an element node $n_e \in N_{El}$, if $label(n_e) = l$, then l is the element type of n_e, and for an attribute node $n_a \in N_{Att}$, if $label(n_a) = @l$, then $@l$ is the attribute name of n_a;

- for all attribute names $@a$, $value_{@a}$ is a partial function that assigns values to attributes, $value_{@a} : N_{El} \rightarrow \mathsf{S}$, i.e. for a node $n \in N_{El}$, if $value_{@a}(n) = v$, then the element node n has a value v for the attribute $@a$;

- id is an identification function that assigns a unique identifier to each node, $id : N \rightarrow \mathsf{ID}$;

- $root$ is the root of the tree.

We denote by $\mathcal{T}$ the set of all XML trees. □

Note that in our definition of XML documents as XML trees, we do not consider the document node of the XML document as a root node of the XML tree. We rather call a *root* the node that is the unique child of the document node, i.e. the element node that contains all other element nodes. We implicitly assume its existence on top of the root node, which is its only child. This is an important remark, as for some query languages the evaluation starts at the document node, while for others, it starts at the node that we call the root node.

The XML trees are also unranked trees, meaning that there is no bound on the number of children a node can have. Sometimes, the sibling ordering of the XML tree can impose constraints and increase the complexity of many XML data management tasks. Thus, in these cases it is useful to disregard the sibling ordering and consider the XML trees as unordered. Also, it is often useful to consider the reflexive and transitive closure of both the child and next sibling relation. These two relations are denoted by $\downarrow^*$ and $\rightarrow^*$ and called the *descendant* and *following sibling* relation respectively. Naturally, we can also distinguish the inverses of the relations defined for the XML tree:

- parent, inverse to the child relation;

- previous sibling, inverse to the next sibling relation;

- ancestor, inverse to the descendant relation;

- preceding sibling, inverse to the following sibling relation.

Example 3.1.1. Consider the XML tree T from Figure 3.1b. It has 18 nodes, 10 of which are element nodes, and the remaining ones are attribute nodes. The root of the tree is the node labeled by a. Let $n_c \in N$ be the single node labeled by c, i.e. $label(n_c) = c$. The pair of nodes $(root, n_c)$ is in the child relation $\downarrow$. We can see that n_c has an attribute named $@g$. The value assigned to this attribute is $value_{@g}(n_c) =$ "v7". △

XML documents may be associated with a document that defines their structure, such as a document type definition (DTD) [47] or an XML Schema (XSD) [50]. Hereafter, we focus on XML documents that use DTD for defining their structure. Formally, a DTD can be seen as an extended context free grammar, where the right-hand sides of the productions can contain regular expressions.

Definition 3.1.2 (DTD)**.** A *DTD* D over the finite sets $El \subset \mathcal{E}$ of elements and $Att \subset \mathcal{A}$ of attributes is a triple $(content, attlist, r)$ where:

- $content : El \to El_R$ is a function that assigns to each element of El a regular expression from the set El_R of regular expressions over El. The regular expressions in El_R are defined as follows:

$$e \quad ::= \quad \varepsilon \quad | \quad l \quad | \quad (e|e) \quad | \quad e,e \quad | \quad e^* \quad | \quad e^+ \quad | \quad e?$$

  where ε is the empty string, $l \in El$ is an element name, $(e|e)$ corresponds to a choice of children, e, e is a sequence of children, e^*, e^+ and $e?$ stand for zero or more occurrences, one or more occurrences and zero or one occurrences of e, respectively;

- $attlist : El \to 2^{Att}$ is a function that assigns to every element of El a set of attribute names;

- r is the element type of the root element node, which does not have a parent node and its set of attribute names is empty. □

It is usual to write the content of an element like a production rule, rather than using the function $content$. For example, let $El = \{a, b, c\}$, $Att = \emptyset$, and $content(a) = (b^*|c^+)$, $content(b) = \varepsilon$, $content(c) = \varepsilon$. Then we can write the following production rules:

$$a \to (b^*|c^+), \qquad b \to \varepsilon, \qquad c \to \varepsilon$$

When a DTD is specified, it is possible to check if the XML document indeed follows the rules described by it, i.e. if it has the desired structure. Given a DTD D and an XML tree $T \in \mathcal{T}$, we say that T conforms to D if:

- the label of the root of T is r;

- if an element node $n \in N_{El}$ has children $n_1, \ldots, n_m$, and $label(n) = l$, then the string $label(n_1) \ldots label(n_m)$ is in the language defined by the regular expression $content(l)$;

- for every element node $n \in N_{El}$ with $label(n) = l$, $value_{@a}(n)$ is defined iff $@a \in attlist(l)$.

DTDs can come in many layouts and flavors. Suppose that a DTD D is modeled as a connected directed graph $G = (V, E)$ where the set of nodes is the set El of element names and there exists an edge between two nodes n_1, n_2 if $\tilde{n}_2$ appears in $content(n_1)$, where $\tilde{n}_2$ is one of the following: $n_2, n_2?, n_2^*$ or n_2^+. Depending on the form of $\tilde{n}_2$, the edge between n_1 and n_2 is labeled with $1, ?, *$ or $+$ respectively. The node representing the root r of D does not have incoming edges, while all other nodes have an incoming degree of at least 1. The DTD D is said to be *recursive* if the graph G contains a cycle, and is called *non-recursive* otherwise. A special case of DTDs that reduces the complexity of many problems in the XML context and that is used in practice is the class of *nested-relational* DTDs. Formally, a nested-relational DTD D is a non-recursive DTD which contains productions of the form $a \rightarrow \tilde{b}_1, \ldots, \tilde{b}_n$, where all b_i's are different and each $\tilde{b}_i$ has one of the following forms: b_i, $b_i?$, b_i^* or b_i^+.

3.2 XML Query Languages

After the definition of the first version of XML, the need for a query language for querying the data stored in XML documents has arisen. [36] proposed a summary of the functionalities that a query language for XML should support, from a database point of view. Some of the recommended guidelines include: the query should have an XML representation and produce XML output, the language should support basic query operations such as selection, extraction, reduction, restructuring and combination, an additional support for mutual embedding with XML should be provided, etc. Over the years there have been many efforts of defining a suitable query language for XML documents. Early works have tried to adapt query languages for semistructured data such as Lorel [5] and make them compatible for querying XML documents. Other works have designed dedicated query languages for XML, including XQL [43], XML-QL [19], XQuery [51] and XPath [49].

XPath and XQuery

XPath is a query language for extracting data from XML documents. It is a navigational and declarative language, whose syntax is based on path expressions. It can be used for navigating an XML document, selecting nodes or computing values from the data stored in the document. Its expressions are in the core of the XML query language XQuery. Additionally, it is frequently used by XSLT [52], which is a language for transforming XML documents. Each path expression contains multiple location steps. Each location step is comprised of:

- an axis, which defines the direction of navigation w.r.t. the current node. It also defines the relationship between the current node and the selected nodes w.r.t. the XML tree. It can have one of the following values: `ancestor`, `ancestor-or-self`, `attribute`, `child`, `descendant`, `descendant-or-self`, `following`, `following-sibling`, `namespace`, `parent`, `preceding`, `preceding-sibling`, `self`;
- a node test, which is used for filtering the nodes selected by the axis;
- a sequence of zero or more predicates, which is used to impose further constraints on the selected nodes;

and has the following syntax: `axis::node-test[predicate]`.

The XPath syntax also defines shorthand notations for the axes. Furthermore, a library of built-in functions is available and it should be supported by each implementation of XPath, which, among others, includes functions for string, numeric, and Boolean value manipulation, functions on nodes and sequences of nodes, etc.

XQuery is another query language for XML. It has been accepted as a W3C Recommendation in 2007, and since then it has been used as a standard query language for XML documents. It is a functional language, and it is not wrong to say that it is to XML what SQL is to relational databases. It subsumes the XPath language, in the sense that every valid XPath expression is a valid XQuery query. Additionally, it supports FLWOR (`FOR`, `LET`, `WHERE`, `ORDER BY`, `RETURN`) expressions, which are used to extract, navigate and restructure data from a single or multiple XML documents. As a result of an XQuery query, one can obtain a set of nodes, an arbitrary subtree of the XML document being queried, as well as a whole XML document, created in the `RETURN` clause. In order to navigate through the XML documents, XQuery uses path expressions, which are indeed valid XPath expressions.

An XPath Fragment and Tree Patterns

A well studied fragment of XPath is $\mathsf{XP}^{\{/,//,[],*\}}$, which contains expressions that allow the child and descendant relations, branches using predicates and wildcards. Many works on optimization and rewriting XPath queries consider this fragment [7, 8, 31, 38, 54]. It is also common to investigate the properties of its subclasses:

- $\mathsf{XP}^{\{/,//,[]\}}$, containing expressions that support children, descendants and branches;
- $\mathsf{XP}^{\{/,//,*\}}$, containing expressions that support children, descendants and wildcards;
- $\mathsf{XP}^{\{/,[],*\}}$, containing expressions that support children, branches and wildcards;

Definition 3.2.1 (Syntax of $\mathsf{XP}^{\{/,//,[],*\}}$)**.** Given the finite sets $El \subset \mathcal{E}$ and $Att \subset \mathcal{A}$, an *XPath expression q in $\mathsf{XP}^{\{/,//,[],*\}}$* over $El \cup Att \cup \{*\}$ is an expression that can be built using the following grammar:

$$q \quad ::= \quad l \quad | \quad * \quad | \quad q/q \quad | \quad q//q \quad | \quad q[q]$$

where l is a label of a node, $*$ is the wildcard symbol, $/$ and $//$ denote child and descendant navigation respectively, and [] denotes a predicate. □

The semantics of an $\mathsf{XP}^{\{/,//,[],*\}}$ expression posed over an XML tree T is defined as a set of nodes from T which are selected by it. XPath expressions, and therefore $\mathsf{XP}^{\{/,//,[],*\}}$ expressions are evaluated w.r.t. a context node. When evaluating an $\mathsf{XP}^{\{/,//,[],*\}}$ expression over an XML tree T, we take the document node of the XML document as a context node, i.e. we start the evaluation of the expression from the document node [53].

Definition 3.2.2 (Semantics of $\mathsf{XP}^{\{/,//,[],*\}}$)**.** The result of applying an XPath expression q in $\mathsf{XP}^{\{/,//,[],*\}}$ to an XML tree $T = (N, E, \rightarrow, \downarrow, label, value_a, id, root)$ is a set of nodes in T,

denoted by $q(T)$. The elements of $q(T)$ are the nodes from T that have been obtained by applying q to the document node of T.

For a node $n \in N$ of the tree T, the result of applying an expression q is defined inductively as follows:

$$\begin{aligned} l(n) &= \{m \mid m \in N \wedge (n,m) \in E \wedge label(m) = l\} \\ *(n) &= \{m \mid m \in N \wedge (n,m) \in E\} \\ (q_1/q_2)(n) &= \{m \mid m' \in q_1(x) \wedge m \in q_2(m')\} \\ (q_1//q_2)(n) &= \{m \mid n' \in q_1(n) \wedge (n',m') \in E^* \wedge m \in q_2(m')\} \\ (q_1[q_2])(n) &= \{m \mid m \in q_1(n) \wedge q_2(m) \neq \emptyset\} \end{aligned}$$

where E^* is the reflexive and transitive closure of the edge relation. □

Example 3.2.1. Recall the XML tree T from Figure 3.1b. The following $\mathsf{XP}^{\{/,//,[],*\}}$ expression

$$a/b[//@d]/e$$

searches for all nodes labeled with e, that are placed under a node labeled with b, which in turn are placed under the root a, and moreover the node labeled with b has a descendant which has an attribute labeled with $@d$. △

The expressions in the XPath fragment $\mathsf{XP}^{\{/,//,[],*\}}$ can be represented by a formalism called tree patterns. Consider again the finite sets $El \subset \mathcal{E}$ of elements and $Att \subset \mathcal{A}$ of attributes. Arbitrary tree patterns can be defined as follows.

Definition 3.2.3 (Tree pattern)**.** A *tree pattern* p over $El \cup Att \cup \{*\}$ of arity $k, k \geq 0$, is a tree $(N_p, E_p, label_p, root_p, \overline{o}_p)$ where:

- N_p is the set of nodes;
- E_p is the edge relation. Note that tree patterns might have two types of edges - child and descendant edges. Thus, $E_p = E_/ \cup E_{//}$ where $E_/$ and $E_{//}$ are the sets of child and descendant edges respectively;
- $label_p : N_p \to El \cup Att \cup \{*\}$ is a labeling function that assigns a label to each node of the pattern;
- $root_p$ is the root node of the tree pattern;
- $\overline{o}_p$ is a k-tuple of output nodes.

The set of all tree patterns is denoted by $\mathsf{P}^{\{/,//,[],*\}}$. We also consider the three subclasses $\mathsf{P}^{\{/,//,[]\}}$, $\mathsf{P}^{\{/,//,*\}}$ and $\mathsf{P}^{\{/,[],*\}}$, that correspond to patterns without wildcards, predicates and descendants respectively. □

Given the definition of a tree pattern, we can see that an expression from $\mathsf{XP}^{\{/,//,[],*\}}$ can be represented by a tree pattern of arity 1 (that is, a tree pattern with one output node). Tree

patterns of arity 0 are called Boolean tree patterns. In [38], it is shown how the translation from tree patterns to XPath expressions can be done, while maintaining the semantics.

In order to define the semantics of applying a tree pattern p to an XML tree T, we need to define the notion of embedding.

Definition 3.2.4 (Embedding)**.** Let $p = (N_p, E_p, label_p, root_p, \overline{o}_p)$ be a tree pattern, $p \in \mathsf{P}^{\{/,//,[],*\}}$ and $T = (N, E, \downarrow, \rightarrow, label, value_{@a}, id, root)$ be an XML tree, $T \in \mathcal{T}$. An *embedding* is a function $e : N_p \rightarrow N$ that satisfies the following conditions:

- e is root preserving, that is it holds that $e(root_p) = root$;
- e is label preserving, i.e. for all $n \in N_p$ it is the case that either $label_p(n) = *$ or $label_p(n) = label(e(n))$;
- e is child preserving, meaning that for all edges $(n_1, n_2) \in E_/$ it holds that $e(n_2)$ is a child of the node $e(n_1)$.
- e is descendant preserving, meaning that for all edges $(n_1, n_2) \in E_{//}$ it holds that $e(n_2)$ is a proper descendant of the node $e(n_1)$. □

Note that the embedding function is only concerned with mapping the nodes from the tree pattern into the nodes of the tree, such that the structure of the pattern is compatible with the structure of the tree. The values stored in the tree do not have any influence on the way the nodes from the pattern are mapped to the nodes of the tree.

The result of applying a tree pattern p to a tree T depends on the output nodes $\overline{o}_p$ of the tree pattern. It is defined as the subset of N^k, where k is the arity of $\overline{o}_p = (o_1, \ldots, o_k)$, as follows:

$$p(T) = \{(e(o_1), \ldots, e(o_k)) \mid e \text{ is an embedding from } p \text{ to } T\}$$

When the pattern p is Boolean, the result $p(T)$ is either $\{()\}$, corresponding to *true*, or $\emptyset$, corresponding to *false*.

Example 3.2.2. Consider the XML tree in Figure 3.1b. A tree pattern posed over this tree is given in Figure 3.2.

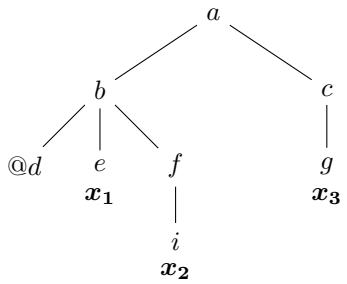


Figure 3.2: *Tree pattern p*

We can see that the left branch of the tree pattern p rooted at b can be mapped to two different nodes in the XML tree T. Thus, we have two different embeddings from the tree pattern to the XML tree, and hence two output tuples in the result $p(T)$. The two embeddings are shown in Figure 3.3 and Figure 3.4.

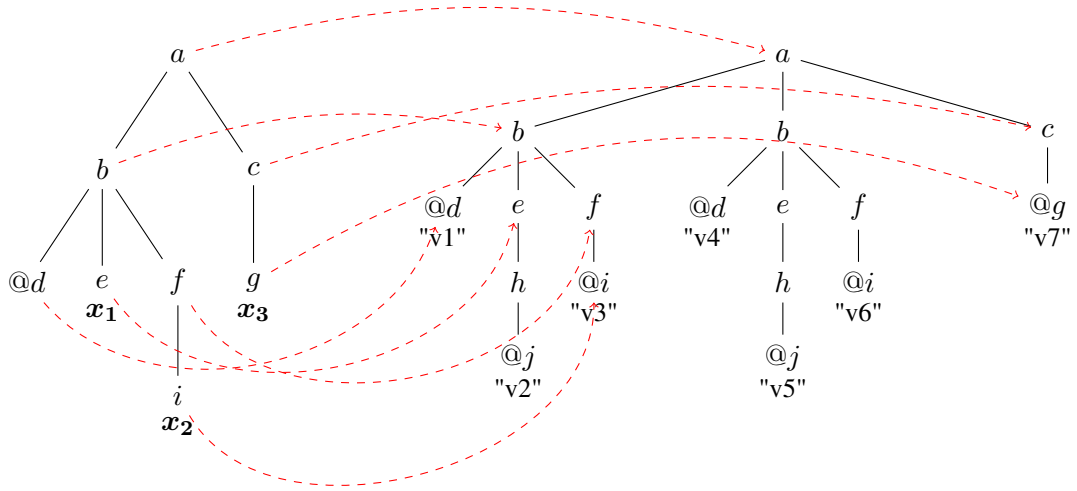


Figure 3.3: *Embedding* e_1

Given the two embeddings, we get the following result of applying p to T:

$$p(T) = \{(e_1(x_1), e_1(x_2), e_1(x_3)), (e_2(x_1), e_2(x_2), e_2(x_3))\}$$

△

It is also useful to define containment and equivalence between tree patterns, as most of the works on XPath query rewriting are focused on computing equivalent rewritings.

Definition 3.2.5 (Containment and equivalence)**.** A tree pattern p_1 is *contained* in the tree pattern p_2, denoted by $p_1 \subseteq p_2$ if for all trees $T \in \mathcal{T}$ it holds that $p_1(T) \subseteq p_2(T)$.

Two tree patterns p_1 and p_2 are *equivalent* if $p_1 \subseteq p_2$ and $p_2 \subseteq p_1$, i.e. for all trees $T \in \mathcal{T}$ it holds that $p_1(T) = p_2(T)$. □

[38] studies the complexity of tree pattern containment and equivalence. In their study, the authors consider Boolean tree patterns, since they show that k-ary tree patterns can be translated into Boolean ones, such that for any k-ary tree patterns p_1, p_2 and their Boolean translations p_1', p_2' it holds that $p_1 \subseteq p_2$ iff $p_1' \subseteq p_2'$. Thus, the containment in the Boolean case boils down to logical implication, i.e. $p_1' \subseteq p_2'$ if and only if $\forall T\ (p_1'(T) \rightarrow p_2'(T))$. It has been shown that the containment problem for $\mathsf{P}^{\{/,//,[],*\}}$ (and hence for $\mathsf{XP}^{\{/,//,[],*\}}$) is coNP-complete [38],

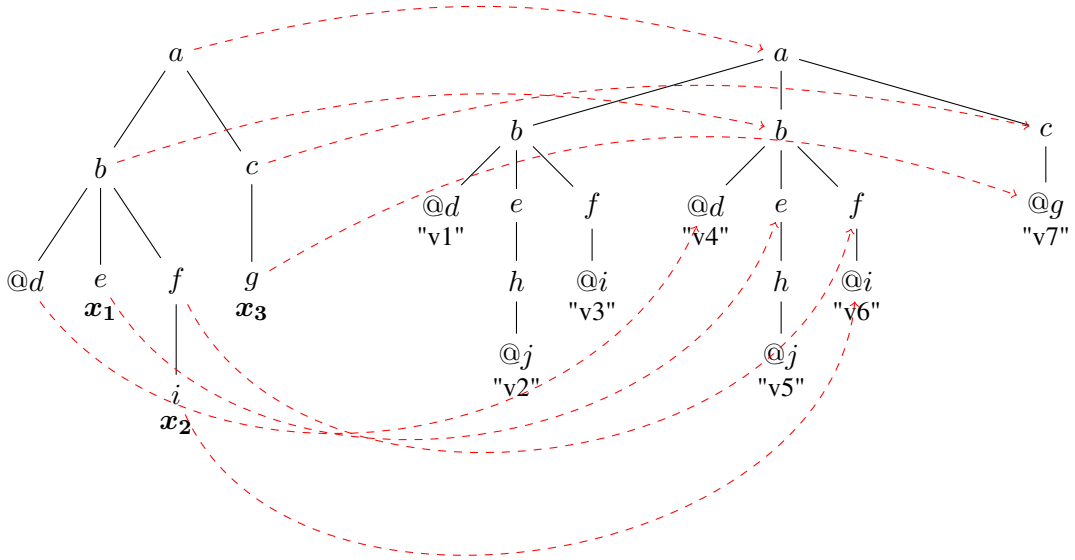


Figure 3.4: *Embedding* e_2

while it is in PTIME for the three subclasses [10, 39, 55]. The containment checking for the subclasses $\mathsf{XP}^{\{/,//,[]\}}$ and $\mathsf{XP}^{\{/,[],*\}}$ is based on finding a homomorphism between two patterns. A homomorphism between two tree patterns p_1, p_2 is a mapping from the nodes of p_1 to the nodes of p_2 that preserves the root, the labels, the child and descendant relation and the output nodes. In the remaining subclass containment checks are done using a relaxed form of homomorphism, called adorned homomorphism [38].

Tree Pattern Formulae and $\mathcal{CTQ}$s

The queries discussed so far return trees, either a subtree of the queried XML tree or a newly defined tree structure. For the purpose of XML data exchange, [12, 9, 18] propose another query language for XML documents, called *conjunctive tree queries,* $\mathcal{CTQ}$*s*. This query language navigates through the trees and returns tuples of values extracted from the attributes using variables. In XML data exchange, the tuples are extracted from a source document and used to populate a target document, which conforms to a target DTD. We discuss the works on XML data exchange in more detail in Section 5.2. The expressions in $\mathcal{CTQ}$ are based on tree pattern formulae [9].

Definition 3.2.6 (Syntax of tree pattern formulae)**.** Given the finite sets $El \subset \mathcal{E}$ of elements and $Att \subset \mathcal{A}$ of attributes, a *tree pattern formula* φ is an expression whose syntax is given by the

following grammar:

$$
\begin{array}{lcll}
\varphi & ::= & \pi, \alpha & \text{pattern formulae} \\
\pi & ::= & l(\bar{t})[\lambda] & \text{pure pattern formulae} \\
\lambda & ::= & \varepsilon \mid \mu \mid //\pi \mid \lambda, \lambda & \text{sets} \\
\mu & ::= & \pi \mid \pi \to \mu \mid \pi \to^{+} \mu & \text{sequences}
\end{array}
$$

where $l \in El \cup \{*\}$, $\bar{t}$ is a tuple of terms that correspond to the attributes of the element labeled by l, and α is a conjunction of equalities and inequalities over terms.

The terms are defined inductively over a set of variables Var and a set of Skolem function symbols Fun:

1. every variable from Var is a term;

2. if f is an m-ary function symbol from Fun and $t_1, \ldots, t_m$ are terms, then $f(t_1, \ldots, t_m)$ is also a term. □

The semantics of the tree pattern formulae is defined w.r.t. an XML tree T, a node $n \in N$ of the tree T and an interpretation F for the function symbols.

Definition 3.2.7 (Semantics of tree pattern formulae)**.** Given a pattern $\varphi(\overline{x})$, we denote that $\varphi(\overline{x})$ is *satisfied* in a node n of the tree T, such that its variables $\overline{x}$ are assigned values from the tuple $\overline{a}$ and the function symbols are interpreted w.r.t. F as $(T, n, F) \models \varphi(\overline{a})$. The meaning of the satisfaction relation is defined as follows:

$$
\begin{array}{rl}
(T, n, F) \models l(\bar{t}) & \text{if } label(n) = l \text{ or } l = *, \text{ and } \bar{t} \text{ is interpreted under } F \text{ as the} \\
& \text{tuple of attributes of } n \\
(T, n, F) \models l(\bar{t})[\lambda_1, \lambda_2] & \text{if } (T, n, F) \models l(\bar{t})[\lambda_1] \text{ and } (T, n, F) \models l(\bar{t})[\lambda_2]; \\
(T, n, F) \models l(\bar{t})[\mu] & \text{if } (T, n, F) \models l(\bar{t}) \text{ and there exists a node } n' \text{ such that } n \downarrow n' \\
& \text{and } (T, n', F) \models \mu; \\
(T, n, F) \models l(\bar{t})[//\pi] & \text{if } (T, n, F) \models l(\bar{t}) \text{ and there exists a node } n' \text{ such that } n \downarrow^{+} n' \\
& \text{and } (T, n', F) \models \pi; \\
(T, n, F) \models \pi \to \mu & \text{if } (T, n, F) \models \pi \text{ and there exists a node } n' \text{ such that } n \to n' \\
& \text{and } (T, n', F) \models \mu; \\
(T, n, F) \models \pi \to^{+} \mu & \text{if } (T, n, F) \models \pi \text{ and there exists a node } n' \text{ such that } n \to^{+} n' \\
& \text{and } (T, n', F) \models \mu; \\
(T, n, F) \models \pi, \alpha & \text{if } (T, n, F) \models \pi \text{ and } \alpha \text{ holds under } F. \qquad \square
\end{array}
$$

When the evaluation of the tree pattern formula starts from the root of the tree, we write $(T, F) \models \varphi(\overline{a})$, and moreover, if there are no function symbols, we write $T \models \varphi(\overline{a})$.

Note that in the tree pattern formulae, the square brackets denote vertical navigation. Unlike in XPath, where the square brackets denote predicates which are expressions for filtering the results of the navigation, in the tree pattern formulae they are used for accessing multiple children of a node at once. For filtering the results, the α part of the tree pattern formula is used.

Example 3.2.3. Consider again the XML tree T from Figure 3.1b and the tree pattern formula

$$\varphi(x_d, x_j, x_i, x_g) = a[b(x_d)[e[h(x_j)], f(x_i)], c(x_g)]$$

The following two tuples satisfy this tree pattern formula, if the evaluation starts from the root and there is no presence of function symbols:

("v1", "v2", "v3", "v7"), i.e. $T \models \varphi$("v1", "v2", "v3", "v7")
("v4", "v5", "v6", "v7"), i.e. $T \models \varphi$("v4", "v5", "v6", "v7")

△

On top of these tree pattern formulae, $\mathcal{CTQ}$ expressions additionally allow existential quantification. Conjunction is supported implicitly, as the tree pattern formulae are closed under conjunction. In fact, the conjunction of two tree pattern formulae is defined as follows. Consider two tree pattern formulae $l_1(\overline{x})[\lambda_1], \alpha_1$ and $l_2(\overline{y})[\lambda_2], \alpha_2$. The conjunction $(l_1(\overline{x})[\lambda_1], \alpha_1 \wedge l_2(\overline{y})[\lambda_2], \alpha_2)$ is defined as the following tree pattern formula:

$$l_1(\overline{y})[\lambda_1, \lambda_2], \alpha_1 \wedge \alpha_2 \wedge x_1 = y_1 \wedge \cdots \wedge x_n = y_n$$

only if $l_1 = l_2$ or either l_1 or l_2 is the wildcard symbol, and $\overline{x}$ and $\overline{y}$ have the same arity. If these conditions are not met, the result of the conjunction is false, represented by a special element type symbol $\perp$, $\perp \notin \mathcal{E}$.

Definition 3.2.8 (Syntax of $\mathcal{CTQ}$s)**.** A *query* in $\mathcal{CTQ}$ is an expression that has the following syntax:

$$\exists \overline{y}\ \varphi(\overline{x}, \overline{y})$$

where $\varphi(\overline{x}, \overline{y})$ is a tree pattern formula such that it does not contain any function symbols and its free variables $\overline{x}$ fulfill the safety condition. The safety condition states that for a variable $x_i \in \overline{x}$, either x_i is used in the π part of φ, or there exists a chain $x_i = t_1, t_1 = t_2, \ldots, t_{k-1} = t_k$ of equality atoms in α such that $t_1, \ldots, t_{k-1}$ are terms and t_k is a variable used in π.

Unions of $\mathcal{CTQ}$s are referred to as $\mathcal{UCTQ}$s. □

Definition 3.2.9 (Semantics of $\mathcal{CTQ}$s)**.** Given a $\mathcal{CTQ}$ $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$, we say that the $\mathcal{CTQ}$ is *satisfied* in a tree T given a tuple of atomic values $\overline{a}$ iff its free variables $\overline{x}$ are interpreted as $\overline{a}$ and there exists a tuple of atomic values $\overline{b}$ that corresponds to an interpretation of the existentially quantified variables such that $T \models \varphi(\overline{a}, \overline{b})$. □

The result of applying a $\mathcal{CTQ}$ $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$ to a tree T is a set of tuples of atomic values that correspond to the values given to the free variables, such that the $\mathcal{CTQ}$ is satisfied in T, and is denoted by:

$$\varphi(T) = \{\overline{a} \mid \exists\, \overline{b} \text{ of the same arity as } \overline{y} \text{ such that } T \models \varphi(\overline{a}, \overline{b})\}$$

Example 3.2.4. Consider again the XML tree T from Figure 3.1b. If we pose the following $\mathcal{CTQ}$ to it

$$\exists y_d \, \varphi(x_i, x_g, y_d) = \exists y_d \; a[b(y_d)[f(x_i)], c(x_g)]$$

we will get the following answers in the result $\varphi(T)$:

$$\varphi(T) = \{(\text{"v3"}, \text{"v7"}), (\text{"v6"}, \text{"v7"})\}$$

since we have that there exist two values "v1" and "v7" for the variable y_d corresponding to the two valuations of the free variables such that

$$T \models \varphi(\text{"v3"}, \text{"v7"}, \text{"v1"}) \text{ and}$$
$$T \models \varphi(\text{"v6"}, \text{"v7"}, \text{"v4"})$$

△

Different classes of tree pattern formulae can be defined using the symbols $\Downarrow, \Rightarrow, \mathsf{Fun}, \sim$, where $\Downarrow$ stands for vertical navigation and wildcard ($\downarrow, \downarrow^+, *$), $\Rightarrow$ for horizontal navigation ($\rightarrow, \rightarrow^+$), Fun for Skolem function symbols and $\sim$ for equalities and inequalities ($=, \neq$). The different classes of tree pattern formulae are used for defining different classes of schema mapping assertions. When defining different classes of $(\mathcal{U})\mathcal{CTQ}$s, the symbols $\Downarrow, \Rightarrow, =$ are used. The use of inequalities is forbidden in the queries, since it makes the computation of certain answers undecidable.

Prefix-selection Queries

Another simple query language for XML trees introduced in [6], and used in [41] is the language of *prefix-selection queries*, or *ps-queries* for short. This query language is simple in the sense that it allows selecting of prefixes of input trees based on selection conditions given for every node. Specification of existential patterns in the trees is also allowed. A prefix of an XML tree T is an XML tree T' such that there exists a homomorphism h from the nodes of T' to the nodes of T that is root, child relation and label preserving, and for every node n' from T' that stores a value, its homomorphic image $h(n')$ stores the same value. We write that $T' \leq T$, if T' is a prefix of T. Two XML trees T and T' are isomorphic if both $T' \leq T$ and $T \leq T'$ hold, and we write $T \simeq T'$. The formal definition of ps-queries is given below.

Definition 3.2.10 (Ps-query)**.** A *ps-query* over the sets $El \subset \mathcal{E}$ and $Att \subset \mathcal{A}$ is a quadruple $\langle t, \lambda, cond, sel \rangle$, where

- t is a rooted tree;
- λ is a labeling function, assigning to each node from the tree a label from $El \cup Att$ such that sibling nodes have distinct labels;
- $cond$ is a partial function that assigns to each node n from t a condition c, which is a Boolean formula that has the following form: $p_0 b_0 p_1 b_1 \ldots p_{m-1} b_m p_m$, where $b_i, 0 \leq i \leq m$ are logical connectives and $p_i, 0 \leq i \leq m$ are predicates that are applied to nodes that store values. The predicates p_i are of the form $op \; v$, where $op \in \{=, \neq, \leq, \geq, <, >\}$, and v is a value;

- sel is a total function that assigns a Boolean value to each node n from t. If $sel(n) = \mathbf{true}$, then the node n is selected by the query. On the other hand, if $sel(n) = \mathbf{false}$, the node is not selected by the query, and moreover, none of its descendants can be selected by the query. □

The answer of a ps-query $q = \langle t, \lambda, cond, del \rangle$ over an XML tree T is a minimal tree that is isomorphic to the *positive subset* of T, which is a subset of nodes from T selected by q. In order to specify the positive subset, an auxiliary function called valuation is used. A *valuation* γ from the query q to the tree T is a homomorphism from the nodes of q to the nodes of T such that the root, the child relation and the labels are preserved. Moreover, for each node n_q from the query, if $cond(n_q)$ is defined, then $\gamma(n_q)$ stores a value, and this value satisfies the condition specified for the node n_q. A node n is in the positive subset of the XML tree T if there exists a valuation γ such that $sel(\gamma^{-1}(n)) = \mathbf{true}$. Example 3.2.5 shows the graphical representation of a ps-query and its answer over an XML tree.

Example 3.2.5. We take this example from [41]. Consider the XML tree T in Figure 3.5. Figure 3.6a shows a ps-query q over this tree, and Figure 3.6b depicts the answer $q(T)$ of the query q over the tree T.

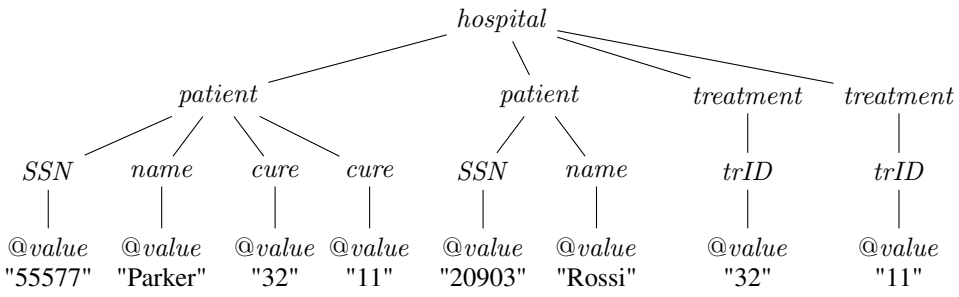
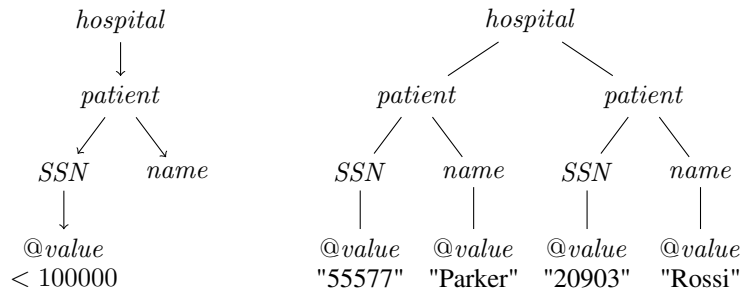


Figure 3.5: *XML tree T*

Note that in the graphical notation of the ps-queries, we use arrows to represent the edges between the nodes in the ps-query. We do this in order to distinguish the ps-queries from the (extended) tree patterns. The attribute node $@value$ of SSN has a specified condition on its value, hence the query q selects only patients whose SSN is smaller than 100000. △

3.3 XML Constraints

Constraints in XML documents can be specified as both type constraints and integrity constraints. The former are implied by the DTD that is defined along the XML document. These type constraints are used for ensuring that the values stored in the respective nodes are from the right data type. On the other hand, the integrity constraints, as in relational databases, are used to identify certain elements in the document. Another type of constraints imposed by the



(a) *ps-query* q **(b)** *answer to the ps-query* $q(T)$

Figure 3.6: *A ps-query and its answer over the tree* T *from Figure 3.5*

DTD attached to the XML document is the specification of the subelement order of a given element. These constraints define in which order the sibling elements at some level in the XML tree should appear. They are implied by the regular expressions found on the right-hand side of the production rules in a DTD.

There have been many attempts to formalize and define a framework for describing integrity constraints for XML in order to provide the same services as the integrity constraints for relational databases, but in an XML setting. The most common kind of integrity constraints used in relational databases are primary key and foreign key constraints, which are a special case of functional and inclusion dependencies respectively. Some of the works that focus on XML constraints are [15, 25, 26, 27]. A natural way to define primary keys would be to use the built-in specification of ID attributes in a DTD or an XML Schema. However, this has its drawbacks, since the ID attributes need to have a unique value throughout the whole document, rather than only in some specific part of the document. Hence, new formalisms are needed for capturing the intended meaning of the XML constraints. In [25], different kinds of key constraints have been examined, and problems such as consistency and implication have been analyzed. The constraints that are taken into account are the keys and foreign keys as introduced in [15], constraints in XML Schema [50], functional dependencies defined in [11] and XML integrity constraints (XICs), introduced in [21].

The work in [15] proposes a formalism for expressing key constraints in XML, by introducing a language for specifying key constraints that is based on path expressions, which can be expressed by using regular expressions or XPath. A key specification is defined as a path expression Q followed by a set of path expressions $\{P_1, \ldots, P_n\}$. A node n in an XML tree T satisfies the key iff for any two nodes n_1, n_2 reachable from n through Q, if the values of the nodes reached from both n_1 and n_2 through $\{P_1, \ldots, P_n\}$ are equal, then n_1 and n_2 are the same node. Sometimes, it is useful to define a key constraint w.r.t. a given node, rather than the root of the tree. In this case, relative keys are used. They have an additional path expression in the key specification that specifies the context nodes where the key constraint should be evaluated. A foreign key constraint is defined as $P_1[L_1] \subseteq P_2[L_2]$ where P_1, P_2 are path expressions and L_1, L_2 are lists of path expressions. A node n in an XML tree T satisfies a foreign key iff for all

nodes n_1 reached from n through P_1 there exists a node n_2 reached from n through P_2 such that the list of values reached from n_1 through L_1 are equal to the values reached from n_2 through L_2. In a similar way as for the keys, relative foreign key constraints can be defined.

In [11], functional dependencies for XML have been proposed. They are of the form $S_1 \rightarrow S_2$, where S_1 and S_2 are sets of path expressions. The semantics of these constraints is defined in terms of a mapping t that maps path expressions into tuples of values from the tree. Hence, an XML tree T satisfies a functional dependency $S_1 \rightarrow S_2$ iff for any two mappings t_1, t_2, if $\forall p_1 \in S_1 \ t_1(p_1) = t_2(p_1)$ and $t_1(p_1)$ is well defined, then $\forall p_2 \in S_2 \ t_1(p_2) = t_2(p_2)$. In contrast to [15], relative functional dependencies of this form cannot be defined. However, these functional dependencies are a generalization of the key constraints in [15].

In [21], the authors formalize relational embedded constraints for XML and call them XML integrity constraints, XICs. For specifying the XICs, the authors use a fragment of XPath that allows the use of variables, called Simple XPath. The XICs are of the form $\forall x_1, \ldots, x_n (B(x_1, \ldots, x_n) \rightarrow \bigvee_{i=1}^{l} \exists y_{i,1}, \ldots, y_{i,k_i} \ C_i(x_1, \ldots, x_n, y_{i,1}, \ldots, y_{i,k_i}))$, where B, C_i are conjunctions of atoms that can be of the form: (1) $v \ p \ w$, where p is a Simple XPath expression and v, w are variables or constants; (2) equality atoms on variables or constants, of the form $v = w$. An XIC is satisfied if for any binding of the variables $x_1, \ldots, x_n$ that satisfies all the atoms in B, there exists $i, 1 \leq i \leq l$ and an extension of the binding to the variables $y_{i,1}, \ldots, y_{i,k_i}$ that satisfies all the atoms in C_i. The binding of v to a node a and of w to a node b satisfies the atom $v \ p \ w$ if b is in the set of nodes returned by p starting from a as a context node. The equality atoms are satisfied in a standard way. This type of constraints is more general than the ones we have seen before, i.e. it subsumes both the key and foreign key constraints from [15], as well as the functional dependencies from [11].

Integrity constraints in XML have many applications, as pointed out in [25]. As we will see in the following sections, some approaches for XML data integration, XML data exchange and answering XML queries using XML views take integrity constraints into account. Other areas where such constraints are typically used include querying XML using an RDBMS, updating XML documents and removing inconsistencies.

CHAPTER 4

XML Mapping Assertions

In Chapter 2, we have seen that relational schema mappings play a crucial role in defining several relational data management problems. This also applies when the respective problems are tackled in an XML setting. Thus, in this section, our aim is to define schema mapping assertions for XML, which will provide a uniform framework for describing different approaches in solving XML data management tasks that we have encountered in the literature. We define the XML mapping assertions, based on a newly defined query language, called *extended tree patterns*. We take inspiration from the different XML query language formalisms discussed so far, and define a common query language which will be used throughout this thesis and that subsumes several query languages w.r.t. the expressive power. We also provide algorithms for translating expressions in these query languages into extended tree patterns. These translations will then allow us to translate mapping assertions from different works into mapping assertions that use extended tree patterns.

4.1 Extended Tree Patterns

The language of extended tree patterns is an extension of the k-ary tree patterns described before. Rather than only allowing a query to return a subtree of the queried tree or a tuple of values stored in its attribute nodes, we aim at bringing the two types of results together. Hence, when using extended tree patterns, we return a tuple of either subtrees, or values, or both. We achieve this by annotating each variable, such that in the evaluation of the extended tree pattern, it is bound to the desired content depending on its annotation.

Definition 4.1.1 (Extended tree pattern)**.** An *extended tree pattern* p over $El \cup Att \cup \{*\}$ of arity $k, k \geq 0$ is a tree $(N_p, E_p, label_p, root_p, \overline{x}_p, \overline{y}_p, var_p)$ where

- $N_p, E_p, label_p$ and $root_p$ are defined as for tree patterns;
- $\overline{x}_p$ is the tuple of output variables of arity k;

- $\overline{y}_p$ is the tuple of existentially quantified variables;
- $var_p : N_p \rightarrow \overline{x}_p \cup \overline{y}_p$ is a partial surjective function that attaches variables from $\overline{x}_p \cup \overline{y}_p$ to nodes from N_p.

Additionally, each variable is annotated by an annotation $a \in \{val, sub\}$, denoted by z^a, such that the following conditions are met:

attachment: if $a = sub$, then z^a is attached to exactly one node $n \in N_p$, i.e. $var_p(n) = z^a$. Otherwise, if $a = val$, then z^a can be attached to multiple nodes, i.e. to a subset of nodes $\{n_1, \ldots, n_m\} \subseteq N_p$, such that $\forall n_i \; var_p(n_i) = z^a, \; 1 \leq i \leq m$;

binding: if $a = sub$, then z^a is bound to the subtree rooted at the selected element node. Otherwise, if $a = val$, then z^a is bound to the string value stored at the selected attribute node;

existential quantification: if $z^a \in \overline{y}_p$, then it can only be annotated by val;

We denote the set of extended tree patterns by $\mathsf{EP}^{\{/,//,[],*\}}$.
The extended tree patterns use the following syntax:

p	::=	$l[b]$		extended tree pattern
l	::=	$el \mid *$	$\mid z^{sub} : el$	element node labels
b	::=	$\varepsilon \mid l \mid z^{val} : @att \mid b, b \mid p \mid //p$		branches

where $el \in El$ is an element node label, $z^{val} : @att$ denotes a variable annotated by val attached to an attribute node labeled by $@att$, $z^{sub} : el$ is a variable annotated by sub attached to an element node labeled by el, $*$ is the wildcard label, $//p$ denotes descendant navigation and $l[b]$ denotes branching. □

Note that when the branch does not contain an occurrence of a variable, it is considered as a predicate. Otherwise, it is used to navigate through the tree and extract the variable value of interest. Furthermore, by allowing a single variable annotated by val to be attached to multiple different nodes in the tree, we implicitly allow specification of equalities between values. In the syntax of the extended tree patterns, we omit the use of existential quantifiers, since we always assume that the tuple of variables $\overline{x}_p$ corresponds to the free variables and the tuple $\overline{y}_p$ to the existentially quantified variables. We may sometimes slightly abuse the notation, and refer to an extended tree pattern as $\exists \overline{y}_p \, p(\overline{x}_p, \overline{y}_p)$. We will do this in cases where it is important to spell out the names of the variables that are free and those that are existentially quantified.

We also extend the notion of embedding, in order to be able to define the semantics of the extended tree patterns. Given an extended tree pattern $p = (N_p, E_p, label_p, root_p, \overline{x}_p, \overline{y}_p, var_p)$, and an XML tree $T = (N, E, \downarrow, \rightarrow, label, value_{@a}, id, root)$, an extended embedding is a function $e : N_p \rightarrow N$ that satisfies the root, label, child and descendant preservation conditions from Definition 3.2.4. Additionally, e is value equality preserving, in such a way that it must not allow nodes from N_p that have the same variable z^{val} attached to be mapped to attribute nodes from N that store different values. More formally, for a variable z^{val}, let $\{n_1, \ldots, n_m\} \subseteq N_p$ denote the

set of nodes such that $var_p(n_i) = z^{val}, 1 \leq i \leq m$. Then for each n_i, with $label_p(n_i) = @a_i$, there exists a node n'_i in the tree T such that $(n'_i, e(n_i)) \in E$. Moreover, it holds that

$$value_{@a_1}(n'_1) = \cdots = value_{@a_m}(n'_m).$$

Using the extended embedding, we define the result of applying an extended tree pattern p to an XML tree T. As for tree patterns from $\mathsf{P}^{\{/,//,[],*\}}$, the result is a set of tuples of arity k, which coincides with the arity of the output tuple $\overline{x}_p$ of p.

Definition 4.1.2 (Semantics of extended tree patterns)**.** The result of applying an extended tree pattern p to an XML tree T is defined as follows:

$$p(T) \quad = \quad \{(b(e(n_1), a_1), \ldots, b(e(n_k), a_k)) \quad | \quad \begin{array}{l} e \text{ is an extended embedding from } p \text{ to } T, \\ b \text{ is a binding function and} \\ n_i \in N_p, 1 \leq i \leq k, \text{ s.t. } var_p(n_i) = x_i^{a_i}\} \end{array}$$

For an output variable $x_i^{a_i}$ from the output tuple of the extended tree pattern p, the binding function takes as parameters the result of the embedding $e(n_i)$ and the annotation a_i, where $var_p(n_i) = x_i^{a_i}$, $label_p(n_i) = l_i$. As a result, it returns the following:

$$b(e(n_i), a_i) = \begin{cases} e(n_i) & \text{if } a_i = sub \text{ and } l_i \in El; \\ value_{l_i}(n'_i) & \text{if } a_i = val, l_i \in Att \text{ and } (n'_i, e(n_i)) \in E. \end{cases}$$

□

Example 4.1.1. Consider the XML tree T from Figure 3.1b. If we pose the extended tree pattern

$$p = a\,[\,b\,[x_1^{val} : @d,\ x_2^{sub} : e\,],\ x_3^{sub} : c\,]$$

over it, we obtain two tuples in the result $p(T)$, w.r.t. the two extended embeddings e_1 and e_2 depicted in Figure 4.1 with red and blue arrows respectively. The variables x_1, x_2 and x_3 are displayed in bold under the node they are attached to. The binding function binds x_1^{val} to the value of the attribute $@d$ of the element node labeled with b, x_2^{sub} and x_3^{sub} to the subtrees rooted at the element nodes labeled with e and c respectively. Hence, the result $p(T)$ is defined as follows:

$$p(T) = \{(\text{"v1"},\ \begin{array}{c} e \\ | \\ h \\ | \\ @j \\ \text{"v2"} \end{array},\ \begin{array}{c} c \\ | \\ @g \\ \text{"v7"} \end{array}),\quad (\text{"v4"},\ \begin{array}{c} e \\ | \\ h \\ | \\ @j \\ \text{"v5"} \end{array},\ \begin{array}{c} c \\ | \\ @g \\ \text{"v7"} \end{array})\}$$

△

Example 4.1.2. This example describes how the extended embedding preserves values. Consider the simple XML tree in Figure 4.2 (on the right) and the extended tree pattern

$$p = a\,[\,b\,[\,x^{val} : @d\,],\ c\,[\,x^{val} : @g\,]]$$

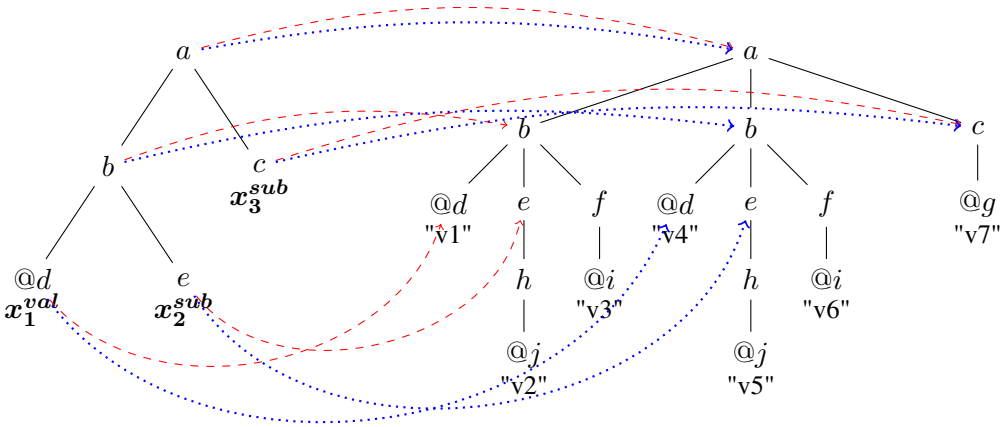


Figure 4.1: *Extended embeddings e_1 (red dashed arrows) and e_2 (blue dotted arrows)*

also shown in Figure 4.2 (on the left). Two embeddings e_1, e_2 are depicted with red dashed and blue dotted arrows respectively. Yet, only one of them, namely e_1, is an extended embedding.

Let n_1, n_2 denote the nodes of the extended tree pattern that have the variable x^{val} attached to them, i..e the attribute nodes labeled with $@d$ and $@f$ respectively. Thus we have the following:

$$\begin{array}{llll} label_p(n_1) = @d, & var_p(n_1) = x^{val}, & e_1(n_1) = n_1', & e_2(n_1) = m_1' \\ label_p(n_2) = @f, & var_p(n_2) = x^{val}, & e_1(n_2) = e_2(n_1) = n_2' & \end{array}$$

We can see that e_1 is an extended embedding, while e_2 is not, since

$$\begin{array}{rl} value_{@d}(n_1'') = value_{@f}(n_2'') = \text{"v1"}, & \text{where } (n_1'', n_1') \in E, (n_2'', n_2') \in E, \text{ but} \\ value_{@d}(m_1'') = \text{"v2"} \neq \text{"v1"} = value_{@f}(n_2''), & \text{where } (m_1'', m_1') \in E, (n_2'', n_2') \in E \end{array}$$

△

4.2 Expressive Power and Translations Into $\mathsf{EP}^{\{/,//,[],*\}}$

Using the extended tree patterns, we can easily express tree patterns from $\mathsf{P}^{\{/,//,[],*\}}$ and $\mathsf{XP}^{\{/,//,[],*\}}$, as well as queries in $\mathcal{CTQ}(\Downarrow, =)$. Namely, patterns from $\mathsf{P}^{\{/,//,[],*\}}$ are extended tree patterns from $\mathsf{EP}^{\{/,//,[],*\}}$, where each output variable $x_i \in \overline{x}$ is annotated by sub. XPath expressions from $\mathsf{XP}^{\{/,//,[],*\}}$ are a sublanguage of $\mathsf{EP}^{\{/,//,[],*\}}$, where only one output variable exists, and it is bound to the subtree rooted at the corresponding node. Finally, queries from $\mathcal{CTQ}(\Downarrow, =)$ are extended tree patterns from $\mathsf{EP}^{\{/,//,[],*\}}$, where the output variables x_i and the free variables y_j are bound to attribute nodes and annotated by val. The remaining XML query languages mentioned in Section 3.2, such as ps-queries, full XPath and XQuery are not expressible using the extended tree patterns.

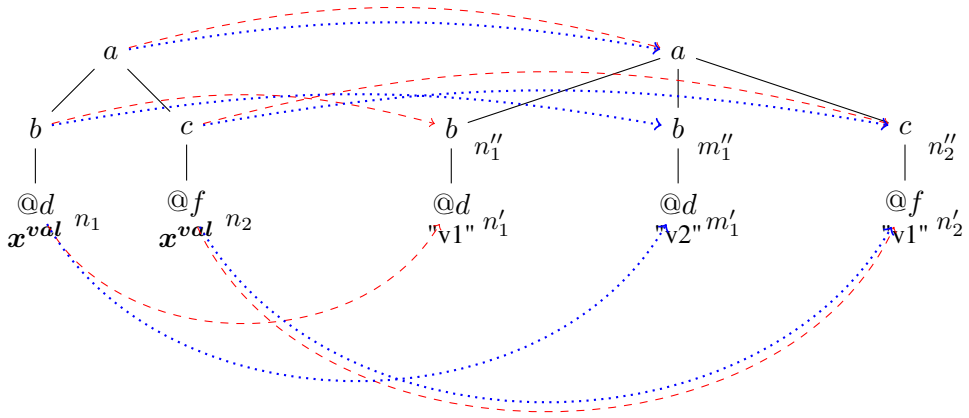


Figure 4.2: *An extended tree pattern and an XML tree, with two embeddings e_1 (red dashed arrows), e_2 (blue dotted arrows). Out of them, only e_1 is an extended embedding*

When translating an expression defined in $\mathsf{P}^{\{/,//,[],*\}}$, $\mathsf{XP}^{\{/,//,[],*\}}$ or $\mathcal{CTQ}(\Downarrow,=)$ into an extended tree pattern, our aim is to find an extended tree pattern that will return the same answers as the original expressions. We provide two algorithms for translating tree patterns from $\mathsf{P}^{\{/,//,[],*\}}$ and queries from $\mathcal{CTQ}(\Downarrow,=)$ into extended tree patterns from $\mathsf{EP}^{\{/,//,[],*\}}$. The translation of XPath expressions from $\mathsf{XP}^{\{/,//,[],*\}}$ into extended tree patterns from $\mathsf{EP}^{\{/,//,[],*\}}$ can be done by first translating the $\mathsf{XP}^{\{/,//,[],*\}}$ expression into a tree pattern from $\mathsf{P}^{\{/,//,[],*\}}$ (by using the translation proposed by [38]), and then applying our translation algorithm to the newly obtained tree pattern. Note that, as mentioned before, expressions in $\mathsf{XP}^{\{/,//,[],*\}}$ are evaluated starting from the document node of the tree. On the other hand, tree patterns, extended tree patterns and queries from $\mathcal{CTQ}(\Downarrow,=)$ are evaluated starting from the root node, i.e. the document node is ignored when computing an embedding, an extended embedding and evaluating the query from $\mathcal{CTQ}(\Downarrow,=)$ respectively.

First, we define the notions of translation of a tree pattern in $\mathsf{P}^{\{/,//,[],*\}}$, an XPath expression in $\mathsf{XP}^{\{/,//,[],*\}}$ and a query from $\mathcal{CTQ}(\Downarrow,=)$ into an extended tree pattern.

Definition 4.2.1 (Translation of $\mathsf{P}^{\{/,//,[],*\}}$ into $\mathsf{EP}^{\{/,//,[],*\}}$)**.** Let q be a tree pattern in $\mathsf{P}^{\{/,//,[],*\}}$ and p an extended tree pattern in $\mathsf{EP}^{\{/,//,[],*\}}$ whose output tuple has the same arity as the output tuple of q. We call p a *translation* of q if for all trees $T \in \mathcal{T}$ it holds that $q(T) = p(T)$. □

Definition 4.2.2 (Translation of $\mathsf{XP}^{\{/,//,[],*\}}$ into $\mathsf{EP}^{\{/,//,[],*\}}$)**.** Let q be an expression in $\mathsf{XP}^{\{/,//,[],*\}}$ and p an extended tree pattern in $\mathsf{EP}^{\{/,//,[],*\}}$ whose output tuple has arity 1. We call p a *translation* of q if for all trees $T \in \mathcal{T}$ it holds that $q(T) = p(T)$. □

Definition 4.2.3 (Translation of $\mathcal{CTQ}(\Downarrow,=)$ into $\mathsf{EP}^{\{/,//,[],*\}}$)**.** Let $\exists\overline{y}\,\varphi(\overline{x},\overline{y})$ be a query in $\mathcal{CTQ}(\Downarrow,=)$ and p an extended tree pattern in $\mathsf{EP}^{\{/,//,[],*\}}$. We call p a *translation* of $\exists\overline{y}\,\varphi(\overline{x},\overline{y})$ if for all trees $T \in \mathcal{T}$ it holds that $\varphi(T) = p(T)$. □

Note that by definition, queries in $\mathcal{CTQ}(\Downarrow, =)$ do not contain function symbols. Since we do not allow inequalities in the queries[1], we assume that equalities between attribute values are expressed by reusing variables [9]. Hence, we assume that expressions in $\mathcal{CTQ}(\Downarrow, =)$ have the form $\exists \overline{y}\, \pi(\overline{x}, \overline{y})$, where $\pi(\overline{x}, \overline{y})$ does not use horizontal navigation and function symbols, and contains multiple occurrences of variables. With the algorithm that we provide for translation of queries in $\mathcal{CTQ}(\Downarrow, =)$, we are also able to translate tree pattern formulae that do not use inequalities, horizontal navigation and function symbols of arbitrary arity. The only function symbols that we are able to support are those function symbols f that have the same arity as the output tuple $\overline{x}$ and moreover have the output tuple $\overline{x}$ as an argument, i.e. are used to construct terms of the form $f(\overline{x})$. These tree pattern formulae can be translated into queries from $\mathcal{CTQ}(\Downarrow, =)$ by replacing each term of the form $f(\overline{x})$ with a fresh existentially quantified variable y. Furthermore, tree pattern formulae without horizontal navigation, inequalities and function symbols are expressions in $\mathcal{CTQ}$ that do not have existentially quantified variables.

We now proceed with the definition of the translation algorithms and proving that the extended tree pattern obtained by the respective algorithms is indeed a translation of the respective input expressions.

Algorithm 1 Translation of a tree pattern into an extended tree pattern

function PTOEP(q)
Input a tree pattern $q = (N_q, E_q, label_q, root_q, \overline{x}_q)$ from $\mathsf{P}^{\{/,//,[],*\}}$
Output an extended tree pattern $p = (N_p, E_p, label_p, root_p, \overline{x}_p, \overline{y}_p, var_p)$ from $\mathsf{EP}^{\{/,//,[],*\}}$
  let q be a tree pattern $(N_q, E_q, label_q, root_q, \overline{x}_q)$
  define $N_p \leftarrow N_q$
  define $E_p \leftarrow E_q$
  define $label_p : N_p \rightarrow El \cup Att \cup \{*\}$ such that $\forall n \in N_p, label_p(n) = label_q(n)$
  define $root_p \leftarrow root_q$
  for all n in N_q **do**
    if $n = x_i$, for some $1 \leq i \leq k$ **then** /* if n is a node $x_i \in \overline{x}_q$ */
      add the variable x_i^{sub} to $\overline{x}_p$
      define $var_p(n) = x_i^{sub}$
    end if
  end for
  define $\overline{y}_p \leftarrow \emptyset$ /* $\overline{y}_p$ is empty */
  return $p = (N_p, E_p, label_p, root_p, \overline{x}_p, \overline{y}_p, var_p)$
end function

Proposition 4.2.1. *Suppose that q is a tree pattern from $\mathsf{P}^{\{/,//,[],*\}}$ and p an extended tree pattern from $\mathsf{EP}^{\{/,//,[],*\}}$ obtained by applying Algorithm 1 to q. Then, the extended tree pattern p is a translation of the tree pattern q.*

[1] As mentioned in Section 3.2, using inequalities in the queries makes the computation of answers undecidable.

Proof. Let $q = (N_q, E_q, label_q, root_q, \overline{x}_q)$ be a tree pattern from $\mathsf{P}^{\{/,//,[],*\}}$. Suppose that $p = (N_p, E_p, label_p, root_p, \overline{x}_p, \overline{y}_p, var_p)$ is an extended tree pattern from $\mathsf{EP}^{\{/,//,[],*\}}$ obtained by applying Algorithm 1 to q. We need to show that for all trees $T \in \mathcal{T}$, $q(T) = p(T)$.

Let T be an arbitrary tree from $\mathcal{T}$. Since both $q(T)$ and $p(T)$ are defined as sets of tuples, we need to show set equality between them. We show this by proving the two inclusion statements: $q(T) \subseteq p(T)$ and $q(T) \supseteq p(T)$.

$\subseteq$ The result of applying the tree pattern q to the tree T is defined as the set of tuples

$$q(T) = \{(e(x_1), \ldots, e(x_k)) \mid e \text{ is an embedding from } q \text{ to } T\}$$

where $x_1, \ldots x_k$ are the output nodes $\overline{x}_q$ of q.

We take an arbitrary tuple in the result $q(T)$, namely $(e(x_1), \ldots, e(x_k))$, where e is an arbitrary embedding from q to T. By the definition of an embedding, we have that e is a mapping from N_q to N that preserves the root, the child and descendant relations between the nodes and the labels of the nodes. The set of nodes, the root node, the set of edges and the labeling function are equal for q and p. Thus we can define a function $e' : N_p \to N$ such that for all nodes $n' \in N_p$ we have $e'(n') = e(n')$. The function e' is an extended embedding, since it is root, child, descendant and label preserving. Moreover, it satisfies the value equality preservation condition trivially, since, according to the algorithm, in p there are no variables annotated by val.

Let $n_1, \ldots, n_k$ be the nodes in the tree T such that $e(x_i) = n_i, 1 \leq i \leq k$. Let $n'_1, \ldots, n'_k$ be the nodes of the extended pattern that have the variables $x_1^{sub}, \ldots, x_k^{sub}$ attached, i.e. where $var_p(n'_i) = x_i^{sub}, 1 \leq i \leq k$. Then, it holds that $e'(n'_i) = e(x_i), 1 \leq i \leq k$. Hence, by the definition of the binding function we have that

$$\begin{array}{rcl} (e(x_1), \ldots, e(x_k)) & = & (n_1, \ldots, n_k) \\ & = & (e'(n'_1), \ldots, e'(n'_k)) \\ & = & (b(e'(n'_1), sub), \ldots, b(e'(n'_k), sub)) \end{array}$$

This yields that $(e(x_1), \ldots, e(x_k)) \in p(T)$.

$\supseteq$ The result of applying p to T is defined as

$$p(T) = \{(b(e'(n'_1), sub), \ldots, b(e'(n'_k), sub)) \mid \begin{array}{l} e' \text{ is an extended embedding} \\ \text{from } p \text{ to } T \text{ and } b \\ \text{is a binding function } \} \end{array}$$

where $n'_1, \ldots, n'_k$ are the nodes that have the variables $x_1^{sub}, \ldots, x_k^{sub}$ attached to them respectively, i.e. $var_p(n'_i) = x_i^{sub}, 1 \leq i \leq k$.

We choose a tuple $(b(e'(n'_1), a_1), \ldots, b(e'(n'_k), a_k))$ from the result of applying p to T where b is the binding function and e' is an arbitrary extended embedding from p to T. Since p is obtained by the translation algorithm, all the output variables are annotated by sub. According to the algorithm, p and q have the same sets of nodes, roots, sets of edges

and labeling functions. We define a function $e : N_q \to N$ such that for all nodes $n \in N_q$ we have $e(n) = e'(n)$. The function e is an embedding from q to T, since e' is root, child, descendant and label preserving.

Let $n_1, \ldots, n_k$ be the nodes in the tree T such that $b(e'(n'_i), sub) = n_i, 1 \leq i \leq k$. Let $x_1, \ldots, x_k$ be the output nodes of q. Then, it holds that $e(x_i) = e'(n'_i), 1 \leq i \leq k$. Therefore, by expanding the definition of the binding function, we get the following:

$$\begin{aligned}(b(e'(n'_1), sub), \ldots, b(e'(n'_k), sub)) &= (e'(n'_1), \ldots, e'(n'_k)) \\ &= (n_1, \ldots, n_k) \\ &= (e(x_1), \ldots, e(x_k))\end{aligned}$$

Thus, $(b(e'(n'_1), sub), \ldots, b(e'(n'_k), sub)) \in q(T)$.

Since the tree T was arbitrarily chosen, we conclude that $q(T) = p(T)$ for all trees $T \in \mathcal{T}$. Hence, p is a translation of q. □

Algorithm 2 Translation of an XPath expression from $\mathsf{XP}^{\{/,//,[],*\}}$ into an extended tree pattern

1: **function** XPTOEP(q)
   Input an XPath expression q from $\mathsf{XP}^{\{/,//,[],*\}}$
   Output an extended tree pattern $p = (N_p, E_p, label_p, root_p, \overline{x}_p, \overline{y}_p, var_p)$ from $\mathsf{EP}^{\{/,//,[],*\}}$
2: $p' \leftarrow$ XPTOP(q) /* translate q into a tree pattern p' [38] */
3: $p \leftarrow$ PTOEP(p') /* translate p' into an extended tree pattern p */
4: **return** p
5: **end function**

Proposition 4.2.2. *Let q be an XPath expression from $\mathsf{XP}^{\{/,//,[],*\}}$ and p an extended tree pattern from $\mathsf{EP}^{\{/,//,[],*\}}$ obtained by applying Algorithm 2 to q. Then, p is a translation of q.*

Proof. The fact that p is a translation of q follows from the fact that one can compute an intermediate tree pattern p' from $\mathsf{P}^{\{/,//,[],*\}}$ that is a translation of q from $\mathsf{XP}^{\{/,//,[],*\}}$, and then translate this tree pattern into an extended tree patten using Algorithm 1. The intermediate tree pattern p' is obtained by a translation algorithm presented in [38], which translates expressions from $\mathsf{XP}^{\{/,//,[],*\}}$ into 1-ary tree patterns from $\mathsf{P}^{\{/,//,[],*\}}$. □

Proposition 4.2.3. *Let $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$ be a query in $\mathcal{CTQ}(\Downarrow, =)$ and p an extended tree pattern from $\mathsf{EP}^{\{/,//,[],*\}}$ obtained by applying Algorithm 3 to $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$. Then, p is a translation of $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$.*

Proof. Let $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$ be a query in $\mathcal{CTQ}(\Downarrow, =)$. Suppose that p is an extended tree pattern obtained by applying Algorithm 3 to $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$. We need to show that for all trees $T \in \mathcal{T}$ it holds that $\varphi(T) = p(T)$.

Algorithm 3 Translation of a query in $\mathcal{CTQ}(\Downarrow, =)$ into an extended tree pattern

function CTQTOEP($\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$)
Input a query $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$ in $\mathcal{CTQ}(\Downarrow, =)$
Output an extended tree pattern $p = (N_p, E_p, label_p, root_p, \overline{x}_p, \overline{y}_p, var_p)$ from $\mathsf{EP}^{\{/,//,[],*\}}$
  let $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$ be of the form $\exists \overline{y}\, \pi(\overline{x}, \overline{y})$
  let $\pi(\overline{x}, \overline{y})$ be of the form $l(x_1, \ldots, x_m, y_{m+1}, \ldots, y_{m'})[\lambda]$
  if $\lambda = \varepsilon$ **then**
    $p \leftarrow l\,[\, x_1^{val} : @att_1, \ldots, x_m^{val} : @att_m, y_{m+1}^{val} : @att_{m+1}, \ldots, y_{m'}^{val} : @att_{m'}\,]$
  else if $\lambda = \pi$ **then**
    $p_1 \leftarrow$CTQTOEP(π)
    $p \leftarrow l\,[\, x_1^{val} : @att_1, \ldots, x_m^{val} : @att_m, y_{m+1}^{val} : @att_{m+1}, \ldots, y_{m'}^{val} : @att_{m'},\ p_1\,]$
  else if $\lambda = //\pi$ **then**
    $p_1 \leftarrow$CTQTOEP(π)
    $p \leftarrow l\,[\, x_1^{val} : @att_1, \ldots, x_m^{val} : @att_m, y_{m+1}^{val} : @att_{m+1}, \ldots, y_{m'}^{val} : @att_{m'},\ //p_1\,]$
  else if $\lambda = \lambda_1, \ldots, \lambda_{m''}$ **then**
    for $i = 1$ to m'' **do**
      $p_i \leftarrow$CTQTOEP(λ_i)
    end for
    $p \leftarrow l\,[\, x_1^{val} : @att_1, \ldots, x_m^{val} : @att_m, y_{m+1}^{val} : @att_{m+1}, \ldots, y_{m'}^{val} : @att_{m'}, p_1, \ldots, p_{m''}\,]$
  end if
  return p
end function

In order to prove this, we need to introduce the notion of homomorphism between a tree pattern formula and an XML tree. The definition of a homomorphism between an arbitrary tree pattern formula and a tree T has originally been defined in [9]. Here we present a definition of a homomorphism between a tree pattern formula $\pi(\overline{x}, \overline{y})$ and a tree T, where the tree pattern formula does not use horizontal navigation, inequalities, and does not contain occurrences of function symbols. Also, for $\pi(\overline{x}, \overline{y})$, we assume that equalities are expressed by repetition of variables. Let $\pi(\overline{x}, \overline{y})$ be such a tree pattern formula and $\overline{a}, \overline{b}$ tuples of atomic values of the same arity as $\overline{x}, \overline{y}$ respectively. A homomorphism $h : S_{\pi(\overline{a},\overline{b})} \to N$ from the set $S_{\pi(\overline{a},\overline{b})}$ of subformulae of $\pi(\overline{a}, \overline{b})$ to the set of nodes N of the tree T is a function that assigns a node from the tree T to each subformula of $\pi(\overline{a}, \overline{b})$ such that:

1. $h(//\pi_1)$ is an ancestor of $h(\pi_1)$;
2. if $h(l(\overline{t})[\mu_1, \ldots, \mu_m]) = n$ then
   a) either $l = *$ or $label(n) = l$;
   b) $\overline{t}$ is the tuple of attributes of n;
   c) if μ_i is of the form π_i then $h(\pi_i)$ is a child of n.

Moreover, it holds that $T \models \pi(\overline{a}, \overline{b})$ iff there exists a homomorphism $h : S_{\pi(\overline{a},\overline{b})} \to N$.

Let T be an arbitrary tree from $\mathcal{T}$. We show the equality between the sets of tuples $\varphi(T)$ and $p(T)$ by showing the two set inclusions $\varphi(T) \subseteq p(T)$ and $\varphi(T) \supseteq p(T)$ separately.

$\subseteq$ Let $\overline{a}$ be an arbitrary tuple in the result $\varphi(T)$ of applying the $\mathcal{CTQ}(\Downarrow, =)$ $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$ to T. This means that there exists a tuple $\overline{b}$ that corresponds to a valuation of the variables $\overline{y}$ such that $T \models \varphi(\overline{a}, \overline{b})$. Recall that the $\mathcal{CTQ}$s that we consider are from the fragment $\mathcal{CTQ}(\Downarrow, =)$, meaning that they are of the form $\exists \overline{y}\, \pi(\overline{x}, \overline{y})$. Hence, we have that $T \models \pi(\overline{a}, \overline{b})$. This implies that there exists a homomorphism $h : S_{\pi(\overline{a},\overline{b})} \to N$ that assigns nodes from T to subformulae of $\pi(\overline{a}, \overline{b})$. It remains to be shown that $\overline{a}$ is also in the result of applying the extended tree pattern p to the tree T. We will prove this by showing the existence of an extended embedding e from the nodes of p to the nodes of T such that:

$$\overline{a} = (b(e(n_1), a_1), \ldots, b(e(n_k), a_k)), \quad \text{where } var_p(n_i) = x_i^{val} \text{ and } a_i = val, 1 \leq i \leq k.$$

Recall that p is obtained from the $\mathcal{CTQ}(\Downarrow, =)$ $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$, which is of the form $\exists \overline{y}\, \pi(\overline{x}, \overline{y})$, where $\pi(\overline{x}, \overline{y})$ is a tree pattern formula that does not use horizontal navigation, inequalities and function symbols, and is of the form

$$\pi(\overline{x}, \overline{y}) = l(x_1, \ldots, x_m, y_1, \ldots, y_{m'})[\lambda_1, \ldots, \lambda_{m''}]$$

and hence $\pi(\overline{a}, \overline{b})$ is of the form $l(a_1, \ldots, a_m, b_1, \ldots, b_{m'})[\lambda_1, \ldots, \lambda_{m''}]$. Then, according to the algorithm, p has the form

$$l[x_1^{val} : @att_1, \ldots x_m^{val} : @att_m,\ y_1^{val} : @att_1, \ldots, y_{m'}^{val} : @att_{m'},\ p_1, \ldots, p_{m''}]$$

where $@att_i, 1 \leq i \leq m$ and $@att_j, 1 \leq j \leq m'$ are the labels of the attributes of the node labeled by l, and $p_1, \ldots, p_{m''}$ are the translations of $\lambda_1, \ldots, \lambda_{m''}$ respectively, as defined in lines 12-16 of Algorithm 3.

We have already stated that there exists a homomorphism $h : S_{\pi(\overline{a},\overline{b})} \to N$. Let n be a node[2] in the tree T such that $h(\pi(\overline{a}, \overline{b})) = n$, and let n' be the root of p, labeled by l, i.e. $label_p(n') = l$. We define a function $e : N_p \to N$ as follows:

1. $e(n') = n$;
2. Let $n_1, \ldots, n_{m+m'}$ and $n'_1, \ldots, n'_{m+m'}$ be the attribute nodes of n and n' respectively. Then $e(n'_i) = n_i, 1 \leq i \leq m + m'$;
3. repeat steps 1 and 2 for each extended pattern p_j, obtained as a translation of $\lambda_j, 1 \leq j \leq m''$.

The function e is child, descendant, root and label preserving, which follows from the definition of the homomorphism h. Moreover, e satisfies the value equality preservation

[2]This is in fact the root node of the tree T, since $\pi(\overline{a}, \overline{b})$ is witnessed at the root.

condition, because the homomorphism h satisfies the equalities between values, which are enforced by repetition of variables. Hence, e is an extended embedding from p to T.

For each variable $x_i \in \overline{x}, 1 \leq i \leq k$ from the query $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$, the corresponding variable in p according to the algorithm is $x_i^{val}, 1 \leq i \leq k$. Let a_i be the value that x_i has in $\pi(\overline{a}, \overline{b})$ and let $n'_i \in N_p$ be the node such that $var_p(n'_i) = x_i^{val}$. Then, we have that

$$b(e(n'_i), val) = a_i, 1 \leq i \leq k, \text{ and hence } \overline{a} \in p(T).$$

$\supseteq$ Let $\overline{a}$ be an arbitrary tuple in the result $p(T)$ of applying p to T. This means that there exists an extended embedding e from the nodes of p to the nodes of T such that $\overline{a} = (b(e(n_1), val), \ldots, b(e(n_k), val))$, where $var_p(n_i) = x_i^{val}$, $1 \leq i \leq k$. We need to show that $\overline{a}$ is also in the result of applying the query $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$ over the tree T. Since $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$ belongs to the fragment $\mathcal{CTQ}(\Downarrow, =)$, we have that it is of the form $\exists \overline{y}\, \pi(\overline{x}, \overline{y})$. Thus, we need to show that $\overline{a}$ is a valuation of the free variables of $\exists \overline{y}\, \pi(\overline{x}, \overline{y})$, such that there exists a tuple of atomic values $\overline{b}$ that is a valuation of the variables $\overline{y}$, such that $T \models \pi(\overline{a}, \overline{b})$. In order to show this, we first define a tuple $\overline{b}$, and then show that $T \models \pi(\overline{a}, \overline{b})$ by showing that there exists a homomorphism $h : S_{\pi(\overline{a},\overline{b})} \to N$ from the set $S_{\pi(\overline{a},\overline{b})}$ of subformulae of $\pi(\overline{a}, \overline{b})$ to the set N of nodes of T.

We define a valuation for the existentially quantified variables $\overline{y}$ as follows:

$$\overline{b} = (b(e(n_1), val), \ldots, b(e(n_{k'}), val)), \text{ where } var_p(n_j) = y_j^{val}, 1 \leq j \leq k'$$

We define a function $h : S_{\pi(\overline{a},\overline{b})} \to N$ as follows. For each subformula $\rho(\overline{c}, \overline{d})$ of $\pi(\overline{a}, \overline{b})$, with $\overline{c} \subseteq \overline{a}$ and $\overline{d} \subseteq \overline{b}$ we have that $\rho(\overline{c}, \overline{d})$ is of the form

$$l(c_1, \ldots, c_m, d_1, \ldots, d_{m'})[\mu_1, \ldots, \mu_{m''}]$$

and hence $\rho(\overline{u}, \overline{v})$ is of the from $l(u_1, \ldots, u_m, v_1, \ldots, v_{m'})[\mu_1, \ldots, \mu_{m''}]$ with $\overline{u} \subseteq \overline{x}$, $\overline{v} \subseteq \overline{y}$, and $\overline{c}$ (resp. $\overline{d}$) is of the same arity as $\overline{u}$ (resp. $\overline{v}$).

Since $\rho(\overline{c}, \overline{d})$ is a subformula of $\pi(\overline{a}, \overline{b})$, according to lines 12-16 of the algorithm, it holds that $\rho(\overline{u}, \overline{v})$ has been translated into a subpattern p' of p that has the form $l[u_1^{val} : @att_1, \ldots, u_m^{val} : @att_m, v_1^{val} : @att_1, \ldots, v_{m'}^{val} : @att_{m'}, p_1, \ldots, p_{m''}]$. Let n' be the root node of the subpattern p' of p with $label_{p'}(n') = l$. Let $n \in N$ be a node of the tree such that $e(n') = n$. We define $h(\rho(\overline{c}, \overline{d})) = n$. Let $n_1, \ldots, n_{m+m'}$ and $n'_1, \ldots, n'_{m+m'}$ be the attribute nodes of n and n' respectively, such that $e(n'_i) = n_i$, $1 \leq i \leq m + m'$. For the particular extended embedding e, we have that

$$c_i = b(e(n'_i), val) = b(n_i, val) = value_{@att_i}(n_i), 1 \leq i \leq m$$

and

$$d_j = b(e(n'_{m+j}), val) = b(n_{m+j}, val) = value_{@att_j}(n_{m+j}), 1 \leq j \leq m'$$

Thus the tuple $(\overline{c}, \overline{d}) = (c_1, \ldots, c_m, d_1, \ldots, d_{m'})$ is exactly the tuple of attribute values of $n = h(\rho(\overline{c}, \overline{d}))$.

The function h satisfies the condition 1 of the homomorphism definition, since e is descendant preserving. Condition 2(a) is satisfied since e is label preserving. We have seen that the tuple of atomic values $(\overline{c}, \overline{d})$ exactly corresponds to the attribute values of the node $h(\rho(\overline{c}, \overline{d}))$, hence condition 2(b) is also satisfied. Condition 2(c) is satisfied due to the fact that e is child preserving.

Thus, we have shown that there exists a homomorphism $h : S_{\pi(\overline{a},\overline{b})} \to N$, which yields that $T \models \pi(\overline{a}, \overline{b})$. Hence, $\overline{a} \in (\exists \overline{y}\, \varphi(\overline{a}, \overline{b}))(T)$.

Since the tree T was arbitrarily chosen, we conclude that for all XML trees T it holds that $\varphi(T) = p(T)$, i.e. p is a translation of $\exists \overline{y}\, \varphi(\overline{x}, \overline{y})$. □

Example 4.2.1. Consider again the XML tree T depicted in Figure 3.1b. We give the translations of the XPath expressions in $\mathsf{XP}^{\{/,//,[],*\}}$ from Example 3.2.1, the tree pattern in $\mathsf{P}^{\{/,//,[],*\}}$ from Example 3.2.2 and the tree pattern formula from Example 3.2.3 into extended tree patterns from $\mathsf{EP}^{\{/,//,[],*\}}$.

$\mathsf{XP}^{\{/,//,[],*\}}$: The translation of the $\mathsf{XP}^{\{/,//,[],*\}}$ expression from Example 3.2.1, $a/b[//@d]/e$, in the $\mathsf{EP}^{\{/,//,[],*\}}$ syntax is as follows:

$$p_1 = a[b[//@d, x^{sub} : e]]$$

$\mathsf{P}^{\{/,//,[],*\}}$: The following extended tree pattern corresponds to the translation of the tree pattern from Example 3.2.2, depicted in Figure 3.2:

$$p_2 = a[b[@d, x_1^{sub} : e, f[x_2^{val} : @i]], c[x_3^{val} : @g]]$$

$\mathcal{CTQ}(\Downarrow, =)$: The tree pattern formula from Example 3.2.3, $a[b(x_d)[e[h(x_i)], f(x_i)], c(x_g)]$, is also a query in $\mathcal{CTQ}(\Downarrow, =)$ without free variables. Hence, it is translated into an $\mathsf{EP}^{\{/,//,[],*\}}$ expression as follows:

$$p_3 = a[b[x_1^{val} : @d, e[h[x_2^{val} : @j]], f[x_3^{val} : @i]], c[x_4^{val} : @g]]$$

△

Table 4.1 summarizes the XML query languages we have discussed in this section. The table shows the type of output returned by each class of queries along with the possibility to translate a query expressed in one language into a query in another language.

4.3 XML Mapping Assertions

In Chapter 2, we have introduced several problems for relational databases whose foundations are based on schema mappings. In this thesis, we focus on the same problems in an XML setting, and give an overview of the works that have addressed these problems so far. In order to describe the different types of schema mappings, in this section we define several kinds of

Table 4.1: *Comparison between the XML query languages*

language	output	translates to
$\mathsf{P}^{\{/,//,[],*\}}$	tuples of subtrees	$\mathsf{EP}^{\{/,//,[],*\}}$
$\mathsf{XP}^{\{/,//,[],*\}}$	single subtree	$\mathsf{P}^{\{/,//,[],*\}}$, $\mathsf{EP}^{\{/,//,[],*\}}$
$\mathsf{EP}^{\{/,//,[],*\}}$	tuples of subtrees and values	$\mathsf{EP}^{\{/,//,[],*\}}$
$\mathcal{CTQ}(\Downarrow,=)$	tuples of values	$\mathsf{EP}^{\{/,//,[],*\}}$
ps-queries	prefix of the queried tree	/
XQuery	single subtree or newly created tree	/

mapping assertions, which differ both in the language and in the meaning of the relationship between the queries over the two schemas. Throughout the literature, we have come across with works that define XML data exchange settings using GLAV mappings expressed in different classes of $\mathcal{CTQ}$, approaches that perform XPath query rewriting using XPath views (that correspond to LAV mapping assertions expressed in $\mathsf{XP}^{\{/,//,[],*\}}$), XML data integration systems that are based on both LAV and GLAV mapping assertions and rewriting of XQuery queries using XQuery views which use mapping assertions whose expressive power goes beyond the one of the GLAV mapping assertions. In this section, we will introduce these different kinds of mapping assertions, and we will later use them when comparing the different approaches.

Another important feature of the mapping assertions is their granularity. The *granularity* of a mapping assertion defines what kind of values are transferred from the left to the right-hand side of the mapping assertion. It depends on the annotation of the output variables in the mapping assertion. We can define the following different types of mapping assertions based on the granularity:

1. mapping assertions that transfer atomic values. In these mapping assertions, all the output variables are annotated by val, and none of them is annotated by sub;

2. mapping assertions that transfer subtrees. If at least one of the output variables is annotated by sub, the mapping assertion transfers one or multiple subtrees bound to the output variables annotated by sub;

3. mapping assertions that transfer subtrees and can further modify these subtrees. These mapping assertions are an extension of the previous ones. The difference between the two is that these mapping assertions do not only transfer the subtrees, but they also support restructuring of the elements, their renaming, as well as introduction of new elements.

Based on the extended tree patterns, we are able to define a new type of schema mapping assertions for XML[3], that essentially capture the expressive power of the mapping assertions expressed in the languages that can be translated into the extended tree patterns. These mapping assertions will then allow us to uniformly view all the related works and facilitate the comparison between the approaches. For this purpose, we define both XML LAV and GLAV mapping assertions based on extended tree patterns. Moreover, we define XML LAV mapping assertions based on ps-queries. As we will see in Section 5.2, we can use the XML LAV mapping assertions to express XPath views used for XPath query rewriting. Since these XPath views are materialized, on the left-hand side of the LAV mapping assertion we will have an extended tree pattern that provides access to the data stored in the materialized view, while on the right-hand side we will have the view definition expressed as an extended tree pattern. The query that needs to be rewritten and the view definition agree on the vocabulary.

Definition 4.3.1 (XML mapping assertion based on $\mathsf{EP}^{\{/,//,[],*\}}$)**.** An *XML mapping assertion based on* $\mathsf{EP}^{\{/,//,[],*\}}$ is an expression of the form

$$\forall \overline{x}\,(\exists \overline{y}\, q_{\mathcal{S}}(\overline{x},\overline{y}) \rightsquigarrow \exists \overline{z}\, q_{\mathcal{G}}(\overline{x},\overline{z}))$$

where $\exists \overline{y}\, q_{\mathcal{S}}(\overline{x},\overline{y})$ and $\exists \overline{z}\, q_{\mathcal{G}}(\overline{x},\overline{z}))$ are extended tree patterns from $\mathsf{EP}^{\{/,//,[],*\}}$ that share the same tuple $\overline{x}$ of free variables and have $\overline{y}$ and $\overline{z}$ respectively as existentially quantified variables.

Depending on the layout of $q_{\mathcal{S}}$, we distinguish the following two types of XML mapping assertions:

- *XML LAV mapping assertion based on* $\mathsf{EP}^{\{/,//,[],*\}}$, where $q_{\mathcal{S}}$ is an extended tree pattern used to extract the data from the materialized view, which conforms to a simple schema, and $q_{\mathcal{G}}$ is an extended tree pattern over the global schema $\mathcal{G}$ that corresponds to the view definition;
- *XML GLAV mapping assertion based on* $\mathsf{EP}^{\{/,//,[],*\}}$, where $q_{\mathcal{S}}$ is an extended tree pattern over the source schema $\mathcal{S}$, and $q_{\mathcal{G}}$ is an extended tree pattern over the global schema $\mathcal{G}$. □

In the context of XML GLAV mapping assertions, we assume that the source and the global schema are given in the form of a DTD. The tuple $\overline{x}$ is a tuple of output variables, each corresponding to an output node and annotated according to the type of the output (value or subtree). Since the source and the global patterns share the same output tuple, $\overline{x}$, we refer to it as the output tuple of the mapping assertion. Depending on the interpretation of the relationship between the two extended tree patterns, we say that the mapping assertion can be sound ($\forall \overline{x}\,(\exists \overline{y}\, q_{\mathcal{S}}(\overline{x},\overline{y}) \rightarrow \exists \overline{z}\, q_{\mathcal{G}}(\overline{x},\overline{z}))$) or exact ($\forall \overline{x}\,(\exists \overline{y}\, q_{\mathcal{S}}(\overline{x},\overline{y}) \leftrightarrow \exists \overline{z}\, q_{\mathcal{G}}(\overline{x},\overline{z}))$).

As we have already mentioned in Section 2.2, the query processing in the presence of GLAV mapping assertions can be done by splitting each GLAV mapping assertion into a GAV and a LAV mapping assertion, by means of an intermediate view. We follow the same approach for query processing in the presence of XML GLAV mapping assertions. The intermediate view

[3]We refer to the schema mapping assertions for XML as XML mapping assertions. We omit the word *schema* in order to avoid ambiguity with XML Schema.

has the following schema. The root element is labeled with $view$, and has zero or more children named $tuple$, where each tuple has k children labeled with $t_1, \ldots, t_k$. Each t_i corresponds to a variable $x_i^{a_i}, 1 \leq i \leq k$ in the k-ary output tuple $\overline{x}$ of the mapping assertion. More precisely:

- if the i-th variable in $\overline{x}$ is annotated by sub, then the node labeled by t_i has a single child labeled by l_i, same as the node to which x_i^{sub} is attached in the extended tree pattern $q_{\mathcal{S}}$, i.e. $l_i = label_{q_{\mathcal{S}}}(n_i)$, $var(n_i) = x_i^{sub}$, where n_i is a node in $q_{\mathcal{S}}$. Moreover, the variable x_i^{sub} is also attached to the node labeled with l_i;
- if the i-th variable in $\overline{x}$ is annotated by val, then the node labeled by t_i has a single attribute node, that is labeled by $l_i = @value$, and that has the variable x_i^{val} attached to it.

The splitting of the XML GLAV mapping assertions is depicted in Figure 4.3. Note that, the LAV part of the split XML GLAV mapping assertion in fact coincides with the definition of the XML LAV mapping assertion, which on the left-hand side stores the result of a materialized view, and on the right-hand side expresses the view definition using an extended tree pattern.

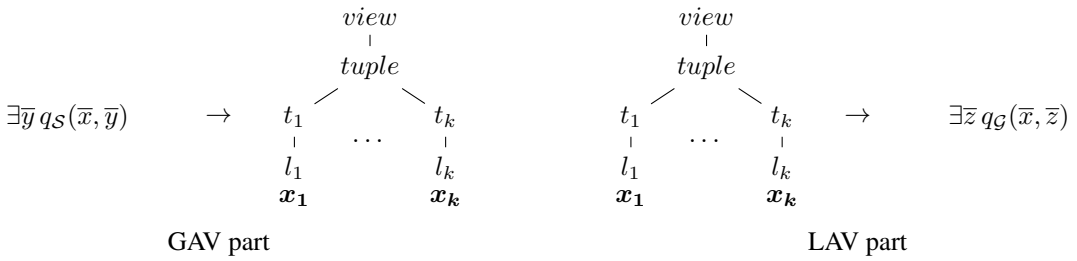


Figure 4.3: *Split XML GLAV mapping assertion*

Another type of schema mapping assertions that we have come across with in the literature are XML LAV mapping assertions that use ps-queries [41]. As we have argued before, ps-queries are not expressible using extended tree patterns. Also, the context in which these LAV mapping assertions are used is a rather limited one, as it only allows the source schemas to be portions of the global schema.

Definition 4.3.2 (XML mapping assertion in ps-queries)**.** An *XML LAV mapping assertion based on ps-queries* is a triple (S, M, as), where S is a source schema, M is a ps-query coherent with the source schema S and as is a specification of the semantics of the mapping, such that $as \in \{sound, exact\}$.

A ps-query M is *coherent* with a source schema S if for every XML tree D that conforms to the schema S, there exists an XML tree T such that $D \leq T$ and $M(T) \simeq D$. □

We will use the three different types of XML mapping assertions to describe the different approaches that can be found in the literature on XML data management.

CHAPTER 5

Data Integration, Data Exchange and Query Answering Using Views in an XML Setting

Lately, many problems that are already well studied in the case of relational databases, have been revisited and dealt with in the case of XML databases. The research community has been challenged to find definitions of the well known concepts in relational database theory and translate them to fit the XML setting. The translation process is not as trivial as it may appear. This is due to the structural difference between the relational databases (formally defined as n-ary relations) and XML documents (represented as ordered labeled trees). Some of these problems that are of interest to our work include XML with incomplete information, XML data exchange, XML data integration and answering XML queries using XML views. In this section, we give an overview of several works that address these problems and present results which reflect the relational ones, and improve the understanding of which formalisms to use when doing research in XML data management.

5.1 XML With Incomplete Information

We have already introduced the notion of relational databases with incomplete information in Section 2.1. Here, we give an overview of some approaches that deal with incomplete information in XML documents, as well as the definition of certain answers for query answering over XML documents with incomplete information.

Incomplete information in XML documents may appear as null values, but also as missing information about the structure of the document. In practice, incomplete information in XML documents can be modeled by specifying optional elements or optional attributes in the DTD associated with the XML document. The work in [6] proposes a way of dealing with incomplete information in the form of nulls in the XML setting. They consider incomplete XML trees, sim-

plified DTDs that ignore the order imposed by the production rules and queries that are prefixes (i.e. subtrees starting from the root) of the incomplete XML tree. In this setting, they propose a representation system for XML trees that follows the definition of c-tables [30]. The authors define query answering over incomplete XML trees using the notion of certain prefix, which is an analog of certain answers. In [14], a representation for incomplete trees is proposed. This representation deals with different sources of incompleteness: both null values (missing attribute values) and structural incompleteness (missing node identifiers, missing node labels replaced by wildcards, absence of a precise specification of the vertical and horizontal relationships between the nodes) are considered. The authors propose a formalism called incomplete tree descriptions, that is used for describing XML trees with incomplete information and characterization of classes of trees with the same parameters that cause incompleteness. Given these incomplete tree descriptions, the authors study the computational tasks concerning incomplete information in XML, such as consistency and query answering. They show that query answering using an incomplete XML tree is in general intractable, but tractable results can be achieved by imposing several restrictions on the query language (use unions of $\mathcal{CTQ}$s) and on the incomplete trees (use incomplete trees with missing data values and labels).

Certain Answers

We have introduced the notion of certain answers for relational databases in Chapter 2 as a concept that emerged from databases with incomplete information. When querying incomplete XML documents, one is also interested in getting certain answers from the XML documents. The need of computing certain answers also arises when querying data that comes from different documents, such as in data integration systems and data exchange settings. As we have seen, many query languages are used for querying XML documents throughout the literature. The existence of many query languages implies that for each query language, there exists a separate definition of what a certain answer is. Thus, we have encountered definitions of notions such as certain prefix for ps-queries, certain answers for queries returning tuples tuples of atomic values and max-descriptions for queries returning trees. In this section, we give a brief overview of the different characterizations of certain answers, and redefine the decision problem of finding certain answers in terms of these characterizations.

Query languages such as tree pattern formulae and $(\mathcal{U})\mathcal{CTQ}s$ return tuples of atomic values stored in the attribute nodes of the queried XML tree. Hence, since the answer to such queries is a set of tuples, it is straightforward to define the set of certain answers. In fact, the definition of certain answers for XML queries returning tuples of atomic values coincides with the definition of certain answers in relational databases. The *certain answers* of a query q returning tuples of atomic values given a set of XML trees $\mathbf{T} \subseteq \mathcal{T}$ is defined as an intersection of the results obtained by evaluating q over each tree $T \in \mathbf{T}$.

Certain answers for ps-queries are defined in terms of the answer a ps-query returns, namely a prefix of a tree, called *certain prefix*. The tree that is being queried to obtain the certain prefix is an intersection of the trees in the set of trees $\mathbf{T} \subseteq \mathcal{T}$. In the setting of ps-query answering and computing certain prefixes, an intersection between two trees T_1 and T_2 is defined as the maximal common prefix of T_1 and T_2. More formally, the intersection $T_1 \cap T_2$ of T_1 and T_2 is a tree that satisfies the following conditions: (a) $T_1 \cap T_2 \leq T_1$ and $T_1 \cap T_2 \leq T_2$; (b) for all T'

such that T' is not isomorphic to $T_1 \cap T_2$, if $T' \leq T_1$ and $T' \leq T_2$, then $T' \leq T_1 \cap T_2$. Recall the definitions of a tree prefix and tree isomorphism presented in Section 3.2

A more recent work ([18]) proposes theoretical grounds for defining certain answers for queries returning trees, such as XQuery. In the spirit of maximality of certain answers for the relational case, the authors define max-descriptions for XML trees in terms of a restricted form of tree pattern formulae defined in the previous section. These restricted tree pattern formulae do not support the equality and inequality atoms, the next and following sibling relations and the descendant relation, as well as the wildcard label. Moreover, they define a ground variant of such restricted tree pattern formulae by assigning values from the set of strings S to the variables in the tree pattern formula. The certain answers over an XML tree are thus defined w.r.t. max-descriptions. Given a set $\mathbf{T}$ of XML trees, the theory of $\mathbf{T}$ is defined as the set of tree pattern formulae that are satisfied by all the trees in $\mathbf{T}$, i.e. $Th(\mathbf{T}) = \{\pi \mid \forall T \in \mathbf{T} : T \models \pi\}$. Note that the satisfaction relation used does not consider function symbols, and starts the evaluation from the root of the tree. A *max-description* of a set of XML trees $\mathbf{T}$ is a tree pattern formula π such that the trees satisfying π are exactly those trees satisfying all the tree pattern formulae in the theory of $\mathbf{T}$. A tree pattern formula π is a *certain answer* to a query q over a set of trees $\mathbf{T} \subseteq \mathcal{T}$ if it is a max-description of $q(\mathbf{T})$, where $q(\mathbf{T}) = \{q(T) \mid T \in \mathbf{T}\}$. Note that, in this case, XML queries returning trees (such as XQuery, XPath, tree patterns and fragments thereof) need to be considered.

Now that we have seen the different definitions of certain answers for different kinds of queries, we can revisit the problem of finding certain answers in an XML setting and propose the following decision problems, given a set of trees $\mathbf{T} \subseteq \mathcal{T}$.

PROBLEM:	CERTAINANSWER$^{tuple}_{\mathbf{T}}(q, \bar{t})$
shorthand:	CA$^{tuple}_{\mathbf{T}}(q, \bar{t})$
INPUT:	A query q returning tuples of atomic values and a tuple $\bar{t}$ of the same arity as q
QUESTION:	Is $\bar{t}$ a certain answer of q?

PROBLEM:	CERTAINANSWER$^{ps}_{\mathbf{T}}(q, p)$
shorthand:	CA$^{ps}_{\mathbf{T}}(q, p)$
INPUT:	A ps-query q and a prefix p
QUESTION:	Is p a certain answer of q?

PROBLEM:	CERTAINANSWER$^{tree}_{\mathbf{T}}(q, \pi)$
shorthand:	CA$^{tree}_{\mathbf{T}}(q, \pi)$
INPUT:	A query q returning trees and a tree pattern formula π
QUESTION:	Is π a certain answer of q?

In the following, we consider several works that compute certain answers of XML queries based on schema mappings. Hence, for a given set of mapping assertions $\mathcal{M}$ we can analogously define the problems CA$^{tuple}_{\mathcal{M}}(q, \bar{t})$, CA$^{ps}_{\mathcal{M}}(q, p)$, CA$^{tree}_{\mathcal{M}}(q, \pi)$. The mapping assertions are used

to obtain a set of target trees $\mathbf{T}$ which are used in computing the certain answers as in the case of $\mathrm{CA}_{\mathbf{T}}^{tuple}(q, \bar{t}), \mathrm{CA}_{\mathbf{T}}^{ps}(q, p), \mathrm{CA}_{\mathbf{T}}^{tree}(q, \pi)$.

5.2 Data Integration, Data Exchange and Query Answering Using Views in an XML Setting

In Chapter 2 we have introduced several problems in a relational database setting, such as data integration, data exchange and query answering using views. In the remainder of this section, we will focus on various works that address these problems in an XML setting. We try to find features that characterize each of the approaches, and propose several criteria along which we compare them. In order to analyze the approaches and to match them accordingly, we will represent them using the XML mapping assertions, introduced in Section 3.2. All of the works that we will consider use schema mappings in some manner, either for specifying the relationship between two XML documents, or between an XML document and a view. For some of the approaches, the expressive power of the LAV mapping assertions is enough to capture the relationships between the documents. Others use the full expressive power of the XML GLAV mapping assertions. Finally, there are also those which go beyond the expressive power of our XML GLAV formalism. These works utilize nested mapping assertions and a procedural query language (XQuery) as opposed to the XML GLAV mapping assertions which are flat (i.e. not nested) and use a declarative query language (extended tree patterns).

Thus, the first classification of the approaches is done by the type of the schema mappings used. Along each class of approaches, we further characterize each one of them, by using the following criteria:

- granularity of the mapping assertions;
- interpretation of the mapping assertions;
- type of query answering;
- number of sources;
- use of constraints;
- preservation of order in the XML document.

The granularity of the mapping assertions is a crucial feature of each of the approaches and is tightly connected with the choice of a mapping language. As discussed previously, we distinguish three types of mapping assertions based on granularity: mapping assertions that transfer atomic (scalar) values, mapping assertions that transfer subtrees, and mapping assertions that transfer scalar values, subtrees and additionally allow modification of the transferred subtrees, as well as creation of new trees by renaming and restructuring the transferred subtrees. Another feature that considers the mapping assertions is their interpretation, which is also known as view semantics. In particular we are interested if the mapping assertions are interpreted as sound or exact (i.e. if the view semantics is defined under the OWA or CWA, respectively). There are

several different types of query answering under schema mappings which have been considered throughout the literature. One of them performs query reformulation (or rewriting) of the query over the global schema into a query over the source schema(s). Another approach is to use the data in the sources and the mapping assertions to materialize a global instance, and to answer the query over the global schema using this instance. No matter how the global query is answered, the data can be provided either by a single source, or its answers can originate from multiple different sources. When multiple sources are used, the answers to the query should be obtained such that there are no clashes between them. That is why some approaches use constraints and make sure that these constraints are not violated, if the data comes from multiple sources. Constraints can also be used if only a single source is available, especially if a materialized global instance is built. These constraints are used in particular for making sure that there are no clashes between the data in the new materialized instance. Finally, as we have defined XML documents to be ordered labeled trees, we are interested in identifying which approaches respect the order of an XML tree, and which ones do not. By order in an XML tree, we consider the horizontal order, i.e. the next sibling order. As we will see, it is not uncommon that the approaches tend to disregard the sibling order, since in some settings, the horizontal navigation makes query answering intractable.

In the rest of this section, we proceed as follows. In Section 5.2, Section 5.2 and Section 5.2 we show which of the works can be expressed with LAV, GLAV and nested GLAV mapping assertions respectively. We describe these works and compare them along the criteria we outlined above. We rephrase examples and adapt them to fit in our XML GLAV mapping assertion formalism. Finally, in Section 5.3, we summarize the similarities and differences between the approaches.

LAV mapping assertions

In this section we overview several approaches that use LAV mapping assertions. In particular, we consider several works on XPath query rewriting using XPath views, as well as an XML data integration system based on LAV mapping assertions. An example of a LAV mapping assertion in the XML setting is presented in Example 5.2.1. An overview of the approaches that we will compare in this section is given in Table 5.1. In the second column of the table, XP is a shorthand notation for $\mathsf{XP}^{\{/,//,[],*\}}$, XP_1 for $\mathsf{XP}^{\{/,//,*\}}$, UXP_1 for union of $\mathsf{XP}^{\{/,//,*\}}$ and XP_2 for $\mathsf{XP}^{\{/,//,[]\}}$.

Example 5.2.1. This example is taken from [54]. In Figure 5.1 we show the LAV mapping assertion, consisting of a query over an intermediate view document on the left-hand side and the view definition on the right-hand side.

The output node of both the intermediate view and the view definition is x_b^{sub}. Once a compensation pattern (which will be defined below) has been computed using the view definition on the right-hand side of the LAV mapping assertion, it can be applied to the materialized intermediate view in order to obtain the answers to the query. △

XPath query rewriting using XPath views [7, 8, 33, 54] has received a lot of attention and has been an active topic of research recently. Most of the works focus on an XPath fragment,

Figure 5.1: *XML LAV mapping assertion based on $\mathsf{EP}^{\{/,//,[],*\}}$ in XPath query rewriting using views*

Table 5.1: *Works that use LAV mapping assertions*

work	problem addressed	mapping language	granularity	number of sources	interpretation of mappings	type of query answering	use of constraints	preservation of order
[7]	$\mathrm{ER}_{\mathcal{M}}^{\mathsf{XP},\mathsf{XP}}(q)$	$\mathsf{XP}^{\{/,//,[],*\}}$	subtrees	single	sound	rewriting	no	no
[8]	$\mathrm{ER}_{\mathcal{M}}^{\mathsf{XP}_1,\mathsf{UXP}_1}(q)$	$\mathsf{XP}^{\{/,//,*\}}$	subtrees	single	sound	rewriting	no	no
[33]	$\mathrm{MCR}_{\mathcal{M}}^{\mathsf{XP}_2,\mathsf{XP}_2}(q)$	$\mathsf{XP}^{\{/,//,[]\}}$	subtrees	multiple	sound	rewriting	yes,	yes
[41]	complexity of $\mathrm{CA}_{\mathcal{M}}^{ps}(q,p)$	ps-queries	subtrees	multiple	sound, exact, mixed	answering over a materialized target instance	yes, key constraints	no
[54]	$\mathrm{ER}_{\mathcal{M}}^{\mathsf{XP},\mathsf{XP}}(q)$,	$\mathsf{XP}^{\{/,//,[],*\}}$	subtrees	single	sound	rewriting	no	no

namely $\mathsf{XP}^{\{/,//,[],*\}}$, as well as its subfragments, all of them defined in Section 3.2. In particular, in [7, 54] the full fragment $\mathsf{XP}^{\{/,//,[],*\}}$ is considered. Additionally, [54] also considers the three sublanguages. [8] considers only the subfragment $\mathsf{XP}^{\{/,//,*\}}$, while [33] the subfragment $\mathsf{XP}^{\{/,//,[]\}}$. Another different approach that uses LAV mappings is the XML data integration described in [41]. The authors propose a system that uses LAV mappings and schemas (both source and global) expressed in terms of DTD documents. The XML data integration system also takes into account integrity constraints, and uses a node identity function in order to identify nodes coming from different sources, but carrying the same information. The language used for posing queries and expressing mapping assertions is the language of ps-queries.

Granularity of the Mapping Assertions The works on XPath query rewriting using XPath views consider the fragment $\mathsf{XP}^{\{/,//,[],*\}}$ and its subfragments, and they all agree on the type of mapping assertions w.r.t. the granularity. Namely, these approaches use mapping assertions that transfer subtrees, in particular a single tree. For describing these mapping assertions using the XML LAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ we proceed as follows. For each view

available for XPath rewriting, we have a single LAV mapping assertion. The LAV mapping assertion is characterized by the view that stores the data on the left-hand side and the view definition on the right-hand side, that provides the vocabulary for the queries. There is only one output variable x in both the materialized view and the view definition, which is bound to a subtree rooted at the single output node, and hence it is annotated by x^{sub}. Example 5.2.1 shows the translated version of the XPath query rewriting using views into our XML LAV mapping formalism.

The XML data integration system from [41], defined as the triple $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, uses LAV mapping assertions. In $\mathcal{I}$, $\mathcal{G}$ stands for the global schema, $\mathcal{S}$ for the set of source schemas and $\mathcal{M}$ for the set of mapping assertions. Each mapping assertion in $\mathcal{M}$ is given as (S_i, M_i, as_i), where S_i is a source schema, M_i is a ps-query that is used to describe the data stored in the source that conforms to the schema S_i, and as_i is used for specifying the semantics of the mapping assertion, i.e. whether it is sound or exact. In this work, the global schema is a generalization of the source schemas, which yields that the ps-query over a source is also a query over the global schema. Example 5.2.2 describes the LAV mapping assertions based on ps-queries.

Example 5.2.2. Consider the XML data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ given with the following source ($\mathcal{S} = \{S_1, S_2\}$) and global ($\mathcal{G} = \langle S_\mathcal{G}, \Phi_K, \Phi_{FK} \rangle$, where Φ_K and Φ_{FK} are sets of keys and foreign keys respectively) schemas:

S_1:

$$\begin{aligned} hospital &\rightarrow patient^* \\ patient &\rightarrow name, SSN \\ attlist(name) &= \{@value\} \\ attlist(SSN) &= \{@value\} \end{aligned}$$

S_2:

$$\begin{aligned} hospital &\rightarrow patient^* \\ patient &\rightarrow SSN \\ attlist(SSN) &= \{@value\} \end{aligned}$$

$S_\mathcal{G}$:

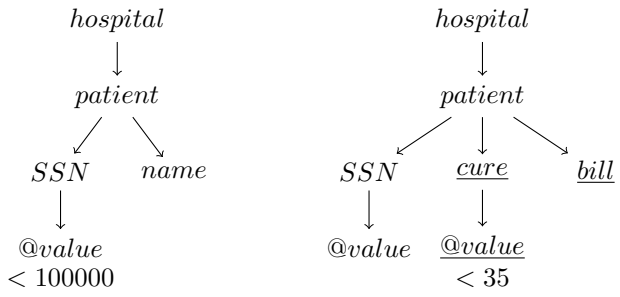
$$\begin{aligned} hospital &\rightarrow patient^+, treatment^+ \\ patient &\rightarrow SSN, name, cure^*, bill? \\ treatment &\rightarrow trID, procedure? \\ procedure &\rightarrow treatment^+ \\ attlist(name) &= \{@value\} \\ attlist(SSN) &= \{@value\} \\ attlist(cure) &= \{@value\} \\ attlist(trID) &= \{@value\} \end{aligned}$$

$$\begin{aligned} \Phi_K &: \{patient.SSN.@value \rightarrow patient; \\ &\quad\ treatment.trID.@value \rightarrow treatment\} \\ \Phi_{FK} &: \{patient.cure.@value \subseteq treatment.trID.@value\} \end{aligned}$$

The mapping $\mathcal{M}$ is a set of triples of the form (S_i, M_i, as_i), $i \in \{1, 2\}$, where M_1 and M_2 are shown in Figure 5.2 and $as_i \in \{sound, exact\}$.

Note that the underlined nodes in the ps-query M_2 in Figure 5.2b represent existential subtree patterns in the query. △

Interpretation of the Mapping Assertions The semantics of the views in the works on XPath query rewriting using XPath views is defined under the OWA. In other words, the mapping assertions are interpreted as sound. On the other hand, the mappings in the data integration system from [41] can have three different interpretations: (1) sound, if all the mapping assertions are interpreted as sound; (2) exact, if all the mapping assertions are interpreted as exact; and (3) mixed, if some of the mapping assertions are interpreted as sound, and others as exact.



(a) M_1 **(b)** M_2

Figure 5.2: *Ps-queries that are a part of the mapping assertions from* $\mathcal{M}$

Type of Query Answering As the name suggests, the works on XPath query rewriting using XPath views perform query answering by reformulation. Given a view specified in XPath, the goal of the rewriting algorithms is to compute another XPath expression, called *compensation pattern*, which combined with the view, provides the same answers as the query. When speaking about query rewriting, it is also interesting to identify whether the query rewriting algorithm produces an equivalent or a maximally contained rewriting. In [7, 54] the problem of whether an equivalent rewriting exists for a query using views in the respective languages is considered. Moreover, in [7, 8, 54], the computation of an equivalent rewriting is addressed. The equivalent rewriting produced in [54] is also a minimal one, in the sense that it is obtained by a minimization algorithm proposed by the authors. A union rewriting based on the union of the views is produced in [8]. The problem of computing a maximally contained rewriting is tackled in [33].

A data integration system, as defined for the relational case, should also perform query answering by query reformulation. However, for the XML data integration system defined in [41], this is not the case. The query answering in this data integration system is performed by materializing a legal global XML tree from the data residing in the source trees. Note that this XML data integration system captures a very restricted setting, since it assumes that the source schemas are sub-schemas of the global schema. This is because the query answering algorithm first expands the sources such that they conform to the global schema and then merges them in a single materialized global instance.

Number of Sources Earlier works on XPath rewriting using XPath views address rewritings using a single XPath view [7, 54]. More recent ones, however, try to extend the rewriting setting and consider multiple (materialized) views [8, 33].

The data integration system from [41] works with multiple sources. In particular, the data integration system considers a set of DTDs as a source schema. The actual data is stored in XML trees, such that each XML tree conforms to some source DTD.

Use of Constraints The only work on XPath query rewriting using XPath views that we consider in this section and that takes constraints into account is [33]. The constraints are imposed

by a schema specified in a form of a DTD. Two flavors of query rewriting are considered: one when the schema constraints are not taken into account, and one with the presence of schema information. As schemas in the latter case, non-recursive DTDs are used.

The data integration system in [41] takes into account integrity constraints, namely unary key and foreign key constraints, as discussed in Example 5.2.2. The sets of unary and foreign key constrains are a part of the global schema together with the global DTD. Although [41] defines a data integration system, the queries are answered over a materialized global instance. Thus, the set of constraints from the global schema is later used for defining a node identity for each node from the global instance in terms of an identification function. This function is used to identify nodes from the source documents in such a way that nodes having same values for their keys correspond to the same node in a global sense. The identification of the nodes starts by first expanding the original data sources with new nodes, such that they conform to the global schema. Skolem constants are used as values for nodes if a node that has to exist in the global schema does not exist in the source schema. The identification function is then applied to such extended sources. The identifier of each node is a concatenation of the identifier of its parent, its node name and an optional value, which is either the value of its key (if one exists) or a fresh Skolem constant. The root of the source document has an empty identifier.

Preservation of Order Given all the approaches that are expressible using LAV mapping assertions that we have considered so far, the only one that respects the order of the elements is [33]. They work with ordered trees, rather than [7, 8, 41, 54] who drop the order of the XML document even in the definition of an XML tree.

Description of the Query Answering Algorithms Here we give a more detailed overview of the query answering algorithms described in the works that are based on LAV mapping assertions. As mentioned before, the works on XPath query rewriting using XPath views follow a similar approach. They are based on computing a contained or equivalent rewriting of a pattern p using a view v by generating a compensation pattern p'. A compensation pattern p' is a pattern which composed with a view v, produces a rewriting pattern r. The composition is performed by merging of the two patterns. The merged pattern is obtained by collapsing the root node of the compensation pattern with the output node of the view. Then, in order to check if r is a contained or equivalent rewriting, a containment or equivalence check is applied respectively. Consider again the translation presented in Example 5.2.1. We have seen that when a query is posed, the goal of the algorithms is to compute a compensation pattern whose root is labeled as the label of the output node of the view definition. When such a pattern is obtained (by some algorithm for XPath rewriting), it can also be applied to the output node of the materialized view, as they are equal. Thus, the answers to the query are obtained from the materialized view on the left-hand side of the LAV mapping assertion. The XML data integration system from [41] performs query answering by materializing a global tree using data stored at the sources. In the following, we give details on some features of the query answering algorithms in the works we consider in this section.

[7] The query answering algorithm in [7] is based on computing a compensation pattern. The compensation pattern that yields an equivalent rewriting w.r.t. the views is chosen from

a set of the natural rewriting candidates. The notion of natural rewriting candidates is defined given an $\mathsf{XP}^{\{/,//,[],*\}}$ expression p and a view v. There are two natural candidates w.r.t. p and v. The first natural candidate is the expression $p^{\geq k}$, which is a subexpression of p that contains nodes of depth greater or equal to k, where k is the depth of the output node of v. The second natural candidate is obtained from $p^{\geq k}$ by forcing all the outgoing edges from the root to be descendant edges. The authors also define the notion of potential rewriting and show completeness of the approach, in the sense that if the potential rewriting is not a rewriting, then one does not exist. Moreover, they give conditions under which one of the two natural candidates is a potential rewriting.

[8] This work proposes an algorithm for producing an equivalent union rewriting based on the union of multiple views. Given an expression p and a set $\mathcal{V}$ of views, all in $\mathsf{XP}^{\{/,//,*\}}$, a union rewriting of p using $\mathcal{V}$ is a set $\mathcal{R}$ of $\mathsf{XP}^{\{/,//,*\}}$ expressions such that for each $r \in \mathcal{R}$ there is a subset $\mathcal{V}_r$ of $\mathcal{V}$ such that for all trees $T \in \mathcal{T}$ it holds that $\bigcup_{r \in \mathcal{R}} r(\mathcal{V}_r(T)) = p(T)$. The motivation behind using unions of view expressions is that there might not exist a rewriting using a single view from a set of multiple views, but if these views are united, a rewriting can be obtained. The proposed rewriting algorithm is sound, but not complete.

[33] The existence of a maximally contained rewriting is tackled in [33]. Two different algorithms for computing a maximally contained rewriting are proposed: one that computes a rewriting without a schema, and another that computes a rewriting in the presence of a schema. For the former case, the algorithm for generating a maximally contained rewriting (which is a union of contained rewritings) runs in exponential time w.r.t. the size of the query in the worst case. In the presence of a schema, it is possible to infer a set of constraints Σ in time polynomial in the size of the schema, and to then apply the chase procedure to the view expression v such that it satisfies the constraints in Σ. The query expression does not need to be chased since the trees that it is posed over already satisfy the schema, and therefore the inferred constraints. The proposed algorithm for checking the existence of the maximally contained rewriting also returns a rewriting if one exists and runs in polynomial time.

[41] In this work, the authors provide three algorithms for query answering and analyze their complexity. The three different flavors of the query answering algorithm come from the different interpretations of the mappings, which can be sound, exact or mixed, as discussed above. The notion of certain answers of a query is defined as an intersection of the answers of the query over the trees in the semantics of the data integration system.

The semantics of the data integration systems is a set of trees that satisfy the global schema, and additionally, depending on the type of the mapping, have to fulfill the following condition for all data sources:

- the data source is a prefix of the answer of the mapping query over the tree if the mapping assertion is interpreted as sound;
- the data source is isomorphic to the answer of the mapping query over the tree if the mapping assertion is interpreted as exact.

The trees in the semantics of the XML data integration system are the legal trees, and their representation as a weak representation system lies in the core of the query answering algorithms. Once the weak representation system is known, the goal is to find a representation for the answer of a query over it. The representation for the legal trees and the representation for the answer have to satisfy the following condition: the intersection of the trees in the semantics of the latter has to be equal to the intersection of the answers of the queries over the trees in the semantics of the former. The complexity of the algorithms comes from the complexity of computing the representation for the legal trees, the complexity of finding the representation for the answers over the legal trees and from the complexity of the intersection. When using sound mappings, the authors show that the query answering algorithm is PTIME w.r.t. data complexity (i.e. the size of the set of data sources). On the other hand, when using exact and mixed mappings, the algorithm is NP-complete w.r.t. data complexity.

[54] This work deals with the problem of checking the existence of an equivalent rewriting, as well as the problem of computing this rewriting. Moreover, during the computation of the equivalent rewriting, a minimization technique is applied, making the final result of the rewriting algorithm a minimal equivalent rewriting. For the rewriting existence problem, coNP-hardness is shown, and an upper bound of Σ_3^p for the computation of minimal rewritings is asserted. However, the proof of the upper bound used results that have been refuted in [31]. For the three fragments $\mathsf{XP}^{\{/,//,[]\}}$, $\mathsf{XP}^{\{/,[],*\}}$, $\mathsf{XP}^{\{/,//,*\}}$, it is shown that the problem of existence of a rewriting and computing a minimal one are in P.

Conclusion In this section, we have overviewed approaches that use the expressive power of the XML LAV mapping assertions. We have seen that the works on XPath query rewriting using XPath views use the LAV mapping assertions based on extended tree patterns, while the data integration system from [41] uses the LAV mapping assertions based on ps-queries. Upon identifying the different works on XPath query rewriting using XPath views, we can see that although there exist many different rewriting algorithms, they are based on the same idea of computing a compensation pattern. In Example 5.2.1, we showed how to translate the setting of XPath query answering using XPath views in an XML LAV mapping assertion based on extended tree patterns. On the left-hand side of this assertion we used a simple view schema, and on the right-hand side we have the view definition. Also, we have seen a data integration system that uses XML LAV mapping assertions based on ps-queries. Although this data integration system is formalized in a similar way as a data integration system for relational databases, in essence it does not perform query reformulation, but rather answers global queries by materializing a global instance and computing certain answers over such an instance.

GLAV Mapping Assertions

In this section we consider works that utilize the expressive power of our XML GLAV mapping assertions. We give insights on one approach that addresses XML data integration [56] and two works on XML data exchange [9, 12]. A summary on the features of the works we are going to overview in this section is given in Table 5.2. In the second column, CQ denotes the core query

language (fragment of XQuery) used as a query language in [56]. Example 5.2.3 shows what kind of mapping assertions we will be dealing with in the remainder of this section.

Table 5.2: *Works that use GLAV mapping assertions*

work	problem addressed	mapping language	granularity	number of sources	interpretation of mappings	type of query answering	use of constraints	preservation of order
[9]	complexity of $\mathrm{CA}_{\mathcal{M}}^{tuple}(q, \bar{t})$	SM($\Downarrow, \Rightarrow, \mathsf{Fun}, \sim$) and subclasses	scalar values	single	sound	answering over a materialized target instance	yes, schema	yes
[12]	complexity of $\mathrm{CA}_{\mathcal{M}}^{tuple}(q, \bar{t})$	SM($\Downarrow$)	scalar values	single	sound	answering over a materialized target instance	yes, schema	yes
[56]	$\mathrm{ER}_{\mathcal{M}}^{CQ,CQ}(q)$	SM($\downarrow, =$)	scalar values	multiple	sound	rewriting	yes, key constraints	no

Example 5.2.3. We take the following example of a GLAV mapping assertion from [12]. Consider the following source and target schema.

Source DTD

$$\begin{aligned} db &\rightarrow book^* \\ book &\rightarrow author^* \\ attlist(book) &= \{@title\} \\ attlist(author) &= \{@name, @aff\} \end{aligned}$$

Target DTD

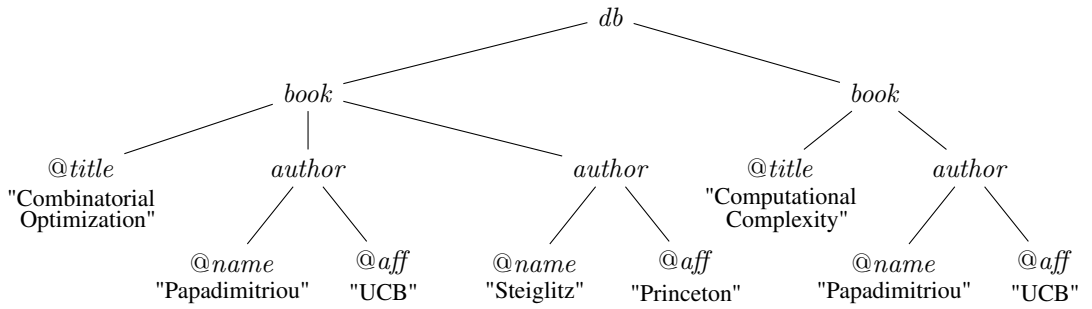
$$\begin{aligned} bib &\rightarrow writer^* \\ writer &\rightarrow work^* \\ attlist(writer) &= \{@name\} \\ attlist(work) &= \{@title, @year\} \end{aligned}$$

We can specify the following GLAV mapping assertion, with the output variables x_1^{val} and x_2^{val}:

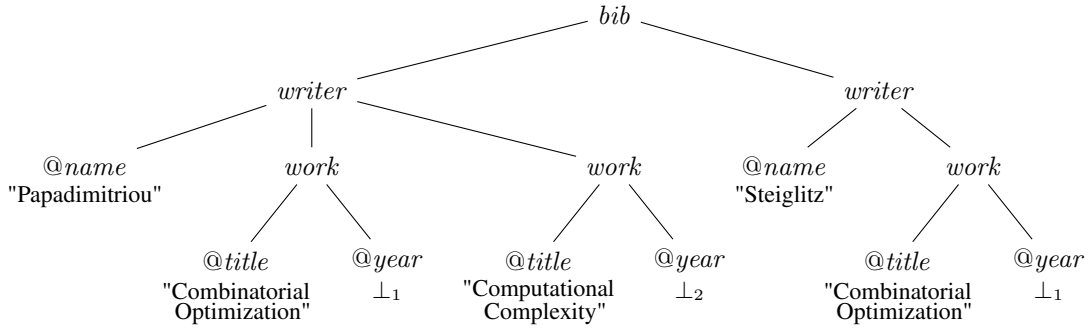
$$\begin{aligned} db[book[x_1^{val} : @title, author[x_2^{val} : @name]]] \rightarrow \\ \exists y_1^{val}\ bib[writer[x_2^{val} : @name, work[x_1^{val} : @title, y_1^{val} : @year]]] \end{aligned}$$

This GLAV mapping assertion is extracting tuples of book titles and author names from the source XML tree and transforms them into work titles of writer and writer names respectively. Given the source XML tree in Figure 5.3a and the mapping assertion, the target XML tree in Figure 5.3b is obtained.

Note that in the target XML tree we have occurrences of null values, denoted by $\perp_1$ and $\perp_2$. △



(a) *Source XML tree*



(b) *Target XML tree*

Figure 5.3: *Source XML tree and the obtained target XML tree using the XML GLAV mapping assertion*

In [56], the authors propose an algorithm for XML query rewriting of queries posed over a global schema, provided that the mapping assertions between the global schema and the sources are previously known, and provided that the data is stored at the sources. The query rewriting is based on the mapping assertions, which despite that their syntax resembles XQuery, can easily be translated into XML GLAV mapping assertions. The query language used for posing queries over the data integration system is called *core query language (CQ)* and is a fragment of XQuery that allows nested subqueries. The output of the query can either be a tuple of label-value pairs, or a nested structure. The CQ language also supports the use of Skolem function symbols, but does not support the descendant relation. The descendant relation is not allowed in the schema mappings either. For the schemas, a nested-relational representation is used, which can describe both relational and XML schemas.

In [9, 12], a framework that performs transformation of an XML document that conforms to one DTD into an XML document that conforms to a different DTD is proposed. An XML data exchange setting is defined as a triple $\langle D_{\mathbf{S}}, D_{\mathbf{S}}, \Sigma_{\mathbf{ST}} \rangle$, where $D_{\mathbf{S}}$ is a source DTD, $D_{\mathbf{T}}$ is a target DTD and $\Sigma_{\mathbf{ST}}$ is a set of source-to-target dependencies that specify the correspondence between $D_{\mathbf{S}}$ and $D_{\mathbf{T}}$. Target dependencies are not taken into account. Given an XML data exchange

setting $\langle D_{\mathbf{S}}, D_{\mathbf{S}}, \Sigma_{\mathbf{ST}} \rangle$ and an XML tree S conforming to the source DTD $D_{\mathbf{S}}$, a solution for S is an XML tree T that conforms to the target DTD $D_{\mathbf{T}}$, such that S and T together satisfy all the source-to-target dependencies in $\Sigma_{\mathbf{ST}}$. As a query language for posing queries over the target data in [9, 12], conjunctive tree queries, denoted by $\mathcal{CTQ}$, and unions of conjunctive tree queries, denoted by $\mathcal{UCTQ}$ as defined in Section 3.2, are used.

Granularity of the Mapping Assertions Although syntactically, the mapping assertions in [56] and the source-to-target dependencies from [9, 12] differ, in fact they coincide in terms of granularity. We can see the source-to-target dependencies as GLAV mapping assertions, and thus in all of the three approaches, the mapping assertions transfer tuples of values. In [56], the mapping assertions are expressed in a language that does not support descendants or wildcards, and can be easily translated into XML GLAV mapping assertions. The source-to-target dependencies from [9, 12] are expressions of the form $\psi_{\mathbf{T}}(\overline{x}, \overline{z})$:- $\varphi_{\mathbf{S}}(\overline{x}, \overline{y})$ where $\varphi_{\mathbf{S}}$ and $\psi_{\mathbf{T}}$ are tree pattern formulae over the source and target schema respectively. In [12], a restricted version of the tree pattern formulae is used for specifying the dependencies. These tree pattern formulae allow wildcards and vertical navigation (including descendants), but they do not support horizontal navigation. Amano et al. [9] present schema mappings that are an extension of the ones defined in [12]. Their aim is to define more expressive languages for specifying XML mapping assertions, which will support ideally all structural characteristics of XML documents, such as horizontal and vertical navigation, use of descendants and wildcards, and allow techniques which are extensively used in relational schema mappings, such as joins of variables. The language for schema mappings presented in [9] coincides with the tree pattern formulae we introduced in Section 3.2. Additionally, the authors in [9] propose a classification of the schema mappings which then allows for a study of their complexity. The most general class of schema mappings discussed is SM($\Downarrow, \Rightarrow, \sim, \mathsf{Fun}$). The class SM($\Downarrow$) corresponds to the schema mappings described in [12]. As this mapping language from [56] transfers atomic values, we will use the same notation to refer to it. Hence, the schema mappings from [56] are in the class SM($\downarrow, =$).

Using our XML GLAV mapping formalism, we can straightforwardly describe the GLAV mappings used in [56]. The mapping assertions used in this work describe which data values from the source instances correspond to which elements of the global schema. The query rewriting consists of reformulating the query over the global schema into a query over the sources using the mapping assertions. Example 5.2.4 shows the mapping assertions considered in this work, as well as their translation into our XML GLAV mapping formalism.

Example 5.2.4. As an example of a data integration system, the authors consider a setting with two sources and one global schema. The DTDs corresponding to the schemas are as follows:

src_1 :

$$
\begin{array}{rcl}
src_1 & \rightarrow & students \\
students & \rightarrow & student^* \\
attlist(student) & = & \{@studentID, @name, @course, @grade\}
\end{array}
$$

src_2 :

$$
\begin{array}{rcl}
src_2 & \rightarrow & students, courseEvals \\
students & \rightarrow & student^* \\
attlist(student) & = & \{@studentID, @name, @courseID\} \\
courseEvals & \rightarrow & courseEval^* \\
attlist(courseEval) & = & \{@courseID, @course, @file\}
\end{array}
$$

tgt :

$$
\begin{array}{rcl}
tgt & \rightarrow & students, evals \\
students & \rightarrow & student^* \\
student & \rightarrow & courses \\
attlist(student) & = & \{@studentID, @name\} \\
courses & \rightarrow & courseInfo^* \\
attlist(courseInfo) & = & \{@course, @evalID\} \\
evals & \rightarrow & eval^* \\
attlist(eval) & = & \{@evalID, @grade, @file\}
\end{array}
$$

Given the two source documents, two mapping assertions are defined:

M_1 **foreach** s **in** $src_1.students$
exists s' **in** $tgt.students, c'$ **in** $s'.students.courses, e'$ **in** $tgt.evals$
where $c'.courseInfo.@evalID = e'.eval.@evalID$
with $s'.student.@studentID = s.@studentID$ **and** $s'.student.@name = s.@name$
and $c'.courseInfo.@course = s.@course$ **and** $e'.eval.@grade = s.@grade$

M_2 **foreach** s **in** $src_2.students.student, c$ **in** $src_2.courseEvals.courseEval$
where $s.@courseID = c.@courseID$
exists s' **in** $tgt.students, c'$ **in** $s'.students.courses, e'$ **in** $tgt.evals$
where $c'.courseInfo.@evalID = e'.eval.@evalID$
with $s'.student.@studentID = s.@studentID$ **and** $s'.student.@name = s.@name$
and $c'.courseInfo.@course = c.@course$ **and** $e'.eval.@file = c.@file$

Although these mapping assertions use a different syntax, essentially they are mapping assertions from SM$(\downarrow, =)$. The **foreach** and the **exists** parts of such a mapping assertion correspond to the left- and right-hand side of an XML GLAV mapping assertion based on $\mathsf{EP}^{\{/,//,[],*\}}$. The **where** and **with** parts give a more precise definition of which variables should be existentially and universally quantified, respectively. The universally quantified variables are those whose values are transferred by the mapping assertion, while the existentially quantified ones are used for expressing join conditions. Hence, M_1 and M_2 can be translated into XML GLAV mapping

assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ as follows:

$$\begin{aligned}
\mathrm{M}_1': \quad & src_1[students[student[x_1{}^{val} : @studentID, x_2{}^{val} : @name, x_3^{val} : @course, x_4^{val}: @grade]]] \\
& \rightarrow \ \exists y_1^{val}, y_2^{val}\ tgt[students[student[x_1^{val} : @studentID, x_2^{val} : @name, \\
& \qquad\qquad courses[courseInfo[x_3^{val} : @course, y_1^{val}: @evalID]]]]] \\
& \qquad evals[eval[y_1^{val} : @evalID, x_4^{val} : @grade, y_2^{val}: @file]]
\end{aligned}$$

$$\begin{aligned}
\mathrm{M}_2': \quad & src_2[students[student[x_1^{val} : @studentID, x_2^{val} : @name, x_3^{val} : @courseID]], \\
& \quad courseEvals[courseEval[x_3^{val} : @courseID, x_4^{val} : @course, x_5^{val} : @file]]] \\
& \rightarrow \ \exists y_1^{val}, y_2^{val}\ tgt[students[student[x_1^{val} : @studentID, x_2^{val} : @name, \\
& \qquad\qquad courses[courseInfo[x_4^{val} : @course, y_1^{val} : @evalID]]]]] \\
& \qquad evals[eval[y_1^{val} : @evalID, y_2^{val} : @grade, x_5^{val} : @file]]
\end{aligned}$$

The graphical representation of the split XML GLAV mapping assertions can be seen in Figure 5.4 and 5.5. We only show the mapping assertion M_1', as the other one is analogous. $\triangle$

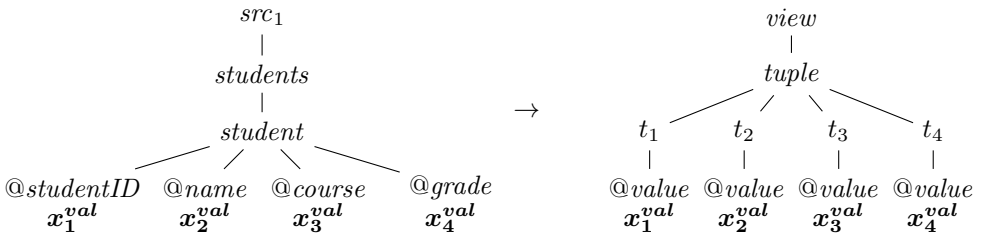


Figure 5.4: *GAV part of the XML GLAV mapping* M_1'

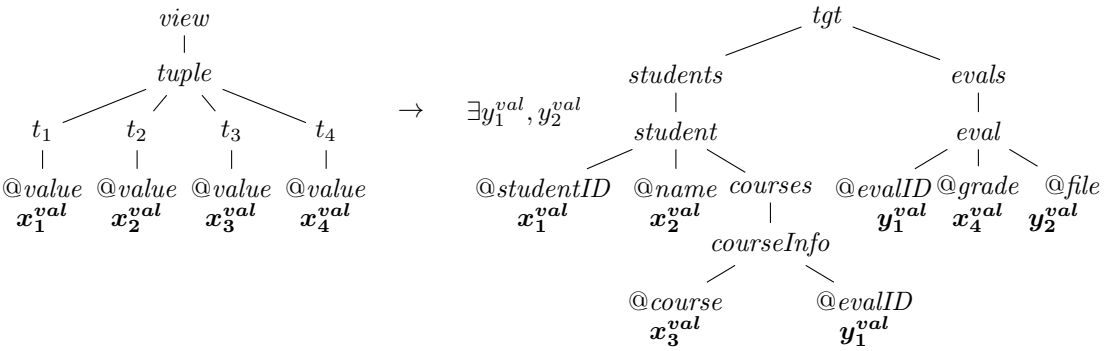


Figure 5.5: *LAV part of the XML GLAV mapping* M_1'

We can apply our XML GLAV mapping formalism to represent the dependencies in the XML data exchange setting as well. As we have seen, [9] uses a more expressive language for specifying the source-to-target dependencies. This language supports horizontal navigation,

which is not supported by the extended tree patterns that we have defined in Section 3.2. Thus, a source-to-target dependency $\psi_{\mathbf{T}}(\overline{x}, \overline{z})$:- $\varphi_{\mathbf{S}}(\overline{x}, \overline{y})$ that does not use horizontal navigation can be seen as an XML GLAV mapping assertion based on $\mathsf{EP}^{\{/,//,[],*\}}$:

$$\forall \overline{x} \, (\exists \overline{y} \, \varphi'_{\mathbf{S}}(\overline{x}, \overline{y}) \rightarrow \exists \overline{z} \, \psi'_{\mathbf{T}}(\overline{x}, \overline{z}))$$

where $\varphi'_{\mathbf{S}}$ and $\psi'_{\mathbf{T}}$ are the translations to extended tree patterns of the tree pattern formulae $\varphi_{\mathbf{S}}$ and $\psi_{\mathbf{T}}$ respectively. The two extended tree patterns have a tuple of output variables $\overline{x}$, each one corresponding to an attribute node, and bound to its value. We describe the translation in more detail in Example 5.2.5.

Example 5.2.5. We take an example schema mapping assertion from [9] that does not use horizontal navigation. Consider the following source DTD D_1 and target DTD D_2:

$$\begin{array}{llcl} D_1: & europe & \rightarrow & country^* \\ & country & \rightarrow & ruler^* \\ & attlist(country) & = & \{@name\} \\ & attlist(ruler) & = & \{@name\} \end{array} \qquad \begin{array}{llcl} D_2: & europe & \rightarrow & succession^* \\ & succession & \rightarrow & country, ruler, \\ & & & successor \\ & attlist(country) & = & \{@name\} \\ & attlist(ruler) & = & \{@name\} \\ & attlist(successor) & = & \{@name\} \end{array}$$

For the given DTDs, the following source-to-target dependency can be specified (in the language of tree pattern formulae):

$$europe[country(x_c)[ruler(x_r)]] \rightarrow \exists y_s \; europe[succession[country(x_c), ruler(x_r), successor(y_s)]]$$

If we translate this source-to-target dependency that uses tree pattern formulae into a XML GLAV mapping assertion based on extended tree patterns, we obtain the following:

$$europe[country[x_1^{val} : @name, ruler[x_2^{val} : @name]]] \rightarrow$$
$$\exists y_1^{val} \; europe[succession[country[x_1^{val} : @name, ruler[x_2^{val} : @name], successor[y_1^{val} : @name]]]]$$

This XML GLAV mapping assertion can be further split into two GAV and LAV mapping assertions, depicted in Figure 5.6 and 5.7.

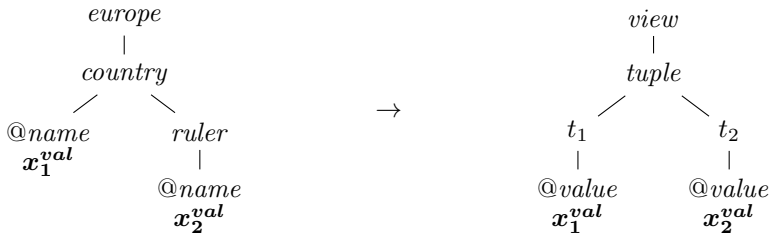


Figure 5.6: *GAV part of the XML GLAV mapping*

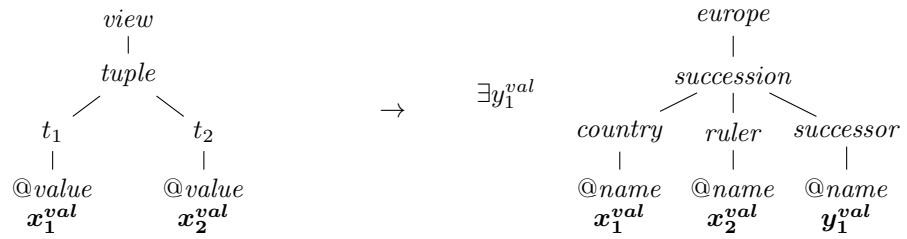


Figure 5.7: *LAV part of the XML GLAV mapping*

The extended tree patterns used in the XML GLAV mapping assertion have two output variables, bound to the values of the attributes $@name$ of $country$ (x_1^{val}) and $@name$ of $ruler$ (x_2^{val}). The variable y_1^{val} is existentially quantified and is used to represent the attribute $@name$ of $successor$. △

Interpretation of the Mapping Assertions The mapping assertions in the three works we consider in this section are interpreted as sound. Although this is not explicitly stated, the soundness of the mapping assertions in [9, 12] follows from their definition, i.e. from the form of the source-to-target dependencies. The mapping assertions in [56] are also sound due to the fact that they specify the relationship between the data in the sources and the global schema only in one direction, from the sources to the global schema.

Type of Query Answering In contrast to the XML data integration system from [41] we discussed in the previous section, the XML data integration system from [56] performs query answering by rewriting the query over the global schema into a query over the sources (which coincides with the definition of a data integration system in the case of relational databases). The XML data exchange settings proposed by [9, 12] perform query answering over a materialized target instance. Certain answers are defined as in the relational case: the set of certain answers for a $\mathcal{UCTQ}$ query Q w.r.t. the XML tree S conforming to the source DTD $D_{\mathbf{S}}$ is the intersection of the answers of Q over all possible solutions T for S.

Number of Sources The XML data integration system in [56] supports multiple source documents. The sources are specified using nested relational schemas, which provides support for both XML and relational sources. On the other hand, the XML data exchange settings from [9, 12] take into account a single source, which they use to build a materialized target instance.

Use of Constraints All of the three approaches that we overview in this section take into account constraints when addressing the respective problems. We can go into further detail and identify the type of constraints used in each of them. In [9, 12], the constraints come from the definition of the structure of the documents, i.e. from the DTDs that represent the source and the target schema. In [56], as a part of the mapping between the source and the global schema, the authors take constraints on the global schema into account. They refer to these constraints

as target constraints, and they are in the form of key constraints. The target constraints allow expression of data merging rules. A need for data merging arises when the data from different sources coincide.

Preservation of Order The XML data integration system from [56] does not respect the order of the document when performing query rewriting. In [9, 12], the source document is defined as an ordered labeled tree. Moreover, [9] allows horizontal navigation through the document, while [12] does not. However, in the restricted tractable case of query answering in [12], the order is not preserved. Also, in [9] it has been shown that horizontal navigation, even only next-sibling navigation, makes query answering intractable. Hence, in the XML data exchange settings, the tractability of query answering can be achieved if the document order is not taken into account.

Description of the Query Answering Algorithms As mentioned earlier, the XML data exchange settings perform $(\mathcal{U})\mathcal{CTQ}$ query answering using a materialized target instance. On the other hand, the XML data integration system performs query rewriting of a query expressed in an XQuery fragment using a set of mapping assertions. Here we outline more details on the query answering algorithms used in the XML data integration system and the two XML data exchange settings based on GLAV mapping assertions.

[12] In [12], the authors show that the upper bound for answering $\mathcal{UCTQ}$s is coNP. However, a restricted setting of query answering is tractable. The restrictions are imposed on the source-to-target dependencies and on the DTDs. In particular, the source-to-target dependencies are restricted to be fully-specified, i.e. the target pattern of the dependency has to start at the root of the target schema and must not include descendants and wildcards. The other restriction is that the target DTD contains univocal regular expressions. The precise definition of univocal regular expressions can be found in [12]. It is important to note that DTDs that contain univocal regular expressions extend nested-relational DTDs. Thus, given this restricted setting, it has been proved that computing the certain answers for a $\mathcal{UCTQ}$ can be done in PTIME.

[9] This work extends the query answering setting presented in [12]. Their aim is to get a tractable result by examining fragments of the mapping specification and the query language, while not breaking the coNP upper bound. They show that for a query in $\mathcal{UCTQ}(\Downarrow, =)$ and a schema mapping in SM$(\Downarrow, \Rightarrow, \sim, \mathsf{Fun})$ which is fully specified, non-bounding and $\sim$-monotonic, the problem of checking whether a tuple $\bar{t}$ is in the certain answers of Q is in PTIME. They also show that the upper bound remains coNP when the schema mapping is from the class SM$(\Downarrow, \Rightarrow, \sim, \mathsf{Fun})$ and the query is from the class $\mathcal{UCTQ}(\Downarrow, \Rightarrow, =)$. Additionally, they study the consistency of the extended data integration setting, as well as the complexity of composition of schema mappings, which is a usual operation in the relational case.

[56] Two original algorithms for query rewriting are introduced in [56]. The first algorithm does not consider constraints, while the second one does. The first rewriting algorithm presented in this article is called the *basic query rewriting algorithm*. It consists of four

phases: rule generation, query translation, source query optimization and query assembly. The first phase transforms the mapping assertions into mapping rules which are needed in the query translation phase. In the second phase, the target query is translated into a set of source queries, where the queries containing a subquery in the **return** part are decorrelated (i.e. the subquery is replaced by a Skolem term and translated separately). This set of source queries is then optimized in the third phase. Finally, in the query assembly phase, the queries that were decorrelated in the second phase are reassembled, by replacing the Skolem terms in the **return** by the corresponding translation of the subqueries. The authors show that the rewritings obtained with this algorithms produce answers that correspond to the answers of the original target query obtained over a materialized target schema, i.e. the query rewriting algorithm is sound and complete. Moreover, they show that the number of rewritings generated in the second phase is $\mathcal{O}(n^k)$, where n is the number of mappings and k is the size of the query.

The second algorithm is an extension of the first one by including *query resolution*, which handles the target constraints. When taking the target constraints into account, completeness is lost, i.e. it may happen that some answers of the original query over a materialized target schema might not be obtained as an answer of the rewritten queries over the sources. The query resolution algorithm first translates the target constraints using the translation algorithm for the queries, used in the second phase of the basic query rewriting algorithm. Also, minimization is applied to the translated target constraints. The result is a set of source constraints which introduce new equalities between Skolem terms. These constraints are used to generate new rewritings in which the equalities between Skolem terms are resolved. This is called a *resolution step*. The rewriting of a query by taking the target constraints into account is obtained by exhaustively applying resolution steps. Termination of the resolution steps is guaranteed by imposing acyclicity on the target constraints. The authors show that the extended query rewriting algorithm that handles the target constraints is sound, but not complete.

Conclusion The approaches that we have compared in this section use mapping assertions that can be translated into our XML GLAV mapping formalism based on $\mathsf{EP}^{\{/,//,[],*\}}$. We have seen two works on XML data exchange and one on XML data integration. The former focus on computing certain answers for XML queries returning tuples, over a materialized target instance built using the source XML tree and the mapping assertions. The latter computes an equivalent rewriting of a query in a language that is subsumed by XQuery. Two novel query rewriting algorithms are proposed, one of which takes into account constraints on the global schema. The granularity of the mapping assertions in all the three approaches is the same, although they differ in the syntax.

Nested GLAV Mapping Assertions

In this section we give an overview of two works [37, 40] on XQuery query rewriting using XQuery views. As query language in [37], a fragment of XQuery is used. This fragment allows returning (i.e. storing) information (such as the value, content or identifier) from several vari-

ables, and value joins, but allows neither nested queries nor the specification of nested views. In [40], query rewriting is tackled for both the full XQuery language, as well as a fragment of XQuery that includes nesting, duplicate elimination, existential quantification and equalities between values and identifiers. This sublanguage, referred to as OptXQuery, has first been introduced in [20]. Table 5.3 summarizes the features of these two works. In the second column of the table, XQ is a shorthand notation for the XQuery fragment used in [37], and OptXQ for the OptXQuery.

Table 5.3: *Works that use nested GLAV mapping assertions*

work	problem addressed	mapping language	granularity	number of sources	interpretation of mappings	type of query answering	use of constraints	preservation of order
[37]	$\mathrm{ER}_{\mathcal{M}}^{\mathsf{XQ},\mathsf{XQ}}(q)$	expressive fragment of XQuery	subtrees that can be modified	multiple	sound	rewriting	no	no
[40]	$\mathrm{ER}_{\mathcal{M}}^{\mathsf{OptXQ},\mathsf{OptXQ}}(q)$	OptXQuery and full XQuery	subtrees that can be modified	multiple	exact	rewriting	no	yes

Granularity of the Mapping Assertions The XQuery view specifications used in both works require the expressive power of the nested GLAV mapping assertions. Based on granularity, these mapping assertions transfer subtrees and scalar values and are able to modify the subtrees by renaming the elements and adding new ones. Since our XML GLAV mapping formalism is unable to express modification of the subtrees, we are unable to represent the nested GLAV mapping assertions, and hence the XQuery views using it.

Interpretation of the Mapping Assertions The XQuery views in [37] are interpreted under the sound semantics. On the contrary, in [40], the views are interpreted under the exact semantics.

Type of Query Answering The two works that we summarize here perform query answering by rewriting an XQuery query using a set of XQuery views. They propose original rewriting algorithms, which will be explained in more detail in the rest of the section.

Number of Sources Both approaches on XQuery query rewriting using XQuery views allow multiple views to be taken into account by the respective rewriting algorithms. Moreover, [37] addresses the problem of computing minimal rewritings in the sense that the authors are interested in computing the rewriting that uses the minimal number of views, and is equivalent to the original query.

Use of Constraints In these two works on query rewriting using views for XQuery, no constraints are taken into account.

Preservation of Order The query rewriting algorithm in [37] does not consider the sibling order when computing a rewriting of the original query. On the other hand, in [40], a query rewriting setting where the sibling order is respected can be supported. As we will see in the next paragraph, the query rewriting algorithm from [40] involves finding an isomorphism between the return trees of the original query and the rewriting. The isomorphism can be either ordered (if the sibling ordering is preserved) or unordered (if the sibling ordering is not taken into account), and hence there may exist either ordered or unordered rewritings.

Description of the Query Answering Algorithms Here we provide more details on the two original algorithms for XQuery query rewriting using XQuery views.

[37] The goal of [37] is to obtain minimal equivalent rewritings, which is achieved by an iterative bottom-up approach for query rewriting, which at each step creates a partial rewriting by combining a smaller rewriting with a previously unused view. A minimal rewriting of a query Q using views $\mathcal{V}$ is defined as a rewriting Q' s.t. no other rewriting of Q using $\mathcal{V}$ uses a proper subset of the views referenced in Q'. As previously mentioned, the fragment of XQuery used in this work allows view joins, which is not possible with the rewriting algorithm from [40]. This is due to the possibility of storing and returning node identifiers. The rewriting algorithm has exponential complexity in the total size of the query and the views, which the authors show that in practice is not a problem through a set of experiments.

[40] In [40] the authors propose an approach for rewriting an XQuery query Q given a set of XQuery views $\mathcal{V}$, following the definition from the relational case presented in [29]. The presented algorithm is sound for full XQuery queries and views, and completeness of the algorithm is guaranteed for queries from the OptXQuery fragment. When rewriting an OptXQuery query using views defined in the same language, an equivalent rewriting is obtained, whenever one exists. In an XQuery context, the obtained rewriting is equivalent to the original query if the trees that they return are isomorphic.

Conclusion In this section we have seen two approaches that do not fit our comparison framework. This is because they use query and mapping languages that have a higher expressive power than the extended tree patterns. These two works focus on finding equivalent rewritings for queries in XQuery and some of its fragments. One of the challenges that remain as future work is to try to adapt and add new functionalities in our framework, and thus make it able to express mapping assertions of this kind.

5.3 Summary

So far, we have seen several approaches that deal with either XML data integration, data exchange or query rewriting using views. Here, we summarize the similarities and differences

Table 5.4: *Summary of the results on XML query rewriting using views*

work	language	nr. of views	problem	complexity	view semantics
[7]	$\mathsf{XP}^{\{/,//,[],*\}}$	single	existence of equivalent rewriting	coNP-complete	sound
			computation of equivalent rewritings using natural candidates, constructed in linear time	coNP-complete	
[8]	$\mathsf{XP}^{\{/,//,*\}}$	multiple	computation of equivalent rewritings using unions of views	not specified (sound, but not complete algorithm)	sound
[33]	$\mathsf{XP}^{\{/,//,[]\}}$	single	computation of maximally contained rewritings with schema	polynomial	sound
			computation of maximally contained rewritings without schema	exponential	
[37]	expressive XQuery dialect	multiple	computation of minimal equivalent rewritings	exponential in the size of the query and the views	sound
[40]	full XQuery and OptXQuery	multiple	computation of equivalent rewritings	exponential in the width[a] of the query, polynomial if the query is acyclic	exact
[54]	$\mathsf{XP}^{\{/,//,[],*\}}$ and fragments	single	existence of equivalent rewriting	coNP-hard, tractable for fragments	sound
			computation of minimal equivalent rewritings	Σ_3^p (refuted), tractable for fragments	

[a] The width of a query is a measure defined in [40] and is in general smaller than the query size.

between the approaches.

Most of the works we have considered in this section directed their attention to adapting the scenario of answering queries using views as defined in [29] to the XML setting. A survey on the issues of defining views in the XML context, alongside informal motivations behind their usability are presented in [1]. Key issues in answering queries using views include: selecting which views are going to be incorporated in the rewriting, actually performing the rewriting of the original query and obtaining the answer of the rewriting using the (materialized) views. The work presented in [44] deals with the view selection process in query rewriting. The authors propose an approach for view selection based on nondeterministic finite automata when multiple views are available. This approach efficiently filters out those views that are not significant for the query rewriting, and based on its output, a minimal view subset is determined. Based on this minimal subset of views, a rewriting of the query is obtained. Other works [7, 8, 37, 40, 54] have proposed algorithms for producing equivalent rewritings of XML queries using views, while [33] has proposed an algorithm for producing a maximally contained rewriting. Table 5.4 summarizes the works on query rewriting using views in the XML case, by showing which problems are tackled by each work and the results of the corresponding solutions that they propose.

As we have already argued, we are able to express the various approaches using the XML mapping assertion formalisms introduced in Section 3.2. We also proposed different criteria based on which we can compare separate works. Since, in general, we have shown that schema mappings play a crucial role in XML data management, an important feature of each approach is the type of schema mappings used. The expressive power of a mapping language is a crucial characteristic of each of the approaches, and it can be used for determining the performance of the query answering algorithms.

First and foremost, the question that arises is what kind of values are transferred using such schema mappings. One type of schema mappings consists of mapping assertions that extract tuples of atomic (string) values from the sources and fit them correspondingly in the global schema. Such schema mapping assertions are used in [9, 12, 56]. As we have already outlined, the mapping assertions in [9, 12] are quite similar. In fact, the ones in [9] build upon the ones in [12]. They are based on a restriction imposed on the XML document, stating that the only place where atomic values can be stored is as attribute values. This yields that they transfer tuples of attribute values from the source to the target schema. Although they use quite a different syntax, the schema mappings in [56] in fact are very similar to the ones in [9, 12]. In [56], the assumption that the atomic values are stored in the attributes is dropped, hence the schema mappings extract values stored at text nodes. Another type of schema mappings considers mapping assertions that copy a structure of a node along with the values stored in its descendants. This type of mappings corresponds to the setting of answering XPath queries using XPath views. In this case, only one tree (rooted at the output node of the XPath expression) is transferred by the mapping assertion. This tree remains intact, in the sense that it is not allowed to neither change the structure of the subtree nor to rename its elements. The schema mappings in the XPath query rewriting setting are slightly more expressive than the schema mappings which copy tuples of atomic values. The most expressive kind of schema mappings that we have come across with are the ones that transfer (single or multiple) trees and support modification of the structure, renaming of the elements, as well as introducing new element names. This kind of schema mapping

assertions are used in XQuery query rewriting using XQuery views. We have previously seen that our XML GLAV mapping assertion formalism is not expressive enough to capture this kind of schema mappings. One of the reasons for this is that the XQuery language is procedural, while our extended tree patterns are declarative. Moreover, the approaches that we discussed, such as [40], allow definition of nested views and rewriting of nested queries. This is yet another construct which is common in the XQuery language, but cannot be expressed in the language of the extended tree patterns.

The schema mappings can further be used to differentiate the approaches, by taking into account the semantics of the mapping assertions, i.e. whether the mapping assertions are interpreted as sound or exact. In other words, sound semantics corresponds to the open world assumption (OWA), while the exact semantics to the closed world assumption (CWA). Most of the works that we have considered so far implicitly use sound mapping assertions. However, in [41], explicit use of sound, exact and mixed mapping assertions is studied. In [40], equivalent rewriting of nested XQuery queries using XQuery views under the exact semantics is examined.

The approaches that we mention can also be classified by the type of query answering they perform. Namely, we can distinguish two types of query answering: (1) query answering by reformulation of the query (rewriting); or (2) query answering over a materialized target instance. As the name suggests, the approaches on XML query rewriting using views perform query answering by reformulation. Moreover, the data integration system in [56] also reformulates queries into the vocabulary of the sources. On the contrary, the data integration system proposed in [41] does not reformulate the queries, and performs query answering over a materialized instance that conforms to the global schema, in particular a global DTD. The works on XML data exchange also answer queries by materializing a target instance. This is due to the fact that the main task in XML data exchange is to transform a source XML document into a new target document conforming to a target DTD, by materializing the corresponding target document.

Another criterion that can be used to characterize the works that have been discussed so far, is the question of whether a single source or multiple sources are used as input to the corresponding problems addressed in each of the works. As sources, we take into account either source XML documents when dealing with XML data integration or exchange, or views when discussing XML query rewriting using views. Following the definition of XML data exchange, we classify [9, 12] as works where a single source is used. Naturally, in data exchange, the source document is transformed into a target one, using the schema mappings. Other works that consider XPath query rewriting using views also use a single view (i.e. a source) when computing a compensation pattern. Such works include [7, 33, 54]. On the other hand, there are works focused on XPath query rewriting that consider multiple views (sources), such as [8, 13]. The two works on XQuery query rewriting that we have mentioned ([37, 40]) also consider multiple views. Furthermore, the XML data integration systems in [41, 56] by definition support multiple source documents. Consequently, the answers of the queries posed over the data integration systems are coming from multiple different XML documents.

An important feature when addressing XML data management problems is the usage of constraints. Some works consider constraints as additional input of the corresponding problems, while others do not. The latter include [7, 8, 12, 37, 40, 54]. Among the works that consider

constraints, we can further specialize by considering the types of constraints used. The most common constraints applicable in XML data management tasks are constraints imposed by the definition of the structure of the XML documents (i.e. by the DTDs), as well as integrity constraints. [9, 12] use constraints that come from the definition of the structure of a document using a DTD. The data integration systems in [41, 56] consider both DTDs and integrity constraints. In [41] both key and foreign key constraints are taken into account, while [56] define target constraints, which resemble key constraints. In [33], two flavors of an XPath query rewriting algorithm are proposed: one without and one with the presence of constraints imposed by a DTD.

By definition, XML documents can be seen as ordered[1] labeled trees. However, in some cases when dealing with XML data it is useful to disregard the order, and hence not to allow horizontal navigation through the document. From the aforementioned approaches, the ones that disregard the order in the XML documents include [7, 8, 12, 37, 41, 54, 56]. In [40] the produced equivalent rewritings of XQuery queries using views can either respect the order of the document, or not take it into account, if the keyword **unordered** is present in the query. [33] generates maximally contained rewritings of XPath queries using XPath views without violation of the element order. In [9], schema mappings that perform horizontal navigation are proposed. However, it has been shown that query answering in a setting where the schema mapping assertions perform next-sibling navigation is intractable. Thus, in order to achieve tractability, one has to restrict the query answering setting, and not take horizontal navigation into account.

Table 5.5: *Overview of works, and the mapping and query languages used in them*

work	mapping language	granularity	type	query language
[7]	$\mathsf{XP}^{\{/,//,[],*\}}$	subtrees	LAV	$\mathsf{XP}^{\{/,//,[],*\}}$
[8]	$\mathsf{XP}^{\{/,//,*\}}$	subtrees	LAV	$\mathsf{XP}^{\{/,//,*\}}$
[9]	SM($\Downarrow$)	atomic values	GLAV	$\mathcal{CTQ}(\Downarrow,=)$
[12]	SM($\Downarrow,\Rightarrow,\mathsf{Fun},\sim$)	atomic values	GLAV	$\mathcal{UCTQ}(\Downarrow,\Rightarrow,=)$
[33]	$\mathsf{XP}^{\{/,//,[]\}}$	subtrees	LAV	$\mathsf{XP}^{\{/,//,[]\}}$
[37]	XQuery fragment	subtrees that can be modified	nested GLAV	XQuery fragment
[40]	OptXQuery	subtrees that can be modified	nested GLAV	OptXQuery
[41]	ps-queries	subtrees	LAV	ps-queries
[54]	$\mathsf{XP}^{\{/,//,[],*\}}$	subtrees	LAV	$\mathsf{XP}^{\{/,//,[],*\}}$
[56]	SM($\downarrow,=$)	atomic values	GLAV	XQuery fragment

So far, we have mainly focused on pointing out the features of the works that use XML mapping assertions in their attempts to adapt various well known database problems into the

[1] When speaking about order, we consider the sibling order between the nodes appearing on the same level in the XML tree.

XML setting. We concentrated on those approaches whose schema mapping assertions transfer either atomic values or subtrees w.r.t. the granularity. We only briefly mentioned two approaches whose schema mapping assertions are the most expressive ones w.r.t. the granularity, since their expressive power goes beyond the one of the XML mapping assertions defined in our framework. Table 5.5 summarizes the type of mapping assertions and query languages used in the different approaches that we consider. From the works that we studied in this thesis, we can draw the following conclusions about the granularity of the mapping assertions:

- the works that use XML LAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ transfer subtrees and have exactly one output variable;
- the works that use XML GLAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ transfer atomic values and have multiple output variables;
- none of the works that we consider utilizes XML LAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer atomic values or XML GLAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer subtrees;
- the XML LAV mapping assertions based on ps-queries can only transfer subtrees. Tuples of atomic values cannot be transferred by ps-queries.

Considering the problems on which the approaches are concentrated, it is interesting to mention which problems have been considered for which types of XML mapping assertions, and to indicate those that still remain as open questions.

The solved problems include:

- computation of equivalent and maximally contained rewritings w.r.t. XML LAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer subtrees. These problems are addressed in the works on XPath query rewriting using XPath views;
- computation of equivalent rewritings w.r.t. XML GLAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer atomic values. This problem is addressed in [56];
- computation of certain answers to queries in $(\mathcal{U})\mathcal{CTQ}$ w.r.t. XML GLAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer atomic values;
- computation of certain answers to ps-queries, i.e. computation of a certain prefix w.r.t. XML LAV mapping assertions based on ps-queries.

To the best of our knowledge, XML LAV and XML GLAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer atomic values and subtrees respectively have not been used. Hence, we identify the problems of finding certain answers, equivalent rewriting and maximally contained rewriting of a query w.r.t. a set of mapping assertions $\mathcal{M}$ as open problems for these types of mapping assertions. Also, computing certain answers of queries w.r.t. XML LAV mapping assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ is a problem that has not been considered yet. Moreover, the problems of computing an equivalent and maximally contained rewriting w.r.t. XML LAV mapping assertions based on ps-queries have not been solved either. Table 5.6 gives an overview of the

problems of query answering under a set of mapping assertions $\mathcal{M}$ that contains different types of mapping assertions. In the column labeled $\text{CA}_{\mathcal{M}}^{type}(q, a)$, $type$ is one of $\{tuple, ps, tree\}$ and a one of $\{\overline{t}, p, \pi\}$ respectively, depending on the expressive power of q. In the columns labeled $\text{ER}_{\mathcal{M}}^{\mathcal{L}_1,\mathcal{L}_2}(q)$ and $\text{MCR}_{\mathcal{M}}^{\mathcal{L}_1,\mathcal{L}_2}(q, a)$, $\mathcal{L}_1, \mathcal{L}_2$ are one of $\{\mathsf{XP}, \mathsf{XP}_1, \mathsf{UXP}_1, \mathsf{XP}_2, CQ, \mathsf{XQ}, \mathsf{OptXQ}\}$[2], depending on the approach.

Table 5.6: *Overview of the problems addressed in the works that we consider*

type of assertions in $\mathcal{M}$	**problem**		
	$\text{CA}_{\mathcal{M}}^{type}(q, a)$	$\text{ER}_{\mathcal{M}}^{\mathcal{L}_1,\mathcal{L}_2}(q)$	$\text{MCR}_{\mathcal{M}}^{\mathcal{L}_1,\mathcal{L}_2}(q)$
XML LAV assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer subtrees	open	considered in [7, 8, 54]	considered in [33]
XML LAV assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer atomic values	open	open	open
XML GLAV assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer subtrees	open	open	open
XML GLAV assertions based on $\mathsf{EP}^{\{/,//,[],*\}}$ that transfer atomic values	considered in [9, 12]	considered in [56]	open
XML LAV assertions based on ps-queries	considered in [41]	open	open

[2]For the definition of the shorthand notations, see Section 5.2, Section 5.2 and Section 5.2.

CHAPTER 6

Conclusion

In this thesis, we have focused on uniformly representing different solutions to a multitude of data management problems, addressed in the context of XML. We have overviewed works on XML data integration, XML data exchange and answering XML queries using XML views. We have illustrated similarities and differences between the works by introducing a unified framework for comparing the different approaches. This framework incorporates a new query language for XML trees, called extended tree patterns, that successfully captures the expressive power of several query languages that have been extensively used for querying XML trees throughout the literature on XML data management. We have introduced translation algorithms between these query formalisms and our formalism of extended tree patterns. Moreover, based on the extended tree patterns, we have defined schema mapping assertions for XML, that can be either LAV or GLAV mapping assertions. The translation algorithms, together with the mapping assertions for XML based on extended tree patterns, have allowed us to make approaches that tackle different problems using different query languages comparable. Furthermore, we have proposed a classification of the existing works in three different classes inside which we additionally characterize each approach by comparing them along several dimensions. By using our unified framework, we have been able to identify different open problems in XML data management. We have seen that there is still plenty of space for examining different variants of the problems connected with XML data management.

What remains as future work is to try to further extend and develop the unified framework we have proposed. Namely, it is of great interest to enhance the extended tree patterns and the XML mapping assertions based on them in order to enable them to capture the expressive power of the mapping assertions used in the context of XQuery query rewriting using XQuery views. Another interesting direction for future research is to propose a way of answering more expressive queries such as XQuery using mapping assertions based on extended tree patterns.

Bibliography

[1] Serge Abiteboul. On Views and XML. *SIGMOD Record*, 28(4):30–38, 1999.

[2] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. On the Representation and Querying of Sets of Possible Worlds. *Theoretical Computer Science*, 78(1):158–187, 1991.

[5] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[6] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Representing and Querying XML with Incomplete Information. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2001, May 21-23, 2001, Santa Barbara, California, USA*. ACM, 2001.

[7] Foto N. Afrati, Rada Chirkova, Manolis Gergatsoulis, Benny Kimelfeld, Vassia Pavlaki, and Yehoshua Sagiv. On Rewriting XPath Queries Using Views. In *12th International Conference on Extending Database Technology, EDBT 2009, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 168–179. ACM, 2009.

[8] Foto N. Afrati, Matthew Damigos, and Manolis Gergatsoulis. Union Rewritings for XPath Fragments. In *15th International Database Engineering and Applications Symposium, IDEAS 2011, September 21 - 27, 2011, Lisbon, Portugal*, pages 43–51. ACM, 2011.

[9] Shun'ichi Amano, Leonid Libkin, and Filip Murlak. XML Schema Mappings. In *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, pages 33–42. ACM, 2009.

[10] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of Tree Pattern Queries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, SIGMOD 2001, Santa Barbara, CA, USA, May 21-24, 2001*, pages 497–508. ACM, 2001.

[11] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. *ACM Transactions on Database Systems*, 29:195–232, 2004.

[12] Marcelo Arenas and Leonid Libkin. XML Data Exchange: Consistency and Query Answering. *Journal of the ACM*, 55(2), 2008.

[13] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3, 2004*, pages 60–71. Morgan Kaufmann, 2004.

[14] Pablo Barceló, Leonid Libkin, Antonella Poggi, and Cristina Sirangelo. XML with incomplete information. *Journal of the ACM*, 58(1):4, 2010.

[15] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Keys for XML. In *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*, pages 201–210. ACM, 2001.

[16] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Query Processing under GLAV Mappings for Relational and Graph Databases. *Proceedings of the VLDB Endowment*, 6(2):61–72, 2012.

[17] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[18] Claire David, Leonid Libkin, and Filip Murlak. Certain Answers for XML Queries. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 191–202. ACM, 2010.

[19] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. XML-QL: A Query Language for XML, 1998.

[20] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT Logical Framework for XQuery. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3, 2004*, pages 168–179. Morgan Kaufmann, 2004.

[21] Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases, KRDB 2001, Rome, Italy, September 15, 2001*, volume 45 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[22] Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive Query Plans for Data Integration. *The Journal of Logic Programming*, 43(1):49–73, 2000.

[23] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[24] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data Exchange: Getting to the Core. *ACM Transactions Database Systems*, 30(1):174–210, 2005.

[25] Wenfei Fan. XML Constraints: Specification, Analysis, and Applications. In *16th International Workshop on Database and Expert Systems Applications, DEXA 2005, 22-26 August 2005, Copenhagen, Denmark*, pages 805–809. IEEE Computer Society, 2005.

[26] Wenfei Fan and Leonid Libkin. On XML Integrity Constraints in the Presence of DTDs. *Journal of the ACM*, 49(3):368–406, 2002.

[27] Wenfei Fan and Jérôme Siméon. Integrity Constraints for XML. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2000, May 15-17, 2000, Dallas, Texas, USA*, pages 23–34. ACM, 2000.

[28] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational Plans For Data Integration. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, AAAI/IAAI 1999, July 18-22, 1999, Orlando, Florida, USA*, pages 67–73. AAAI Press / The MIT Press, 1999.

[29] Alon Y. Halevy. Answering Queries Using Views: A Survey. *The VLDB Journal*, 10(4):270–294, 2001.

[30] Tomasz Imielinski and Witold Lipski Jr. Incomplete Information in Relational Databases. *Journal of the ACM*, 31(4):761–791, 1984.

[31] Benny Kimelfeld and Yehoshua Sagiv. Revisiting Redundancy and Minimization in an XPath Fragment. In *11th International Conference on Extending Database Technology, EDBT 2008, Nantes, France, March 25-29, 2008, Proceedings*, volume 261 of *ACM International Conference Proceeding Series*, pages 61–72. ACM, 2008.

[32] Phokion G. Kolaitis. Schema Mappings, Data Exchange, and Metadata Management. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2005, June 13-15, 2005, Baltimore, Maryland, USA*, pages 61–75. ACM, 2005.

[33] Laks V. S. Lakshmanan, Wendy Hui Wang, and Zheng (Jessica) Zhao. Answering Tree Pattern Queries Using Views. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB 2006, Seoul, Korea, September 12-15, 2006*, pages 571–582. ACM, 2006.

[34] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2002, June 3-5, Madison, Wisconsin, USA*, pages 233–246. ACM, 2002.

[35] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of 22th International Conference on Very Large Data Bases, VLDB 1996, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann, 1996.

[36] David Maier. Database Desiderata for an XML Query Language. In *QL*, 1998.

[37] Ioana Manolescu, Konstantinos Karanasos, Vasilis Vassalos, and Spyros Zoupanos. Efficient XQuery Rewriting Using Multiple Views. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 972–983. IEEE Computer Society, 2011.

[38] Gerome Miklau and Dan Suciu. Containment and Equivalence for a Fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[39] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Database Theory, 7th International Conference, ICDT 1999, Jerusalem, Israel, January 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 1999.

[40] Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, and Emiran Curtmola. Rewriting Nested XML Queries Using Nested Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD Conference 2006, Chicago, Illinois, USA, June 27-29, 2006*, pages 443–454. ACM, 2006.

[41] Antonella Poggi and Serge Abiteboul. XML Data Integration with Identification. In *Database Programming Languages, 10th International Symposium, DBPL 2005, Trondheim, Norway, August 28-29, 2005, Revised Selected Papers*, volume 3774 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 2005.

[42] Rachel Pottinger and Alon Y. Levy. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of 26th International Conference on Very Large Data Bases, VLDB 2000, September 10-14, 2000, Cairo, Egypt*, pages 484–495. Morgan Kaufmann, 2000.

[43] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). In *QL*, 1998.

[44] Nan Tang, Jeffrey Xu Yu, M. Tamer Özsu, Byron Choi, and Kam-Fai Wong. Multiple Materialized View Selection for XPath Query Rewriting. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 873–882. IEEE, 2008.

[45] Ron van der Meyden. Logical Approaches to Incomplete Information: A Survey. In *Logics for Databases and Information Systems (the book grow out of the Dagstuhl Seminar 9529: Role of Logics in Information Systems, 1995)*, pages 307–356. Kluwer, 1998.

[46] Victor Vianu. A Web Odyssey: From Codd to XML. *SIGMOD Record*, 32(2):68–77, 2003.

[47] W3C. Document Type Definition. `http://www.w3.org/TR/html4/sgml/dtd.html`.

[48] W3C. Extensible Markup Language (XML). `http://www.w3.org/TR/REC-xml`.

[49] W3C. XML Path Language (XPath). `http://www.w3.org/TR/xpath/`.

[50] W3C. XML Schema Part 1: Structures Second Edition. `http://www.w3.org/TR/xmlschema-1/`.

[51] W3C. XQuery: An XML Query Language. `http://www.w3.org/TR/xquery`.

[52] W3C. XSL Transformations. `http://www.w3.org/TR/xslt`.

[53] Philip Wadler. A Formal Semantics of Patterns in XSLT and XPath. *Markup Languages*, 2(2):183–202, March 2000.

[54] Wanhong Xu and Z. Meral Özsoyoglu. Rewriting XPath Queries Using Materialized Views. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, Trondheim, Norway, August 30 - September 2, 2005*, pages 121–132. ACM, 2005.

[55] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, VLDB 1981, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

[56] Cong Yu and Lucian Popa. Constraint-Based XML Query Rewriting for Data Integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD Conference 2004, Paris, France, June 13-18, 2004*, pages 371–382. ACM, 2004.