Querying Semistructured Data Based On Schema Matching

D I S S E R T A T I O N

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herr Dipl.-Inf. André Bergholz
geboren am 20. März 1971 in Berlin

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Dr. h.c. Hans Meyer

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. sc. nat. Bodo Krause

Gutachter:

1. Prof. Johann Christoph Freytag, Ph. D.
2. Prof. Dr. Heinz Schweppe
3. Prof. Alex Brodsky, Ph. D.

eingereicht am: 25. November 1999
Tag der mündlichen Prüfung: 24. Januar 2000

Abstract

Most of today's data is still stored in files rather than in databases. This fact has become even more evident with the growth of the World Wide Web in the 1990s. Because of that observation, the research area of semistructured data has evolved. Semistructured data is typically stored in documents and has an irregular, partial, and implicit structure. The thesis presents a new framework for querying semistructured data. Traditional database management requires design and ensures declarativity. The possibilities to design are limited in the field of semistructured data, thus, a more flexible approach is needed.

We argue that semistructured data should be represented by a set of partial schemata rather than by one complete schema. Because of irregularities of the data, a complete schema would be very large and not representative. Instead, partial schemata can serve as good representations of parts of the data. While finding a complete schema turns out to be difficult, a database designer may be able to provide partial schemata for the database. Also, partial schemata can be extracted from user queries if the query language is designed appropriately. We suggest to split the notion of query into a "What"- and a "How"-part. Partial schemata represent the "What"-part. They cover semantically richer concepts than database schemata traditionally do. Among these concepts are predicates, variable definitions, and path descriptions. Schemata can be used for query optimization, but they also give users hints on the content of the database. Finding the occurrences (matches) of such a schema forms the most important part of query execution. All queries of our approach, such as the focus query or the transformation query, are based on this matching. Query execution can be optimized using knowledge about containment relationships between different schemata.

Our approach and the optimization techniques are conceptually modeled and implemented as a prototype on the basis of Constraint Satisfaction Problems (CSPs). CSPs form a general class of search problems for which many techniques and heuristics exist. A CSP consists of variables that have a domain associated to them. Constraints restrict the values that variables can simultaneously take. We transform the problem of finding the matches of a schema in a database to a CSP. We prove that under certain conditions the matches of a schema can be found without any search and in polynomial time. For optimization purposes the containment relationship between schemata is explored. We formulate a sufficient condition for schema containment and test it again using CSP techniques. The containment relationship can be used in two ways depending on the direction of the containment: It is either possible to reduce the search space when looking for matches of a schema, or it is possible to present the first few matches immediately without any search. Our approach has been implemented into the constraint system ECLiPSe and tested using XML documents.

Keywords:
Semistructured data, Query languages, Query processing, Constraint Satisfaction Problems

Zusammenfassung

Daten werden noch immer größtenteils in Dateien und nicht in Datenbanken gespeichert. Dieser Trend wird durch den Internetboom der 90er Jahre nur noch verstärkt. Daraus ist das Forschungsgebiet der semistrukturierten Daten entstanden. Semistrukturierte Daten sind Daten, die meist in Dokumenten gespeichert sind und eine implizite und irreguläre Struktur aufweisen. HTML- oder BibTEX-Dateien oder in ASCII-Dateien gespeicherte Genomdaten sind Beispiele. Traditionelles Datenbankmanagement erfordert Design und sichert Deklarativität zu. Dies ist im Umfeld der semistrukturierten Daten nicht gegeben, ein flexiblerer Ansatz wird gebraucht. In dieser Arbeit wird ein neuer Ansatz des Abfragens semistrukturierter Daten präsentiert.

Wir schlagen vor, semistrukturierte Daten durch eine Menge von partiellen Schemata zu beschreiben, anstatt zu versuchen, ein globales Schema zu definieren. Letzteres ist zwar geeignet, einen effizienten Zugriff auf Daten zu ermöglichen; ein globales Schema für semistrukturierte Daten leidet aber zwangsläufig an der Irregularität der Struktur der Daten. Wegen der vielen Ausnahmen vom intendierten Schema wird ein globales Schema schnell sehr groß und wenig repräsentativ. Damit wird dem Nutzer ein verzerrtes Bild über die Daten gegeben. Hingegen können partielle Schemata eher ein repräsentatives Bild eines Teils der Daten darstellen. Mit Hilfe statistischer Methoden kann die Güte eines partiellen Schemas bewertet werden, ebenso können irrelevante Teile der Datenbank identifiziert werden. Ein Datenbanksystem, das auf partiellen Schemata basiert, ist flexibler und reflektiert den Grad der Strukturierung auf vielen Ebenen. Seine Benutzbarkeit und seine Performanz steigen mit einem höheren Grad an Struktur und mit seiner Nutzungsdauer.

Partielle Schemata können auf zwei Arten gewonnen werden. Erstens können sie durch einen Datenbankdesigner bereitgestellt werden. Es ist so gut wie unmöglich, eine semistrukturierte Datenbank komplett zu modellieren, das Modellieren gewisser Teile ist jedoch denkbar. Zweitens können partielle Schemata aus Benutzeranfragen gewonnen werden, wenn nur die Anfragesprache entsprechend entworfen und definiert wird. Wir schlagen vor, eine Anfrage in einen „Was“- und einen „Wie“-Teil aufzuspalten. Der „Was“-Teil wird durch partielle Schemata repräsentiert. Partielle Schemata beinhalten reiche semantische Konzepte, wie Variablendefinitionen und Pfadbeschreibungen, die an Konzepte aus Anfragesprachen angelehnt sind. Mit Variablendefinitionen können verschiedene Teile der Datenbank miteinander verbunden werden. Pfadbeschreibungen helfen, durch das Zulassen einer gewissen Unschärfe, die Irregularität der Struktur der Daten zu verdecken. Das Finden von Stellen der Datenbank, die zu einem partiellen Schema passen, bildet die Grundlage für alle Arten von Anfragen. Im „Wie“-Teil der Anfrage werden die gefundenen Stellen der Datenbank für die Antwort modifiziert. Dabei können Teile der gefundenen Entsprechungen des partiellen Schemas ausgeblendet werden oder auch die Struktur der Antwort völlig verändert werden. Wir untersuchen die Ausdrucksstärke unserer Anfragesprache, in dem wir einerseits die Operatoren der relationalen Algebra abbilden und andererseits das Abfragen von XML-Dokumenten demonstrieren.

Wir stellen fest, daß das Finden der Entsprechungen eines Schemas (wir nennen ein partielles Schema in der Arbeit nur Schema) den aufwendigsten Teil der Anfragebearbeitung ausmacht. Wir verwenden eine weitere Abstraktionsebene, die der Constraint Satisfaction Probleme, um die Entsprechungen eines Schemas in einer Datenbank zu finden. Constraint Satisfaction Probleme bilden eine allgemeine Klasse von Suchproblemen. Für sie existieren bereits zahlreiche Optimierungsalgorithmen und -heuristiken. Die Grundidee besteht darin, Variablen mit zugehörigen Domänen einzuführen und dann die Werte, die verschiedene Variablen gleichzeitig annehmen können, über Nebenbedingungen zu steuern. In unserem Ansatz wird das Schema in Variablen überführt, die Domänen werden aus der Datenbank gebildet. Nebenbedingungen ergeben sich aus den im Schema vorhandenen Prädikaten, Variablendefinitionen und Pfadbeschreibungen sowie aus der Graphstruktur des Schemas. Es werden zahlreiche Optimierungstechniken für Constraint Satisfaction Probleme in der Arbeit vorgestellt. Wir beweisen, daß die Entsprechungen eines Schemas in einer Datenbank ohne Suche und in polynomialer Zeit gefunden werden können, wenn das Schema ein Baum ist, keine Variablendefinitionen enthält und von der Anforderung der Injektivität einer Einbettung abgesehen wird. Zur Optimierung wird das Enthaltensein von Schemata herangezogen. Das Enthaltensein von Schemata kann auf zwei Weisen, je nach Richtung der Enthaltenseinsbeziehung, genutzt werden: Entweder kann der Suchraum für ein neues Schema reduziert werden oder es können die ersten passenden Stellen zu einem neuen Schema sofort präsentiert werden.

Der gesamte Anfrageansatz wurde prototypisch zunächst in einem Public-Domain Prolog System, später im Constraintsystem ECLiPSe implementiert und mit Anfragen an XML-Dokumente getestet. Dabei wurden die Auswirkungen verschiedener Optimierungen getestet. Außerdem wird eine grafische Benutzerschnittstelle zur Verfügung gestellt.

Schlagwörter:
Semistrukturierte Daten, Anfragesprachen, Anfragebearbeitung, Constraint Satisfaction Probleme

Acknowledgment

I am grateful to Professor Johann Christoph Freytag, Professor Heinz Schweppe, and Professor Alex Brodsky for reviewing my thesis. For the past three years, Professor Freytag has supervised my work and helped me with numerous discussions and comments. He also made it possible for me to meet other researchers, in particular the Lore group at Stanford University. Professor Schweppe also helped me to improve my work with many discussions.

I would like to thank all members of the Graduate School in "Distributed Information Systems", in particular its chair Professor Oliver Günther. The internal workshops provided a critical forum, that effected the progress of my work in a very positive manner. Furthermore, every student benefited from the many invited talks by prominent researchers. Many thanks for numerous discussions to all the members of the Graduate School, in particular to Professor Bernhard Thalheim, Dr. Myra Spiliopoulou, Lukas Faulstich, Marcus Jürgens, Ulf Leser, and Felix Naumann. The research of the Graduate School is supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).

The database group at Humboldt-University made my everyday life very enjoyable. I am most grateful to Felix Naumann for reviewing many papers and drafts, for an infinite number of comments, and for his patience with me. I would like to thank all members of our group for numerous discussions and comments, and for providing a pleasant atmosphere to work in during the past three years. Karsten Lücke helped me with the implementation of the project. I thank Heinz Werner for his continuous technical support and Ulrike Scholz for solving many administrative problems.

Last, but most important, I would like to thank my mother and Michael Wiederhold for their love and their imperturbable faith in me. Thank you all!

Berlin, 26th May 2000

André Bergholz

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Semistructured Data

The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.
(Sir William Bragg)

1.1 Those good old days: Traditional data management

With the amount of electronically available data growing tremendously during the past decade the management of electronic data becomes more and more important. Database management systems (DBMSs) are extremely valuable tools for achieving this. They are widely used in all areas of today's society. The major advantage of database management systems is that they provide a "centralized control of ... data" [Dat95]. Date explains the advantages of DBMSs in more detail:

- Redundancy of data can be avoided, because applications share data instead of having their own private files.
- Thus, inconsistency of data can be avoided.
- Data can be shared. Not only existing applications can share data, but new applications can be built on top of a database.
- Standards on data representation can be enforced. This is very desirable e.g., for data exchange.
- Security restrictions can be applied. Rules can be specified for each type of access on each piece of data. The centralized management of data in fact requires a good security system.
- Integrity of data can be maintained.

- Most important, data independence can be provided. Knowledge of data organization and access techniques is no longer needed on the application level.

Relational DBMSs are most widely used today. There are a number of good reasons for their popularity. First, they have a solid theoretical foundation, the relational model [Cod70]. Second, they offer declarative access to the data. That is, access to the data is based on the structure rather than the content. Third, query processing techniques are well-developed and relational systems scale to today's demands on data management.

The downside of relational database management is, however, that it requires design. A data administrator defines the structure of the data; he enforces standards. Only then does querying based on this structure become possible for every user; and the vision of many applications sharing the same data can become true. We will see in the next section that one of the most important challenges of today's database research is to handle data that cannot be put into a predefined structure.

1.2 These scary new days: Semistructured data

Semistructured data has emerged as an important research topic within the database field [Abi97, Bun97]. There are several reasons for this development. First, the amount of electronically available data in a vast variety of representations has grown enormously over the last couple of years. The World Wide Web is the driving force behind this growth. Treating the Web like a database is desirable, however, the Web cannot be constrained by a schema. A second reason for the importance of semistructured data is data exchange. We would like to have flexible means for exchanging data between different places. Third, even when dealing with structured data it may be convenient to view them as semistructured for certain purposes, such as browsing.

Documents of markup languages are the most cited examples of semistructured data. BibTEX-files [Lam94] are related examples, because they have many properties in common with documents of markup languages. The example in Figure 1.1 presents a BibTEX-file containing two entries. They share attributes, yet they are not derived from some "superclass". Within the entries there are attributes that are mandatory and others that are optional. Furthermore, arbitrary annotation is permitted. Most important, the structure is given within the document itself.

Informally, semistructured data is "data that is neither raw data nor strictly typed" [Abi97]. We pick three aspects of semistructured data that seem very characteristic.

1. The structure may be irregular. The data may be more specific or less specific than some intended schema. Attributes may be missing, extra annotation may be given. Different types may be used for the same attribute. An address can

```
@inproceedings{Abi97,
author="Abiteboul, S.",
title="Querying Semi-Structured Data",
booktitle="Proceeding of the
  International Conference on Database Theory (ICDT)",
year=1997}

@book{Dat95,
author="Date, C. J.",
title="An Introduction To Database Systems",
publisher="Addison-Wesley",
year=1995,
series="The System Programming Series",
edition="6th"}
```

Figure 1.1: An excerpt from a BibTEX-file

be just a string or be split into street, city and zip code. The latter may be an integer or a string.

2. The structure may be partial. Over a discourse world the degree of structure may vary. Bitmaps or text have little structure, whereas other parts of the data may be well structured.

3. The structure of the data may be implicit. The structure becomes clear only after analyzing the actual value of the data. Because semistructured data is very often stored in documents, the aspect of implicity of structure is very typical. BibTEX is a good example for this.

We recognize that semistructured data is really a new and demanding challenge for database researchers. The most important question, that arises in this context, is certainly: How we can query semistructured data in the same declarative manner, that we are used to with, say, relational data? But there is more: Many traditional areas of database research, such as query optimization, indexing, or views have to be adapted to cope with the new challenge.

1.3 Queries, Queries, Queries

Because querying is probably the most important requirement that people have for a database system, we take a closer look at various kinds of queries that users pose.

grep command The command `grep 'Carpenter' *.txt` in a Unix-system takes all files in the current directory with the extension `txt` as its "database" and returns occurrences of the string given. These occurrences are in a sense the answer to the query. They take the form `<filename> : <line_of_occurence>`. Instead of a string a regular expression can be given as the first argument. This type of querying does not respect any kind of structure of the database, it is a purely *content-based* kind of querying.

Search engines Search engines on the World Wide Web provide a somewhat different kind of functionality. For them, the "database" is the set of HTML-pages they are aware of. The user can enter one or several strings and the engine returns URLs of HTML-pages containing these strings. Most search engines also support boolean operations, e.g., the user can specify that the page should contain one string, but not another. So far, this kind of querying is also purely content-based. However, because HTML is a markup language the user can typically also pose queries based on tags. At AltaVista [Alt] such a query takes the form `<tag>:<string>`; the system returns all pages where `<string>` occurs in a text marked up by `<tag>`. This introduces a little structure to querying. However, because HTML tags are predefined and do not usually reflect the structure of a document very well, this type of search engines really does only slightly better than a Unix-`grep`. Other search engines, such as Yahoo! [Yah], provide a hierarchically organized catalog of preselected pages. With such a system, the query is first matched against the catalog returning a set of categories and pages from the preselected set. Querying the complete set of pages is performed afterward or can be performed on demand.

Relational algebra In contrast to the above, this query language of the relational model is purely *structure-based*. Queries are expressed in terms of relation and attribute names. For example, the user can specify a query asking for the names of all persons born in 1971. This query is purely based on the relation name `Person` and on the attribute names `name` and `yearOfBirth`. Operators of the relational algebra include the selection (denoted by σ), the projection (π), the Cartesian product ($\times$), the Join ($\bowtie$), the division ($\div$) and the set operations union ($\cup$), intersection ($\cap$), and difference ($\setminus$). The example query can be written as $\pi_{\{name\}}(\sigma_{yearOfBirth=1971}(Person))$. The answer consists of a relation containing one attribute `name` filled with 1-tuples of names. Querying in this structure-based manner is possible, because the data is well-structured and organized in relations. Note that this language has a formal set-based semantic.

SQL SQL is the standard query language for relational database systems. Its theoreti-

cal foundation is the tuple relational calculus. It is also a structure-based language (also called *declarative* language). The example query from the previous point can be expressed in SQL as `SELECT name FROM Person WHERE yearOfBirth=1971;`. However, SQL provides a richer functionality. The user can express queries that include concepts, such as outer joins or aggregations. These concepts cannot be formalized within the set-theoretic approach of the relational algebra. The former introduces `NULL`-values and the latter often requires the presence of an ordering on the values. A query language similar to SQL is what we would like to have for semistructured data as well.

1.4 A flexible query framework for semistructured data

In this section we present the main idea of this thesis, namely a new framework for querying semistructured data. The following chapters demonstrate how this framework is put into practice. A preliminary version of the ideas presented in this thesis can also be found in [Ber99].

Consider again relational systems. We identify three abstract layers: the operational layer, the schema layer and the instance layer. The tuples form the instance layer; and the tables form the schema layer. On the operational layer there are concepts, such as queries, views, or constraints. We note that the items of the operational layer are expressed using the items of the schema layer, i.e., queries are expressed using tables. We would like to adapt this framework for querying semistructured data. As we have learned in the previous sections, the serious problem of semistructured data is its lack of known-in-advance structure. We observe that for relational data every item on the instance layer (every tuple) belongs to exactly one item on the schema layer (one table). Certainly, this constraint has to be relaxed in the context of semistructured data.

The query framework, adapted to cover semistructured data, is shown in Figure 1.2. Because semistructured data is usually represented as a graph, we show example graphs for the two bottom layers. Partial schemata in the middle layer conform to some parts of the database in the bottom layer. There is no further restriction, i.e., a partial schema can have an arbitrary number of instances in the database, and instances can conform to an arbitrary number of schemata.

The crucial layer of this approach is the middle one, the layer of the schemata. There are a number of interesting questions: How do we get those schemata? What are they good for? How do we manage them? The simplest way to get them is from a database designer. Remember that the data is called semistructured rather than unstructured. So at least some parts of a database can potentially be modeled.

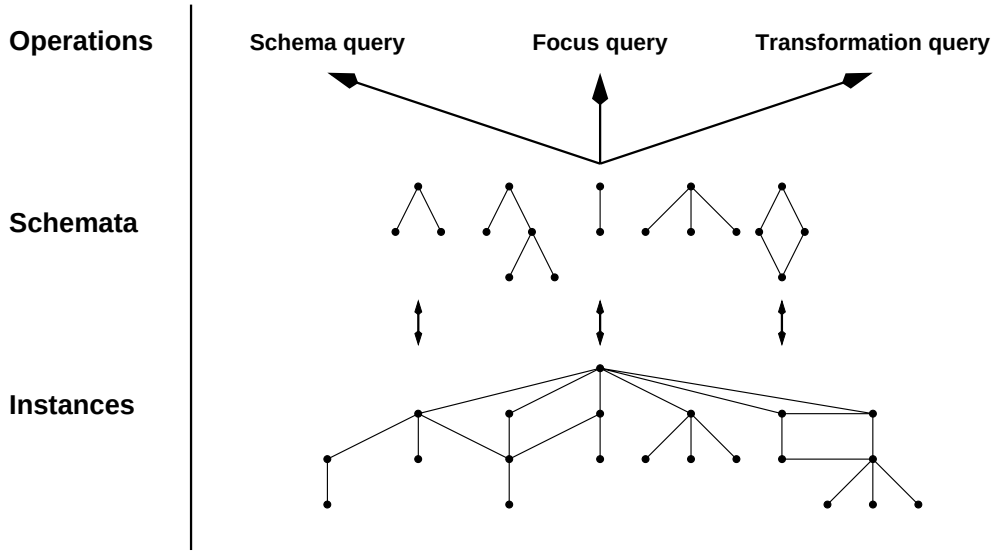

Figure 1.2: A new query framework for semistructured data

A database designer may thus be able to provide some meaningful partial schemata. Another way to get partial schemata is the following. A query posed to the system uses both schema and operational layer. In other words, a query consists of a "What"-part (i.e., a partial schema) and a "How"-part (i.e., an operation). As an analogy, in the relational world we can consider a selection to correspond to the "What"-part and a projection correspond to the "How"-part of a query. Now, an obvious approach is to cache the "What" 's, i.e., to extract partial schemata out of queries. To make this possible we lift some concepts typically found in queries (such as selection conditions) to the layer of the schemata. Partial schemata are useful for two main purposes. First, they can give users hints on the content of a database. Second, they can be used for query optimization. Note, that schemata being good for the former are not necessarily good for the latter and vice versa.

What are the advantages of our approach? A system designed in this way reflects the degree of structure of a database on many levels. If a database is well structured there will be large schemata with many instances. Thus, users will get a lot of information about the data; and the performance of the system will be good as well. If, however, the database is not well structured there will be only some useful schemata. Thus, the user will not get full knowledge about the database; and the performance will suffer as well. The schema layer can serve as an indication on the degree of structure of the database. The existence of large schemata with many instances indicates that the database is rather well structured. Parts of the database, that are not covered by any schema, are probably not very interesting or have a rather obscure structure. We will

conclude this section with three paradigms that shall guide our approach:

1. Answering a query *works without* schema information.
2. Answering a query *benefits from* schema information.
3. Answering a query *induces new* schema information.

1.5 Related areas

In this section we introduce areas that substantially influence this thesis. For now, we only give overviews of the areas. More specific relationships will be pointed out whenever appropriate in the later chapters. We start by introducing other projects on semistructured data and observe two of the areas that strongly influenced semistructured data: Data integration and Web querying. Ideas from the area of graph transformations are used in our approach for specifying queries. Constraint satisfaction techniques play an important role in this work, they are used for query optimization.

Semistructured data We already introduced the notion of semistructured data in Section 1.2. The Lore project at Stanford [MAG+97] and the UnQL query language at the University of Pennsylvania [BDHS96] will be discussed in detail. The reason why is that the two have semistructured data in general and not just one specific application in mind.

Lore initially used a simple and flexible data model, the Object Exchange Model (OEM) [PGMW95], to represent the data. Recently, Lore switched to using XML [Xml]. In OEM all objects are self-describing; there is initially no need for classes or schemata. For this model a query language named Lorel ("Lightweight Object Repository Language") has been developed [QRS+95]. Syntactically, Lorel is similar to SQL/OQL. It supports simple queries, boolean connectors, subqueries, and label markers to distinguish prefixes in paths. The semistructured flavor comes through by the introduction of general path expressions. They serve two purposes: One can use wildcards for paths, and one can define regular expressions over the labels. In a next step some schema information in the form of dynamically created and maintained DataGuides is introduced [GW97]. They help the user to get a better view on the data, e.g., for query formulation in a QBE-like manner. Additionally, DataGuides are useful for optimization; they might serve as an index.

A similar, more theoretical project is the University of Pennsylvania's UnQL project. The query language UnQL ("Unstructured Query Language") has been developed [BDHS96]. Edge-labeled trees are used as the data model. The query language UnQL

has a similar functionality as Lorel. Edge-labeled trees or labels can be the result of a query. One interesting aspect is the explicit treatment of restructurings of a database by using the `traverse`-command. Again, post-defined schemata for the purpose of optimization are introduced [BDFS97].

Abiteboul, Cluet and Milo provide techniques for the management of data stored in files [ACM93]. The translation between structured strings and databases is investigated. Thus, file querying and manipulating by using database technology becomes possible. Furthermore, optimization techniques from relational databases are adapted.

Data integration Projects on semistructured data often originated together with information integration projects. Often these integration projects focused on a low-level, syntactical integration of data sources. A good example is the Lore project, which originated from The Stanford-IBM Manager of Multiple Information Sources (TSIMMIS, [CGMH+94]). TSIMMIS provides tools for integration of heterogeneous information sources. It uses the wrapper/mediator architecture to translate and combine information from different sources [Wie92]. Furthermore, TSIMMIS allows browsing of data sources over the Web.

Similarly, IBM's partner project Garlic aims at enabling large-scale multimedia information systems [CHN+95]. It shall be capable of integrating data from a variety of repositories. Garlic uses wrappers and a metadata repository. It is based on an object-oriented data model.

Querying the World Wide Web More related work arises in the context of querying the World Wide Web. A main focus lies on query languages suited for the Web.

AT&T's Strudel is a Web-site management system that addresses the problems of handling multiple data sources and of automating the management of site content and structure [FFK+98]. It selects and manages data at Web sites, organizes the structure of the data at individual pages as well as between multiple pages, and designs a visual presentation of pages. In this context, the query language StruQL for semistructured data is presented.

The ARANEUS project, located at the University of Rome and the University della Basilicata, aims at developing tools for the management of data coming from the World Wide Web [AMM97]. Web sites are described using a formal data model. Based on this model, tools and methodologies for wrapping, querying, integrating, designing and implementing Web sites have been developed.

W3QS ("WWW Query System"), a system developed at the Technion Israel Institute of Technology, uses the SQL-like query language W3QL that addresses both

structure and content of WWW nodes [KS95]. Similar languages are Concordia's WebLog and Toronto's WebSQL [LSS96, MMM96].

Abiteboul and Vianu address the problem from the theory point of view [AV97]. They consider first oder logic, Datalog, and Datalog with negation in the context of the Web and characterize them with respect to computability.

Graph transformations Graph transformations address the dynamic aspects of graphs. Implemented systems are typically rule-based and can be used to model behavior or workflow. A graph transformation rule can be described by two graphs: a left-hand side and a right-hand side. Informally, in an application of the rule a match for the left-hand side in the host graph is replaced by the right-hand side. An introduction into computation by graph transformations can be found in the book by Rozenberg [Roz97].

Two popular systems are PROGRES [Sch97] and AGG [Agg]. Particularly interesting from our point of view is how the matching of left-hand side graphs into the host graph is performed. PROGRES uses a database-like approach [Zue93] AGG uses a constraint-based approach [Rud98].

Constraint Satisfaction Problems Constraint Satisfaction Problems (CSPs) form a general class of search problems for which many techniques and heuristics exist. In a CSP variables with domains associated to them are given; and constraints restrict the values that variables can simultaneously take. A solution to a CSP is an assignment of values from the domains to the variables such that all constraints are satisfied. CSPs have a wide variety of applications ranging from airport counter allocation to multiple DNA and protein sequence alignment. Bartak [Bar98] and Kumar [Kum92] provide an introduction to the field. They give various algorithms, heuristics and useful background information to efficiently solve CSPs.

1.6 Thesis outline

This thesis is organized as follows. The query language in accordance with the framework outlined in Section 1.4 is described in the Chapters 2, 3, and 4. The three chapters correspond to the three layers shown in Figure 1.2. Chapter 2 describes the underlying syntax of labeled directed graphs as well as mappings between them. We also introduce our running examples. In Chapter 3 we introduce the notion of schema and define conformity between schemata and objects. Schemata can include predicates, variable definitions, and path descriptions. The queries on top of the schemata are described in Chapter 4. We investigate the expressiveness of the query language. Furthermore,

we give an introduction to the field of graph transformations, which is a conceptual inspiration for our approach.

The following Chapters 5, 6, and 7 deal with the problem of query optimization. In Chapter 5 we point out that the challenging part of answering a query is to find the matches of a given schema in a database graph. We give an introduction to the field of Constraint Satisfaction Problems (CSPs) and reformulate the problem of finding schema matches in this framework. Chapter 6 discusses various optimization techniques for CSPs. Among them are domain reduction by applying consistency techniques, search algorithms, and the order of instantiation of the variables. We also prove an interesting property of our approach: If the injectivity requirement is ignored, matches of a tree-shaped schema without variable definitions can be found without search and in polynomial time. In Chapter 5 we incorporate the notion of schema containment into our optimization. To this end, we define the notion of schema containment and give a sufficient condition for it. We describe how to test this condition, again using CSP techniques. Furthermore, we present how we make use of the knowledge about schema containment once we detect it.

Chapter 8 describes the implementation part of this work. We made our initial experiences with a Prolog prototype of a schema matcher and then switched to the commercial constraint solver ECLiPSe. In this chapter we also outline how we incorporate XML documents into our system. We conclude with a summary and a discussion of the thesis in Chapter 9. In the appendices we answer frequently asked questions, give fundamentals of graph theory and partially ordered sets, provide a documentation of our ECLiPSe-based answering system, and give a list of the mathematical symbols used in this thesis.

Part II

The Query Language

Chapter 2

Labeled Graphs for Data Representation

If it can't be expressed in figures,
it is not science; it is opinion.
(Lazarus Long)

This chapter describes the underlying syntax of our approach. We use a very general graph model that we introduce in Section 2.1. Mappings between graphs play an important role in later parts of this thesis when we talk about schemata and instances and about query processing. Definitions of such mappings are given in Section 2.2. To broaden our understanding of graphs, two alternative characterizations of graphs and mappings are presented in Section 2.3: an algebraic and a relational one. In Section 2.4 we give three examples that will be used several times throughout the whole work. Section 2.5 takes a look at other work on semistructured data and their data representations. In particular, we give a short introduction into XML, a model that has recently enjoyed great popularity. Finally, Section 2.6 summarizes this chapter.

2.1 Labeled directed graphs

We use a general graph model to represent the data we are interested in. Graph models seem to be "the unifying idea in semi-structured data" [Bun97]. We do not require any specific restrictions to our graphs. In particular we allow labels on both vertices and arcs and do not require the graph to be acyclic, a tree, connected etc.

Definition 2.1 (Total directed graph). A tuple $G = (V, A, s, t)$ is a *total directed graph* if V is a set of *vertices*, A a set of *arcs*, and s and t are total functions from A to V assigning each arc its *source* and *target* vertex, respectively.

We sometimes use the term *node* instead of vertex. However, we always use the term arc instead of edge to emphasize that we consider directed graphs. In our model two nodes can be linked by more than one arc. Furthermore, cycles are allowed. The following definition introduces labels on nodes and arcs.

Definition 2.2 (Labeled directed graph). Let $\mathcal{L}$ be an arbitrary set of *labels*. A tuple $G = (V, A, s, t, l)$ is a *($\mathcal{L}$-)labeled directed graph* if (V, A, s, t) is a total directed graph and $l : V \cup A \longrightarrow \mathcal{L}$ is a total *label function* assigning each vertex and arc a label from $\mathcal{L}$.

Now, an *object* is a labeled directed graph. We also use the term *database* instead of object when we talk about a "large" object that is to be queried. Note that we usually denote objects with lower-case letters (i.e., $o_1, o_2, \dots$), but graphs with upper-case letters (i.e., $G_1, G_2, H, \dots$) to be consistent with both worlds.

Figure 2.1 presents an example that we shall use many times throughout the work. It shows a semistructured database on persons having names, surnames, a year of birth, a profession etc. Additionally, a sibling relationship relates different people.

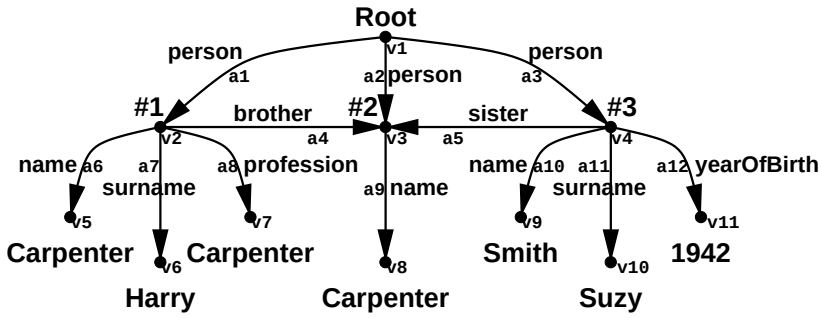

Figure 2.1: A labeled directed graph

Paths play an important role in semistructured data, because it is usually not exactly known where to find a certain piece of information. Thus, fuzziness and traversal are needed. We take a closer look at paths from the graph theory point of view.

Definition 2.3 (Walk, Trail, Path). A nonempty sequence $(v_{i_0}, a_{i_1}, v_{i_1}, \dots, a_{i_m}, v_{i_m})$ is called a *walk* in the graph $G = (V, A, s, t)$ if $s(a_{i_j}) = v_{i_{j-1}}$ and $t(a_{i_j}) = v_{i_j}$ for all positive $j \leq m$. If all arcs in a walk are pairwise distinct the walk is called a *trail*. If additionally all vertices are pairwise distinct the walk is called a *path*.

Usually we only give the arc sequences when specifying walks, trails and paths. This notion suffices, because the missing vertices are determined by the source and the target functions of the respective graph. Only walks consisting of exactly one vertex are exceptions. We call the number of arcs in a walk the *length* of the walk. Walks of length one are called *atomic*.

In our approach we are specifically interested in the trails of a graph. The main advantage over the walks is that in an arbitrary total directed graph the set of all trails is always finite. The main advantage over the paths is that the notion of trail is more general. Because in semistructured data the term path is well-established, we compromise at this point. We denote the set of all nonempty trails by P^+ and include this set in a graph (i.e., $G = (V, A, P^+, s, t)$) when appropriate. The source and target functions can be naturally extended to cover walks, trails, and paths: $s((v_{i_0}, a_{i_1}, v_{i_1}, \ldots, a_{i_m}, v_{i_m})) := v_{i_0}$ and $t((v_{i_0}, a_{i_1}, v_{i_1}, \ldots, a_{i_m}, v_{i_m})) := v_{i_m}$. We call two walks p_1 and p_2 *concatenable* if $t(p_1) = s(p_2)$. The *concatenation* of concatenable walks $p_1 = (v_{1_0}, a_{1_1}, v_{1_1}, \ldots, a_{1_m}, v_{1_m})$ and $p_2 = (v_{2_0}, a_{2_1}, v_{2_1}, \ldots, a_{2_n}, v_{2_n})$ is defined as $p_1 \circ p_2 = (v_{1_0}, a_{1_1}, v_{1_1}, \ldots, a_{1_m}, v_{1_m} = v_{2_0}, a_{2_1}, v_{2_1}, \ldots, a_{2_n}, v_{2_n})$. The concatenation is associative, i.e., for concatenable walks p_1, p_2 and p_3 the proposition $(p_1 \circ p_2) \circ p_3 = p_1 \circ (p_2 \circ p_3)$ holds.

For specifying answers to queries we will need the notion of a subobject of a database. The following definitions and lemmata assume some basic knowledge of partially ordered sets. For an introduction see e.g., [Tro92].

Definition 2.4 (Subobject). An object $o_2 = (V^{(o_2)}, A^{(o_2)}, s^{(o_2)}, t^{(o_2)}, l^{(o_2)})$ is a *subobject* of $o_1 = (V^{(o_1)}, A^{(o_1)}, s^{(o_1)}, t^{(o_1)}, l^{(o_1)})$ if $V^{(o_2)} \subseteq V^{(o_1)}$, $A^{(o_2)} \subseteq A^{(o_1)}$, $s^{(o_2)} = s^{(o_1)}|_{A^{(o_2)}}$, $t^{(o_2)} = t^{(o_1)}|_{A^{(o_2)}}$, and $l^{(o_2)} = l^{(o_1)}|_{V^{(o_2)} \cup A^{(o_2)}}$. We denote this by $o_2 \subseteq o_1$.

Intuitively, if $o_2 \subseteq o_1$, then o_2 is the "more general" object and o_1 is the "more specific" object. For a given object o we denote the set of all its subobjects by $\mathfrak{P}(o)$.

Lemma 2.1. *For a given object o the structure $[\mathfrak{P}(o), \subseteq]$ is a partially ordered set, i.e., $\subseteq$ is a reflexive, antisymmetric, and transitive binary relation over $\mathfrak{P}(o)$.*

Proof. $\subseteq$ is reflexive, because $o \subseteq o$ holds for all objects o. Let o_1 and o_2 be two objects with $o_1 = (V^{(o_1)}, A^{(o_1)}, s^{(o_1)}, t^{(o_1)}, l^{(o_1)})$ and $o_2 = (V^{(o_2)}, A^{(o_2)}, s^{(o_2)}, t^{(o_2)}, l^{(o_2)})$. Now, $o_1 \subseteq o_2$ and $o_2 \subseteq o_1$ immediately imply that $V^{(o_1)} = V^{(o_2)}$ and $A^{(o_1)} = A^{(o_2)}$ and thus, $o_1 = o_2$. Hence, $\subseteq$ is antisymmetric. Let $o_3 = (V^{(o_3)}, A^{(o_3)}, s^{(o_3)}, t^{(o_3)}, l^{(o_3)})$. $o_1 \subseteq o_2$ and $o_2 \subseteq o_3$ imply $V^{(o_1)} \subseteq V^{(o_3)}$ and $A^{(o_1)} \subseteq A^{(o_3)}$. We prove the restriction condition for the source functions. For the target and label functions it can be shown in a similar manner. $o_1 \subseteq o_2$ and $o_2 \subseteq o_3$ imply $s^{(o_3)} = s^{(o_2)}|_{A^{(o_3)}} = (s^{(o_1)}|_{A^{(o_2)}})|_{A^{(o_3)}}$. Because $A^{(o_3)} \subseteq A^{(o_2)}$ this is equal to $s^{(o_1)}|_{A^{(o_3)}}$. Hence, $\subseteq$ is transitive. $\square$

Lemma 2.2. *For a given object o the structure $[\mathfrak{P}(o), \subseteq]$ is a lattice, i.e., every nonempty subset of $\mathfrak{P}(o)$ has a least upper and a greatest lower bound.*

Proof. Let o be an object and $M = \{o_1, \ldots, o_m\}$ be an arbitrary subset of $\mathfrak{P}(o)$. Let $o_{glb} = (\bigcap_i V^{(o_i)}, \bigcap_i A^{(o_i)}, s^{(o)}|_{\bigcap_i A^{(o_i)}}, t^{(o)}|_{\bigcap_i A^{(o_i)}}, l^{(o)}|_{\bigcap_i V^{(o_i)} \cup A^{(o_i)}})$ be the "intersection

object". We show that o_{glb} is a greatest lower bound for M. A construction of a least upper bound can be done in a similar manner using a "union object". First, we observe that o_{glb} is indeed an object. If an arc is in $A^{(o_{glb})} = \bigcap_i A^{(o_i)}$ it is also in all $A^{(o_i)}$. Then its source and target vertices are in all $V^{(o_i)}$ and thus, in $V^{(o_{glb})}$. Now, $o_{glb} \subseteq o_i$ holds for all $1 \leq i \leq m$, because $V^{(o_{glb})} \subseteq V^{(o_i)}$ and $A^{(o_{glb})} \subseteq A^{(o_i)}$. Hence, o_{glb} is a lower bound for M. Suppose we have another lower bound o' for M, and $o' \subseteq o_{glb}$ does not hold. Then there exists a vertex or an arc $x \in V^{(o')} \cup A^{(o')}$ with $x \notin V^{(o_{glb})} \cup A^{(o_{glb})}$. This implies that there exists an object o_i with $x \notin V^{(o_i)} \cup A^{(o_i)}$. But this immediately leads to $o' \nsubseteq o_i$, which is a contradiction to the fact that o' is a lower bound for M. $\square$

With these lemmata we proved that we are in a well structured environment where such notions as "minimal element", "maximal antichain" etc. are defined. As an example we show in Figure 2.2 the Hasse diagram of all subgraphs of a directed tree with three nodes. Appendix B gives an introduction into notions related to partially ordered sets.

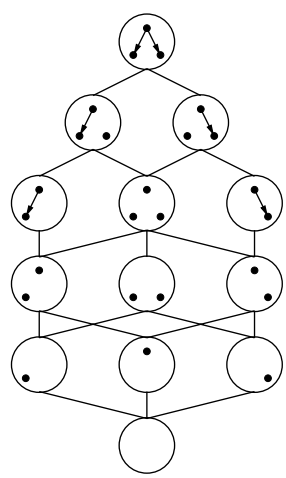

Figure 2.2: The lattice of all subgraphs of a simple tree

2.2 Mappings between graphs

To define a notion of conformity between schemata and objects, we introduce mappings between graphs based on their structure. We adopt the notion of graph morphism.

Definition 2.5 (Graph morphism). A *graph morphism* from one directed graph $G = (V^{(G)}, A^{(G)}, s^{(G)}, t^{(G)})$ into another directed graph $H = (V^{(H)}, A^{(H)}, s^{(H)}, t^{(H)})$ is a function $m : V^{(G)} \cup A^{(G)} \longrightarrow V^{(H)} \cup A^{(H)}$, such that

1. for all $x \in V^{(G)} \cup A^{(G)}$ it is true that $m(x) \in V^{(H)}$ if and only if $x \in V^{(G)}$, and

$m(x) \in A^{(H)}$ if and only if $x \in A^{(G)}$ (vertices are mapped to vertices and arcs are mapped to arcs) and

2. $s^{(H)} \circ m|_{A^{(G)}} = m \circ s^{(G)}$ and $t^{(H)} \circ m|_{A^{(G)}} = m \circ t^{(G)}$ (source and target of arcs are preserved by the morphism).

As opposed to people from the area of graph transformation we do not require that labels are preserved by graph morphisms. This requirement is too strong to be suitable for modeling mappings between different "kinds" of graphs. A graph morphism m is called *partial* if not all elements from $V^{(G)} \cup A^{(G)}$ are in the range of m, and *total* otherwise. Similarly, we introduce the notions of *surjective*, *injective*, and *bijective* graph morphisms. General characterizations of an arbitrary mapping $f : M \longrightarrow N$ are shown in Table 2.1.

	$M \longrightarrow N$	$M \longleftarrow N$
Totality	total	surjective
Uniqueness	unique	injective

Table 2.1: Characterizations of mappings

Definition 2.6 (Graph isomorphism). A total and bijective graph morphism is called a *graph isomorphism*.

2.3 An algebraic and a relational characterization

In this section we give two alternative views on labeled directed graphs. We start with an algebraic characterization that is frequently used by the people in the area of graph transformations. Second, we give a characterization based on relations that may be interesting for database people.

An algebraic characterization We start with the definition of an *algebra*. Any algebra is based on a *signature*.

Definition 2.7 (Signature). A *signature* Σ is a tuple (S, Ω). $S = \{s_1, \ldots, s_m\}$ is the set of the *sorts* and $\Omega = \{\omega_1, \ldots, \omega_n\}$ is the set of the *operation symbols* on these sorts. Every $\omega_j \in \Omega$ has an operation symbol *type* from $S^* \times S$, where S^* is the union of all lists of sorts of arbitrary length.

As an example consider this signature for operating on natural numbers. Remember that a signature is something purely syntactical. The intuition that is induced by the sorts and the operation symbols is not semantically manifested anywhere.

Signature $NUMBERS$

Sorts and operation symbols

$$\begin{array}{llll} \{Nat, Bool\} & T, F: & & \longrightarrow & Bool \\ & succ: & Nat & \longrightarrow & Nat \\ & add, mult: & NatNat & \longrightarrow & Nat \\ & eq: & NatNat & \longrightarrow & Bool \end{array}$$

Definition 2.8 (Algebra). An *algebra* $A = (A_S, A_\Omega)$ over a signature $\Sigma = (S, \Omega)$ consists of *carrier sets* A_{s_i} for every $s_i \in S$ and of *operations* $a_{\omega_j} : A_{s_{j_1}} \times \cdots \times A_{s_{j_{(k-1)}}} \longrightarrow A_{s_{j_k}}$ for every operation symbol $\omega_j : s_{j_1}, \ldots, s_{j_{(k-1)}} \longrightarrow s_{j_k}$.

The following algebra A provides the natural semantics for the signature $NUMBERS$ that was previously introduced.

Signature $NUMBERS$

Algebra A

Carrier sets and operations

$$\begin{array}{llllll} A_{Nat} := \{0, 1, 2, 3, 4, \ldots\} & a_T: & () & := & True \\ A_{Bool} := \{True, False\} & a_F: & () & := & False \\ & a_{succ}: & (x) & := & x + 1 \\ & a_{add}: & (x, y) & := & x + y \\ & a_{mult}: & (x, y) & := & x * y \\ & a_{eq}: & (x, y) & := & (x = y) \end{array}$$

Before we present a signature for labeled directed graphs we introduce two more notions in the context of algebras. We will see that in the context of labeled directed graphs they correspond to notions we introduced earlier in this chapter. The first notion is that of a *subalgebra*.

Definition 2.9 (Subalgebra). A *subalgebra* A' of an algebra A over the signature $\Sigma = (S, \Omega)$ consists of a family of subsets $A'_{s_i} \subseteq A_{s_i}$ for every sort $s_i \in S$. These subsets have to be closed with respect to the operations in A, i.e., for every operation $a_{\omega_j} : A_{s_{j_1}} \times \cdots \times A_{s_{j_{(k-1)}}} \longrightarrow A_{s_{j_k}}$ and all $(x'_{s_{j_1}}, \ldots, x's_{j_{(k-1)}}) \in A'_{s_{j_1}} \times \cdots \times A'_{s_{j_{(k-1)}}}$ it is true that $a_{\omega_j}(x'_{s_{j_1}}, \ldots, x's_{j_{(k-1)}}) \in A'_{s_{j_k}}$.

Definition 2.10 (Σ-Homomorphism). A *homomorphism* between two algebras A and B over the same signature $\Sigma = (S, \Omega)$ is a tuple of functions $f = (f_{s_1}, \ldots, f_{s_m})$

where every f_{s_i} is a total mapping between the carrier sets of the sort $s_i \in S$ ($f_{s_i} : A_{s_i} \longrightarrow B_{s_i}$). The homomorphism property holds for all operation symbols and their operations, i.e., for all operation symbols $\omega_j : s_{j_1}, \ldots, s_{j_{(k-1)}} \longrightarrow s_{j_k}$ and arbitrary $(x_{s_{j_1}}, \ldots, x_{s_{j_{(k-1)}}})$ from $A_{s_{j_1}} \times \cdots \times A_{s_{j_{(k-1)}}}$ it is true that $f_{s_{j_k}}(a_{\omega_j}(x_{s_{j_1}}, \ldots, x_{s_{j_{(k-1)}}})) = b_{\omega_j}(f_{s_{j_1}}(x_{s_{j_1}}), \ldots, f_{s_{j_{(k-1)}}}(x_{s_{j_{(k-1)}}}))$.

We provide an example of a second algebra B over the signature $NUMBERS$ that implements a group of cardinality ten. Then we define a tuple (f_{Nat}, f_{Bool}) that is a Σ-homomorphism between A and B.

Signature $NUMBERS$

Algebra B

Carrier sets and operations

$$
\begin{array}{llllll}
B_{Nat} := \{0,1,2,3,4,5,6,7,8,9\} & b_T : & () & := & True \\
B_{Bool} := \{True, False\} & b_F : & () & := & False \\
 & b_{succ} : & (x) & := & x+1 (\text{mod } 10) \\
 & b_{add} : & (x,y) & := & x+y (\text{mod } 10) \\
 & b_{mult} : & (x,y) & := & x*y (\text{mod } 10) \\
 & b_{eq} : & (x,y) & := & (x=y)
\end{array}
$$

Functions of the homomorphism

$$
\begin{aligned}
f_{Nat}(x) &:= x (\text{mod } 10) \\
f_{Bool}(x) &:= x
\end{aligned}
$$

An inverse homomorphism does not exist, because f_{Nat} is not an injective function. Just as before we also have the notion of *isomorphism*.

Definition 2.11 (Σ-Isomorphism). An *isomorphism* between two algebras A and B over the same signature $\Sigma = (S, \Omega)$ is a homomorphism $f = (f_{s_1}, \ldots, f_{s_m})$, such that $f^{-1} := (f_{s_1}^{-1}, \ldots, f_{s_m}^{-1})$ is a homomorphism between B and A as well.

We now present a signature for labeled directed graphs. Any algebra over this signature is a labeled directed graph.

Signature $GRASIG$

Sorts and operation symbols

$$
\begin{array}{llll}
\{Node, Arc, Label\} & source, target : & Arc & \longrightarrow & Node \\
 & nlabel : & Node & \longrightarrow & Label \\
 & alabel : & Arc & \longrightarrow & Label
\end{array}
$$

This notion of a labeled directed graph is equivalent to the one introduced in Section 2.1. The notions of subgraph from that section and of subalgebra from this one are equivalent as well.

The algebraic characterization presented is heavily used in the area of graph transformations. It forms the base for describing graph rules using the pushout concept from category theory.

A relational characterization A completely different kind of characterization of graphs is the relational one. It is based on binary relations between arcs and vertices.

Definition 2.12 (Directed graph). A tuple $G = (V, A, S, T)$ is called *directed graph* (or *digraph*) if V is a set of vertices, A is a set of arcs and $S, T \subseteq V \times A$ are unique relations called *source incidence* and *target incidence*.

The basic operation is the *product* of two relations. It is defined only for relations of type $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ for arbitrary sets X, Y, Z and the result will be of type $R \circ S \subseteq X \times Z$. The product of two relations is defined as

$$R \circ S := \{(x, z) \in X \times Z | \exists Y \in Y : (x, y) \in R \wedge (y, z) \in S\}$$

Usually we write RS instead of $R \circ S$. The product can be used to characterize relations. Recall Table 2.1 on Page 28. The properties introduced in that table can be described using the product of relations. Let I denote the identity relation. A relation R is called

1. *total* if and only if $I \subseteq RR^T$,
2. *unique* if and only if $R^T R \subseteq I$,
3. *surjective* if and only if $I \subseteq R^T R$ and
4. *injective* if and only if $RR^T \subseteq I$.

In this context R^T denotes the transpose of the relation R, i.e., $R^T := \{(x, y) | (y, x) \in R\}$. The subset relation $R \subseteq S$ means that $(x, y) \in R$ implies $(x, y) \in S$.

A simpler kind of directed graph is the *1-graph*. No parallel arcs are allowed in a 1-graph. Hence, it can be defined by a set of nodes V and a binary association relation $A \subseteq V \times V$ only. A directed graph does not have parallel arcs if and only if $SS^T \cap TT^T \subseteq I$. The corresponding 1-graph can be found by setting $A := S^T T$.

Look at the example in Figure 2.3. It shows a simple directed graph. This graph

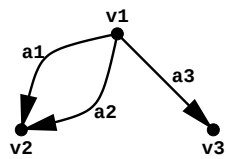

Figure 2.3: A simple directed graph

can be described by relations as follows:

$$
\begin{aligned}
V &= \{v_1, v_2, v_3\} \\
A &= \{a_1, a_2, a_3\} \\
S &= \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \\
T &= \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}
\end{aligned}
$$

The corresponding 1-graph "glues" the arcs a_1 and a_2 together. We are left with two arcs linking v_1 to v_2 and v_3, respectively. This fact can also be verified using the previously given formula:

$$
\begin{aligned}
A &:= S^T T \\
&= \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}
\end{aligned}
$$

Note that the matrix representation of the relations allows the product of relations to be computed similar to the traditional matrix product. Instead of $+$ and $\cdot$ the boolean operations $\vee$ and $\wedge$ are used. Using the other formula given above we check whether

there are any parallel arcs in the graph.

$$
\begin{aligned}
SS^T \cap TT^T &= \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cap \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \cap \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}
\end{aligned}
$$

We observe that the result is not a subset of the identity matrix. Hence, parallel arcs exist. In fact, we can use the result to recognize that a_1 and a_2 are parallel.

The definitions of morphisms are also based on the product of relations.

Definition 2.13 (Graph homomorphism, isomorphism). Let G_1 and G_2 be two directed graphs with $G_1 = (V_1, A_1, S_1, T_1)$ and $G_2 = (V_2, A_2, S_2, T_2)$. The tuple (M_V, M_A) is called a *homomorphism* between G_1 and G_2 if M_V and M_A are total and unique mappings, i.e., $M_V^T M_V \subseteq I$, $I \subseteq M_V M_V^T$, $M_A^T M_A \subseteq I$ and $I \subseteq M_A M_A^T$, and the morphism properties $S_1 \subseteq M_A S_2 M_V^T$ and $T_1 \subseteq M_A T_2 M_V^T$ hold. (M_V, M_A) is an *isomorphism* if both (M_V, M_A) and (M_V^T, M_A^T) are homomorphisms.

As an example consider a second graph consisting of just one arc, i.e., let V_2 be $\{v_1, v_2\}$, A_2 be $\{a_1\}$ and $S_2 = \begin{pmatrix} 1 & 0 \end{pmatrix}$ and $T_2 = \begin{pmatrix} 0 & 1 \end{pmatrix}$. Let the homomorphism map this arc to a_2 in graph in Figure 2.3. Therefore the two nodes v_1 and the two nodes v_2 are also mapped to each other. The homomorphism (M_V, M_A) is defined as

$$
M_V := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad M_A := \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}
$$

We check the uniqueness and the totality just for M_V. Uniqueness is given if and only if $M_V^T M_V \subseteq I$ holds. Intuitively, this formula says that going back from G_2 and forth again returns back to the same point.

$$
\begin{aligned}
M_V^T M_V &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
&\subseteq I
\end{aligned}
$$

Intuitively, totality means that going forth from G_1 and back again covers all the original points, i.e., $I \subseteq M_V M_V^T$ must hold.

$$\begin{aligned} M_V M_V^T &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ &\supseteq I \end{aligned}$$

The morphism properties $S_1 \subseteq M_A S_2 M_V^T$ and $T_1 \subseteq M_A T_2 M_V^T$ intuitively say that source and target vertices of an arc in G_1 can also be reached by going forth from the arc, then to the source or target of its image in G_2 and then back to G_1 again.

$$\begin{aligned} M_A S_2 M_V^T &= \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \end{pmatrix} \\ &\supseteq S_1 \end{aligned}$$

$$\begin{aligned} M_A T_2 M_V^T &= \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \end{pmatrix} \\ &\supseteq T_1 \end{aligned}$$

For 1-graphs the notion of homomorphism can be defined in an easier manner, because the arcs are not treated separately and only one mapping M (the equivalent of M_V) is needed. The requirements change to $M^T M \subseteq I$, $I \subseteq M M^T$ and $A_1 \subseteq M A_2 M^T$.

This relational characterization demonstrated that describing and manipulating graphs is possible on a purely operational basis. With these two alternative views on labeled directed graphs we hope to give the impression that various mathematical formalisms exist to cope with our syntactical model.

2.4 Three examples

This section presents three examples of databases that we shall use throughout the thesis. The first one is a toy example and will be used to illustrate the concepts of the

query language. The second example is derived from a relational database, the third one from an XML document. Both will be used to illustrate the expressiveness of the language. The first example is presented in Figure 2.4. Incidently, it is the same as the one in Figure 2.1. The example consists of three persons with varying attributes, such as name, surname etc. Furthermore, a sibling relationship is illustrated.

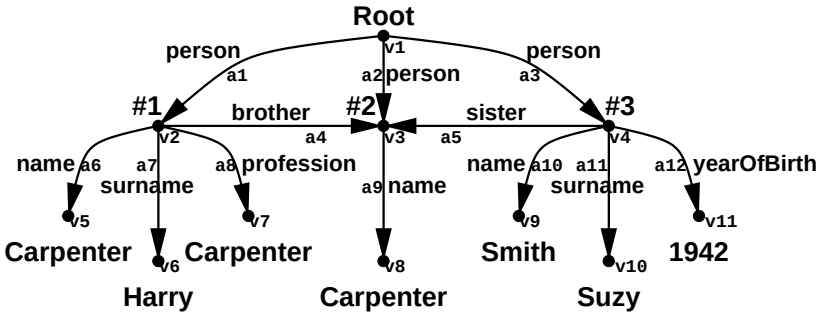

Figure 2.4: Example 1: A simple person database

The second example is a relational database. It consists of two tables, one describing persons and the other one relating persons to projects they are working on. Figure 2.5 shows the relational representation of the example.

Person:

ID	Name	Surname
01	Smith	John
02	Miller	Steve
03	Smith	Rita

WorksOn:

ID	Project
01	Holiday
02	Holiday
02	GetRich

Figure 2.5: Example 2: A relational database

We transform this relational representation into a graph representation using the ideas presented by Buneman and associates [BDHS96]. From the root down we split the database into relations, then into tuples and finally according to the attributes of the relations. Figure 2.6 shows the result of this transformation. Note that this transformation always leads to a tree of fixed height.

Finally, we use an XML representation of our database group as a third example. We split the aspects concerning our group into members, research, lectures and publications. Next we present an excerpt from the complete XML document. How this document is transformed into the graph representation is described in Section 8.2.

```
<DBIS name="LFE Datenbanken und Informationssysteme">

  <MEMBERS>
    <HEAD id="jcf">
      <NAME> Freytag </NAME>
      <SURNAME> Johann </SURNAME>
```

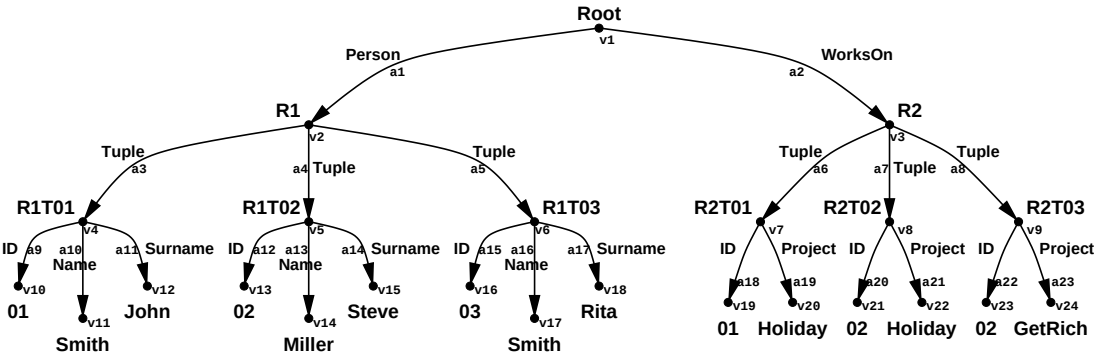

Figure 2.6: Example 2: A relational database as labeled directed graph

```
    <SURNAME> Christoph </SURNAME>
    <DEGREE> Ph.D. </DEGREE>
    <EMAIL> freytag@dbis.informatik.hu-berlin.de </EMAIL>
    <PHONE> 2093 3009 </PHONE>
    <FAX> 2093 3010 </FAX>
  </HEAD>
  <STAFF>
    <PERSON id="rc">
      <NAME> Conrad </NAME>
      <SURNAME> Rainer </SURNAME>
      <DEGREE> Dr. </DEGREE>
      <EMAIL> rconrad@dbis.informatik.hu-berlin.de </EMAIL>
      <PHONE> 2093 3020 </PHONE>
    </PERSON>
    <PERSON id="ab">
      <NAME> Bergholz </NAME>
      <SURNAME> Andre </SURNAME>
      <EMAIL> bergholz@dbis.informatik.hu-berlin.de </EMAIL>
      <PHONE> 2093 3024 </PHONE>
      <HOBBY> Go </HOBBY>
    </PERSON>
    ...
  </STAFF>
  <SECRETARY id="us">
    ...
  </SECRETARY>
  <TECHNICAL>
    ...
  </TECHNICAL>
  <STUDENTS>
    ...
    <PERSON id="kl">
      <NAME> Luecke </NAME>
```

```
      <SURNAME> Karsten </SURNAME>
      <EMAIL> luecke@dbis.informatik.hu-berlin.de </EMAIL>
    </PERSON>
  </STUDENTS>
  <ALUMNS>
    ...
  </ALUMNS>
</MEMBERS>

<RESEARCH>
  <PROJECT>
    <NAME> AQUES - An Agent-based Query Evaluation System </NAME>
    <HEAD idref="jcf"/>
    <MEMBER idref="ms"/>
  </PROJECT>
  <PROJECT>
    <NAME> CABS - Comprehensive Analysis of Biological Sequences </NAME>
    <HEAD idref="jcf"/>
    <MEMBER idref="ab"/>
    <PARTNER> Kelman GmbH </PARTNER>
  </PROJECT>
  ...
</RESEARCH>

<LECTURES>
  <LECTURE>
    <TITLE> Grundlagen von Datenbanksystemen </TITLE>
    <LECTURER idref="jcf"/>
    <ASSISTANT idref="rc"/>
    <ASSISTANT idref="ds"/>
  </LECTURE>
  ...
  <SEMINAR>
    <TITLE> Forschungsseminar: Neue Entwicklungen im Datenbankbereich </TITLE>
    <LECTURER idref="jcf"/>
    <ASSISTANT idref="ab"/>
  </SEMINAR>
</LECTURES>

<PUBLICATIONS>
  <PUBLICATION>
    <AUTHORS>
      <AUTHOR idref="fn"/>
      <AUTHOR> Ulf Leser </AUTHOR>
      <AUTHOR idref="jcf"/>
    </AUTHORS>
    <TITLE> Quality-driven Integration of Heterogeneous Information Sources
```

```
      </TITLE>
      <BOOKTITLE> Proceedings of the
        International Conference on Very Large Databases (VLDB 99)
      </BOOKTITLE>
      <LOCATION> Edinburgh </LOCATION>
      <YEAR> 1999 </YEAR>
      <MONTH> September </MONTH>
    </PUBLICATION>
    <PUBLICATION url="99dbpl.ps">
      <AUTHORS>
        <AUTHOR idref="ab"/>
        <AUTHOR idref="jcf"/>
      </AUTHORS>
      <TITLE> Querying Semistructured Data based on Schema Matching
      </TITLE>
      <BOOKTITLE> Proceedings of the International Workshop
        on Database Programming Languages (DBPL, in conjunction with VLDB'99)
      </BOOKTITLE>
      <LOCATION> Kinloch Rannoch </LOCATION>
      <YEAR> 1999 </YEAR>
      <MONTH> September </MONTH>
    </PUBLICATION>
    ...
  </PUBLICATIONS>

</DBIS>
```

2.5 Other representations for semistructured data

Most other approaches to semistructured data also use a graph-based syntax. The recently popular language *Extensible Markup Language* (*XML*, [Xml]) is no exception. An XML document consists of constructs of one of the following types:

- Element: An element is a collection object that can have child objects (such as data or more elements). Elements are denoted by tags and can be further specified by attributes with values.
- Data: Data is nothing but plain text. It can appear anywhere in an XML document.
- Document type definition: A DTD specifies a grammar for documents. It can by included within an XML document itself, or it can be specified externally.

- Processing instruction: A processing instruction is information meant for a potential application using the XML document. Typical examples are command names or parameters.

- Comment

Syntactically, XML is closely related to HTML. The important difference is that the tags defining the elements can be arbitrary (i.e., user-defined) in XML, but are predefined in HTML. XML documents can be accompanied by a *Document Type Definition* (*DTD*), which is essentially a grammar describing valid languages of XML documents (primarily by specifying allowed tags). Let us take a look at a simple example:

```
<?xml version="1.0" standalone="yes"?>
<!-- This is a most simple example. -->
<EXAMPLE id="1" foo="bar">
This is a test.
</EXAMPLE>
```

The first line is an introductory head line. There is a comment on the second line and an element with two attributes on the third to fifth line. The `id`-attribute has a predefined meaning; it introduces a symbolic object identifier. Elements provide the platform for nesting. In the example there is data nested in the `EXAMPLE`-element. Such a document can naturally be represented as a graph, e.g., as in Figure 2.7.

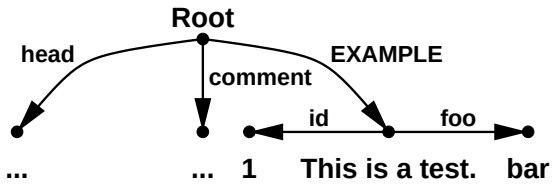

Figure 2.7: A simple XML document represented as a labeled directed graph

The advantage of XML is that it has a richer semantics, i.e., more constructs with a "meaning". On the other hand, this can also easily be the source of problems. Clearly, XML has great advantages over HTML, because the tags introducing some structure can be user-defined. This gives users a tool for directly describing the structure of a document that they have in mind. Take a look at this HTML example from my publications page:

```
<UL>
  <LI>Bergholz, A., Heymann, S., Schenk, J. A., Freytag, J. C.:
    <A HREF="97ideas.ps">
    "Sequence comparison using a relational database approach"</A>,
```

```
    Proceedings of the International Database Engineering
    and Applications Symposium (IDEAS) 1997, Montreal, August 1997
    (<A HREF="97ideatk.ps">talk slides</A>)
  </LI>
</UL>
```

Clearly, this is not very well-structured. Most important, the structure is mainly layout-driven and not content-driven. The same content could be encoded in XML as follows:

```
<PUBLICATIONS>
  <PUBLICATION url="97ideas.ps" url2="97ideatk.ps">
    <AUTHORS>
      <AUTHOR id="ab"> Bergholz, A. </AUTHOR>
      <AUTHOR id="sh"> Heymann, S. </AUTHOR>
      <AUTHOR id="js"> Schenk, J. A. </AUTHOR>
      <AUTHOR id="jcf"> Freytag, J. C. </AUTHOR>
    </AUTHORS>
    <TITLE> Sequence comparison using a relational database approach
    </TITLE>
    <BOOKTITLE> Proceedings of the International Database Engineering
      and Applications Symposium (IDEAS)
    </BOOKTITLE>
    <LOCATION> Montreal </LOCATION>
    <YEAR> 1997 </YEAR>
    <MONTH> August </MONTH>
  </PUBLICATION>
</PUBLICATIONS>
```

This representation is much better, because it reflects the structure that the author had in mind. To sum it up, XML is an emerging standard for representing information in documents. An XML document can be seen as a labeled directed graph, although XML has richer semantic concepts that graphs have not. Nevertheless, from a database point of view the graph representation seems like a reasonable abstraction.

Before switching to XML the Lore project at Stanford used a very similar model, the *Object Exchange Model* (*OEM*, [PGMW95]). OEM is a data model that is particularly suited for data exchange in heterogenous, dynamic environments. An object in OEM is a tuple (`label`, `type`, `value`, `object-ID`), where `label` denotes the kind of the object, `type` is a data type (atomic, composed or reference), `value` denotes the actual value and `object-ID` gives an identifier. This model is very flexible and simple. All objects

are self-describing, there is initially no need for classes or schemata. The component `label`, on the other hand, serves two purposes. It describes the semantics of the object, but it also identifies the object within a super-object. OEM is, like our own approach, semantically poorer, but simpler and more general than XML. There is for instance no distinction between nested elements and attributes in OEM; both of them would be represented as subobjects. We present an example for a composed OEM object next.

<person-record, set, $\{cmpnt_1, cmpnt_2, cmpnt_3\}$>
$cmpnt_1$ is <person-name, string, "Fred">
$cmpnt_2$ is <office-number-in-building, integer, 333>
$cmpnt_3$ is <department, string, "toy">

Represented as a graph an OEM object is a connected directed graph with a root, where labels are allowed on the arcs and on the leaf nodes, but not on the inner nodes. This is very similar to XML.

The UnQL project at UPenn uses a slightly different model [BDHS96]. The main difference is that labels are allowed only on the arcs. As shown in Figure 2.8 there are simple transformations between graphs with labels on both nodes and arcs and graphs with labels on arcs only or on nodes only. So allowing labels on arcs only certainly makes some formalisms easier and more elegant. On the other hand it is less intuitive. There are good reasons for allowing labels at least on the leaf nodes, because these labels typically represent values or data, whereas the other labels typically represent attribute names.

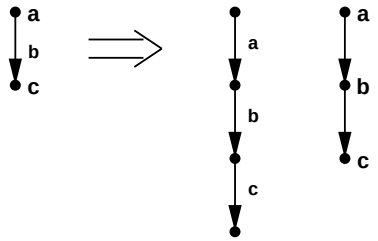

Figure 2.8: Labels on nodes and arcs - Transformations

In UnQL a database is an *edge-labeled tree*. An arbitrary edge-labeled tree can be constructed using

1. the empty tree: $\{\}$
2. the addition of a root and an edge labeled l to an already existing tree t: $\{l \Rightarrow t\}$
3. and the union of the roots of two trees t_1 and t_2: $t_1 \cup t_2$.

Furthermore, tree markers identify subtrees to introduce cycles. Using the abbreviations l for $l \Rightarrow \{\}$ and $\{l_1 \Rightarrow t_1, l_2 \Rightarrow t_2\}$ for $\{l_1 \Rightarrow t_1\} \cup \{l_2 \Rightarrow t_2\}$, and omitting the braces around singleton trees the "tree" in Figure 2.9 is described by X_1 where

$$X_1 = \{a \Rightarrow X_2, c \Rightarrow \{d, e \Rightarrow X_1\}\}$$
$$X_2 = \{b \Rightarrow X_2\}.$$

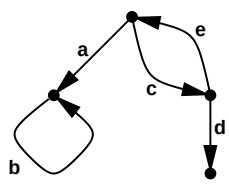

Figure 2.9: An edge-labeled tree with cycles

We described typical representations for semistructured data in this section. All approaches to semistructured data and querying the WWW use similar models. Table 2.2 summarizes the properties of the different approaches presented in this chapter. The conclusion is that graph models in one way or another seem to be an appropriate choice. Our approach is the most general one, because it does not restrict the structure of the graph nor does it restrict where to put the labels. Other models certainly have other advantages, such as a more elegant theory or a closer relationship to currently popular technologies, such as XML.

	Our approach	XML	OEM	UnQL model
Labels are allowed	everywhere	arcs / leaves	arcs / leaves	arcs only
Root node	not necessary	necessary	necessary	necessary
Cycles	yes	yes	yes	yes
Parallel arcs	yes	yes	yes	yes
Semantic concepts	none	many	object types	none

Table 2.2: Comparison of different models for semistructured data

2.6 Summary

This chapter introduced the underlying syntax of our approach. The basic notion is that of a labeled directed graph. We characterized this notion in different ways. The traditional way presented in Section 2.1 will be used throughout the thesis. Additionally we looked at an algebraic representation used in the area of graph transformations, and

at a relational representation based on operations between relations. Mappings between graphs have also been characterized in these different manners. We provided three running examples that will be used in the following chapters. Finally we characterized other representations for semistructured data, such as XML and other graph models.

Chapter 3

Schemata and Instances

There are no facts, only interpretations.
(Friedrich Nietzsche)

This chapter introduces the notions of schema and conformity between schemata and objects. These concepts form the base for querying as introduced in the next chapter. Potentially, although not within the scope of this work, it can also be the base for other database concepts, such as views or constraints. Is a view not nothing but a named schema or a named query? This notion of conformity provides "flexible declarativity", if you like.

When we talk about schemata we do not mean a complete database schema as we know it for instance from relational databases. Rather, we talk about something that describes certain parts of a database. In that sense, the term partial schema that we used in the introduction chapter would be more correct. From now on we abbreviate this notion and talk about schemata. We thought about using other terms, such as description or pattern, but schema still seems to be more appropriate.

This chapter is organized as follows. Section 3.1 introduces preliminary simple notions of schema and conformity by using predicates as object labels. We gradually enhance the schema notion by adding variable definitions in Section 3.2 and path descriptions in Section 3.3. To achieve the latter, we introduce a notion related to a graph closure, the corresponding trail graph. Finally, Section 3.4 looks at other work in this area; and Section 3.5 gives a summary.

3.1 Predicate schemata and naive conformity

This section presents a preliminary notion of schema for our approach. Informally, a schema is an object that describes a set of objects. In the simpler syntactic framework of the label world, this schema concept certainly exists as well. One label might de-

scribe a set of other labels. This is frequently done; data types, predicates and regular expressions are examples. As as first step toward schemata in the graph world we assign schemata from the label world to the elements of the graph. We choose predicates to be the label world schemata.

Definition 3.1 (Predicate schema). Given a set of unary predicates $\mathcal{P}$, a *predicate schema* (over $\mathcal{P}$) is an object $s = (V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)})$ where the elements are labeled with predicates ($l : V^{(s)} \cup A^{(s)} \longrightarrow \mathcal{P}$).

We give an example in Figure 3.1. Note that we treat a quoted constant c (such as 'Carpenter' in the example) as an abbreviation for the predicate $X = c$. The predicate $true()$ serves as a wildcard; it holds for every label in the database.

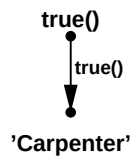

Figure 3.1: A simple predicate schema

To establish a relationship between a schema and the objects described by it, we establish the notion of *conformity* between both of them. Depending on the direction of the mapping, we say that we *match* a schema into an object, or we *interpret* an object by a schema.

Definition 3.2 (Naive conformity). A *match* (or a *match function*) of a predicate schema s into an object o is an isomorphic embedding of s into o, i.e., a total, injective graph morphism $m : s \longrightarrow o$, such that for all $x \in V^{(s)} \cup A^{(s)}$ the predicate $l^{(s)}(x)$ is true for $l^{(o)}(m(x))$.

If there exists a match of the schema s into the object o we say that o *conforms* to s (or o can be *interpreted* by s) and we call o an *instance* (or also a *match*) of s.

Let o be a database, s be a schema and $o_1 \subseteq o$ a match of s. Then every superobject o_2 of o_1 (i.e., $o_1 \subseteq o_2 \subseteq o$) is also a match of s. Let $\mathfrak{M}^{(s)}(o)$ denote the set of all matches of s in o. Because $\mathfrak{M}^{(s)}(o)$ is a subset of $\mathfrak{P}(o)$ the structure $[\mathfrak{M}^{(s)}(o), \subseteq]$ is also a partially ordered set. We call a minimal element in this partially ordered set a *minimal match* (or a *minimal instance*) of s in o. We denote the set of minimal matches of s in o with $\mathfrak{M}^{(s)}_{min}(o)$. In Figure 3.2 we show the same schema as in Figure 3.1, but this time together with its minimal matches in the database from Figure 2.4.

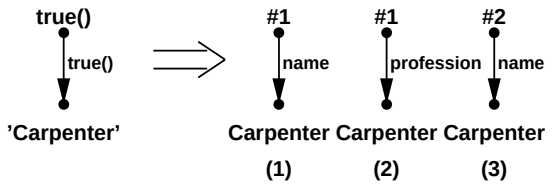

Figure 3.2: The predicate schema and its minimal matches

3.2 Adding variables

We gradually improve the expressiveness of the schemata. Remember that we propose to have richer concepts on the schema layer, because they ensure more flexibility in representing data. Thus, we lift some concepts known in query languages to the schema layer. In this section we add variable definitions. They enable us to enforce links between different parts of a database based on the labels. We add variable definitions in the following manner: Let s be a predicate schema, $\mathcal{V}$ be a set of variables and $v^{(s)} : V^{(s)} \cup A^{(s)} \longrightarrow \mathcal{V}$ be a partial mapping from the nodes and arcs in the schema into the variables. Then we call $(V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)}, v^{(s)})$ a *predicate schema with variables*. Intuitively, nodes and arcs that are mapped to the same variable are "joined", i.e., their labels must be the same on the instance level. Thus, we additionally require for a mapping m to be a match of s into an object o, that for all $x_1, x_2 \in V^{(s)} \cup A^{(s)}$, if $v(x_1)$ and $v(x_2)$ exist and $v(x_1) = v(x_2)$ then $l^{(o)}(m(x_1)) = l^{(o)}(m(x_2))$. A predicate schema with variables and its minimal matches in the database from Figure 2.4 are shown in Figure 3.3. In this example the set of variables $\mathcal{V}$ consists of just one variable; and the two nodes at the bottom are mapped to it. This is indicated by the label $X : true()$. Intuitively, the schema matches every object where the name equals the profession.

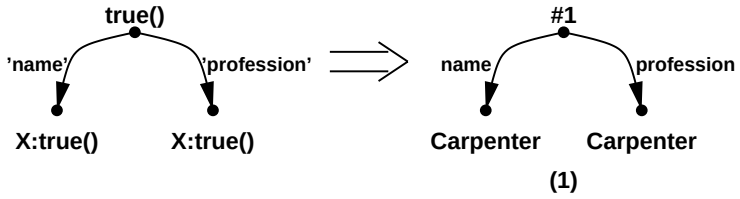

Figure 3.3: Adding variables

This section has been on a rather informal level. We will give a complete definition of schemata covering variable definitions and more concepts in the next section.

3.3 Adding paths

In this section we give our final definition of a schema. We further enhance the notion given before by representing paths, a very important concept in semistructured data. The reason behind is that it is often not known how the data is structured. For example, a date can be represented as an atomic value or it can be split into year, month and day. The latter makes it necessary to go one level deeper into the database. Skipping some levels can be achieved by making use of paths.

Let $G = (V, A, s, t)$ be a total directed graph. As defined before, a *trail* is an arc sequence $(a_{i_1}, \ldots, a_{i_m})$ where all a_{i_j} are distinct and there exist nodes $v_{i_0}, \ldots, v_{i_m}$, such that for all a_{i_j} $s(a_{i_j}) = v_{i_{j-1}}$ and $t(a_{i_j}) = v_{i_j}$. Note that this notion does not require the $v_{i_0}, \ldots, v_{i_m}$ to be distinct. The number of arcs in a trail is called the *length* of the trail. Despite the fact that we are talking about trails we denote the set of all trails in a graph by P and the set of nonempty trails by P^+, because from the intuition point of view we are talking about paths. For a nonempty trail $p_i = (a_{i_1}, \ldots, a_{i_m}) \in P^+$ we introduce a source and target function $s_P, t_P : P^+ \longrightarrow V$, which are defined in a canonical manner as $s_P(p_i) = s(a_{i_1})$ and $t_P(p_i) = t(a_{i_m})$, respectively.

The ultimate goal of this section is to give a notion of conformity between schemata and objects representing paths. To this end, we need some structural relationship between graphs that allows arcs to be in relationship with paths. As a first step toward achieving this goal we introduce the notion of a *corresponding trail graph*.

Definition 3.3 (Corresponding trail graph). The *corresponding trail graph* to a graph $G = (V^{(G)}, A^{(G)}, s^{(G)}, t^{(G)})$ is defined as $G_P = (V^{(G)}, P^{+(G)}, s_P^{(G)}, t_P^{(G)})$.

Intuitively, in the corresponding trail graph the trails are materialized as arcs. This notion is related to the notion of *transitive closure* of a graph as defined in [Jun90]. The only difference between the two notions is as follows: The transitive closure includes only one arc for every pair of reachable nodes, whereas we include an arc for every trail via which they are reachable. Nonetheless, the corresponding trail graph is always finite, because only finite many trails exist for any directed graph. Figure 3.4 shows three examples of directed graphs and their corresponding trail graphs.

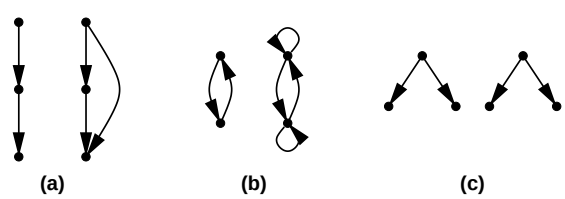

Figure 3.4: Three directed graphs and their corresponding trail graphs

Lemma 3.1. *A directed graph is always a subgraph of its corresponding trail graph.*

The lemma holds, because there is a natural embedding $a_i \longrightarrow (a_i)$ of the arcs in A into the trails in P^+. The following lemma is equally obvious.

Lemma 3.2. *If a directed graph G_1 is subgraph of G_2 then the corresponding trail graph of G_1 is a subgraph of the corresponding trail graph of G_2.*

Now we can extend our notion of schema. We introduce two additional functions q_{min} and q_{max}, that let us specify length constraints on paths in the matching objects. Furthermore, as a remnant from the previous section, we need a set of variables $\mathcal{V}$ and a variable mapping v.

Definition 3.4 (Schema). Given a set of labels $\mathcal{L}$ and a set of variables $\mathcal{V}$ a *schema* s is a tuple $(V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)}, v^{(s)}, q^{(s)}_{min}, q^{(s)}_{max})$ where

1. $V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)}$ are defined as before,
2. $v : V^{(s)} \cup A^{(s)} \longrightarrow \mathcal{V}$ is the *variable mapping*, a partial mapping from the nodes and arcs in the schema into the variables, and
3. $q^{(s)}_{min} : A^{(s)} \longrightarrow \mathbb{N}^+$ and $q^{(s)}_{max} : A^{(s)} \longrightarrow \mathbb{N}^+ \cup \{+\infty\}$ are *length restrictions.*

Furthermore, if for an arbitrary arc $a_i \in A^{(s)}$ a variable binding $v^{(s)}(a_i)$ exists, then $q^{(s)}_{min}(a_i) = q^{(s)}_{max}(a_i) = 1$ holds.

To assign some meaning to a schema we (re-)define the notion of conformity between schemata and objects.

Definition 3.5 (Conformity). Let $s = (V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)}, v^{(s)}, q^{(s)}_{min}, q^{(s)}_{max})$ be a schema and $o = (V^{(o)}, A^{(o)}, s^{(o)}, t^{(o)}, l^{(o)})$ be an object. A *match* of s into o is an isomorphic embedding of s into o_P, i.e., an isomorphic embedding of $(V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)})$ into $(V^{(o)}, P^{+(o)}, s^{(o)}_P, t^{(o)}_P)$, so that the following properties hold:

1. For all nodes $x \in V^{(s)}$ the predicate $l^{(s)}(x)$ is true for $l^{(o)}(m(x))$.
2. For all arcs $x \in A^{(s)}$ the predicate $l^{(s)}(x)$ is true for the labels $l^{(o)}(y_j)$ of all the arcs y_j in the trail $m(x)$.
3. For all elements $x_1, x_2 \in V^{(s)} \cup A^{(s)}$ for which $v^{(s)}(x_1)$ and $v^{(s)}(x_2)$ exist and $v^{(s)}(x_1) = v^{(s)}(x_2)$, the labels are the same $l^{(o)}(m(x_1)) = l^{(o)}(m(x_2))$.
4. For all arcs $x \in A^{(s)}$ the length of the trail $m(x)$ is at least $q^{(s)}_{min}(x)$ and no greater than $q^{(s)}_{max}(x)$.

If a match between a schema s and an object o exists we say that o *conforms* to s.

The following theorem states that we indeed enhanced our initial notion of schema, i.e., our new notion of schema does not contradict the initial one.

Theorem 3.3. *A predicate schema s conforms to an object o in the naive manner if and only if it conforms to o, assuming that $v^{(s)}$ is the empty mapping, and $q_{min}^{(s)}$ and $q_{max}^{(s)}$ equal one for all arcs in s.*

Proof. Let $s = (V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)})$ be a predicate schema. Let o be an object conforming to s in the naive manner and m' be the match between s and o. Then we can construct a match m between s and o in the following manner:

$$m(x) := \begin{cases} m'(x) & \text{if } x \in V^{(s)} \\ (m'(x)) & \text{if } x \in A^{(s)} \end{cases}$$

First, we observe that we indeed defined a graph morphism m. For every arc x the property $s_P^{(o)}(m(x)) = m(s^{(s)}(x))$ holds, because $s_P^{(o)}(m(x)) = s_P^{(o)}((m'(x))) = s^{(o)}(m'(x)) = m'(s^{(s)}(x)) = m(s^{(s)}(x))$ and m' being a graph morphism. Analogously, $t_P^{(o)}(m(x)) = m(t^{(s)}(x))$ holds.

Now we check if m fulfills the conditions of a match. The condition on node labels holds for m, because it holds for m'. An arbitrary arc $x \in A^{(s)}$ is mapped to $(m'(x))$. This trail has always length one, hence the fourth condition on trail lengths holds. The condition on arc labels also holds, because it holds for m'. Finally, the condition on variable bindings holds, because the variable mapping is empty.

Vice versa, let m be a match of s into o; and let $v^{(s)}$ be the empty mapping, and $q_{min}^{(s)}$ and $q_{min}^{(s)}$ equal one for all arcs in s. Then we can construct a graph morphism m' between s and o as follows:

$$m'(x) := \begin{cases} m(x) & \text{if } x \in V^{(s)} \\ first(m(x)) & \text{if } x \in A^{(s)} \end{cases}$$

In this definition, $first$ returns the first arc of a trail. Note, that $m(x)$ always contains exactly one arc, because $q_{min}^{(s)}$ and $q_{max}^{(s)}$ equal one. Thus, m' preserves source and target, because m does so as well, i.e., $s^{(o)}(m'(x)) = s^{(o)}(first(m(x))) = s_P^{(o)}(m(x)) = m(s^{(s)}(x)) = m'(s^{(s)}(x))$ and the same for the target function. Furthermore, for all $x \in V^{(s)} \cup A^{(s)}$ the predicate $l^{(s)}(x)$ holds for $l^{(o)}(m'(x))$, because it holds for $l^{(o)}(m(x))$ if x is a vertex and for all $l^{(o)}(m(x))$ if x is an arc. □

Consider the example in Figure 3.5. There is a '+'-sign on the first arc in the schema. It indicates that the length of the paths it matches is bound by 1 and $+\infty$. Thus, the schema intuitively matches every object that emanates from the root via a path of positive length and leads to a 'name'-arc. We observe that the second person

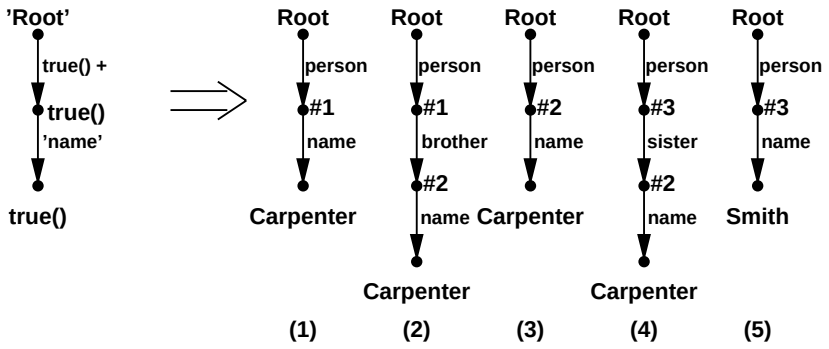

Figure 3.5: Adding paths

in the database can be reached via different paths: either "directly" from the root, or via the sibling relationships. Note that we started to use the term "path", although strictly speaking we meant "trail". We shall use the term path when talking about the generally useful concept in semistructured data, but we will continue to use the term trail when mathematical correctness is needed.

Let us look at this example in more detail. Figure 3.6 shows the same schema together with the second match from the database graph. Remember, that the match is a subgraph of the corresponding trail graph of the database. The match of the schema is indicated by solid lines, whereas the dashed lines represent arcs from the rest of the corresponding trail graph of the database.

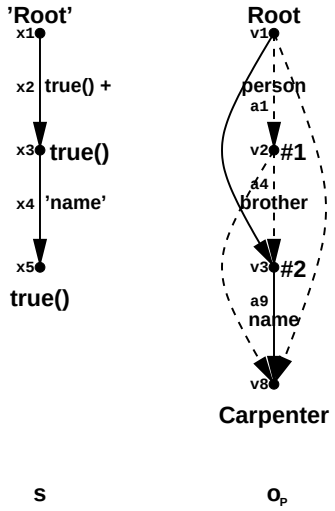

Figure 3.6: A more detailed look into conformity

The schema can formally be written as:

$$
\begin{aligned}
V &= \{x_1, x_3, x_5\} \\
A &= \{x_2, x_4\} \\
s &= \{(x_2, x_1), (x_4, x_3)\} \\
t &= \{(x_2, x_3), (x_4, x_5)\} \\
l &= \{(x_1, 'Root'), (x_2, true()), (x_3, true()), (x_4, 'name'), (x_5, true())\} \\
v &= \{\} \\
q_{min} &= \{(x_2, 1), (x_4, 1)\} \\
q_{max} &= \{(x_2, +\infty), (x_4, 1)\}
\end{aligned}
$$

The match m between the schema and the part of the database is now as follows:

$$
\begin{aligned}
m = \{&(x_1, v_1), (x_3, v_3), (x_5, v_8), \\
&(x_2, (a_1, a_4)), (x_4, (a_9))\}
\end{aligned}
$$

The example makes some subtleties apparent. A match of s in o is supposed to be a subobject of o. However, the scope $m(s)$ of the match function m is a subobject of o_P. These subtleties become a serious problem when we adapt the definition of minimal match. The notion of minimal match is important for the definition of queries as we will see in the next section. Consider Figure 3.7. (We omitted the node labels, because they are not relevant to this problem.) The schema on the left is matched into

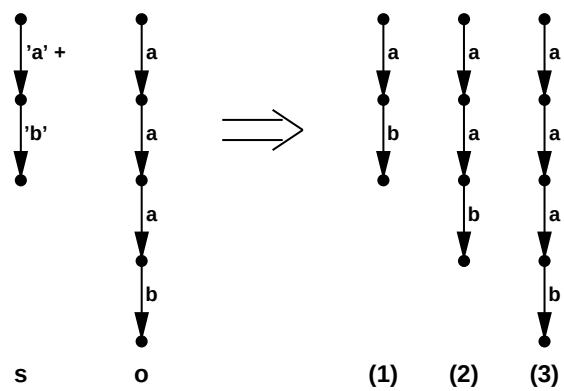

Figure 3.7: A problem with minimal matches

the database right beside it. All the three matches are potentially "interesting", but only the first one is minimal, because it is a subobject of the others. Beside, if one of the matches was more interesting than the others, wouldn't it be the one with the longest path, i.e., the one on the right? But we observe, that all the three matches result from different match functions. The scopes of their respective match functions

are incomparable subobjects of o_P. Thus, we define minimal matches with respect to the match function. To achieve this we need a $flatten$-function that takes a subobject of o_P and produces a subobject of o. Informally, $flatten$ decomposes the trails into arcs and adds all source and target nodes to the node set. Formally, let o' be a subobject of o_P then:

$$\begin{aligned} flatten(o') :=& (V^{(o')} \cup \{v | \exists a \in A^{(o')} : \exists a_i \in a : v = s^{(o)}(a_i) \vee v = t^{(o)}(a_i)\}, \\ & \{a_i | \exists a \in A^{(o')} : a_i \in a\}, \\ & s^{(o)}|_{V^{(flatten(o'))} \cup A^{(flatten(o'))}}, t^{(o)}|_{V^{(flatten(o'))} \cup A^{(flatten(o'))}}, \\ & l^{(o)}|_{V^{(flatten(o'))} \cup A^{(flatten(o'))}}) \end{aligned}$$

Now, we can define the set of *minimal matches* of s in o as:

$$\mathfrak{M}^{(s)}_{min}(o) := \{flatten(m(s)) | m \text{ is a match of } s \text{ into } o\}$$

We observe, that every $flatten(m(s))$ is indeed a match of s in o, because s can be embedded into $flatten(m(s))_P$ using m. Furthermore, the following lemma says that we indeed gave a useful definition of minimal matches.

Lemma 3.4. *For every match o' of s in o (i.e., every element of $\mathfrak{M}^{(s)}(o)$) there exists a minimal match in $\mathfrak{M}^{(s)}_{min}(o)$ being a subobject of o'.*

Proof. Let o' be a match of s in o. Hence, there exists a match function m embedding s into o'_P. Because $o' \subseteq o$ and thus $o'_P \subseteq o_P$, the mapping m embeds s also into o_P. Hence, $flatten(m(s))$ is an element of $\mathfrak{M}^{(s)}_{min}(o)$. If we can show, that $flatten(m(s)) \subseteq o'$ holds, the lemma is proven.

Let v be a node in $flatten(m(s))$. We can distinguish two cases: If v is also in $m(s)$ then there exists a node v_s in s, such that $m(v_s) = v$. This implies $v \in V^{(o'_P)}$; and thus, $v \in V^{(o')}$. If v is not in $m(s)$ it was introduced by the $flatten$-function. In this case there exists an arc a in a trail $(a_1, \ldots, a_k) \in m(s)$, such that v is either the source or the target of a. Because $A^{(o')} \subseteq A^{(o)}$ and $(a_1, \ldots, a_k) \in A^{(o'_P)}$, all the a_js are in $A^{(o')}$. In particular, $a \in A^{(o')}$. Because o' is an object, v is in $V^{(o')}$.

Let a be an arc in $flatten(m(s))$. There exists an arc a_s in s, such that $a \in m(a_s)$. Because $m(a_s) \in A^{(o'_P)}$ and $A^{(o')} \subseteq A^{(o)}$, it follows that all arcs in $m(a_s)$ are also in $A^{(o')}$. In particular, $a \in A^{(o')}$. We have shown, that $V^{(flatten(m(s)))} \subseteq V^{(o')}$ and $A^{(flatten(m(s)))} \subseteq A^{(o')}$; and because both $flatten(m(s))$ and o are objects, the proposition $flatten(m(s)) \subseteq o'$ follows. □

With the revised definition of minimal match all the three matches on the right hand side of Figure 3.7 are minimal.

Up to this point we have only talked about nonempty trails and paths. The reason for this is that it makes many things easier, for instance to introduce source and target

functions. However, if we take a closer look we realize that incorporating the empty path is not difficult. The key point to realize is, that in an arbitrary graph there is no single empty path, but rather $|V|$-many empty paths, i.e., one for every vertex. Formally, they are walks consisting of exactly one vertex. Now things become simple. The notions of source and target function are well-defined and the notion of corresponding trail graph can easily be adapted. Thus, the empty path can be integrated into the notions of schema and conformity.

In this section we have given full definitions of the notions of schema and conformity between schemata and objects. To incorporate path descriptions we have introduced the notion of a corresponding trail graph. We have proven that our new notion of conformity is not contradictory to, but rather an enhancement of the previous notion of naive conformity. Finally, we revised to notion of minimal match.

3.4 Other notions of schema

Other people's work in this area concentrates on defining a complete schema for a semistructured database. An important example of such a schema is the DataGuide [GW97] used in the Lore project. They point out, that a schema serves two important purposes. First, a schema enables users to understand the structure of the database and form meaningful queries over it. Second, a schema can help the query processor to devise efficient plans for computing query results. Hence, a DataGuide is a hybrid concept between schema and index. It is intended to be a concise, accurate and convenient summary of the structure of a database. Conciseness means, that every unique label path of the source appears exactly once in the DataGuide. Accuracy means, that the DataGuide does not encode a label path that does not appear in the source. Convenience means, that a DataGuide itself can be treated as an OEM object, the basic data model of Lore.

Creating a DataGuide over a source database is equivalent to conversion of a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA) [NUWC97]. From automata theory it is known, that a single NFA may have many equivalent DFAs. Similarly, one source OEM database can have multiple DataGuides. This fact is illustrated in Figure 3.8. The DataGuide on the right is in fact minimal in the sense that no other DataGuide with fewer nodes exists. Minimal DataGuides are, however, not the best possible choice when it comes to the question of which DataGuide to choose. First, incremental maintenance of a minimal DataGuide can be very expensive. Just imagine in Figure 3.8 a new child object to object 10 being added. Second, minimal DataGuides are harder to annotate.

Therefore the notion of a strong DataGuide is introduced. A DataGuide is strong if

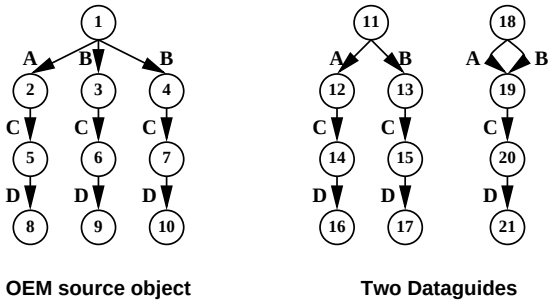

Figure 3.8: An OEM source together with two DataGuides

each set of label paths, that share the same (singleton) target set in the DataGuide, is exactly the set of label paths, that share the same target set in the source. The minimal DataGuide in Figure 3.8 is not strong. $B.C$ has the target set $\{6, 7\}$ in the source. No other label path in the source has this target set. In the minimal DataGuide, however, $B.C$ has the target set $\{20\}$, which is also the target set of $A.C$. The first DataGuide in Figure 3.8 is strong.

A strong DataGuide guarantees a one-to-one mapping between target sets in the source and DataGuide objects. This makes strong DataGuides easy to compute and to maintain. This property is also of use for query optimization, because a strong DataGuide can serve as path index. In polynomial time to the length of a path, a strong DataGuide can be used to find all source objects reachable via that path. This property holds independently of the size of the source.

A DataGuide can also be used for query formulation in a QBE-like manner [Zlo77]. In Lore the user can interactively explore the DataGuide and formulate queries using the DataGuide. No knowledge of the Lore language is necessary.

Because DataGuides can be expensive to compute for large, cyclic databases, their definition has recently be relaxed. An approximate DataGuide may have "false positives, i.e., it is no longer required that all DataGuide paths must exist in the source database [GW99].

Theoretical research results within the UnQL project are presented in [BDFS97]. They give a formal definition of graph schema based on bisimulation between graphs (rather than isomorphy). Bisimulation relaxes the constraints in the way, that a relation between nodes is required rather than a mapping. The advantage of this approach is, that simulations can be computed in polynomial time [HHK95]. The disadvantage is, that the notion of isomorphy seems to reflect the notion of conformity in a more natural manner. Their approach is similar to ours in that predicates are used as labels in the schema, and similar to DataGuides in that a schema always models the complete database. Subsumption, ordering, and equivalence of schemata are discussed in some

detail. The knowledge about such schemata can be used for optimization in the query language UnQL.

There is no real notion of schema in the context of XML yet, although there are many activities within this field. Document Type Definitions are more in the flavor of grammars. In Figure 3.9 we present a DTD for articles that was first presented in [CACS94]. As an example a section can either consist of a title and at least one "body" (a figure or a paragraph) or of a title, an arbitrary number of bodies and at least one subsection.

```
<!DOCTYPE article [
<!ELEMENT article  - - (title, (author+), affil, abstract,
        (section+), (bib & ack))>
<!ELEMENT title  - O (#PCDATA)>
<!ELEMENT author  - O (#PCDATA)>
<!ELEMENT affil  - O (#PCDATA)>
<!ELEMENT abstract  - O (#PCDATA)>
<!ELEMENT ack  - O (#PCDATA)>
<!ELEMENT bib  - O (#PCDATA)>
<!ELEMENT section  - O ((title, (body+)) |
        (title, (body*), (subsectn+)))>
<!ELEMENT subsectn  - O (title, (body+))>
<!ELEMENT body  - O (figure | paragr)>
<!ELEMENT figure  - O (picture, (caption?))>
<!ELEMENT picture  - O (#PCDATA)>
<!ELEMENT caption  - O (#PCDATA)>
<!ELEMENT paragr  - O (#PCDATA)>
]>
```

Figure 3.9: A Document Type Definition for articles

How grammar-like DTDs and database schemata can be integrated will be one of the challenging research issues in the near future.

Our notion of schema is significantly different from all of the above work. First, our schemata include advanced concepts, that are typical for query languages. Among them are variable definitions and path descriptions. In a sense, a schema in our approach is more like a view. Second, we believe, that partial schemata (rather than complete schemata) are more appropriate within the field of semistructured data. Although complete schemata are closer to the traditional notion of a database schema we believe, that for semistructured databases they will lead to some unwanted results. A

complete schema will be useful for serving as an index. Due to the irregular structure of semistructured databases, however, a complete schema will cover many exceptions. Thus, it will be much larger than it ought to be. Additionally, it will not be very useful in giving users information about the database, because the many exceptions cannot be distinguished from the representative parts of the schema.

3.5 Summary

This chapter presented our notion of schema, which forms the base for queries. We incorporate rich semantical concepts typically found in query languages into our notion of schema. As a first step we introduced a simple notion of schema. A predicate schema is an object labeled with unary predicates. The notion of conformity between a predicate schema and an object is based on the graph theoretic notion of isomorphic embedding. Gradually we integrated variable definitions and path descriptions into our notion of schema. The notion of conformity between a schema and an object is based on an isomorphic embedding of the schema into the corresponding trail graph of the database, which is similar to a graph closure of the database. We close the chapter with a discussion of similar notions, in particular DataGuides used in the Lore project.

Chapter 4

Queries and Answers

Computers are useless. They can only give you answers.
(Pablo Picasso)

This chapter concludes the definition of the query language of our approach. All the queries we introduce are based on matching a schema as introduced in the previous chapter. Whereas the schemata form the "What"-part of a query, the operations defined in this chapter form the "How"-part. By splitting a query in this manner we set up a more natural environment. It can for instance be used for query optimization. We store and reuse the "What"-part of a query, i.e., the schema.

In Section 4.1 we observe that a schema itself is already a simple form of a query. We introduce the notion of answer. The first operation, the focus, is introduced in Section 4.2. Section 4.3 shows how a schema match can be restructured completely. The idea is based on graph transformations, which we briefly introduce in Section 4.5. Before that, we examine the expressiveness of our query language in Section 4.4. We look at other query languages for semistructured data in Section 4.6 and finish the chapter with a summary in Section 4.7.

4.1 Simple schema queries

We observe that a schema itself already forms the most simple kind of query. It queries all subobjects of a database that conform to it.

Definition 4.1 (Schema query). A *schema query* is a tuple $q = (s)$ where s is a schema.

When we pose such a query we are usually not interested in all matches of the schema, but only in the minimal ones.

Definition 4.2 (Answer). The *answer* to a schema query $q = (s)$ with respect to a database o is the set of minimal matches of s in o, i.e., $\mathfrak{M}_{min}^{(s)}(o)$.

As an example of a schema query you can imagine any of the schemata from the previous chapter.

4.2 Adding a focus

With a schema query we are able to formulate conditions. This roughly corresponds to a selection operation in the relational world. It, again, reflects the "What"-part of a query. Now we touch the subject of the "How"-part of a query. With a schema query you can express something like the following SQL query:

```
SELECT *
FROM Person
WHERE name = 'Carpenter'
```

However, a query including a projection operation, e.g., something like

```
SELECT surname
FROM Person
WHERE name="Carpenter"
```

cannot be expressed. In this type of query we explicitely state that we want only the surnames of the persons, i.e., we explicitly say how we want the answer to look like. Hence, we are talking about the "How"-part of a query. We start by introducing an operation that is comparable to the projection operation of the relational world. We give a focus to the schema forming the base of the query.

Definition 4.3 (Focus query). A *focus query* is a tuple $q = (s_1, s_2)$ where s_1 is a schema and s_2 is a subobject of s_1. We call s_2 the *focus* of the query.

Definition 4.4 (Answer). The *answer* to a focus query $q = (s_1, s_2)$ with respect to a database o is the union of the minimal matches of s_2 over all minimal matches of s_1 in o, i.e., $\bigcup_{x \in \mathfrak{M}_{min}^{(s_1)}(o)} \mathfrak{M}_{min}^{(s_2)}(x)$.

As an example we use the second of the preceding SQL-statements. The query in Figure 4.1 queries for the surnames of all persons with the name 'Carpenter'. The focus of the query is indicated by the dashed box.

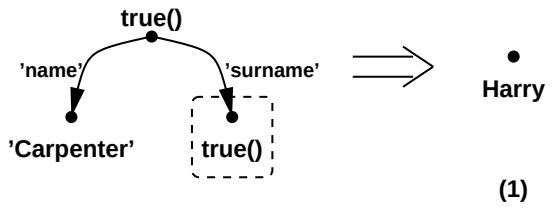

Figure 4.1: A focus query

4.3 Transforming the answer

Sometimes we prefer to restructure the answer to a query completely. To achieve this we adopt concepts from the area of graph transformations. In Section 4.5 we look into this area in more detail. The general idea of the area of graph transformations is the rule-based manipulation of a host graph. In our context, the idea is that the user can specify a graph structure for the query answers. Furthermore, new labels can be computed by using terms over the old ones. With the operation that we describe in this section, we can express something like the following, which could not be expressed before.

```
SELECT id, (1999 - yearOfBirth) AS age
FROM Person
WHERE name = 'Smith'
AND surname = 'Suzy'
```

Definition 4.5 (Transformation query). A *transformation query* is a tuple $q = (s, t)$ where s is a schema and t is an object labeled with terms over the elements in s.

Definition 4.6 (Answer). The *answer* to a transformation query $q = (s, t)$ with respect to a database o is built by creating for every match of s in o, i.e., for every element of $\mathfrak{M}^{(s)}_{min}(o)$, a new object isomorphic to t, labeled with the evaluated terms of t, instantiating the terms by using the match.

Again, we use the preceding SQL-statement as guidance for our example in Figure 4.2. It queries for the age of Suzy Smith. The age is derived from the year of birth.

Because this part of the work is conceptual we are not going into detail about specific terms, typing and applicability of terms. Of course, the term $1999 - x_7$ we use in the example, is only applicable to instantiations of x_7 that are numbers. Correct typing can for instance be enforced via the predicates in the schema. So we used the predicate $integer()$ at x_7 instead of, say, $true()$.

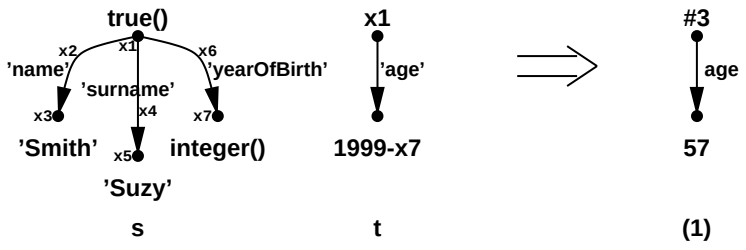

Figure 4.2: A transformation query

The transformation query forms the most general query in our approach. Note that schema and focus queries can be expressed as transformation queries.

This concludes the introduction of our query language. The types of queries proposed in this section should be seen as instances of the operational layer as described in the introduction in Chapter 1 or, which is essentially the same, as the "How"-part of a query.

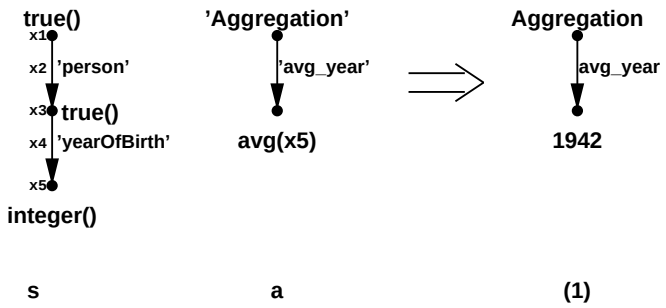

Figure 4.3: An idea to introduce aggregation

An obvious limitation of our approach is that every type of query always returns one answer per schema match. Thus, we currently do not support aggregation. A first idea how to overcome this limitation is to adapt the notion of transformation query. Figure 4.3 presents an aggregation query. The matches of the schema on the left are not presented one at a time. Rather, they are collected and the aggregation on the labels is performed and one result graph is presented. Again, the same problems of applicability of specific functions to specific kinds of labels arise. Furthermore, paths introduce another kind of problem. If some node included in aggregation is reached via different paths, it will appear several times during the aggregation.

Our language was presented in [BF99b], and in a preliminary form in [BF98]. We think that our approach can also be extended to cover restructurings of a database. To this end, ideas from the area of graph transformations can play an even more significant role.

4.4 On the expressiveness of our query language

This section takes a deeper look at the expressiveness of our query language. We concentrate on comparing it to the operations of the relational algebra, because they are well-known and well-studied. On the other hand, we also investigate peculiarities of the semistructured world by looking at some XML-related examples.

First, let us classify the operations of the relational algebra into "What" and "How" as well. Our result is shown in Table 4.1. Only the crossproduct is not so easy to classify,

Operation	"What"	"How"
Selection	x	
Projection		x
Crossproduct	(x)	(x)
Union	x	
Difference	x	

Table 4.1: Classification of the relational operators

it does not really belong solely to either class. The set operations are not part of our approach. If they were they should probably be part of the schema language, because they clearly belong to the "What"-part of a query.

We move on to show how these operations can be performed using our approach. We will use the example shown in Figures 2.5 and 2.6 on Page 35.

Selection As an example we want to have all the information belonging to the person with the identifier '01', i.e., we want the answer to the query $\sigma_{ID='01'}(Person)$. This can be done using a simple schema query or, optionally, a focus query. The schema we need is shown in Figure 4.4. Optionally, a focus, for instance the one indicated by the dashed box, can be specified. One problem is the need to specify all the attributes

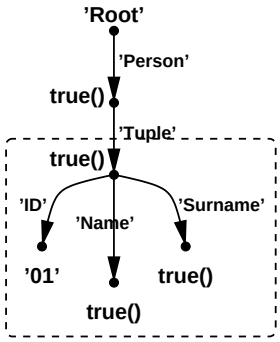

Figure 4.4: Representing the selection

explicitely. This is due to the fact that in an arbitrary graph we cannot know whether a certain node has only outgoing arcs that lead directly to leaf nodes. The sample database we are using for this example is well-structured; every leaf node has exactly depth three.

Projection We use the query $\pi_{Name}(Person)$ as an example. Contradictory to what we have said so far, e.g., with Table 4.1, we do not necessarily need a focus query, i.e., concepts belonging to the "How"-part of a query, to express the projection. In our approach, wanted attributes have to be "projected in" (see the preceding example for the selection operation), whereas in relational algebra unwanted attributes are "projected out". Furthermore, our example shows a "pure" projection, i.e., one that is not combined with a selection. We can again express this query using a simple schema query or, optionally, a focus query. Figure 4.5 shows the schema.

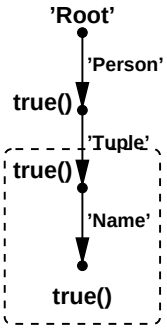

Figure 4.5: Representing the projection

Crossproduct How can we express $Person \times WorksOn$? Now we definitely need a transformation query. The schema on the left of Figure 4.6 produces on match per tuple in the crossproduct of the two relations. This match has to be transformed in such a way, that it appears as a tuple of one result relation. The term-labeled graph on the right does exactly that. Keep in mind, that you get one graph isomorphic to it per match of the schema. By the way, the complexity of this example is a consequence of the redundant representation of relational data, which is a necessity in graph models.

Join As a special case of what we have seen so far, we demonstrate the natural join $Person \bowtie WorksOn$. The selection involved in this join compares labels in the database graph to other labels. This is different from what we have seen in the example about the selection, where we compared labels to constants or checked predicates on the labels. The only difference in the schema in Figure 4.7 compared to the one in Figure 4.6

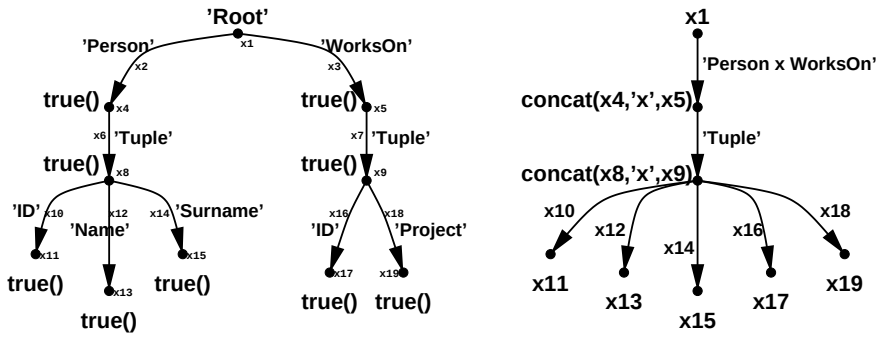

Figure 4.6: Representing the crossproduct

is that a variable links the two nodes representing the ID's. The term-labeled graph on the right is adapted accordingly; we need one "column" less.

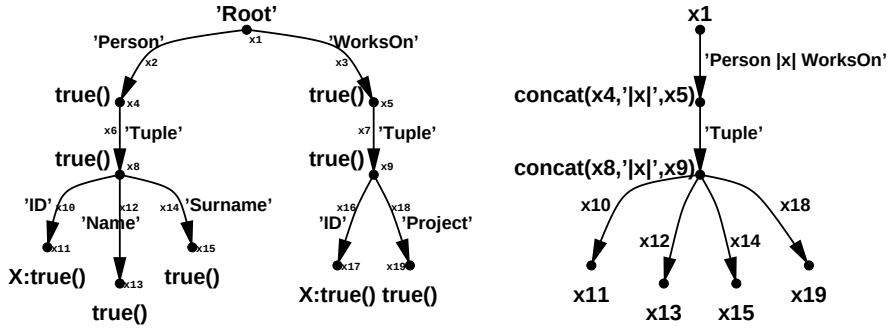

Figure 4.7: Representing the natural join

Set operations The set operations lead to some unexpected problems. Of course, one can easily introduce set operations on sets of schema matches. Then a query, such as $\pi_{Name}(Person) \cup \pi_{Surname}(Person)$, can easily be expressed and give the results we expect. However, with the difference operation things are a little more subtle. Let us use $\pi_{ID}(Person) \setminus \pi_{ID}(WorksOn)$ as an example. We would want to get $\{(03)\}$ as the answer, because only the person with the identifier '03' is not involved in any projects. In our approach, however, we would get $\{(01), (02), (03)\}$ as the answer. The reason is that the identifiers of the Person-part of the database and the ones of the WorksOn-part of the database are strictly distinct, because they are unique nodes, i.e., they are different subgraphs of the database. The semantics of our approach is strictly identity-based rather than value-based. This also influences the projection operation. Our projection behaves like the `SELECT`-statement in SQL rather than like the `SELECT DISTINCT`-statement. Using a composition of queries we can define a little workaround for this problem. We can use the semijoin to find the identifiers in the $Person$ table that have partners in the $WorksOn$ table. Afterward we can perform

the difference operation solely on the identifiers of the $Person$ table, which will lead to the intended result. Altogether, we simulate the query $\pi_{ID}(Person) \setminus \pi_{ID}(Person \ltimes WorksOn)$. Although the difference operation can be used to define the intersection operation (i.e., $R \cap S = R \setminus (R \setminus S) = S \setminus (S \setminus R)$) we would like to point out that in our approach the result would be a "table-specific" intersection, i.e., the result would be either the intersection using the tuples from R or using the tuples from S.

Other issues Due to the very nature of semistructured data, our approach can be used to link data and structural parts of the database. This cannot be achieved using relational algebra. The schema in Figure 4.8 matches tuples having the same value at the same attribute. The answer to this schema query with respect to the database in Figure 2.5 on Page 35 consists of the tuples with $ID = 01$ and $ID = 02$ from the tables $Person$ and $WorksOn$, respectively. Due to the injectivity requirement of the match function (in particular with respect to the arcs indicating the relations) this schema does not match the two different tuples with $ID = 02$ in table $WorksOn$.

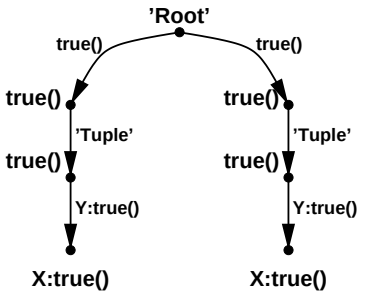

Figure 4.8: Equal attribute-value pairs

Examples involving paths can also not be expressed in relational algebra; however, they are not very useful in such a structured and unnested environment.

Querying XML documents Of course, our language is well-suited for querying XML documents. With the predicates, the variable definitions, and the path descriptions general concepts of other languages for semistructured data are represented. A special feature of our approach, however, is that no knowledge of a root node or of specific paths going out from the root node is required. Consider the simple example in Figure 4.9 together with the XML document covering various kinds of information on our database group. This document was introduced in Section 2.4. The schema matches every object with a name, a surname, and a phone number, regardless of where they appear in the document.

Due to the way XML documents are transformed into labeled directed graphs, different parts of a document can be linked. Remember, that the example document

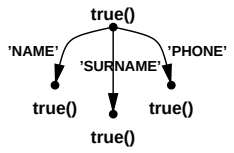

Figure 4.9: A simple schema query for XML documents

basically consists of four parts covering the member, the research, the lectures, and the publications of our group. The example schema in Figure 4.10 matches the title of every publication in 1999 I am involved in. The arc labeled $true()+$ matches the linking path `AUTHORS.AUTHOR.idref`. Again, no knowledge of a possible path from the root node is necessary.

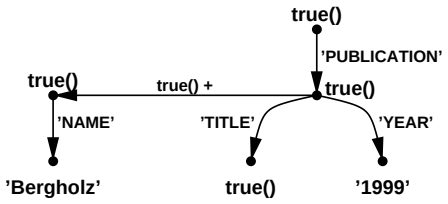

Figure 4.10: Linking different parts of an XML document

4.5 Intermezzo: Graph transformations

In this section we will provide a brief introduction to the field of graph transformations. This area has significantly influenced our work, in particular on concepts like the transformation query, but also on query processing using constraints. In a sense, we extend the notion of graph transformation through our schema concept and provide a richer notion of left-hand sides for rules. We will concentrate on the *Single-Pushout* approach to graph transformations [Löw93].

Graph transformations are about making changes to some host graph. In the context of graph transformations graphs are usually attributed. Every vertex and arc in a graph is associated with a type; and the label of the vertex or arc must be of this type. Now, a *graph rule* is simply a graph morphism $r : L \longrightarrow R$. On the left-hand side a constellation of objects to be rearranged is described. This is related to what we call a schema. The morphism describes the changes to be done. Objects being involved in the morphism are preserved; all other objects are deleted from the host graph. In addition, new objects can be created by the morphism. A *redex* of a left-hand side L in a host graph G is a total graph morphism $m : L \longrightarrow G$. The redex indicates occurrences of the left-hand side of a rule in the host graph. This morphism has to

be total, because only in this case the conditions formulated in the left-hand side are completely fulfilled in the host graph. Of course, the host graph can, and typically will, be much larger than the left-hand side of a rule. Hence, there exists typically more than just one redex for every rule, just as there exist multiple matches of a schema in a database. The part of the host graph, that is not in the scope of the redex, is called the *context*.

The result of applying a graph rule $r : L \longrightarrow R$ to a host graph G using a redex $m : L \longrightarrow G$ can formally be described using the pushout concept from category theory (hence the name Single-Pushout). Instead, we outline the construction of the result for the non-attributed case and give an example later on. The result graph H is constructed in two steps:

1. Glue and add: Every object in the left-hand side L of the rule r has a corresponding object in the host graph G via the redex m. We add all the objects to G, that are added by r to R. Furthermore, we glue objects being unified by m together; i.e., let $x, y \in L$ be glued (i.e., $m(x) = m(y)$), then everything, that is added in R to $r(x)$ and $r(y)$, is added to the single object $m(x) = m(y)$ in G.

2. Delete: We delete all the objects $m(x)$ for which no $r(x)$ exists. During this process dangling arcs can occur. They are also deleted. The constructed graph is the result graph H.

Note that a graph rule specifies three components: the part of the graph to be deleted, i.e., $L \backslash dom(r)$, the subobject of L to be preserved, i.e., $dom(R)$ and the added structure, i.e., $R \setminus r(L)$.

A rule, such as the one in Figure 4.11, can be formulated in the graph transformation system AGG [Agg]. A box stored in a depot is to be put onto a vehicle. The morphism r between the left-hand side of the rule and the right-hand side of the rule is implicitly given by the object names, i.e., the depot object on the left is mapped to the depot object on the right etc. Note that rules can be accompanied by attributes. After the application of a rule the number of boxes in the depot is reduced by one and the weight of the box is added to the weight of the vehicle. In Figure 4.12 we show a host graph before and after the application of the rule in Figure 4.11. There are two vehicles, one on a parking lot and one in front of the depot. Three boxes carrying a variety of goods are in the depot and on the second vehicle, respectively. After applying the rule one box is moved from the depot to the second vehicle. Note that the values of the attributes are updated accordingly. In the example we observe that an arbitrary redex is used in this transformation. Instead of the box with cookies the box carrying apples could have been moved to the vehicle as well.

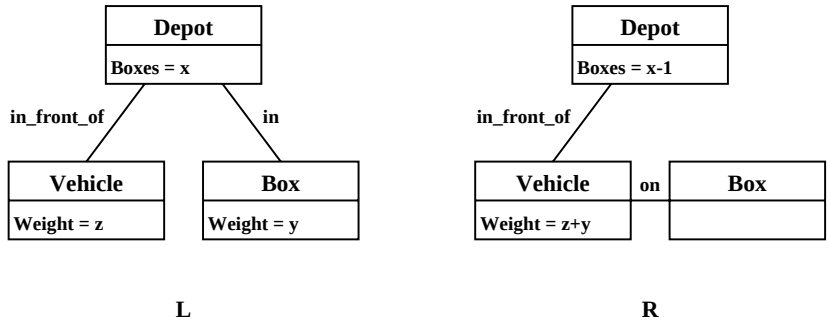

Figure 4.11: An attributed graph rule

To avoid unwanted results one can specify *application conditions* for redices. A redex m is called

- *injective*, if $m(x) = m(y)$ implies $x = y$ for all $x, y \in L$. An injective redex never glues nodes together.
- *d-injective* (from delete-injective), if $m(x) = m(y)$ implies $x = y$ or $x, y \in dom(R)$. D-injectivity requires a one-to-one correspondence between candidates for deletion in G and L.
- *conflict-free*, if $m(x) = m(y)$ implies either $x, y \in dom(R)$ or $x, y \notin dom(R)$. This guarantees that an element of G is either meant to be preserved or meant to be deleted. D-injectivity ensures conflict-freeness. A conflict-free redex is also said to fulfill the *identification condition*.
- *d-complete*, if for every edge $e \in G$ with $s(e) \in codom(m|_{L \setminus dom(r)})$ or with $t(e) \in codom(m|_{L \setminus dom(r)})$ also $e \in codom(m|_{L \setminus dom(r)})$ holds. This condition ensures, that the complete structural context of the elements in G to be deleted, is described in L, i.e., no dangling edges can occur.

As opposed to the Single-Pushout approach the older *Double-Pushout* approach makes use of a gluing graph [EPS73]. A graph rule is thus a tuple (L, K, R) and K, typically a subgraph of both L and R, describes the part of the left-hand side that is to be preserved by an application of the rule. A rule is applicable to a host graph G if G contains a homomorphic image of L. In the first step, the application of the rule removes the part of G that corresponds to $L - K$. This leads to the context graph D. Then all items in $R - K$ are added to D. Finally, these new items from $R - K$ are embedded into D in accordance with the relation between $R - K$ and K.

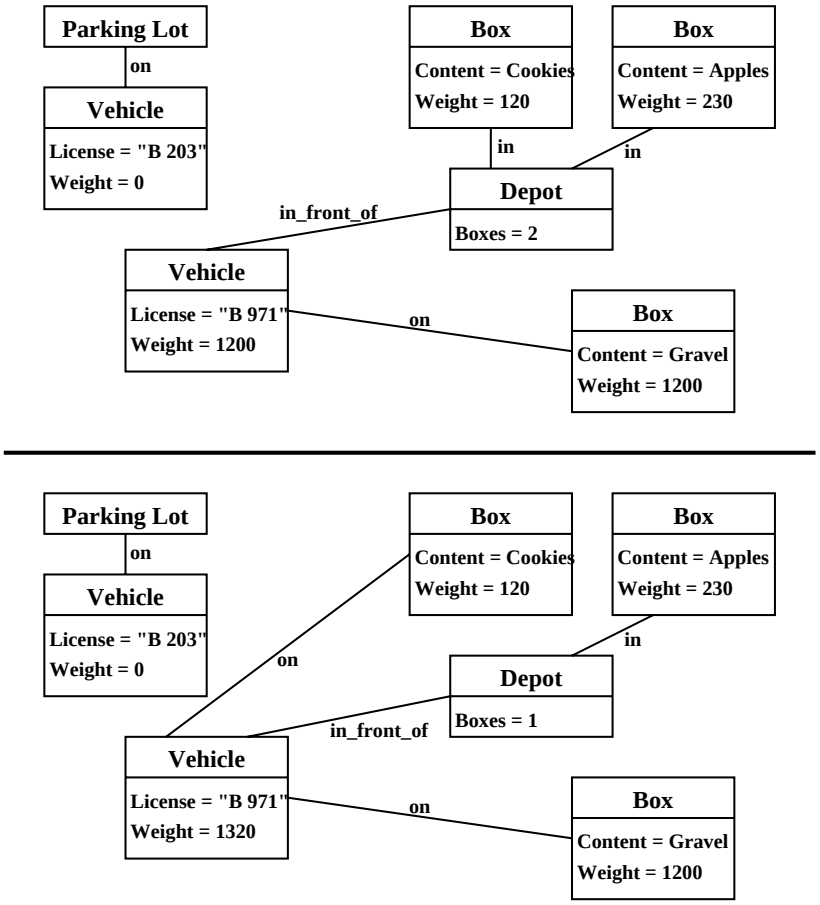

Figure 4.12: The host graph before and after the application of the rule

4.6 Other query languages for semistructured data

In this section we summarize other query languages for semistructured data. We primarily present two languages, Stanford's Lorel [AQM+97] and University of Pennsylvania's UnQL [BDHS96].

Lorel Lore is a pioneering project in semistructured data. Originally, the language Lorel (i.e., Lore language) was implemented on top of the object-oriented DBMS O_2 [BDK92a]. Now it has been rebuilt as a stand-alone system. We describe Lorel in its refined version called Lorel96 as presented in [AQM+97]. The original language, now dubbed Lorel1, was presented in [QRS+95]. Lorel was designed to be a query language for the Object Exchange Model OEM, but it can now also be used to query XML documents.

We describe Lorel's functionality using an example with data on restaurants listed in some guide. Lorel supports OQL-like basic functionality as shown in the next example.

The user can specify conditions; they can also be connected using boolean connectors.

```
select Guide.restaurant
(from Guide)
where Guide.restaurant.entree.name = "Green_curry";
```

We observe that a `from`-clause is not really necessary. Furthermore, equality is a difficult issue in this language. The path expression `Guide.Restaurant.Entree.Name` returns a set of objects if there exist entrees with several names. In that case the comparison would return true if there existed a name in the set of names equaling the value.

A very important concept of Lorel is that of a path expression. It is used for a navigational querying of the data. Label marker can be used to fix paths or prefixes. They can be useful if the user for instance wants to query a book with two authors. Then each of the two author paths has to be distinguished. Label markers can also make the semantics of a query clearer. The previous example query becomes clearer if written like this:

```
select Guide.restaurant R
where R.entree.name = "Green_curry";
```

General path expressions allow both regular expressions and label completion to be used in paths. Consider the following examples of path expressions.

```
Guide.restaurant(.address)?.zip%
Guide.restaurant.#@P.name
Guide.restaurant(.nearby)*{R}.name
```

The first expression matches paths starting from `Guide` followed by a `restaurant` edge, then an edge with a label beginning with `zip` (the intention being `zip` or `zipcode`, of course), possibly with an `address` edge inbetween. The second expression matches every path starting with `Guide.restaurant` and ending with a `name` edge. An arbitrary subpath can occur inbetween. This subpath is matched to `#` and bound to the variable `P`. In the third example the variable `R` is bound to the object immediately before the `name` edge. It is obvious that these kinds of path expressions can be used to pose powerful queries, such as the one presented next.

```
select R.name
from Guide.restaurant R
where R.zip% = 92310
and R.% = "cheap"
```

Label markers can be used for at least two more purposes. The `path-of` function allows the user to discover the structure of the database. The next query would return paths, such as `restaurant`, `restaurant.address`, or `restaurant.nearby.address`.

```
select distinct path-of(P)
from Guide.#@P.zipcode
```

Furthermore, label markers can be used to construct results. The following example illustrates this fact.

```
select R.name, R.address
from Guide.restaurant R
```

In its O_2 implementation Lorel also featured elements of data manipulation.

UnQL UnQL is a simple yet powerful language for querying semistructured data. In fact, it is strictly more powerful than Lorel, because of its possibilities to restructure query results.

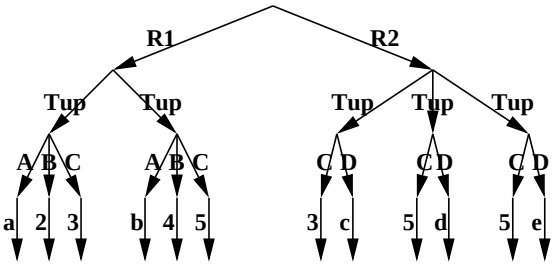

Figure 4.13: A relational database as edge-labeled tree

When a relational database is transformed into an edge-labeled tree as in Figure 4.13 UnQL is equivalent to relational algebra. An SQL-query

```
SELECT A, D
FROM R1, R2
WHERE R1.C = R2.C;
```

would be expressed in UnQL as:

```
select {Tup => {A => x, D => z}}
where
  R1 => Tup => {A => \x, C => \y} <-- DB,
  R2 => Tup => {C => y, D => \z} <-- DB
```

Understanding this query requires understanding the notion of edge-labeled trees introduced in Section 2.5. In the previous query the two edge-labeled tree patterns `R1 => Tup => {A => \x, C => \y}` and `R2 => Tup => {C => y, D => \z}` will be searched in the database. The variables are instantiated at the points where they are prefixed with a backslash. The join is achieved through the variable `y`, and in the `select`-clause the projection is performed.

Path expressions similar to those in Lorel are introduced. The following query returns the set of all strings in the database.

```
select {l}
where _* => \l => _ <-- DB, isstring(l)
```

With UnQL it is also possible to powerfully restructure the database. This makes UnQL unique among the query languages for semistructured data. The following query replaces all `foo`-edges by `bar`-edges. Additionally, it is also possible to change the graph structure of a database.

```
traverse DB giving X
  case foo => _ then X := {bar => X}
  case \l => _ then X := {l => X}
```

Other languages: XML-QL Closely related are also the query languages for the World Wide Web mentioned in Section 1.5 mentioned. The query language XML-QL, designed specifically for XML, is similar to our approach in that it uses element patterns as the "What"-part of a query [DFF+99]. Consider the following example.

```
WHERE <book>
    <publisher> <name> Addison-Wesley </> </>
    <title> $t </>
    <author> $a </>
  </> IN "www.foo.net/bib.xml"
CONSTRUCT <result>
    <author> $a </>
    <title> $t </>
  </>
```

With the `WHERE`-clause the element pattern is defined. It matches every book with a publisher named "Addison-Wesley" and arbitrary title and author. The `CONSTRUCT`-clause represents the "How"-part of the query. The user specifies that he wants the result tagged as "result" and within the result element the author first, followed by the

title. Like our approach, XML-QL supports nested queries, tag variables and regular path expressions. Unfortunately the examples in the mentioned paper are sometimes contradictory to the grammar given. It is not clear at this time what the language will finally look like and what influence it will have on the database community.

	Our approach	Lorel	UnQL	XML-QL
Conditions	yes	yes	yes	yes
Boolean operations	partly	yes	yes	yes
Path expressions	yes	yes	yes	yes
Nested queries	no	yes	yes	yes
Root node	not necessary	necessary	necessary	not necessary
Ordering the result	no	yes	no	yes
Answer is a / an	graph	OEM object	"tree" or label	XML doc
Aggregation	no	yes	no	no
Restructurings	no	no	yes	no

Table 4.2: Comparison of different query languages for semistructured data

Table 4.2 summerizes features supported by the different query languages presented in this chapter. Of course, the criteria listed in this table are somewhat subjective, but we tried to pick typical concepts of query languages. A more detailed comparison between five different query languages specifically designed for XML can be found in [BC99].

4.7 Summary

This chapter presented our notion of query. We moved from the very simple notion of a schema query via the focus query to the transformation query. A schema query consists just of a schema as presented in the previous chapter. It queries for the minimal matches of the schema. A focus can be given to project to interesting aspects. With a transformation query the answer can be completely restructured. Ideas for this querying approach were taken from the area of graph transformations, from which basic concepts were also presented in this chapter. We compared the expressiveness of our query language to the relational algebra and pointed out peculiarities when querying XML documents. Finally, we looked at other query languages for semistructured data.

Part III

Query Processing

Chapter 5

Schema Matching as a Constraint Satisfaction Problem

When you have eliminated the impossible,
whatever remains, however improbable, must be the truth.
(Arthur Conan Doyle)

This chapter is the first of three chapters dealing with the problem of query optimization. Our work focuses again on the "What"-part of a query, because finding the matches of a given schema is the most difficult part of processing a query. The operations introduced in the previous chapter can all be performed easily once the schema matches are found.

We choose to base our query processing on constraints. This idea comes from the area of graph transformations where constraints are used in the AGG-system [Agg, Rud98]. There are at least two good reasons for this approach. First, this additional layer of abstraction permits us to extend our schema language later on, without having to change too much on the optimizer. Second, many techniques and heuristics already exist for this general class of search problems.

In this chapter we demonstrate how to find the matches of a schema in a database directly, i.e., without any additional information. In the next chapters we discuss optimization in more detail and demonstrate how to make use of previously matched schemata. This chapter is organized as follows. Section 5.1 gives an introduction to the field of Constraint Satisfaction Problems. In Section 5.2 we demonstrate how the problem of finding a match to a given schema can be transformed into an equivalent Constraint Satisfaction Problem. Optimization techniques for Constraint Satisfaction Problems are discussed in the next chapter. We discuss other approaches to query processing in semistructured data in Section 5.3. Section 5.4 provides a summary of the chapter.

5.1 Introduction to Constraint Satisfaction Problems

Constraint Satisfaction Problems form a general class of search problems. They deal with solving problems by stating properties or constraints that any solution must fulfill.

Definition 5.1 (Constraint Satisfaction Problem). A *Constraint Satisfaction Problem (CSP)* is a tuple (X, D, C) where

- X is a set of *variables* $\{x_1, \ldots, x_m\}$,
- D is a set of finite *domains* D_i, one for each variable $x_i \in X$ and
- C is a set of *constraints* $\{C_{S_1}, \ldots, C_{S_n}\}$ restricting the values that the variables can simultaneously take. The $S_i = (x_{S_{i_1}}, \ldots, x_{S_{i_k}})$ are arbitrary tuples of variables from X; and each C_{S_i} is a relation over the crossproduct of the domains of these variables ($C_{S_i} \subseteq D_{S_{i_1}} \times \cdots \times D_{S_{i_k}}$).

Because variables and domains are linked to each other we also call every tuple $< x_i, D_i >$ with $x_i \in X$ and $D_i \in D$ a *domain variable.* A *variable interpretation* for a given set of domain variables is a mapping $\iota : X \longrightarrow D$, such that $\iota(x_i) \in D_i$ holds for every $x_i \in X$. A variable interpretation ι *satisfies* a constraint $C_{(x_{S_{i_1}}, \ldots, x_{S_{i_k}})}$ if $(\iota(x_{S_{i_1}}), \ldots, \iota(x_{S_{i_k}})) \in C_{(x_{S_{i_1}}, \ldots, x_{S_{i_k}})}$. A *solution* to a CSP is a variable interpretation, such that all constraints are simultaneously satisfied.

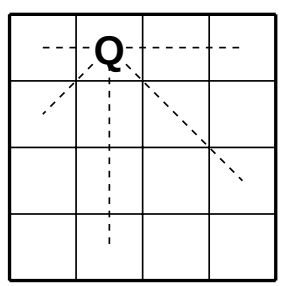

Figure 5.1: A queen restricting the possible positions of the other queens

A typical example for a CSP is the N-queens problem; we illustrate it as the 4-queens problem. The problem is to place four chess queens on a 4×4-board, such that no queen can capture another (in chess semantics). In Figure 5.1 a queen restricting the possible positions of the other queens is depicted. In the corresponding CSP four variables $x_1, \ldots, x_4$ representing the four rows are introduced. This is already a simplification, because it is based on the observation that in any solution there can be at most one queen per row. The domain of every variable is $\{1, 2, 3, 4\}$. $x_i = j$ represents the fact that the queen in the i-th row is located at position (column) j. As shown in Figure 5.1,

three classes of constraints are derived. One class represents that no two queens can be in the same column. The other two classes represent the left and the right diagonal.

$$\forall i,j,i<j\leq 4 : C^{col}_{(x_i,x_j)} = \{(x,y)\in D_i\times D_j | y\neq x\}$$
$$\forall i,j,i<j\leq 4 : C^{left}_{(x_i,x_j)} = \{(x,y)\in D_i\times D_j | y\neq x-(j-i)\}$$
$$\forall i,j,i<j\leq 4 : C^{right}_{(x_i,x_j)} = \{(x,y)\in D_i\times D_j | y\neq x+(j-i)\}$$

For the 4-queens problem there are six constraints in every one of the three classes. In Figure 5.2 we show the set of constraints.

$$\begin{aligned}
C^{col}_{(x_1,x_2)} &= \{(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)\}\\
C^{col}_{(x_1,x_3)} &= \{(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)\}\\
&\ldots\\
C^{col}_{(x_3,x_4)} &= \{(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)\}\\
C^{left}_{(x_1,x_2)} &= \{(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,1),(3,3),(3,4),(4,1),(4,2),(4,4)\}\\
C^{left}_{(x_1,x_3)} &= \{(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),(3,2),(3,3),(3,4),(4,1),(4,3),(4,4)\}\\
&\ldots\\
C^{left}_{(x_3,x_4)} &= \{(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,1),(3,3),(3,4),(4,1),(4,2),(4,4)\}\\
C^{right}_{(x_1,x_2)} &= \{(1,1),(1,3),(1,4),(2,1),(2,2),(2,4),(3,1),(3,2),(3,3),(4,1),(4,2),(4,3),(4,4)\}\\
C^{right}_{(x_1,x_3)} &= \{(1,1),(1,2),(1,4),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3),(3,4),(4,1),(4,2),(4,3),(4,4)\}\\
&\ldots\\
C^{right}_{(x_3,x_4)} &= \{(1,1),(1,3),(1,4),(2,1),(2,2),(2,4),(3,1),(3,2),(3,3),(4,1),(4,2),(4,3),(4,4)\}
\end{aligned}$$

Figure 5.2: The set of constraints for the 4-queens problem

The so defined CSP has two solutions: $(2,4,1,3)$ and $(3,1,4,2)$. They correspond to the actual solutions of the 4-queens problem as depicted in Figure 5.3.

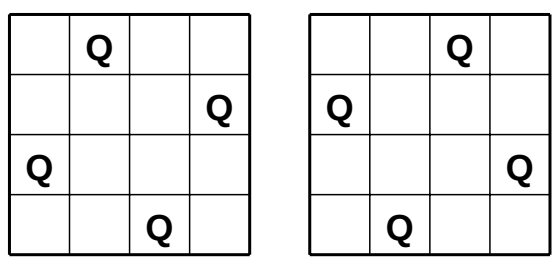

Figure 5.3: Solutions of the 4-queens problem

Determining whether a given CSP has a solution is an NP-complete problem. Because 3-SAT is NP-complete, Constraint Satisfaction with at most two values per do-

main and at most three variables per constraint is NP-complete. However, only the worst-case scenario is exponential. We will see that the need to search is not fatal.

The simplemost idea to solve a CSP is to follow the generate-and-test paradigm. Every possible variable interpretation is generated; and then every constraint is checked. This is, of course, a very inefficient approach, because it always considers the all possible variable instantiations. If there are m variables and every domain has size n then n^m many possible instantiations exist.

In the future, we restrict ourselves to binary CSPs, i.e., to CSPs where the constraints are between at most two variables. Every CSP with arbitrary constraints can be converted to an equivalent binary CSP [RPD89]. The idea of the transformation as as follows. For an arbitrary constraint $C_{S_i} = C_{(x_{S_{i_1}},\ldots,x_{S_{i_k}})}$ we introduce a new variable y_{S_i} that has the constraint C_{S_i} as its domain. Then k new binary constraints $C_{(y_{S_i},x_{S_{i_1}})},\ldots,C_{(y_{S_i},x_{S_{i_k}})}$ link the original constraint represented by y_{S_i} to its k components. Thus, the original constraint can be removed.

A binary CSP can be represented by a *constraint graph*, where every node represents a variable and every arc represents a constraint between two variables. Unary constraints are represented by loops, i.e., by arcs originating and terminating at the same node. The constraint graph for the 4-queens problem is shown in Figure 5.4.

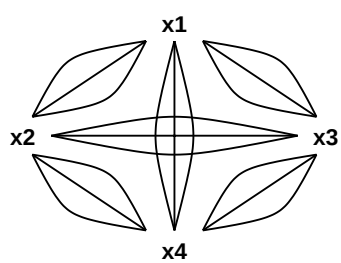

Figure 5.4: The constraint graph for the 4-queens problem

Constraint satisfaction techniques are used for a variety of applications. They range from algorithmic problems, such as graph colorings via DNA sequencing, to airport counter allocation [Bar98]. Scheduling problems are among the most popular applications.

5.2 Transformation of the schema matching problem

This section illustrates how to transform the problem of finding the matches for a given schema in a database to an equivalent CSP. This work was also presented in [BF99a]. The problem we are addressing is related to the SUBGRAPH-ISOMORPHISM problem, which is known to be NP-complete [Coo71]. However, our problem is exponentially

hard in the size of the schema, but only linearly hard in the size of the database. Furthermore, the labels in the graphs greatly reduce the average complexity. We start with giving the basic transformation steps for predicate schemata in the first subsection. Then we move on to show how to deal with variables and paths. In the final subsection we prove the correctness and the completeness of the transformation.

5.2.1 The basic principles of the transformation

We will illustrate the process of transforming a schema matching problem into a CSP in detail for the simplest sort of schema, the predicate schema, first. We use our well-known database graph in Figure 2.4 on Page 35 and the predicate schema in Figure 3.1 on page 45 as an example. The basic idea is the same as for the more sophisticated schema concepts. The database graph is transformed into suitable domains and variables are introduced for the elements in the schema. Furthermore, constraints representing the match semantics are introduced. They can be categorized into the ones that represent the label part and the ones that represent the structural part of the match semantics. The basic idea of the transformation is illustrated in Figure 5.5.

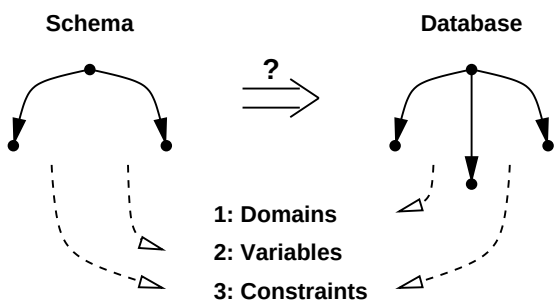

Figure 5.5: The basic idea of the transformation

In general, let us assume we have a database object $o = (V^{(o)}, A^{(o)}, s^{(o)}, t^{(o)}, l^{(o)})$ and a schema $s = (V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)})$. The following transformation is performed:

1. From the database graph we deduce domains $D_V = V^{(o)}$ and $D_A = A^{(o)}$.

   In our example we get:

$$D_V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$$
$$D_A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}\}$$

2. A variable x_i is introduced for each element in $V^{(s)} \cup A^{(s)}$. As a shorthand, we use the variable as a synonym for the corresponding element and say that "x_i is an arc" instead of "x_i represents an arc". X is the set of these variables.

3. For each variable $x_i \in X$ we define its domain D_i as $D_i := D_V$ if x_i is a node, and $D_i := D_A$ if x_i is an arc. D is the set of the D_i.

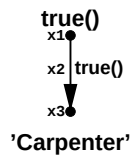

Figure 5.6: The predicate schema revisited

We repeat the schema graph in Figure 5.6 to illustrate the variable assignment. We get:

$$X = \{x_1, x_2, x_3\}$$
$$D_1 = D_3 = D_V$$
$$D_2 = D_A$$

4. For every variable x_i we introduce a unary *label constraint* $C^{lab}_{(x_i)}$. They represent the semantics of the respective predicate in the schema. A node or arc is in the constraint if its label makes the predicate of x_i true ($C^{lab}_{(x_i)} = \{(d_{i_k}) \in D_i | l^{(s)}(x_i)(l^{(o)}(d_{i_k})) = true\}$).

In the example x_1 is labeled with the predicate $true()$, which holds for any label. So there is not really a constraint on D_V and we could actually skip this one. The same holds for the arc x_2. x_3 is labeled $'Carpenter'$, which is a shorthand for the predicate $X = Carpenter$. The nodes v_5, v_7, and v_8 in the database graph carry this label.

$$\begin{aligned}
C^{lab}_{(x_1)} &= \{(d_{1_i}) \in D_1 | true(l^{(o)}(d_{1_i}))\} \\
&= \{(v_1), (v_2), (v_3), (v_4), (v_5), (v_6), (v_7), (v_8), (v_9), (v_{10}), (v_{11})\} \\
C^{lab}_{(x_2)} &= \{(d_{2_i}) \in D_2 | true(l^{(o)}(d_{2_i}))\} \\
&= \{(a_1), (a_2), (a_3), (a_4), (a_5), (a_6), (a_7), (a_8), (a_9), (a_{10}), (a_{11}), (a_{12})\} \\
C^{lab}_{(x_3)} &= \{(d_{3_i}) \in D_3 | Carpenter = l^{(o)}(d_{3_i})\} \\
&= \{(v_5), (v_7), (v_8)\}
\end{aligned}$$

5. For every variable $x_i \in A_S$ and its source and target node x_s and x_t we introduce two *structure constraints* $C^{src}_{(x_i,x_s)}$ and $C^{tar}_{(x_i,x_t)}$. They ensure the preservation of the graph structure of the schema (morphism property, $C^{src}_{(x_i,x_s)} = \{(d_{i_k}, d_{s_l}) \in D_i \times D_s | s^{(o)}(d_{i_k}) = d_{s_l}\}$, $C^{tar}_{(x_i,x_t)} = \{(d_{i_k}, d_{t_l}) \in D_i \times D_t | t^{(o)}(d_{i_k}) = d_{t_l}\}$).

There is only one arc in the schema graph, namely x_2. We link it to its source vertex x_1 and to its target vertex x_3. One should read such a constraint as follows: "If x_2 is assigned to a_1 then v_1 is a valid value for x_1" etc.

$$\begin{aligned}
C^{src}_{(x_2,x_1)} &= \{(d_{2_k}, d_{1_l}) \in D_2 \times D_1 | s^{(o)}(d_{2_k}) = d_{1_l}\} \\
&= \{(a_1, v_1), (a_2, v_1), (a_3, v_1), (a_4, v_2), (a_5, v_4), (a_6, v_2), \\
&\quad (a_7, v_2), (a_8, v_2), (a_9, v_3), (a_{10}, v_4), (a_{11}, v_4), (a_{12}, v_4)\} \\
C^{tar}_{(x_2,x_3)} &= \{(d_{2_k}, d_{3_l}) \in D_2 \times D_3 | t^{(o)}(d_{2_k}) = d_{3_l}\} \\
&= \{(a_1, v_2), (a_2, v_3), (a_3, v_4), (a_4, v_3), (a_5, v_3), (a_6, v_5), \\
&\quad (a_7, v_6), (a_8, v_7), (a_9, v_8), (a_{10}, v_9), (a_{11}, v_{10}), (a_{12}, v_{11})\}
\end{aligned}$$

6. For every pair of variables that are nodes and every pair of variables that are arcs we introduce an *injectivity constraint*. These constraints ensure that no two nodes and no two arcs in the schema are mapped to the same node or arc in the database graph ($C^{inj}_{(x_i,x_j)} = \{(d_{i_k}, d_{j_l}) \in D_i \times D_j | d_{i_k} \neq d_{j_l}\}$).

In our example we have only two nodes and one arc. Thus, we need only one constraint between the two nodes x_1 and x_3.

$$\begin{aligned}
C^{inj}_{(x_1,x_3)} &= \{(d_{1_k}, d_{3_l}) \in D_1 \times D_3 | d_{1_k} \neq d_{3_l}\} \\
&= \{(v_1, v_2), (v_1, v_3), (v_1, v_4), \ldots, (v_1, v_{11}), \\
&\quad (v_2, v_1), (v_2, v_3), (v_2, v_4), \ldots, (v_2, v_{11}), \\
&\quad (v_3, v_1), (v_3, v_2), (v_3, v_4), \ldots, (v_3, v_{11}), \\
&\quad \ldots, \\
&\quad (v_{11}, v_1), (v_{11}, v_2), (v_{11}, v_3), \ldots, (v_{11}, v_{10})\}
\end{aligned}$$

7. C is the set of all introduced constraints.

Our sample CSP has the solutions (v_2, a_6, v_5), (v_2, a_8, v_7), and (v_3, a_9, v_8) for the variables (x_1, x_2, x_3). They correspond to the matches of the schema as depicted in Figure 3.2 on Page 46.

5.2.2 Dealing with variables and paths

We move on to explain how we map the more advanced concepts, e.g., variables and paths, into the CSP. For the variables we reuse the example from Figure 3.3 on Page 46. We show it again in Figure 5.7.

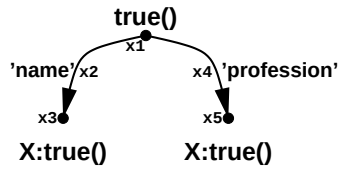

Figure 5.7: Variables in a schema revisited

Just as in the previous subsection we introduce variables $\{x_1, x_2, x_3, x_4, x_5\}$ with their domains. Furthermore, we introduce constraints $C^{lab}_{(x_1)}$, $C^{lab}_{(x_2)}$, $C^{lab}_{(x_3)}$, $C^{lab}_{(x_4)}$, and $C^{lab}_{(x_5)}$ derived from the labels of the schema, constraints $C^{src}_{(x_2,x_1)}$, $C^{tar}_{(x_2,x_3)}$, $C^{src}_{(x_4,x_1)}$, and $C^{src}_{(x_4,x_5)}$ derived from the structure of the schema, and injectivity constraints $C^{inj}_{(x_1,x_3)}$, $C^{inj}_{(x_3,x_5)}$, $C^{inj}_{(x_1,x_5)}$, and $C^{inj}_{(x_2,x_4)}$. To represent the variables located at x_3 and x_5 we do the following:

- For every pair x_i, x_j of variables with $v(x_i) = v(x_j)$ we introduce a constraint that ensures that the labels of the mappings of x_i and x_j are the same ($C^{var}_{(x_i,x_j)} = \{(d_{i_k}, d_{j_l}) \in D_i \times D_j | l^{(o)}(d_{i_k}) = l^{(o)}(d_{j_l})\}$).

In the example we get:

$$\begin{aligned} C^{var}_{(x_3,x_5)} &= \{(d_{3_k}, d_{5_l}) \in D_3 \times D_5 | l^{(o)}(d_i) = l^{(o)}(d_j)\} \\ &= \{(v_1, v_1), (v_2, v_2), (v_3, v_3), (v_4, v_4), (v_5, v_5), (v_5, v_7), (v_5, v_8), (v_6, v_6), \\ &\quad (v_7, v_5), (v_7, v_7), (v_7, v_8), (v_8, v_5), (v_8, v_7), (v_8, v_8), (v_9, v_9), (v_{10}, v_{10}), (v_{11}, v_{11})\} \end{aligned}$$

This CSP then has the solution $(v_2, a_6, v_5, a_8, v_7)$ that directly corresponds to the match shown in Figure 3.3 on Page 46.

For dealing with paths we need an additional domain. We put the trails of the database graph into a new domain.

- From the database graph we deduce the domain $D_P = P^{+(o)}$.

For the sample database in Figure 2.4 on Page 35 we get:

$$\begin{aligned} D_P = \{&(a_1), (a_2), (a_3), (a_4), (a_5), (a_6), (a_7), (a_8), (a_9), (a_{10}), (a_{11}), (a_{12}), \\ &(a_1, a_4), (a_1, a_6), (a_1, a_7), (a_1, a_8), (a_2, a_9), (a_3, a_5), (a_3, a_{10}), (a_3, a_{11}), (a_3, a_{12}), \\ &(a_4, a_9), (a_5, a_9), (a_1, a_4, a_9), (a_3, a_5, a_9)\} \end{aligned}$$

As an example we reuse the schema from Figure 3.5 on Page 50. We show it again in Figure 5.8.

We observe that the variable x_2 gets D_P as its domain, whereas x_1, x_3, and x_5 get D_V and x_4 gets D_A as before. The constraints $C^{lab}_{(x_1)}$, $C^{lab}_{(x_3)}$, $C^{lab}_{(x_4)}$, and $C^{lab}_{(x_5)}$ as well as

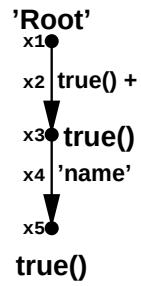

Figure 5.8: Paths in a schema revisited

$C^{src}_{(x_4,x_3)}$ and $C^{src}_{(x_4,x_5)}$, and $C^{inj}_{(x_1,x_3)}$, $C^{inj}_{(x_3,x_5)}$, and $C^{inj}_{(x_1,x_5)}$ are defined in the same manner as before. Only the constraints involving x_2 have to be adapted.

- For every arc x_i with domain D_P we introduce a unary *label constraint* $C^{lab}_{(x_i)}$ that represents the semantics of the respective predicate in the schema. A trail is in the constraint if the labels of all its arcs make the predicate of x_i true ($C^{lab}_{(x_i)} = \{(d_{i_k}) \in D_i | \forall a_j \in d_{i_k} : l^{(s)}(x_i)(l^{(o)}(a_j) = true\}$).
- For every arc $x_i \in A_S$ with domain D_P and its source and target node x_s and x_t we introduce two *structure constraints* $C^{src}_{(x_i,x_s)}$ and $C^{tar}_{(x_i,x_t)}$ that ensure the preservation of the graph structure of the schema ($C^{src}_{(x_i,x_s)} = \{(d_{i_k}, d_{s_l}) \in D_i \times D_s | s^{(o)}_P(d_{i_k}) = d_{s_l}\}$, $C^{tar}_{(x_i,x_t)} = \{(d_{i_k}, d_{t_l}) \in D_i \times D_t | t^{(o)}_P(d_{i_k}) = d_{t_l}\}$).

In the example this leads to:

$$\begin{aligned}
C^{lab}_{(x_2)} &= \{(d_{2_k}) \in D_2 | \forall a_j \in d_{2_k} : true(l^{(s)}(x_i)(l^{(o)}(a_j))\} \\
&= \{((a_1)), ((a_2)), ((a_3)), ((a_4)), ((a_5)), ((a_6)), ((a_7)), ((a_8)), ((a_9)), \\
&\quad ((a_{10})), ((a_{11})), ((a_{12})), ((a_1, a_4)), ((a_1, a_6)), ((a_1, a_7)), ((a_1, a_8)), ((a_2, a_9)), \\
&\quad ((a_3, a_5)), ((a_3, a_{10})), ((a_3, a_{11})), ((a_3, a_{12})), ((a_4, a_9)), ((a_5, a_9)), \\
&\quad ((a_1, a_4, a_9)), ((a_3, a_5, a_9))\}
\end{aligned}$$

$$\begin{aligned}
C^{src}_{(x_2,x_1)} &= \{(d_{2_k}, d_{1_l}) \in D_2 \times D_1 | s^{(o)}_P(d_{2_k}) = d_{1_l}\} \\
&= \{((a_1), v_1), ((a_2), v_1), ((a_3), v_1), ((a_4), v_2), ((a_5), v_4), ((a_6), v_2), \\
&\quad ((a_7), v_2), ((a_8), v_2), ((a_9), v_3), ((a_{10}), v_4), ((a_{11}), v_4), ((a_{12}), v_4), \\
&\quad ((a_1, a_4), v_1), ((a_1, a_6), v_1), ((a_1, a_7), v_1), ((a_1, a_8), v_1), ((a_2, a_9), v_1), \\
&\quad ((a_3, a_5), v_1), ((a_3, a_{10}), v_1), ((a_3, a_{11}), v_1), ((a_3, a_{12}), v_1), \\
&\quad ((a_4, a_9), v_2), ((a_5, a_9), v_4), ((a_1, a_4, a_9), v_1), ((a_3, a_5, a_9), v_1)\}
\end{aligned}$$

$$
\begin{aligned}
C^{tar}_{(x_2,x_3)} &= \{(d_{2_k}, d_{3_l}) \in D_2 \times D_3 | t^{(o)}_P(d_{2_k}) = d_{3_l}\} \\
&= \{((a_1), v_2), ((a_2), v_3), ((a_3), v_4), ((a_4), v_3), ((a_5), v_3), ((a_6), v_5), \\
&\quad ((a_7), v_6), ((a_8), v_7), ((a_9), v_8), ((a_{10}), v_9), ((a_{11}), v_{10}), ((a_{12}), v_{11}), \\
&\quad ((a_1, a_4), v_3), ((a_1, a_6), v_5), ((a_1, a_7), v_6), ((a_1, a_8), v_7), ((a_2, a_9), v_8), \\
&\quad ((a_3, a_5), v_3), ((a_3, a_{10}), v_9), ((a_3, a_{11}), v_{10}), ((a_3, a_{12}), v_{11}), \\
&\quad ((a_4, a_9), v_8), ((a_5, a_9), v_8), ((a_1, a_4, a_9), v_8), ((a_3, a_5, a_9), v_8)\}
\end{aligned}
$$

Furthermore, we need two additional constraints representing the conditions on the length of the path.

- For every arc $x_i \in A_S$ with domain D_P we introduce two *length constraints*. They reflect the conditions on path lengths defined by $q^{(s)}_{min}$ and $q^{(s)}_{max}$ ($C^{min}_{(x_i)} = \{(d_{i_k}) \in D_i | q^{(s)}_{min}(x_i) \leq length(d_{i_k})\}$, $C^{max}_{(x_i)} = \{(d_{i_k}) \in D_i | q^{(s)}_{max}(x_i) \geq length(d_{i_k})\}$).

Because $q^{(s)}_{min}(x_2) = 1$ and $q^{(s)}_{max}(x_2) = +\infty$, there is no real constraint in our example, i.e., the constraints look the same as $C^{lab}_{(x_2)}$. We can observe that the domain of the arcs D_A is actually a special case of the domain of the paths D_P with the length constraints already manifested within the domain.

Hence, we must not forget to introduce somewhat subtle injectivity constraints. Because D_A is a subset of D_P (in the sense that there is a natural embedding of D_A into D_P), we must also link the variables that indicate paths to the variables that indicate arcs.

- For every pair of variables (x_i, x_j) that are arcs we introduce an *injectivity constraint*. These constraints ensure that no arcs in the schema are mapped to the same trail in the database graph ($C^{inj}_{(x_i,x_j)} = \{(d_{i_k}, d_{j_l}) \in D_i \times D_j | d_{i_k} \neq d_{j_l}\}$). This definition assumes that an arc is equal to the atomic trail consisting of that arc.

In the example this leads to:

$$
\begin{aligned}
C^{inj}_{(x_2,x_4)} &= \{(d_{2_k}, d_{4_l}) \in D_2 \times D_4 | d_{2_k} \neq d_{4_l}\} \\
&= \{((a_1), a_2), ((a_1), a_3), ((a_1), a_4), \ldots, ((a_1), a_{12}), \\
&\quad ((a_2), a_1), ((a_2), a_3), ((a_2), a_4), \ldots, ((a_2), a_{12}), \ldots, \\
&\quad ((a_{12}), a_1), ((a_{12}), a_2), ((a_{12}), a_3), \ldots, ((a_{12}), a_{11}), \\
&\quad ((a_1, a_4), a_1), ((a_1, a_4), a_2), ((a_1, a_4), a_3), \ldots ((a_1, a_4), a_{12}), \ldots, \\
&\quad ((a_3, a_5, a_9), a_1), ((a_3, a_5, a_9), a_2), ((a_3, a_5, a_9), a_3), \ldots ((a_3, a_5, a_9), a_{12})\}
\end{aligned}
$$

This step concludes the transformation. Our example problem has five solutions: $(v_1, (a_1), v_2, a_6, v_5)$, $(v_1, (a_1, a_4), v_3, a_9, v_8)$, $(v_1, (a_2), v_3, a_9, v_8)$, $(v_1, (a_3, a_5), v_3, a_9, v_8)$, and $(v_1, (a_3), v_4, a_{10}, v_9)$. They correspond directly to the matches of the schema as shown in Figure 3.5 on Page 50.

5.2.3 Correctness and completeness of the transformation

We conclude this section with the proof that our transformation is correct and complete, i.e., that solutions of the CSP and matches of the schema correspond to each other.

Theorem 5.1. *Let o be an object and s be a schema. Let (X, D, C) be the CSP with k variables $X = \{x_1, \dots x_k\}$ that has been defined in accordance with the above transformation. A tuple $(d_{1_i}, \dots, d_{k_i}) \in D_1 \times \cdots \times D_k$ is a solution of the CSP if and only if the subobject of o induced by it is a minimal match of s in o.*

Before we are going to prove the theorem we clarify what we mean by "by a solution of a CSP induced subobject". Let $(d_{1_i}, \dots, d_{k_i})$ be a solution of the CSP. Let $V^{(0)} := \emptyset$ and $A^{(0)} := \emptyset$. For every $d_{j_i} \in (d_{1_i}, \dots, d_{k_i})$ we do the following. If d_{j_i} is a vertex then let $V^{(j)} := V^{(j-1)} \cup \{d_{j_i}\}$ and $A^{(j)} := A^{(j-1)}$. If d_{j_i} is an arc then let $V^{(j)} := V^{(j-1)}$ and $A^{(j)} := A^{(j-1)} \cup \{d_{j_i}\}$. If d_{j_i} is a trail then let $V^{(j)} := V^{(j-1)}$ and $A^{(j)} := A^{(j-1)} \cup \{a_{j_i} | a_{j_i} \in d_{j_i}\}$. Because the domains were constructed from the original object o, we observe $V^{(k)} \subseteq V^{(o)}$ and $A^{(k)} \subseteq A^{(o)}$. Now we set $V^{(o_i)} := V^{(k)} \cup \{v \in V^{(o)} | \exists a \in A^{(k)} : v = s^{(o)}(a) \vee v = t^{(o)}(a)\}$ and $A^{(o_i)} := A^{(k)}$. The so defined object $o_i = (V^{(o_i)}, A^{(o_i)}, s^{(o)}|_{A^{(o_i)}}, t^{(o)}|_{A^{(o_i)}}, l^{(o)}|_{V^{(o_i)} \cup A^{(o_i)}})$ is the subobject of o that is induced by $(d_{1_i}, \dots, d_{k_i})$. This resembles the $flatten$-function defined in Section 3.3.

Proof. Let $o = (V^{(o)}, A^{(o)}, s^{(o)}, t^{(o)}, l^{(o)})$ be an arbitrary, but fixed object and $s = (V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)}, v^{(s)}, q^{(s)}_{min}, q^{(s)}_{max})$ be a schema. Let (X, D, C) be the CSP with k variables $X = \{x_1, \dots x_k\}$ that has been defined in accordance with the transformation. Let $(d_{1_i}, \dots, d_{k_i}) \in D_1 \times \cdots \times D_k$ be a solution of the CSP. Let $o_i = (V^{(o_i)}, A^{(o_i)}, s^{(o_i)}, t^{(o_i)}, l^{(o_i)})$ be the subobject of o that is induced by $(d_{1_i}, \dots, d_{k_i})$. We first show that o_i is a match of s in o and then prove that o_i is minimal. Let m be a mapping between $V^{(s)} \cup A^{(s)}$ and $V^{(o)} \cup P^{+(o)}$ defined as $m(x_j) := d_{j_i}$, where d_{j_i} is the value of the variable x_j that was introduced in the CSP to represent nodes and arcs of the schema. Let us do two simplifications here. First, we use the CSP variables x_j as a synonym for the node or arc they represent. Second, we do not distinguish between the domains D_A and D_P, but rather see any arc d_{i_j} from D_A as being identical to the atomic trail (d_{i_j}) from D_P. Thus, we do not care if an $m(x_j)$ is in D_A or D_P, but rather view them as members of $P^{+(o)}$. We show that m is a match function, because it fulfills all conditions of Definition 3.5:

1. The mapping m is an isomorphic embedding of s into o_P. It is obvious that m is a total function. Let x_j be an arbitrary arc from s, and x_{j_1} and x_{j_2} be its source and target nodes. Then $(d_{i_j}, d_{i_{j_1}}) \in C^{src}_{(x_j, x_{j_1})}$ and $(d_{i_j}, d_{i_{j_2}}) \in C^{tar}_{(x_j, x_{j_2})}$, because the constraints exist and $(d_{1_i}, \ldots, d_{k_i})$ is a solution of the CSP. Because of the definition of these two constraints, it is clear that $d_{i_{j_1}}$ and $d_{i_{j_2}}$ are the source and target node of d_{i_j} in o_P. We can similarly prove that m is an injective function by using the injectivity constraints $C^{inj}_{(x_{j_1}, x_{j_2})}$.

2. Let x_j be a node. Then $d_{i_j} \in C^{lab}_{(x_j)}$, hence the predicate $l^{(s)}(x_j)$ holds for $l^{(o_P)}(d_{i_j})$. Similarly, let x_j be an arc. Then $d_{i_j} \in C^{lab}_{(x_j)}$ and hence for all arcs in d_{i_j} the predicate $l^{(s)}(x_j)$ holds for their labels. Thus, conditions one and two of Definition 3.5 hold.

3. Let x_{j_1} and x_{j_2} be two variables with $v^{(s)}(x_{j_1}) = v^{(s)}(x_{j_2})$. If no two such variables exist, condition three holds trivially. If they exists then $(d_{i_{j_1}}, d_{i_{j_2}}) \in C^{var}_{(x_j, x_{j_1})}$, because the constraint exists, and $(d_{1_i}, \ldots, d_{k_i})$ is a solution of the CSP. This implies that the labels of $d_{i_{j_1}}$ and $d_{i_{j_2}}$ are the same, because of the definition of the constraint. Thus, condition three is fulfilled.

4. Let x_j be an arbitrary arc. If it has D_A as its domain then $q^{(s)}_{min}(x_j) = 1$ and $q^{(s)}_{max}(x_j) = 1$. Obviously, d_j has exactly length one in this case, so condition four is fulfilled. If x_j has domain D_P then constraints $C^{min}_{(x_j)}$ and $C^{max}_{(x_j)}$ exist; and d_j is a member of both, because it is a member of a solution of the CSP. Because of the definition of the constraints, d_j is no shorter than $q^{(s)}_{min}(x_j)$ and no longer than $q^{(s)}_{min}(x_j)$. Thus, condition four is also always fulfilled.

We have seen that m is a match function. o_i is a minimal match of s in o, because $o_i = flatten(m(s))$. Hence, we proved correctness of the transformation.

Vice versa, let $o_i = (V^{(o_i)}, A^{(o_i)}, s^{(o_i)}, t^{(o_i)}, l^{(o_i)})$ be a minimal match of s in o. Let m be the match function. Let $(d_{1_i}, \ldots, d_{k_i}) := (m(x_1), \ldots, m(x_k))$. We observe that $(d_{1_i}, \ldots, d_{k_i}) \in D_1 \times \cdots \times D_k$, i.e., that $(d_{1_i}, \ldots, d_{k_i})$ is potentially a solution of the CSP. We show that it is indeed a solution by showing that it satisfies all constraints.

1. Let $C^{lab}_{(x_j)}$ be an arbitrary label constraint. If x_j is a node then the predicate $l^{(s)}(x_j)$ is true for $l^{(o)}(m(x_j))$, because m is a match function. This implies $m(x_j) \in C^{lab}_{(x_j)}$ and thus, $d_{j_i} \in C^{lab}_{(x_j)}$. If x_j is an arc then the predicate $l^{(s)}(x_j)$ is true for all arcs in the trail $m(x_j)$, because m is a match function. Thus, $d_{j_i} \in C^{lab}_{(x_j)}$ holds also in this case.

2. Let $C^{src}_{(x_{j_1}, x_{j_2})}$ be an arbitrary structure constraint of type src. Because $s^{(s)}(x_{j_1}) = x_{j_2}$ and m is a match function, $s^{(o)}_P(m(x_{j_1})) = m(x_{j_2})$ holds. This implies

$(m(x_{j_1}), m(x_{j_2})) \in C^{src}_{(x_{j_1},x_{j_2})}$ and thus, $(d_{i_{j_1}}, d_{i_{j_2}}) \in C^{src}_{(x_{j_1},x_{j_2})}$. A similar reasoning can be done for an arbitrary structure constraint $C^{tar}_{(x_{j_1},x_{j_2})}$ of type *tar*.

3. Let $C^{inj}_{(x_{j_1},x_{j_2})}$ be an arbitrary injectivity constraint. Because $x_{j_1} \neq x_{j_2}$ and m is a match function, it is true that $m(x_{j_1}) \neq m(x_{j_2})$. Thus, $(m(x_{j_1}), m(x_{j_2})) \in C^{inj}_{(x_{j_1},x_{j_2})}$ and $(d_{i_{j_1}}, d_{i_{j_2}}) \in C^{inj}_{(x_{j_1},x_{j_2})}$ hold.

4. Let $C^{var}_{(x_{j_1},x_{j_2})}$ be an arbitrary variable constraint. This implies that $v^{(s)}(x_{j_1}) = v^{(s)}(x_{j_2})$. Because m is a match function, we get $l^{(o)}(m(x_{j_1})) = l^{(o)}(m(x_{j_2}))$. Thus, $(m(x_{j_1}), m(x_{j_2})) \in C^{var}_{(x_{j_1},x_{j_2})}$ and $(d_{i_{j_1}}, d_{i_{j_2}}) \in C^{var}_{(x_{j_1},x_{j_2})}$ hold.

5. Let $C^{min}_{(x_j)}$ be an arbitrary length constraint of type *min*. Because m is a match function, it is true that $length(m(x_j)) \geq q^{(s)}_{min}(x_j)$. Thus, $m(x_j) \in C^{min}_{(x_j)}$ and $d_{j_i} \in C^{min}_{(x_j)}$ hold. Similarly, $d_{j_i} \in C^{max}_{(x_j)}$.

We have shown that $(d_{1_i}, \ldots, d_{k_i})$ satisfies all introduced constraints. Hence, it is a solution of the CSP. We also proved completeness of the transformation. □

5.3 Cost-based query processing for semistructured data

A more traditional approach to query optimization is used in the Lore system. A query is parsed and preprocessed; and a single logical query plan providing a high-level description of the execution strategy is generated. Statistics and a cost model are used to transform the logical query plan into a physical plan. Lore uses several different indexing structures. The value index supports finding all atomic objects with a given incoming edge label and satisfying a given predicate. The label index supports finding all parents of a given object via an edge with a given label. The edge index supports finding all parent-child object pairs connected via a specified label. Finally, the DataGuide [GW97] provides the functionality of a path index.

Lore distinguishes different strategies of query evaluation [MW99]. The top-down strategy simply traverses a path starting from the root in a forward manner. This strategy is similar to pointer-chasing in object-oriented systems. The bottom-up strategy uses the value index to find objects that satisfy the given predicate, and the label index to traverse back through the data. A third strategy is to mix the two together: evaluate parts of a path expression top-down and traverse back from the leaf objects simultaneously. A join between the two temporary results leads to complete satisfying paths. This strategy is called the hybrid strategy. For every one of these strategies there are examples making the respective strategy perform best.

For UnQL a calculus called UnCAL provides a formal basis for deriving optimization rewrite rules [BDHS96]. However, a cost-based optimizer does not exist. The Strudel system is different in that the data may reside anywhere in an arbitrary format [FFK+98]. Query languages for XML, such as XML-QL or XQL, are not yet in a stage where a full optimizer exists.

Query processing on the basis of Constraint Satisfaction Problems is significantly different from cost-based query processing. The latter introduces a planning phase, where an efficient query execution plan is generated from a set of alternative plans. When solving a CSP only little planning is performed. At each step in the process of solving the problem a decision on how to carry on is made dynamically. We will focus on techniques for solving CSPs efficiently in the next chapter.

5.4 Summary

This chapter introduced basic ideas of our query processing based on constraints. We identified that the difficult part of answering a query is to find the matches of a schema in a database. We reduced this problem to an equivalent Constraint Satisfaction Problem. We provided an introduction to the field of Constraint Satisfaction Problems and demonstrated the transformation. We proved the correctness and the completeness of the transformation. In the final section we looked at the more traditional cost-based query optimization used in the Lore project.

Chapter 6

Optimization Techniques for Constraint Satisfaction Problems

"Contrariwise", continued Tweedledee, "If it was so, it might be;
and if it were so, it would be;
but as it isn't, it ain't. That's logic."
(Lewis Carrol)

After we have seen how to create a Constraint Satisfaction Problem (CSP) from our original problem of finding schema matches in a database, we move on to discuss optimization techniques for these CSPs. Our basic assumption is that we are interested in all solutions of a CSP, not just in one. The latter would lead to completely different heuristics, because it would not necessarily require to consider the complete search space. Furthermore, we are restricting ourselves to binary CSPs as we explained in Section 5.1.

The chapter is organized as follows. In Section 6.1 we start with domain reduction, i.e., with eliminating values violating constraints. We move on to consider more intelligent search algorithms in Section 6.2 and conclude in Section 6.3 with observations on how the order, in which the variables are instantiated, plays a significant role. In Section 6.4 we prove an interesting property about our approach: Matches of a tree-shaped schema without variable definitions can be found without search and in polynomial time if the requirement of injectivity of the match function is ignored. Section 6.5 provides a summary of the chapter.

6.1 Consistency techniques

A first approach to solving CSPs more efficiently is to reduce the domains of the variables, and thus, to reduce the search space. Eliminating only one value from one

domain in a problem with m variables and domain size n reduces the number of possible instantiations by a factor of $\frac{n}{n-1}$ or, by an absolute size of n^{m-1}. Techniques that eliminate such inconsistent values from domains are called *consistency techniques*. They are deterministic, as opposed to search, which is non-deterministic. Thus, the deterministic computation is usually performed as soon as possible. Nevertheless, the consistency techniques are rarely used alone to solve a CSP completely, although they could be.

An obvious first idea is to eliminate values that violate unary constraints. Suppose there is a variable x with domain $\{1, 2, 3\}$ and a unary constraint $C^{odd}_{(x)} = \{(1), (3)\}$ then a solution to the CSP with $x = 2$ cannot exist.

Definition 6.1 (Node consistency). A variable x in a CSP is called *node consistent* if for every value in the current domain of x each unary constraint on x is satisfied.

The notion of node consistency is derived from the constraint graph. Node consistency can obviously be achieved in polynomial time.

This basic consistency idea can be extended to cover more than one variable. Consider two variables x and y and a binary constraint $C_{(x,y)}$ between them. Values u in the current domain of x for which no value v in the current domain of y exists, such that (u, v) satisfies the constraint, are inconsistent.

Definition 6.2 (Arc consistency). The variable pair (x, y) is *arc consistent* if for every value u in the current domain of x there exists some value v in the current domain of y, such that $x = u$ and $y = v$ is permitted by every binary constraint between x and y.

Actually, the original definition defines arc consistency per arc in the constraint graph, i.e., per binary constraint. Note that arc consistency is directional: If (x, y) is arc consistent then (y, x) is not necessarily arc consistent as well.

Various algorithms have been proposed to achieve arc consistency for every arc in a CSP. The theoretical worst-case time complexity of achieving arc consistency is $O(cn^2)$, where c is the total number of binary constraints and n is the domain size for each variable. To verify arc consistency, each arc must be inspected at least once, which takes $O(n^2)$ time. Mohr and Henderson present an algorithm that achieves exactly this lower bound [MH86]. Another popular algorithm is AC-3 [Mac77]; it has running time $O(cn^3)$.

Arc consistency still does not eliminate the need to search. Consider Figure 6.1. It shows an arc consistent CSP, because for every instantiation of an arbitrary variable one can find a value for the other variables that satisfies the "not equal"-constraint. But there exists no solution to the CSP, which will become clear only after performing search.

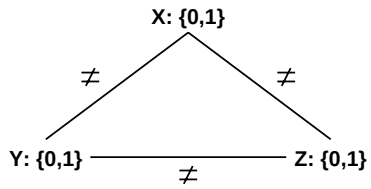

Figure 6.1: An arc consistent CSP with no solution

The last statement is actually not completely true. Search is only necessary if we do not have more advanced consistency notions. The consistency idea can of course be extended to cover even more variables.

Definition 6.3 (k-consistency, strong k-consistency). A constraint graph is *k-consistent* if the following is true: Choose values of any $k-1$ variables that satisfy all the constraints among these variables. Then choose any kth variable. There exists a value in the current domain of this variable that satisfies all the constraints among all the K variables. A constraint graph is *strongly k-consistent* if it is j-consistent for all $j \leq k$.

Node consistency is equivalent to strong 1-consistency; arc consistency is equivalent to strong 2-consistency. Algorithms exist to make a constraint graph strongly k-consistent, but their worst-case running time is exponential [Coo89]. Clearly, if a constraint graph with m nodes is strongly m-consistent, then a solution to the CSP can be found without any search.

Consistency techniques can be applied at any stage of a search procedure. They can be applied "statically", i.e., before the search starts. During the search, if a variable gets instantiated to some value a new, "virtual" CSP is defined, in which this variable has a domain consisting only of the value it got instantiated to. Then consistency techniques can be applied again. The reductions performed in this case are of course temporarily; they have to be revoked once the variable gets instantiated to some other value.

6.2 Search algorithms

Suppose we have a given CSP with m variables and assume for simplicity that all underlying domains have the same size n. Then there exist n^m possible instantiations for the variables. With the inefficient but simple generate-and-test approach a total assignment is generated. Then the validity of the constraints is tested. Of course, more efficient methods are needed. We introduce some basic algorithms in this section. Several other algorithms, in particular hybrid algorithms combining two or more of the methods, have been developed.

A *search space* (or *search tree*) for a CSP is a tree in which the nodes at level i represent an instantiation of i variables. Thus, the set of the leaf nodes represents the set of the possible instantiations. At each level, the tree is split according to the possible values for some fixed uninstantiated variable. The ordering of the variables, according to which the tree is built, influences the number of inner nodes of the search space and thus, the size of the search space. We show one search space for the 3-Queens Problem in Figure 6.2.

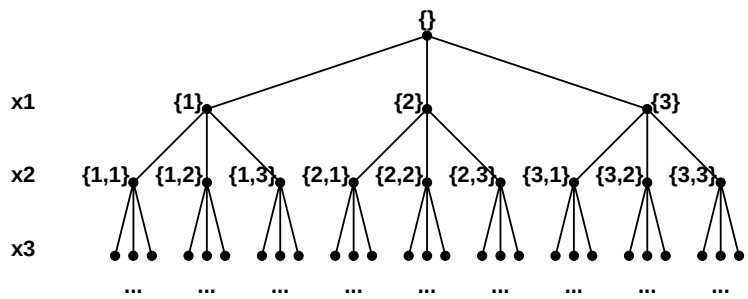

Figure 6.2: The search space for the 3-Queens Problem

Backtracking (BT) We present the simplemost form of backtracking first. The idea of backtracking is to extend partial solutions. An initially empty assignment is extended by assigning one variable at a time. In each step the validity of the constraints is tested, i.e., consistency checks between the instantiation of the current variable and the instantiations of the earlier variables, starting with the first variable, are performed. When a contradictory partial assignment is found, the next domain value of the current variable is tried. If there are no more domain values left, the backtracking to the most recently instantiated variable is performed. A solution is recorded every time all consistency checks succeed at the lowest level. Hence, this method is also called chronological backtracking [BR75]. Standard measures for performance of a backtracking algorithm are the number of nodes visited in the backtrack tree generated by the algorithm, and the number of consistency checks performed.

The main advantage of backtracking is its simplicity. Although backtracking performs better than the naive generate-and-test approach there is still plenty of room for improvement. One major disadvantage is *thrashing* [Mac77]. Suppose you have two variables x_i and x_j, and a constraint $C_{(x_i,x_j)}$ between them. Now suppose that x_i is assigned to a value, for which the constraint can never be true. Then each assignment for x_j results in failure, but unfortunately all the variables between x_i and x_j in the ordering of instantiation are reconsidered again and again after each failure for x_j, although they have nothing to do with that

failure. No doubt we can recognize by now where the name "thrashing" comes from.

Backjumping (BJ) Backjumping addresses the drawback of thrashing. Instead of backtracking chronologically backjumping jumps back to the deepest past variable that conflicted with the current variable [Gas78]. The consistency checks are performed in the original order of instantiation. A drawback of backjumping is that it cannot perform "multiple backjumps". Additionally, a big backjump causes the loss of information gained inbetween.

Conflict-directed backjumping (CBJ) The behavior of conflict-directed backjumping [Pro93] is even more sophisticated. Every variable has its own conflict set consisting of the past variables that failed consistency checks with its current instantiation. Once there are no more values to be tried for the current variable, the backtracking goes to the deepest variable in the conflict set. In this case the conflict set is propagated, so that no conflict information is lost.

Backmarking (BM) Backmarking addresses another drawback of backtracking. Consistency checks are performed without keeping the information, which of them were already performed at an earlier stage. Backmarking imposes a marking scheme [Gas77]. This marking scheme is based on two observations, again identified in [Nad89]:

1. If, at the most recent node where a given instantiation was checked, the instantiation failed against some past instantiation that has not yet changed, then it will fail against it again.
2. If, at the most recent node where a given instantiation was checked, the instantiation succeeded against all past instantiations that have not yet changed, then it will succeed against them again.

Backmarking visits exactly the same nodes of the search tree that backtracking visits. So the disadvantage of thrashing is not addressed. However, the advantage of backmarking is that sometimes no consistency checks are necessary at all.

Forward checking (FC) In contrast to all the above methods forward checking performs the consistency checks forwardly, i.e., values in the domains of future variables that are inconsistent with the current instantiation of the current variable are removed [HE80]. If, during that process, one of these domains is annihilated, then the temporal changes caused by the Forward checking are undone, and backtracking is invoked. Otherwise the next variable gets instantiated to some value

from its now filtered domain. A solution is recorded every time the last variable gets instantiated.

A theoretical evaluation of selected backtracking algorithms is presented in [KvB97]. The algorithms are compared with respect to the number of nodes they visit in the search tree and to the number of consistency checks they perform. The study is based on the assumptions that the variable ordering is static and that all solutions of a CSP are sought. We restrict the results to the algorithms we have discussed and shown them in Figure 6.3. The results are shown as Hasse diagrams. Chronological backtracking visits

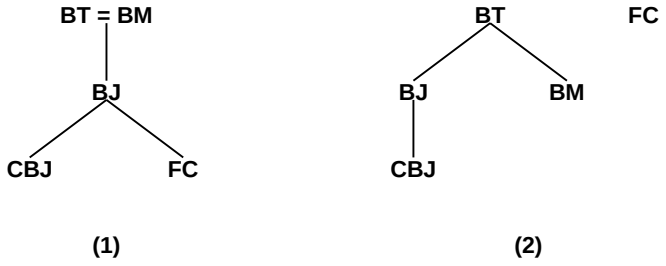

Figure 6.3: Theoretical evaluation of backtracking algorithms

the most nodes in the search tree (left diagram) and performs the most consistency checks (right diagram, incomparable with forward checking). From the diagrams we get the impression that conflict-directed backjumping performs best, although the original paper suggests that a hybrid algorithm between forward checking and conflict-directed backjumping [Pro93] performs even better.

6.3 Variable ordering

The order, in which the variables are instantiated, can have a substantial influence on the performance of an arbitrary backtracking algorithm. The reason is the first-fail heuristic. Once a consistency check fails we get rid of a complete subtree of the search tree. Obviously, we want that subtree to be as large as possible. Hence, we want consistency checks to fail as early as possible. This reasoning is again based on the assumption that we are interested in all solutions of a CSP, because only in this case we have to consider the complete search space. How can the order, in which the variables are instantiated, have an effect on the performance?

1. If we instantiate variables with small domains first, we ensure that once a consistency check fails, the subtree we get rid of is reasonably large, because the variables with the large domain come last. Furthermore, the search space is smallest if the variables are instantiated in this manner. Let $(x_{i_1}, \ldots, x_{i_m})$ be

a fixed variable ordering. The search space built according to that ordering has size $1 + |D_{i_1}| + |D_{i_1}||D_{i_2}| + \cdots + |D_{i_1}| \ldots |D_{i_m}|$. This number is equal to $1 + |D_{i_1}|(1 + |D_{i_2}|(\ldots(1 + |D_{i_m}|)\ldots))$. From this expression we can see that the size of the search space is minimal if $|D_{i_1}| \leq |D_{i_2}| \leq \cdots \leq |D_{i_m}|$.

2. If a variable is involved in many constraints, it is likely that it is difficult to find a consistent value for it and that many consistency checks will fail. Hence, it is a good idea to instantiate such variables first.

An intelligent variable ordering can also help to reduce thrashing by putting variables that are involved in the same constraint close to each other. The variable ordering can also be handled dynamically; a backtracking algorithm that maintains full arc consistency and performs dynamic variable ordering has been proposed in [SF94].

6.4 Observations on the properties of our approach

In this section we prove an important property of our approach. We give a condition, under which a CSP in our approach can be solved without search and in polynomial time. This theorem is based on the following well-known theorem by Freuder [Fre82].

Theorem 6.1. *If a constraint graph is strongly k-consistent, and $k > w$ where w is the width of the constraint graph, then there exists a search order that is backtrack-free.*

To understand this theorem we must clarify the notion of *width* of a constraint graph. An *ordered constraint graph* is a constraint graph whose vertices have been ordered linearly. The width of an ordered constraint graph is defined as the maximum number of arcs leading from an arbitrary vertex to previous vertices. The width of a constraint graph is the minimum width over all its ordered constraint graphs. Intuitively, the width is an indicator of how many already instantiated variables have to be taken into account once an uninstantiated variable gets instantiated. Figure 6.4 shows a constraint graph together with all its ordered constraint graphs and their widths. The width of the constraint graph is 1.

Now, the proof of the above theorem is straightforward. There exists an ordering of the constraint graph, such that the number of arcs leading from any vertex of the graph to the previous vertices, is at most w. If the variables are instantiated using this ordering, then whenever a new variable is instantiated, a value for this variable is consistent with all the previous assignments. The reason is that this value is to be consistent with the assignments of at most w other variables, and the graph is strongly $(w + 1)$-consistent.

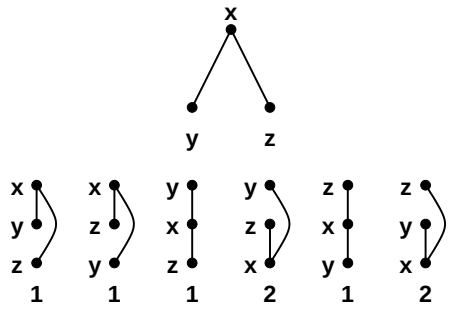

Figure 6.4: The widths of ordered constraint graphs

We are ready to formulate our theorem.

Theorem 6.2. *Let s be a tree-structured schema with an empty variable mapping. Using the previously described transformation into a CSP and ignoring the requirement of injectivity, the matches of s in an arbitrary database can be found without any search and in polynomial time.*

Proof. The key to the proof is the following: If the schema is a tree and does not contain any variable definitions then the resulting constraint graph for the constructed CSP is a tree as well. We observe that the CSP has only label and structure constraints, but no variable or injectivity constraints. The label constraints are unary, so they can be incorporated into the domains and thus, be ignored for this proof. Hence, we only must care about the structure constraints. When we want to construct the constraint graph we simply replace every arc in the schema by a vertex representing the variable introduced for the arc. The only constraints being introduced are the source and the target constraints for these arcs. In the constraint graph two edges are added between every newly introduced vertex and the vertices representing the source and target of the respective arc. Figure 6.5 illustrates this construction by showing a tree-structured schema on the left side and the resulting constraint graph on the right side. During the

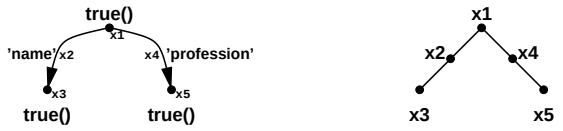

Figure 6.5: A tree-structured schema and the corresponding constraint graph

construction of the constraint graph the properties connectivity and acyclicity remain untouched. Hence, if the schema is a tree the constraint graph is a tree as well.

A tree-structured constraint graph always has width one [Fre82]. Because of the Theorem 6.1, it is clear that after making our CSP node and arc consistent, there exists

a search order that is backtrack-free. This search order can be found by using breath-first traversal of the constraint graph [Fre82]. Making a CSP node and arc consistent can be achieved in polynomial time [Mac77, MH86]. □

It is important to note that the injectivity constraints heavily blow up the constraint graph. Because every pair of vertices and every pair of arcs in the schema is linked by an injectivity constraint, the constraint graph then contains two cliques of sizes $|V^{(s)}|$ and $|A^{(s)}|$. This immediately implies that the constraint graph has a width of at least $max(\{|V^{(s)}|-1, |A^{(s)}|-1\})$. Hence, no polynomial algorithm can ensure the necessary level of consistency. Thus, it might be preferable to postpone the injectivity check and reduce a possibly larger set of CSP solutions in a separate postprocessing step. However, our own practical experiences described in Section 8.1 do not support this.

6.5 Summary

In this chapter we outlined optimization techniques for Constraint Satisfaction Problems. Consistency techniques help to reduce the sizes of domains. Various levels of consistency can be defined; a CSP can be solved with consistency techniques alone. However, because algorithms for achieving arbitrary levels of consistency are not polynomial in time, various search methods exist. They can be compared with respect to the number of nodes in the search tree they visit, and to the number of consistency checks they perform. The order, in which the variables are instantiated, has an influence on the size of the search tree. A good variable ordering must support the first-fail heuristic. In the remainder of the chapter we proved an important observation of our approach. Ignoring the injectivity constraint matches of tree-shaped schemata with no variable definitions can be found without search and in polynomial time.

Chapter 7

Schema Containment and Optimization

As far as the laws of mathematics refer to reality, they are not certain;
and as far as they are certain, they do not refer to reality.
(Albert Einstein)

In this chapter we want to explore the optimization potential of our approach. In Chapter 5 we described how we can find the matches of a schema in a database graph without any further information given. In Chapter 6 we showed how to do this more efficiently. Now we describe how we can incorporate the information of previously matched schemata. This optimization is based on the concept of schema containment.

Section 7.1 defines the notion of schema containment. We prove a sufficient condition for schema containment. In the following Section 7.2 we show how we test this sufficient condition using similar techniques to those in Section 5.2 for transforming the matching problem. We make use of the knowledge of schema containment in two ways and demonstrate them in Section 7.3. If we are looking for the matches of the contained schema and the matches of the containing schema are known, the search space can be reduced. If we are looking for the matches of the containing schema and the matches of the contained schema are known, we can present the first few matches of the containing schema immediately without any search. Because the notion of schema containment is related to the more traditional notion of query containment, we review query containment in Section 7.4. Finally, Section 7.5 gives a summary of this chapter.

7.1 Schema containment

In this section we define the notion of schema containment and give a sufficient condition for it. The definition is straightforward.

Definition 7.1 (Schema containment). A schema s_1 *contains* a schema s_2 (denoted by $s_1 \supseteq s_2$) if for all databases o all matches of s_2 are also matches of s_1.

This definition is related to the more traditional notion of query containment as defined in [ASU79]. Intuitively, the potential for optimization using schema containment is the following. From the definition we immediately observe that if a schema s_1 contains another schema s_2 then

1. matches of s_2 can only be found among the matches of s_1. If we want to find the matches of s_2 and already have the ones for s_1 we can *reduce the search space.*

2. all matches of s_2 are also matches of s_1. If we want to find the matches of s_1 and already have the ones for s_2 we can present the *first few matches immediately*, thereby reducing the latency. Of course, there may exist more matches for s_1.

We will discuss this in more detail in Section 7.3. Before we make use of schema containment we have to find a way to get the information about it. In this section we give a sufficient condition for schema containment; and in the next section we describe how we test this condition using constraints.

The following condition assumes the notion of *predicate containment.* A predicate p_1 contains a predicate p_2 if for all labels x the implication $p_2(x) \longrightarrow p_1(x)$ holds.

Theorem 7.1. *A schema s_1 contains another schema s_2 if*

1. *the graph of s_1 is a subgraph of the graph of s_2,*

2. *for all nodes and arcs in s_1 the predicate of the node or arc contains the predicate of the respective node or arc in s_2,*

3. *for every pair of nodes or arcs in s_1 being mapped to the same variable, their corresponding elements in s_2 are also mapped to the same variable or are labeled with the same constant predicate, and*

4. *for all arcs in s_1 the path descriptions, if present, indicate that paths in s_1 are at least as long as the respective ones in s_2.*

Proof. Let s_1 and s_2 be two schemata that fulfill the conditions of the theorem. Let o be at match of s_2. We have to prove that o is also a match of s_1. Let m be the match function between s_2 and o. Let i be the isomorphic embedding of s_1 into s_2. Let m' be $m \circ i$. We show that m' is a match function between s_1 and o.

1. For any $x \in V^{(s_1)}$ the predicate $l^{(s_2)}(i(x))$ holds for the label of the node $m(i(x))$ in the object o, because m is a match function between s_2 and o. The predicate $l^{(s_2)}(i(x))$ is contained in the predicate $l^{(s_1)}(x)$. Therefore, the predicate $l^{(s_1)}(x)$ also holds for the label of the node $m(i(x))$, i.e., for $l^{(o)}(m'(x))$.

2. Similarly, for any arc $x \in A^{(s_1)}$ the predicate $l^{(s_2)}(i(x))$ holds for the labels of all arcs in the trail $m(i(x))$ in the object o. Therefore $l^{(s_1)}(x)$ also holds for the labels of all arcs in $m(i(x))$.

3. For an arbitrary pair of elements $x_1, x_2 \in V^{(s_1)} \cup A^{(s_1)}$ being mapped to the same variable (i.e., $v^{(s_1)}(x_1) = v^{(s_1)}(x_2)$), we know that either $v^{(s_2)}(i(x_1)) = v^{(s_2)}(i(x_2))$, or the labels of $i(x_1)$ and $i(x_2)$ are the same constant predicate. Either property ensures that the labels of their matching elements in the object o are the same, i.e., that $l^{(o)}(m(i(x_1))) = l^{(o)}(m(i(x_2)))$ and thus, $l^{(o)}(m'(x_1)) = l^{(o)}(m'(x_2))$.

4. For any arc $x \in A^{(s_1)}$ the length of the trail $m(i(x))$ (which is the same as $m'(x)$) is bound by $q^{(s_2)}_{min}(i(x))$ and $q^{(s_2)}_{max}(i(x))$. Because $q^{(s_1)}_{min}(x) \leq q^{(s_2)}_{min}(i(x))$ and $q^{(s_1)}_{max}(x) \geq q^{(s_2)}_{max}(i(x))$, the match condition is also fulfilled for m'.

Hence, m' is a match function between s_1 and o. □

The reverse direction of this implication does not hold as can be seen in Figure 7.1. The three schemata are "semantically identical" (i.e., they match the same objects), but they do not fulfill the conditions of the theorem.

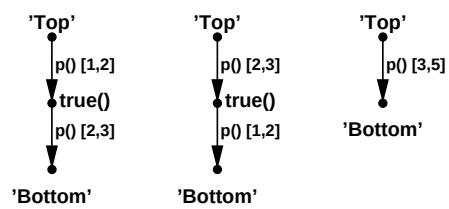

Figure 7.1: Three semantically identical schemata

7.2 Testing schema containment using constraints

We describe the testing of the correctness criterion specified in the previous section again as a CSP. This problem is, like the problem of matching a schema into a database, related to the SUBGRAPH-ISOMORPHISM problem. Similar to the principles shown in Figure 5.5 in Section 5.2, we transform schema s_2 into domains and schema s_1 into variables. Again, we derive constraints representing the label part, and constraints representing the structure part of the containment. Let us call the mapping that we are searching a *containment embedding* from s_1 into s_2. Keep in mind that it is s_2 that is contained in s_1 once we find this mapping.

The simple example in Figure 7.2 shows two schemata s_1 and s_2 where s_2 is contained in s_1. We are thus looking for a containment embedding of s_1 into s_2 to verify

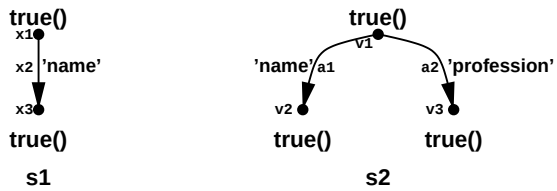

Figure 7.2: Schema containment

this. Variables and domains are introduced as in Section 5.2.

$$X = \{x_1, x_2, x_3\}$$
$$D_1 = D_3 = D_V = \{v_1, v_2, v_3\}$$
$$D_2 = D_A = \{a_1, a_2\}$$

The structure constraints are also the same as before.

$$C^{src}_{(x_2,x_1)} = \{(a_1, v_1), (a_2, v_1)\}$$
$$C^{tar}_{(x_2,x_3)} = \{(a_1, v_2), (a_2, v_3)\}$$

For the label part we must ensure that the predicates in s_1 contain the respective ones in s_2 using the definition of predicate containment. In the example, this is rather easy to achieve. The predicate $true()$ contains every predicate and a constant predicate $X = c$ is contained by a predicate p if $p(c)$ holds.

$$C^{pred}_{(x_1)} = \{(v_1), (v_2), (v_3)\}$$
$$C^{pred}_{(x_2)} = \{(a_1)\}$$
$$C^{pred}_{(x_3)} = \{(v_1), (v_2), (v_3)\}$$

In the general case, we assume that information on predicate containment is explicitely given. Such information can be provided in various ways:

1. A specific predicate can be in an unconditional relationship to all other predicates: $true() \supseteq p$, $false() \subseteq p$.

2. A class of predicates can be in a conditional relationship to all other predicates: $p \supseteq (x = c)$, if $p(c)$.

3. Two specific predicates can be in an unconditional relationship to each other: $number() \supseteq integer()$.

4. Within a class of predicates conditional relationships can exist: $x < a \supseteq x < b$, if $a \geq b$.

In all of the preceding examples, p denotes an arbitrary predicate.

The CSP for the example has the solution (v_1, a_1, v_2), which corresponds to the containment embedding that we already were aware of. We rewrite the embedding as $\{(x_1, v_1), (x_2, a_1), (x_3, v_2)\}$, store it, and see how we can make use of it in the next section.

Variables and paths are also treated in a similar manner as before. If s_1 contains elements that are linked via a variable, we must make sure that the respective elements in s_2 are also linked via a variable or have the same constant predicate as their label. Constraints on the path length ensure that paths in s_1 are not shorter than the respective ones in s_2.

It is important to note that the test for schema containment is independent of the size of the database. Only the number of such containment testings has an influence on the run-time.

7.3 Making use of schema containment

In this section we describe how we make use of schema containment once we detect it using the methods described in the previous section. As we already mentioned before we can make use of both containment directions. If a schema s_1 contains another schema s_2 and know the matches

1. of s_1 then we can *reduce the search space* when looking for the matches of s_2.

2. of s_2 then we can present the *first few matches* of s_1 *immediately.*

We show how we can often dramatically reduce the search space if we have a schema s_1 together with its matches in a database and another schema s_2 that is contained in s_1 and whose matches we are looking for. Consider the example in Figure 7.3. On top of the figure there a schema s_2. The schema s_1 containing s_2 is shown on the left of the figure. The containment is the same as in Figure 7.2 in the previous section. We have renamed the elements a little, the containment embedding is now $\{(y_1, x_1), (y_2, x_2), (y_3, x_3)\}$.

We are interested in the matches of s_2 in our standard sample database shown in Figure 2.4 on Page 35. There is nothing we can do about the variables x_4 and x_5 in our optimization, because they are not in the scope of the containment embedding. However, x_1, x_2, and x_3 can only be matched to the matches of their respective partners in the containment embedding. Intuitively, we are looking for those matches of s_1 that

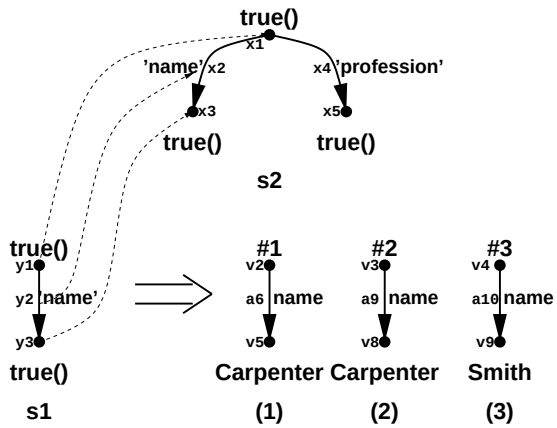

Figure 7.3: Reducing the search space

can be extended by a profession-arc. We get the following variables and domains:

$$
\begin{aligned}
X &= \{x_1, x_2, x_3, x_4, x_5\} \\
D_1 &= \{v_2, v_3, v_4\} \\
D_2 &= \{a_6, a_9, a_{10}\} \\
D_3 &= \{v_5, v_8, v_9\} \\
D_4 &= D_A = \{a_1, a_2, a_3, \ldots, a_{12}\} \\
D_5 &= D_V = \{v_1, v_2, v_3, \ldots, v_{11}\}
\end{aligned}
$$

We immediately observe that we greatly reduced the domains of x_1, x_2 and x_3. The reduction will probably be even more substantial in real life examples. To fully exploit the containment information we introduce an additional constraint.

$$C^{sol}_{(x_1,x_2,x_3)} = \{(v_2, a_6, v_5), (v_3, a_9, v_8), (v_4, a_{10}, v_9)\}$$

Vice versa, consider the case that we have a schema s_2 together with its matches and a schema s_1 containing s_2 whose matches we are looking for. We can immediately derive some matches of s_1 by looking at the containment embedding of s_1 into s_2. Consider the example in Figure 7.4. The schemata are the same as in the previous example, but we are now interested in the matches of s_2 and have the ones for s_1. The containment embedding of s_1 into s_2 is $\{(y_1, x_1), (y_2, x_2), (y_3, x_3)\}$, just as before.

The matches of x_1, x_2 and x_3 are immediately also matches of y_1, y_2 and y_3. Thus, (v_2, a_6, v_5) is one solution of the CSP constructed for s_1. There may be more solutions, though. Incidently, there are more solutions in this case as we demonstrated in the previous example in Figure 7.3.

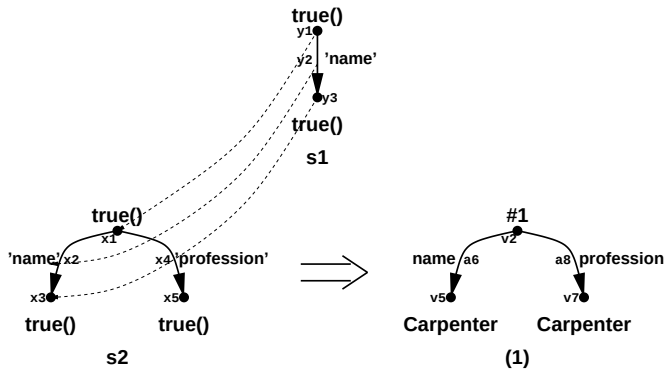

Figure 7.4: First few matches

7.4 Traditional query containment

Query containment deals with the following general problem: Given two queries, can a relationship between them be established that says that the answers to one of the queries is always a subset of the answer to the other one, no matter what database is queried. More formally, a query q_1 contains a query q_2, written $q_1 \subseteq q_2$, if for all databases the answers of q_2 are a subset of the answers of q_1. The queries q_1 and q_2 are equivalent, written $q_1 \equiv q_2$, if $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$.

Query containment has a variety of applications. Originally it was used for query optimization [CM77, ASU79, SY81]. More recently, it has become an important notion in the context of rewriting of queries using views [LMSS95, CKPS95]. In particular, this notion plays a significant role for materializing views in data warehouses. Levy and Sagiv use the notion of query containment to investigate kinds of queries that are independent of updates [LS93]. Query containment can also be applied to deciding which views to materialize in a data warehouse [HRU96].

Query containment for first order conjunctive queries is decidable and even NP-complete [CM77]. On the other hand, containment of Datalog programs is undecidable [Shm93]. In the context of semistructured data Florescu, Levy, and Suciu showed that query containment for a union-free, negation-free subset of their StruQL language is decidable [FLS98]. Additionally, they proved that query containment for a further subset of this language restricting the allowed kinds of path expressions is NP-complete.

7.5 Summary

In this chapter we demonstrated how to make use of schema containment for query optimization purposes. We introduced the notion of schema containment, gave a sufficient

condition for schema containment, and proved its sufficiency. We then incorporated the concept of schema containment into our constraint-based optimization. Testing the sufficient condition is again reduced to an equivalent CSP using similar techniques as before. We showed how the knowledge of schema containment can be used in two different ways depending on the direction of the containment. If we are looking for the matches of the contained schema and the matches of the containing schema are known, the search space can be reduced. Vice versa, if we are looking for the matches of the containing schema and the matches of the contained schema are known, we can present the first few matches of the containing schema immediately without any search, thus reducing latency. Finally, we discussed the more traditional notion of query containment.

Part IV

Implementation and Conclusion

Chapter 8

Implementation

Programming today is a race between software engineers
striving to build bigger and better idiot-proof programs,
and the Universe trying to produce bigger and better idiots.
So far, the Universe is winning.
(Rich Cook)

This chapter describes our implementation efforts and their results. Again, we emphasize matching schemata, because this is the computationally challenging part. We use the optimization techniques described in the previous chapters. Our implementation is threefold: In Section 8.1 we describe our first steps consisting of a schema matcher implemented in Prolog. We test various optimization ideas there. We outline how we integrate XML documents in Section 8.2. Finally, Section 8.3 describes our schema matcher based on the constraint solving system ECLiPSe. The user interface to this system is presented in Section 8.4. We conclude with a summary in Section 8.5.

8.1 First steps: A Prolog-based schema matcher

To gain some experience with the constraint-based optimization, we first implemented a schema matcher based on the public domain SWI-Prolog software [Swi]. Our system consists of several components as shown in Figure 8.1. The arrows indicate, which components are needed by which other components.

Object and schema maintainer These two components manage the databases and the schemata. They provide predicates for creating and destroying objects as well as for creating and destroying vertices and arcs. Automatic checking for dangling arcs is performed. Other auxiliary predicates are provided: for prettyprinting objects, for finding paths and their sources and targets, and for computing induced

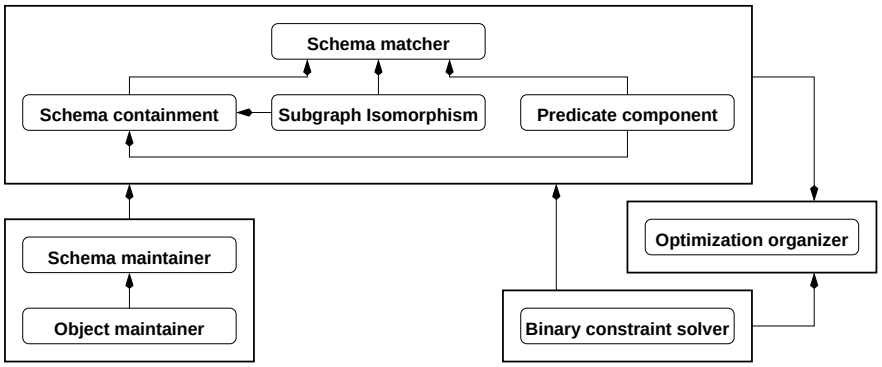

Figure 8.1: Architecture of the Prolog-based schema matcher

subobjects (induced by some set of vertices and arcs). Furthermore, the schema maintainer can analyze the labels. It can detect their "type", i.e., whether it contains variable definitions or path descriptions. The schema maintainer can extract the predicate from a label and store all relevant information in some predefined form.

Binary constraint solver The binary constraint solver can solve binary CSPs. The user can specify a CSP by defining variables, domains, and constraints. The domains can be specified either by assigning them to a variable or by giving them an identifier. The latter is useful if several variables share domains. The following optimization techniques are implemented: node consistency, two different variable orderings (smallest domain first, most constraints first), and domain reducing functions.

Domain reducing functions were initially proposed by Rudolf [Rud98]. We illustrate them in the context of graph matching. Once we instantiate a variable representing an arc, we immediately know that there is only one possible value remaining for the variables representing its source and target vertex. This can also be achieved by dynamically performing arc consistency algorithms, but computing it via functions *source* and *target* is certainly more efficient, because no domain access is necessary. We did not make good experiences with the inverse functions *incoming* and *outgoing*, which we applied to instantiated vertices. The main reason for this is that in our approach these functions also compute all incoming and outgoing paths, which slows down the query processing considerably. Because these functions have to be called whenever a variable gets instantiated, the overhead gets bigger if many instantiations are "useless".

Subgraph isomorphism component This component creates a CSP that solves the SUBGRAPH-ISOMORPHISM problem. This CSP serves as the base for both

direct schema matching and testing schema containment. Only the constraints representing the structure part of the match and the injectivity constraints are introduced here. Optimization potential lies in testing the injectivity of the match afterward (instead of ensuring it by constraints), and in precomputing constraints. The source and target constraints are the same for every arc in a schema and also for every schema to be matched against a fixed database.

Predicate component The predicate component implements the predefined predicates and containment information between them. The supported predicates are:

- $true$ and $false$: These predicates are always or never fulfilled, respectively. The predicate $true$ contains every other predicate, the predicate $false$ is contained in any other predicate.
- $caseignore(Atom)$: This predicate is true for every label that is the equal to $Atom$, ignoring the case of the letters. $caseignore(Atom)$ contains the constant predicate $Atom$. In general, every predicate p contains the predicate $X = c$ if $p(c)$ holds.
- $sgrep(Atom)$: This predicate holds for every label that has $Atom$ as a sublabel. The predicate $sgrep(Atom1)$ contains $sgrep(Atom2)$ if $Atom1$ is a subatom of $Atom2$. In the example in Figure 2.4 on Page 35 the predicate $sgrep('arp')$ would match only the nodes labeled $'Carpenter'$, i.e., v_5, v_7, and v_8. In contrast, $sgrep('ar')$ would additionally match the node v_6 labeled $'Harry'$.
- $integer$ and $number$: They hold if the label is an integer or a number, respectively. $number$ contains $integer$.
- $lessthan(eq)(Num)$ and $greaterthan(eq)(Num)$: These predicates hold if the label is a number and this number is less than, less than or equal, greater than, or greater then or equal to Num, respectively. The containment information is as follows: $lessthan(eq)(Num1)$ contains $lessthan(eq)(Num2)$ if $Num1 \geq Num2$, $greaterthan(eq)(Num1)$ contains $greaterthan(eq)(Num2)$ if $Num1 \leq Num2$.

Schema containment component This component detects containments between schemata. It uses the subgraph isomorphism component and the predicate component. The techniques discussed in Section 7.2 are implemented here.

Schema matcher The schema matcher is the main component of the system and uses all the other components. Its purpose is to find all the matches of a schema in a database. With the help of the subgraph isomorphism component the basic

CSP is constructed. This CSP is adapted in case variable and path descriptions are present in the schema. One can optionally turn on the usage of the schema containment component, both to reduce the search space and to present the first few matches immediately.

The optimization organizer arranges the optimization switches into levels to handle them easier. We will now demonstrate the effect the different optimization techniques have on the performance. We summarize our results in Table 8.1. The database we use is our standard example in Figure 2.4 on Page 35. Schema 1 is the schema in Figure 3.1 on Page 45, Schema 2 is the schema in Figure 3.3 on Page 46, and Schema 3 is the schema in Figure 3.5 on Page 50. Schema 1 contains only constants and predicates, Schema 2 contains a variable definition, and Schema 3 contains a path description. The performance is measured in the number of logical inferences. The tests were run using SWI-Prolog version 3.2.8.

	Schema 1	Schema 2	Schema 3
No optimization	25,895	104,840	268,348
Node consistency (NC)	15,459	57,369	162,868
Variable ordering (smallest domain / VO1)	25,925	104,906	106,791
Variable ordering (most constraints / VO2)	26,046	14,489	91,509
NC + VO1	15,508	12,182	63,855
NC + VO2	15,725	12,902	115,779
External Injectivity check (EI)	26,645	159,538	326,512
Precompute constraints (PC)	26,781	103,265	266,774
Function-based domain reduction (FDR)	26,147	106,036	270,148
NC + VO1 + PC	16,394	10,607	62,281
NC + VO2 + PC	16,611	11,327	114,205
NC + VO1 + PC + FDR	16,598	10,042	61,309
NC + VO2 + PC + FDR	16,815	10,762	74,833

Table 8.1: Performance of the Prolog-based schema matcher

One can easily see that the performance heavily depends on the choice of optimizations. An improvement of a factor of 15 can be achieved. It is not so easy to discover, which of the switches are best, though. VO2 on its own seems better than VO1, however, NC + VO1 is better than NC + VO2. The other optimizations seem to have little influence. Function-based domain reduction has almost no effect on the performance. Combined with PC it significantly improves the performance of NC + VO2 only for Schema 3. To check the injectivity of the match externally does not seem to be a good idea. Precomputing the constraints will improve the performance only when several

schemata are matched against the same database.

	Schema 1	Schema 2	Schema 3	
No optimization	25,895	104,840	268,348	
Search space reduction	15,008	22,156	205,619	79,414
Containment test only	2,087	2,897	2,427	2,911
Number of possible containment tests	6	29	26	65

Table 8.2: Schema containment in the Prolog-based schema matcher

Table 8.2 summarizes our results with respect to making use of schema containment. The first row shows again the performance if no optimization is used. For getting the results in the second row we used one schema containing the respective example and precomputed its matches in the database. So the numbers depend on the choice of the containing schema. For Schema 3 we used two different containing schemata. The third row of the table shows how many inferences were needed to prove the schema containment. Remember that this test is included in the complete matching process, i.e., the numbers in the third row are included in those in the second row. We observe that one containment test alone takes about a constant amount of time. Note that this test is independent of the size of the database. Computed from the first three rows, the fourth row indicates how many containment tests between schemata can be performed, so that the complete number of inferences is still less than the number in the first row. Note that this number depends on the size of the database, because the numbers in the first and second row depend on the size of the database. Because we used a very small database, the numbers in the fourth row will dramatically increase for real life examples, whereas the numbers in the third row will remain the same.

8.2 Integrating XML documents

In this section we describe how we transform XML documents into labeled directed graphs, so that we can use them in our approach. We have already discussed XML in Section 2.5. There we have seen that an XML document consists of constructs of five different types: element, data, document type definition, processing instruction and comment. From the database point of view only elements and data are interesting. Document Type Definitions could be used to generate some schema information.

We have developed a parser that turns XML documents into definitions of labeled directed graphs. Subelements and the attributes attached to elements are turned into subgraphs. A tree structure is constructed in this manner. The only exception, where the database constructed is not a tree, arises when links using the `id`- and the `idref`-attribute are present. For every `idref`-attribute an additional arc to the defining

element (the one that has the appropriate `id`-attribute) is added. Our parser was implemented using lex and yacc, or rather their GNU versions flex and bison. The simple example that we introduced in Section 2.5 was:

```
<?xml version="1.0" standalone="yes"?>
<!-- This is a most simple example. -->
<EXAMPLE id="1" foo="bar">
This is a test.
</EXAMPLE>
```

A graphical representation of this example can be found in Figure 2.7 on Page 39. Our parser generates the following Prolog code.

```
% This file was automatically generated using XML2PL.
:- begin_object('XML_Document').
:- create_vertex('XML_Document',id_0,"XML_Document").
:- create_vertex('XML_Document',id_1,"XML").
:- create_arc('XML_Document',id_6,id_0,id_1,"XML Declaration").
:- create_vertex('XML_Document',id_2,"1.0").
:- create_arc('XML_Document',id_3,id_1,id_2,"version").
:- create_vertex('XML_Document',id_4,"yes").
:- create_arc('XML_Document',id_5,id_1,id_4,"standalone").
:- create_vertex('XML_Document',id_7," This is a most simple example. ").
:- create_arc('XML_Document',id_8,id_0,id_7,"comment").
:- create_vertex('XML_Document',id_9,"This is a test.").
:- create_arc('XML_Document',id_16,id_0,id_9,"EXAMPLE").
:- create_vertex('XML_Document',id_10,"1").
:- create_arc('XML_Document',id_11,id_9,id_10,"id").
:- create_vertex('XML_Document',id_12,"bar").
:- create_arc('XML_Document',id_13,id_9,id_12,"foo").
:- end_object('XML_Document').
```

The `begin_object/1` and `end_object/1` predicates tell the object maintainer that the predicates inbetween contain the complete object definition. Hence, the object maintainer can perform a check that no dangling arcs exist. Only in this case the predicate `end_object/1` will succeed. The predicate `create_vertex/3` takes the database, the identifier and a label as its arguments, `create_arc/5` additionally takes the identifiers of the source and target vertices.

8.3 An ECLiPSe-based answering system

ECLiPSe (ECLiPSe Common Logic Programming System) is a Prolog-based system whose aim is to serve as a platform for integrating various logic programming extensions, in particular Constraint Logic Programming (CLP) [Ecl]. We used the experiences we gained from the Prolog-based schema matcher to build a schema matcher based on ECLiPSe. This system has a similar architecture as the former one (see Figure 8.1). There is of course no constraint solver in this new system, because this functionality is provided by ECLiPSe. Furthermore, we omitted the optimization organizer. We do provide a graphical user interface based on Tcl/Tk. ECLiPSe provides an interface for data exchange between the core of ECLiPSe and Tcl/Tk. Because the system is in its architecture very similar to the Prolog-based system, we omit repeating the description of the components. Rather, this section explains in some detail how the constraints are represented and the search is performed.

For solving the Constraint Satisfaction Problems we use the Finite Domain Library provided by ECLiPSe. This library implements constraints over finite domains. These constraints can contain integer as well as atomic (i.e., atoms, strings, floats, etc.) and ground compound (e.g., f(a, b)) elements. CSP variables are called *domain variables* in the ECLiPSe context. They are associated to domains represented as lists using the infix predicate `::/2`. We implement unary constraints by explicitely reducing the domain of the variable. The following example introduces a variable with domain $\{1, 2, 3\}$ and then implements the unary constraint $C^{odd}_{(x)} = \{(1), (3)\}$.

```
example(X) :-
        % introduce the variable and assign a domain
        X :: [1,2,3],
        % retrieve the domain and unify it with Domain
        dvar_domain(X,Domain),
        % convert the unary constraint from a list into a domain
        list_to_dom([1,3],ConstrainedDomain),
        % build the intersection between the original domain
        % and the constraint
        dom_intersection(Domain,ConstrainedDomain,NewDomain,_),
        % set the domain to its new value
        dvar_update(X,NewDomain).
```

Obviously this technique cannot be used to implement constraints of an arity greater than one. An efficient way of implementing such constraints is to link their components via an index over lists using the `element/3` predicate. The following example introduces two variables x and y with domains $\{1, 2, 3\}$ and $\{a, b\}$ and a binary constraint stating

y to be a if and only if x is odd, i.e., a constraint $C_{(x,y)} = \{(1,a),(2,b),(3,a)\}$. One can easily see that this technique can be used to implement constraints of arbitrary arity.

```
example(X,Y) :-
        % introduce the two variables
        X :: [1,2,3],
        Y :: [a,b],
        % link the possible value pairs by the index variable I
        element(I,[1,2,3],X),
        element(I,[a,b,a],Y).
```

The `element/3` predicates are examples of *delayed goals*. Execution of these goals is suspended until the involved domains are changed, either explicitely by a `dvar_update/2` predicate or implicitly by the backtracking procedure. If the above predicate is called like

```
[eclipse 1]: example(X,Y), X = 2.
```

then the trace of the execution shows that once `X` gets unified with `2` the suspended `element/3` predicates are woken.

```
  (7) 3 RESUME  element(I{[1..3]}, [1, 2, 3], 2)   %> creep
  (7) 3 EXIT  element(2, [1, 2, 3], 2)   %> creep
  (10) 3 RESUME  element(2, [a, b, a], Y{[a, b]})   %> creep
  (10) 3 EXIT  element(2, [a, b, a], b)   %> creep

Y = b
X = 2
yes.
```

Now, the backtracking works as follows. The `solution/1` predicate implements a simple version of chronological backtracking. The argument given is a list of unbound domain variables. The `deleteff/3` predicate selects the variable with the smallest domain to be the chosen one for instantiation. This predicate guarantees a dynamic variable ordering; the variable with the smallest domain at the current point in time, i.e., after performing the full lookahead using the suspend/wake mechanism, gets chosen. An alternative is the `deleteffc/3` predicate, which chooses the variable that is involved in the highest number of constraints. After the selection of a variable the predicate `indomain/1` is used to instantiate the chosen variable. Because the dynamic

domain reduction is performed automatically for all variables, it is sufficient to recursively call the `solution/1` predicate with the remaining list of unbound domain variables.

```
solution([]) :- !.
solution(List) :-
        % use variable with smallest domain
        deleteff(Var,List,Rest),
        % instantiate the variable to a member of its domain
        indomain(Var),
        % continue with the rest list
        solution(Rest).
```

Because ECLiPSe does not provide a tool for counting the number of logical inferences, we decided to implement a counter for the number of backtracks. This number seems to be a good measure for the quality of both constraint propagation and search heuristics. The predicates shown next implement such a counter. We use ideas from the ECLiPSe tutorial here.

```
:- local variable(backtracks), variable(deep_fail).

init_backtracks :-
        setval(backtracks,0).

get_backtracks(B) :-
        getval(backtracks,B).

count_backtracks :-
        % succeeds on the first call...
        setval(deep_fail,false).
count_backtracks :-
        getval(deep_fail,false),
        setval(deep_fail,true),
        % ...but increments counter when backtracking occurs
        incval(backtracks),
        % continue the backtracking
        fail.
```

The flag `deep_fail/0` ensures that backtracking to exhausted choices does not increment the count. The `solution/1` predicate has to be adapted to incorporate the counter.

```
solution(List) :-
        % set the counter to zero
        init_backtracks,
        (
            % from List = List1 to List2 = [] do
            fromto(List,List1,List2,[]) do
            % use variable with smallest domain
            deleteff(Var,List1,List2),
            % call the counting predicate,
            % actual counting only occurs during backtracking
            count_backtracks,
            % instantiate the variable to a member of its domain
            indomain(Var)
        % set List1 to List2
        ),
        % get the number of backtracks and print it
        get_backtracks(B),
        printf("Solution found after %d backtracks.%n",[B]).
```

It turns out that with the used heuristic of always instantiating the variable with smallest domain the results are very good. For our standard example in Figure 2.4 on Page 35 the first solution for all tested schemata was found without any backtracking at all. We get the same positive result when using the second example database (the relational one) in Figure 2.6 on Page 36 together with the schemata from Section 4.4. This is a quite amazing result, because the CSPs resulting from these schemata consist of close to 20 variables.

The largest example that we used to test our system was an XML document containing Shakespeare's "Hamlet". Using our parser described in the previous section this document was turned into a database consisting of more than 13,000 nodes and arcs. This database takes 82 seconds to be loaded into ECLiPSe. Searching the node containing "To be, or not to be: that is the question:" takes 161 seconds. These tests were run on a Sun with four UltraSPARC II processors and one gigabyte of RAM. The tests demonstrate that solving our kinds of CSPs can be done virtually without backtracking, but that building the CSPs can consume a considerable amount of time for large examples. An attempt to solve this problem would be to store the facts about the database vertices and arcs in a relational database management system and build the domains and constraints using SQL.

8.4 The user interface

ECLiPSe provides a programming interface to the script language Tcl/Tk. This gives us the possibility to combine the sophisticated techniques of the ECLiPSe constraint solver with the comfort of graphical user interfaces. We used the programming interface to implement an intuitive graphical user interface to the ECLiPSe-based schema matcher. This interface allows the user to deal with multiple databases and schemata and to get details on matches between them. Figure 8.2 gives the reader an impression of the interface.

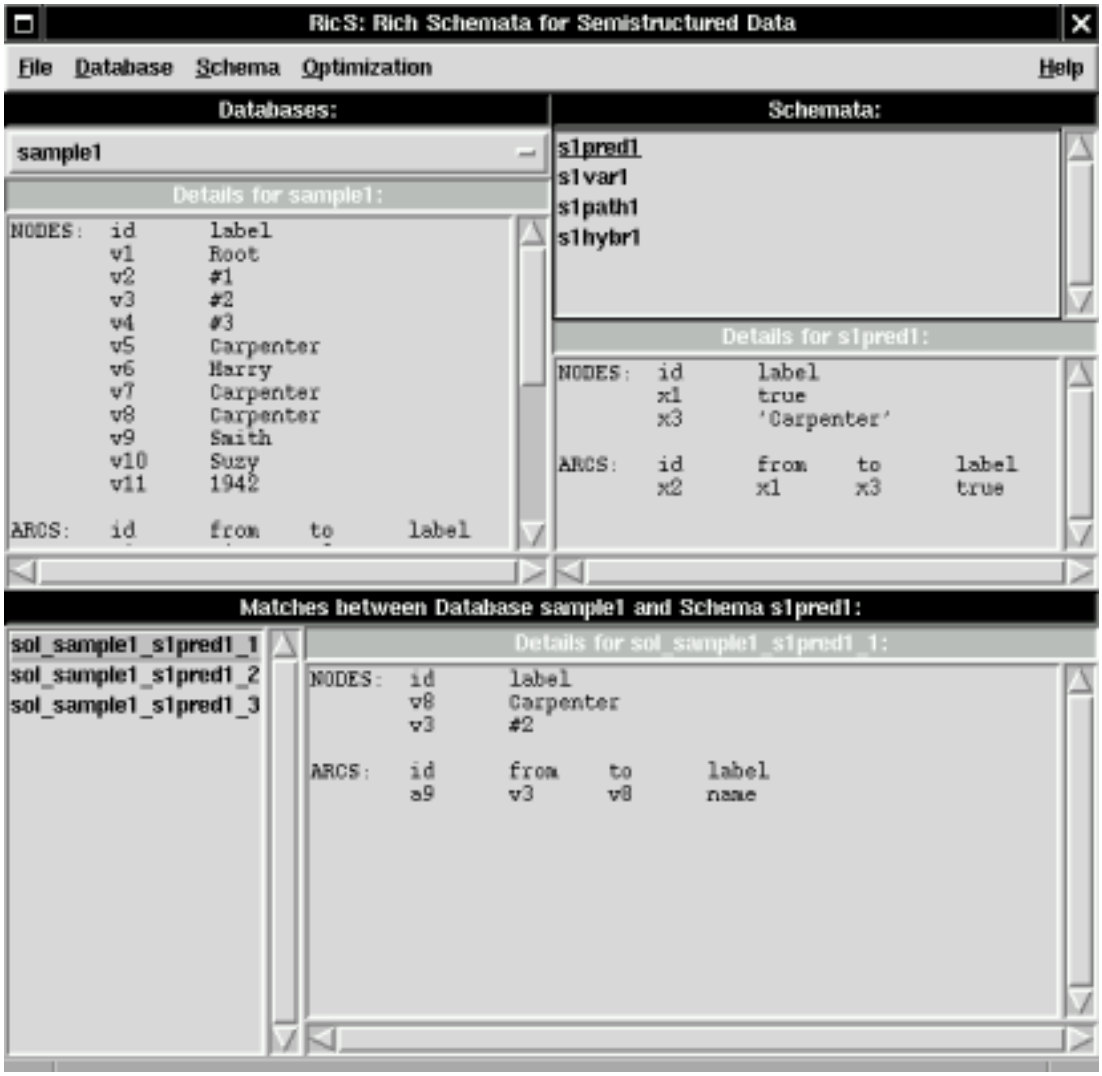

Figure 8.2: Graphical user interface to the ECLiPSe-based schema matcher

With this interface the user can comfortably open, close, and select databases and schemata. The user can also invoke the computation of schema matches. Furthermore,

we incorporated the integration of XML documents.

Technically, we use Tk as a client to the ECLiPSe system. The `tk/1`-predicate allows to start a Tk script from the ECLiPSe command line. The library `tk` must be loaded beforehand. The embedded `prolog` command provides means for communication from the Tk script to the core ECLiPSe system.

8.5 Summary

This chapter presented our implementation efforts. To gain some experiences with programming using constraints we first implemented a schema matcher into a Prolog system. Various optimization techniques have a significant effect on the performance. We showed how we integrated XML documents into the system. The ECLiPSe-based schema matcher uses the finite domain library to solve CSPs. We demonstrated how the constraints are represented. In all our experiments we found out that typically only very little backtracking is needed. Furthermore, we presented our graphical user interface.

Chapter 9

Conclusion

Prediction is very difficult, especially about the future.
(Niels Bohr)

9.1 Summary of the thesis

Traditional database management requires design and ensures declarativity. Semistructured data, "data that is neither raw data nor strictly typed" [Abi97], lacks a fixed and rigid schema. Often their structure is irregular and implicit. Examples for semistructured data include XML and HTML files, BibTEX files, and genome data stored in ASCII files. In this field a more flexible approach for querying is needed. We propose to split the notion of query into a "What"- and a "How"-part. The major advantage of this idea is that the "What"-part can be stored and used as a partial schema for the database. Partial schemata can give users an idea about the content of the database. Furthermore, they can be used for query optimization. A database system designed in this manner reflects the degree of structure of the data on many levels. Its usability and its performance grow with a higher degree of structure and with the time the system has been in usage.

Chapter 1 provides an introduction to the field of semistructured data and illustrates the basic idea of our approach. Furthermore, pointers to related fields are given. In Chapter 2 we present the underlying syntax of labeled directed graphs taking into account graph theoretic aspects. Chapters 3 and 4 introduce the proposed query language. In Chapter 3 the "What"-part of a query is characterized by introducing the notion of (partial) schema. These schemata are represented as labeled directed graphs and cover semantic concepts, such as predicates, variable definitions, and path descriptions. The notion of conformity between schemata and databases is defined using graph morphisms. Chapter 4 addresses the "How"-part of a query. We propose how

to process matches of a schema in a database and introduce schema, focus, and transformation queries. The three chapters also introduce other models for semistructured data, such as XML and OEM, other notions of schema, such as DataGuides, and other query languages, such as Lorel, UnQL, and XML-QL.

We identify that the most challenging part of query execution is to find the matches of a schema in a database. We tackle this problem using techniques from the area of Constraint Satisfaction Problems (CSPs). Chapter 5 provides an introduction to the field of Constraint Satisfaction. In a CSP variables are associated to domains. Constraints restrict the values that variables can simultaneously take. We demonstrate how the problem of finding all matches of a schema in a database can be reduced to a CSP. We prove the correctness and the completeness of this transformation. Chapter 6 discusses optimization techniques for CSPs. We observe that consistency techniques, search algorithms, and the order in which the variables are instantiated have a significant influence on the performance. We prove an interesting property of our approach: The matches of tree-shaped schema without variable definitions can be found without search and in polynomial time if the requirement of injectivity of the match function is ignored. In Chapter 7 we discuss optimization using schema containment. We define the notion of schema containment and provide a sufficient condition for schema containment. We describe how to test this condition again by reducing it to a CSP. Schema containment can be used in two ways depending on the direction of the containment: It is either possible to greatly reduce the search space when looking for matches of a schema, or to present the first few matches immediately without any search. These three chapters also discuss more traditional cost-based optimization used in the Lore system and the notion of query containment.

Chapter 8 presents our implementation efforts. To gain more experience with constraint programming we first implemented a schema matcher into a Prolog system. We experimented with various optimization techniques and concluded that they indeed have a significant influence on the performance. The constraint system ECLiPSe provides a far better environment for our approach. We demonstrate how constraints are represented in ECLiPSe and give some performance results. Furthermore, we show how to integrate XML documents into the system and provide a comfortable graphical user interface.

9.2 Discussion

This work demonstrates two points. First, partial schemata play an important role in handling semistructured data. Second, query processing using constraint techniques applies general search techniques and provides a more abstract and extensible frame-

work.

Semistructured data needs to be handled in a more flexible manner than well-structured data, such as data in relational databases. We argue that trying to provide a complete schema leads to ad-hoc results, which do not represent the data very clearly. Instead, we suggest to use semantically rich partial schemata. Partial schemata can be of more meaningful nature and ignore irrelevant parts of the database. Statistics on the number of matches that a partial schema has in a database can be used to deduce information on the degree of the structure of the database, and to identify relevant and irrelevant parts of a database. If a large partial schema has many matches then the database is probably rather well-structured; if even small partial schemata have only few matches the database is probably unstructured. Around this idea we define a query language based on matching partial schema. We ensure that partial schemata must not necessarily come from a database designer, but can also be extracted from user queries. Furthermore, we argue that containment relationships between partial schemata can be used for query optimization.

Using Constraint Satisfaction Problems lifts the problem of query processing to a more abstract and extensible level. We demonstrate that applying general search techniques can be an alternative to reasoning about tree traversal strategies. In our approach the query processing is dynamic in the sense that at any point in time decisions on how to proceed further are made. Our theoretical and practical results show that in general only very little backtracking is necessary for processing a query. Thus, we provide an interesting alternative to the traditional cost-based query processing.

There are several possible directions into which future research could lead. Most important, managing the partial schemata needs to be improved. Certainly not all schemata induced by queries can be stored for future use. Rather, an intelligent selection has to be made. To achieve this goal, ideas from materializing views in the data warehouse context can be used. Furthermore, the relationship to XML could be investigated in more detail. In particular, it would be interesting to exploit the relationship between DTDs and partial schemata. To increase the usability of a system a more comfortable interface for specifying a query has to be defined.

Part V

Appendices

Appendix A

Frequently Asked Questions

1. Can you put the message of this thesis into one sentence?

   No, but in two, because to me there are two main ideas in this thesis: The first one is to split up a query into a "What"- and a "How"-part, thus enabling a more flexible query framework. The second one is that, because of its generality and its sophisticated search strategies, query processing using constraints seems to be a feasible alternative to traditional cost-based query optimization.

2. Couldn't the data be modeled more intelligently?

   No. One of the key ideas in semistructured data to me is that the data is not modeled beforehand using Entity-Relationship diagrams, UML or similar techniques. Rather, semistructured data is more related to the idea of reverse engineering. The data is taken as it comes and one tries to get the best out of it. Hence, a general and simple model seems to be the appropriate choice.

3. Is this data representation suitable for the Internet?

   Because the model is rather general, virtually everything can be modeled using it. There is, however, even an intuitive way to model Web domains. Clearly, single HTML pages should correspond to nodes and links should correspond to arcs. Furthermore, a single HTML document itself is similar to an XML document in that it can be transformed into a graph using the ideas presented in Sections 2.5 and 8.2.

4. Your notion of subobject does not conform to my understanding of object-oriented systems. Can you enlighten me?

   Let us call my notion of subobject the syntactic one and your notion of subobject the semantic one. The notion in this thesis is based on the notion of subgraph,

which is purely syntactic. Your intuition is reflected by the idea of schema containment. If a predicate schema is a subobject in the syntactic sense of another predicate schema then all matches of the latter are also matches of the former. That makes the latter a subobject of the former in the semantic sense. We prefer to call this property schema containment.

5. Your notion of schema is confusing and in particular contradictory to the traditional notion of database schema. Can you explain that in more detail?

   I admit that this notion is a bit confusing. A database for semistructured data does not have a schema in the traditional sense. Other approaches try to compute a complete schema for such a database once the database is built. Our approach is based on a set of partial schemata covering parts of the database. This approach seems to be more natural, because a computed complete schema is almost guaranteed to be awkward and not very representative. So, instead of having a complete schema for a database, we have a set of partial schemata each of which we call schema. I seriously thought about other names for this concept: pattern, description, view, ... However, I felt that none of them really works.

6. Who is going to ask such queries?

   Although we do not provide a tool that makes this language a visual query language, implementing such a tool seems quite possible. We have not thought about a representation of this language in a select-from-where manner, although this seems quite possible too. An important aspect of this language is to represent concepts of query languages for semistructured data in an abstract manner. So our language could serve as a middle layer for some other language, such as XML-QL.

7. Doesn't the user have to have a lot of knowledge about the schema (in the traditional meaning) to formulate meaningful queries?

   Yes. A system designed in this way should be seen as going through an evolution. The very first query posed will probably be not very meaningful. But as the system is filled with more and more of the partial schemata (ideally filtered in some intelligent way) the user will get a pretty good idea about the database. Furthermore, a designer may provide partial schemata at least for some parts of the database, so that the system starts already at a higher level of usability.

8. Is this whole backtracking not necessarily too slow and, thus, not suited for databases?

   I do not think so. Combined with suitable heuristics and lookahead techniques the search can be performed reasonably efficient. Our theoretical and practical

observations demonstrate this. A more serious problem to me is that constraint systems are main-memory systems and lack efficient storage mechanisms.

9. Can you provide a result on the decidability of schema containment in your approach?

   No. With respect to schema containment this work is rather pragmatic. We only need a sufficient condition (a correctness criterion) for our ideas and that is all we investigated.

10. Didn't you implement the same thing twice? Why? Wasn't it a very time-consuming thing to do?

    Yes, the ECLiPSe implementation is an extension of the Prolog implementation. The Prolog implementation was done to get some experiences with programming using constraints and with various optimization techniques. We could reuse many parts of the code for the ECLiPSe implementation, because ECLiPSe is also a Prolog-based system. Only the core functionality involving the constraints had to be reimplemented completely.

Appendix B

Fundamentals of Graph Theory and Partially Ordered Sets

This appendix introduces some basic notions from graph theory and the theory of partially ordered sets. The first part is mainly based on the books by Foulds and Jungnickel [Fou92, Jun90], the second one on the book by Trotter [Tro92].

B.1 Fundamentals of Graph Theory

Definition B.1 (Graph). A *graph* (V, E) is an ordered pair where V is a finite and nonempty set, whose elements are called *vertices* or *nodes*, and E is a set of unordered pairs of distinct nodes from v, whose elements are called *edges*.

Some authors permit V to be empty or infinite. We say that an edge $e = \{p, q\} \in E$ (or just $e = pq$) links the nodes (or points) p and q. In this case we call p and q *incident* to e and *adjacent* to each other. Edges are called *adjacent*, if they share a node. We denote the set of all nodes adjacent to v by $\Gamma(v)$.

Definition B.2 (Digraph (directed graph)). A *digraph* (or *directed graph*) is an ordered pair (V, A), where V is a finite and nonempty set and A is a set of ordered pairs of distinct elements of V.

In contrast to graphs the elements of A in a digraph are usually called *arcs*. If (u, v) (or uv) is an arc in a digraph then we call u the *predecessor* of v and v the *successor* of u. An *oriented graph* is a digraph that for every pair of nodes $u, v \in V$ contains at most one of the arcs (u, v) and (v, u).

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called *isomorphic* (we write $G_1 \cong G_2$), if there exists a bijection $f : V_1 \longrightarrow V_2$, such that $uv \in E_1 \iff f(u)f(v) \in E_2$. In this case f is called an *isomorphism*. Isomorphic graphs have the same number of

nodes of every degree and the same number of edges. Isomorphisms keep the adjacency. An *automorphism* of a graph G is an isomorphism of G onto itself.

Various notions of subgraphs are useful. Let $G = (V, E)$ be a graph. A graph $G' = (V', E')$ is called *subgraph* of G, if $V' \subseteq V$ and $E' \subseteq E$. Let $V' \subseteq V$. Then we call the graph $G' = (V', E')$ with $E' = \{(u, v) \in E | u \in V' \wedge v \in V'\}$, that contains all edges from G linking nodes from V' the by V' *induced subgraph* of G. Let E' be a (proper) subset of E. The graph $G' = (V, E')$ is called a (proper) subgraph of G.

Definition B.3 (Complete graph). A graph, in which every pair of its n nodes is linked directly via an edge, is called *complete graph* (or *clique*) and is denoted by K_n.

If we remove the node v (all nodes of the graph G') from a graph G together with all incident edges, then we denote the resulting graph by $G - v$ ($G - G'$). If we remove an edge e we denote the resulting graph by $G - e$.

Connectivity A sequence $(e_1, \ldots, e_n)$ of edges in graph is called *walk*, if nodes $v_0, \ldots, v_n$ with $e_i = v_{i-1}v_i$ for all $1 \leq i \leq n$ exist. We say that the walk *links* v_0 and v_n. A walk is *closed* if $v_0 = v_n$ and *open* otherwise. If all the edges are distinct then we call the walk a *trail*. If all the nodes are distinct (and thus all the edges as well) then we call the walk a *path*.

Definition B.4 (Connected graph). A graph is called *connected* if every pair of nodes is linked via a path.

A graph is called k-connected if at least k edges have to be removed to make the graph unconnected.

Definition B.5 ((Connected) Component). A connected subgraph H of an arbitrary graph G is called *(connected) component* if it is maximal in the sense that there exists no supergraph H' of H that is a connected subgraph of G as well.

A closed trail that consists of at least three nodes with all nodes except the first and the last being distinct is called a *cycle*. A cycle is called *even* if its number of edges is even and *odd* otherwise. A cycle with three edges is also called a *triangle*. We denote a cycle with n edges by C_n.

Definition B.6 (Acyclic graph). A graph that does not contain any cycles is called acyclic.

A node v (an edge e) is called *cut point* (*bridge*) if $G - v$ ($G - e$) consists of more components than G. A graph without cut points is called *inseparable*. A maximal inseparable subgraph is called a *block*.

Theorem B.1. *The following properties hold:*

1. *In every graph the number of nodes with an odd degree is even.*

2. *A graph with n nodes and k components cannot have more than $\frac{1}{2}(n-k)(n-k+1)$ edges.*

3. *Every graph that has more edges than nodes contains a cycle.*

Trees We can give alternative definitions for trees, they are at the same time descriptions of properties of trees.

Definition B.7 (Tree). A tree is a connected and acyclic graph.

1. A tree is a connected graph with n nodes and $n-1$ edges.

2. A tree is an acyclic graph with n nodes and $n-1$ edges.

3. A tree is a graph, in which there exists exactly one path between every pair of nodes.

Definition B.8 (Forest). An acyclic graph is called a *forest*.

The components of a forest are trees.

A tree together with a specifically marked node is called *rooted tree* and the node is called the *root* of the tree. Trees without a root are sometimes called *free*.

Definition B.9 (Binary tree). A *binary tree* is a rooted tree consisting of at least three nodes, in which the root has degree two and all the other nodes have degree one or three.

Digraphs We already defined elementary concepts for digraphs. Now we want to present notions and properties that are different from their counterparts for undirected graphs.

Let $G = (V, A)$ be a digraph. An alternating sequence $(v_0, a_1, v_1, \ldots, a_n, v_n)$ is called a *walk* if every arc a_i is $v_{i-1}v_i$ for $1 \leq i \leq n$. The walk is *closed* if $v_0 = v_n$, and *spanning* if $\{v_0, v_1, \ldots, v_n\} = V$. A walk is termed a *trail* if all of its arcs are distinct and a *path* if additionally all of its nodes are distinct. A closed trail that consists of at least two nodes with all nodes except the first and the last being distinct is called a *cycle*. A digraph that contains a cycle is called *cyclic*, otherwise it is called *acyclic*. We call v_2 *reachable* from v_1 if there exists a path from v_1 to v_2.

A *semiwalk* is an alternating sequence $(v_0, a_1, v_1, \ldots, a_n, v_n)$ where every arc a_i is either $v_{i-1}v_i$ or its converse v_iv_{i-1}. A semiwalk is termed a *semitrail* if all of its arcs are distinct, a *semipath* if all of its vertices are distinct, and a *semicycle* if it contains at least

three vertices and all of its vertices are distinct except for the fact that $v_0 = v_n$. These notions form the base for the various notions of connectivity that exist for digraphs.

Definition B.10 (Strongly / unilaterally / weakly connected digraph). A digraph is called *strongly connected* if every two of its distinct vertices v_1 and v_2 are so that v_2 is reachable from v_1 and v_1 is reachable from v_2. The digraph is called *unilaterally connected* if v_2 is reachable from v_1 or v_1 is reachable from v_2. A digraph is called *weakly connected* if v_1 and v_2 are linked by a semipath.

A digraph is called *disconnected* if it is weakly connected.

Theorem B.2. *The following properties hold:*

1. *A digraph is strongly connected if and only if it has a spanning closed walk.*
2. *A digraph is unilaterally connected if and only if it has a spanning walk.*
3. *A digraph is weakly connected if and only if it has a spanning semiwalk.*

Just as there are three kinds of connectivity for digraphs there are three kinds of components as well. Note that a *subdigraph* of a digraph $G = (V, A)$ is a digraph $G' = (V', A')$, such that $V' \subseteq V$ and $A' \subseteq A$.

Definition B.11 (Strong / unilateral / weak component). A *strong (unilateral, weak) component* in a digraph G is a maximal strongly (unilaterally, weakly) connected subdigraph of G.

There are interesting notions of traversability for digraphs. A digraph G is called *Eulerian* if it contains a closed trail that traverses every arc of G exactly once. A digraph G is termed *Hamiltonian* if it has a cycle containing all of the vertices of G. A digraph is Eulerian if and only if it is connected and each of its vertices has an out-degree equal to its in-degree.

Definition B.12 (Directed tree). A *directed tree* is a weakly connected digraph that does not contain a semicycle.

Of particular interest for computer science are those directed trees that have a *root*.

Definition B.13 (Arborescence). A directed tree is said to be an *arborescence* if it contains exactly one vertex, called the *root*, with no arcs directed toward it, and if all the arcs on any semipath are directed away from the root.

Theorem B.3. *The following properties hold for any arborescence G:*

1. *Every vertex in G, other than the root, has exactly one arc directed toward it.*

2. *There is a path from the root of G to any other vertex in G.*

3. *Every vertex in G, other than the root, is reachable from the root. The root is not reachable from any other vertex.*

B.2 Fundamentals of Partially Ordered Sets

Definition B.14 (Partially Ordered Set). A structure $[\mathfrak{M}, \leq]$ is called a *(reflexive) partially ordered set* or simply a *poset* if $\mathfrak{M}$ is an arbitrary set (the *ground set*) and $\leq$ is a reflexive, antisymmetric and transitive binary relation over $\mathfrak{M}$.

The elements of the ground set are also called *points*. A poset is called *finite* if its ground set is finite. Instead of writing $(x, y) \in \leq$ we usually stick to the infix-notation $x \leq y$. In addition, we write $x < y$ for $x \leq y$ and $x \neq y$.

As an example let $M = \{\emptyset, \{a\}, \{a, b\}, \{b, c\}, \{a, b, c\}\}$ and $R = \{(A, B) \in M \times M | A \subseteq B\}$. Then $[M, R]$ is a poset. In general, every family of sets together with the subset-relation is a poset.

Let $x, y \in \mathfrak{M}$ and $x \neq y$. We call x and y *comparable* and write $x \perp y$ if either $x < y$ or $y < x$. Otherwise we call them *incomparable* and write $x \parallel y$. We say that y *covers* x (or x *is covered by* y) and write $x <: y$ if $x < y$ and no z with $x < z$ and $z < y$ exists.

Every subset M of a ground set $\mathfrak{M}$ forms together with the restricted binary relation $\leq |_M$ also a poset. We call it a *subposet*.

The *dual poset* of a given poset $[\mathfrak{M}, \leq]$ is the structure $[\mathfrak{M}, \geq]$, in which for all $x, y \in \mathfrak{M}$ the relation $x \geq y$ holds if and only if in $[\mathfrak{M}, \leq]$ the relation $y \leq x$ holds. A poset $[\mathfrak{M}, \leq]$ is called *self-dual* if $[\mathfrak{M}, \leq] = [\mathfrak{M}, \geq]$.

A poset $[\mathfrak{M}, \leq]$ is called *connected* if for all $x, y \in \mathfrak{M}$ with $x \neq y$ a sequence of points $x = x_0, x_1, \ldots, x_n = y$ with $x_i \perp x_{i+1}$ for all $i = 0, 1, \ldots, n-1$ exists. A subposet $[M, \leq |_M]$ of $[\mathfrak{M}, \leq]$ is called a *component* of $[\mathfrak{M}, \leq]$ if it is connected and no other connected subposet $[N, \leq |_N]$ with $N \supset M$ exists.

Posets are usually visualized using a so called *Hasse diagram*. It consists of a graph where the points in the ground set are the vertices and an edge is included for every pair (x, y) with $x <: y$. Usually the "smaller" points are located at the bottom, i.e.,they have a smaller vertical coordinate. Figure B.1 shows the Hasse diagram for the example given above.

Chains and Antichains; Maximum and Minimum

Definition B.15 (Chain, Antichain). A poset $[\mathfrak{M}, \leq]$ is called a *chain* if every pair of points from $\mathfrak{M}$ is comparable. A poset $[\mathfrak{M}, \leq]$ is called an *antichain* if every pair of points from $\mathfrak{M}$ is incomparable.

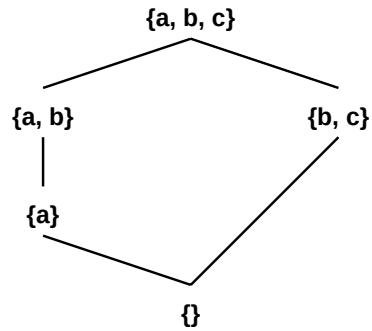

Figure B.1: A Hasse diagram

We can extend these notions to subsets of the ground set.

Definition B.16 (Chain, Antichain 2). A subset M of the ground set of a poset $[\mathfrak{M}, \leq]$ is called a *chain* (*antichain*) if the subposet $[M, \leq|_M]$ is a chain (antichain).

A chain (antichain) in a poset is called a *maximum chain* (*maximum antichain*) if no other chain (antichain) with a higher cardinality exists. The *height* of a poset is the cardinality of the maximum chain, the *width* of a poset is the cardinality of the maximum antichain.

Definition B.17 (Maximal, minimal point). A point $x \in \mathfrak{M}$ is called *maximal point* (*minimal point*) if no $y \in \mathfrak{M}$ with $y > x$ ($y < x$) exists.

We denote the set of the set of maximal points of a poset $[\mathfrak{M}, \leq]$ by $max([\mathfrak{M}, \leq])$ and the set of minimal points by $min([\mathfrak{M}, \leq])$.

Definition B.18 (Maximum, minimum point). A point $x \in \mathfrak{M}$ is called *maximum point* or *greatest point* (*minimum point* or *least point*) of the poset $[\mathfrak{M}, \leq]$ if for all $y \in \mathfrak{M}$ the relation $y \leq x$ ($y \geq x$) holds.

We can apply the notions of maximal and minimal points to chains and antichains. The set of all chains (antichains) of a poset forms together with the subset-relation a new poset. The maximal points of this new poset are called the maximal chains (antichains) of the original poset.

Definition B.19 (Maximal chain, maximal antichain). A chain (antichain) $C \in \mathfrak{M}$ is called *maximal chain* (*maximal antichain*) of the poset $[\mathfrak{M}, \leq]$ if no other chain (antichain) $D \in \mathfrak{M}$ with $D \supset C$ exists.

In a poset $[\mathfrak{M}, \leq]$ both $max([\mathfrak{M}, \leq])$ and $min([\mathfrak{M}, \leq])$ are maximal antichains.

Mappings between Posets Let $[\mathfrak{M}, \leq]$ and $[\mathfrak{N}, \preceq]$ be posets and $f : \mathfrak{M} \longrightarrow \mathfrak{N}$ a mapping between their ground sets. f is called:

- *order preserving*, if for all $x_1, x_2 \in \mathfrak{M}$ with $x_1 \leq x_2$ the relation $f(x_1) \preceq f(x_2)$ holds,
- *order reversing*, if for all $x_1, x_2 \in \mathfrak{M}$ with $x_1 \leq x_2$ the relation $f(x_1) \succeq f(x_2)$ holds.

Definition B.20 (Isomorphism). A mapping $f : \mathfrak{M} \longrightarrow \mathfrak{N}$ between posets $[\mathfrak{M}, \leq]$ and $[\mathfrak{N}, \preceq]$ is called *isomorphism*, if f is a bijection and for all $x_1, x_2 \in \mathfrak{M}$ the relation $x_1 \leq x_2$ holds if and only $f(x_1) \preceq f(x_2)$ holds.

If there exists an isomorphism between the posets $[\mathfrak{M}, \leq]$ and $[\mathfrak{N}, \preceq]$ we call the posets *isomorphic* and write $[\mathfrak{M}, \leq] \cong [\mathfrak{N}, \preceq]$. An isomorphism between $[\mathfrak{M}, \leq]$ and $[\mathfrak{M}, \leq]$ is called *automorphism*. An isomorphism between $[\mathfrak{M}, \leq]$ and a subposet of $[\mathfrak{N}, \preceq]$ is called an *embedding* of $[\mathfrak{M}, \leq]$ into $[\mathfrak{N}, \preceq]$. We usually do not distinguish between isomorphic posets, we rather say that $[\mathfrak{M}, \leq]$ *is contained in* $[\mathfrak{N}, \preceq]$ if an embedding exists. We also write $[\mathfrak{M}, \leq] = [\mathfrak{N}, \preceq]$ instead of $[\mathfrak{M}, \leq] \cong [\mathfrak{N}, \preceq]$

Lower and upper bound; Lattice Let $[\mathfrak{M}, \leq]$ be a poset and $M \subseteq \mathfrak{M}$. A point $x \in \mathfrak{M}$ is called *upper bound* (*lower bound*) for M if $y \leq x$ ($y \geq x$) holds for all $y \in M$. An upper bound (lower bound) x for M is called *least upper bound* (*greatest lower bound*) for M (we write $lub(M)$ and $glb(M)$, respectively) if $x \leq x'$ ($x \geq x'$) holds for every upper (lower) bound x' for M.

Definition B.21 (Lattice). A poset $[\mathfrak{M}, \leq]$ is called a *lattice* if every nonempty subset $M \subseteq \mathfrak{M}$ has a least upper and a greatest lower bound.

Finite lattices always have a greatest and a least point. If the lattice contains more then one point the greatest point is traditionally called *one* and the least point *zero*. For lattices $[\mathfrak{M}, \leq]$ we have naturally defined functions $\vee(join)$ and $\wedge(meet)$ from $\mathfrak{M} \times \mathfrak{M}$ into $\mathfrak{M}$:

$$x \vee y = lub(\{x, y\})$$
$$x \wedge y = glb(\{x, y\})$$

Appendix C

Documentation of the ECLiPSe-based Answering System

In this appendix we describe the implemented software system. We first describe the modules that are primarily involved in the schema matching processes. Next, we describe the graphical user interface to the system. In the final section we explain how to transform XML documents into databases for our system.

C.1 The ECLiPSe modules

The ECLiPSe-based schema matcher consists of a set of hierarchically ordered modules. For a description of the system see Section 8.3. In this appendix we describe the predicates that the individual modules provide. The package consists of the following files:

COPYRIGHT	The copyright note.
README	A short description.
go.pl	A short demo.
misc.pl	Module MISC: Miscellaneous supporting predicates
objmain.pl	Module OBJMAIN: The object maintainer
preds.pl	Module PREDS: The predicate component
samples/	Directory of sample databases and schemata
scontain.pl	Module SCONTAIN: Schema containment component
smain.pl	Module SMAIN: The schema maintainer
smatch.pl	Module SMATCH: The schema matcher (main module)
subgraph.pl	Module SUBGRAPH: The subgraph tester

We describe the set of modules in a top-down fashion, i.e.,we start with the schema matcher and end with the supporting predicates.

Module SMATCH The schema matcher (main module)

Uses modules: SCONTAIN, SUBGRAPH, PREDS, SMAIN, OBJMAIN, MISC

Exports predicates:

- `schema_matches(+Schema,+DB,-List)` gives a list List of newly created objects that are the minimal matches of schema Schema in database DB.
- `is_schema_match(?Schema,?DB,?Match)` succeeds if Match is a minimal match of schema Schema in database DB. The matches are stored as lists of associations Elem(Value).
- `set_opt_oldMatches` activates the optimization of reusing stored schema matches if present.
- `is_opt_oldMatches` tests for the optimization of reusing stored schema matches if present.
- `unset_opt_oldMatches` deactivates the optimization of reusing stored schema matches if present.
- `set_opt_containTest` activates the optimization of using schema containment for reducing the search space.
- `is_opt_containTest` tests for the optimization of using schema containment for reducing the search space.
- `unset_opt_containTest` deactivates the optimization of using schema containment for reducing the search space.
- `set_opt_firstFew` activates the optimization of using schema containment for computing the first few schema matches immediately.
- `is_opt_firstFew` tests for the optimization of using schema containment for computing the first few schema matches immediately.
- `unset_opt_firstFew` deactivates the optimization of using schema containment for computing the first few schema matches immediately.

Module SCONTAIN Schema containment component

Uses modules: SUBGRAPH, PREDS, SMAIN, MISC

Exports predicates:

- `schema_containments(+S1,+S2,-List)` returns a list List of containment mappings from S2 into S1, if S2 contains S1, and the empty list otherwise.

Module SUBGRAPH The subgraph tester

Uses modules: OBJMAIN, MISC

Exports predicates:

- `iso_subgraphs(+Obj1,+Obj2,-List)` returns a list List of subobjects of Obj2 that are isomorphic to Obj1. The objects in this list are created during the process.
- `construct_csp(+Obj1,+Obj2,-VList,-EList)` constructs a CSP for the SUBGRAPH-ISOMORPHISM problem. Variables for the elements of Obj1 are introduced and collected in VList, their corresponding elements are collected in EList.
- `create_domains(+Obj)` creates the domains of the vertices and arcs for object Obj to be used in a CSP.
- `create_var_list(-VList,+EList)` creates a list of domain variables for the elements of EList and unifies the result with VList.
- `precompute_constraints(+Obj)` precomputes the structure constraints for object Obj to be used in a CSP.
- `find_domain_variables(-VList1,+EList1,+VList2,+EList2)` finds those variables from VList2 that correspond to the elements in EList1, a sublist of EList2, and unifies the result with VList1.
- `find_domain_variable(-Var,+Elem,+VList,+EList)` finds the single variable from VList that corresponds to Elem, an element of EList, and unifies the result with Var.
- `association_terms(?VList,?EList,?AList)` produces a list of associations Elem(Var) of elements from EList and their corresponding domain variables and unifies the result with AList.
- `solution(+List)` produces a CSP solution for the list of domain variables List. Backtracking can be used to produce all solutions.
- `labeling(+List)` is a synonym for `solution(+List)`.
- `build_solutions(+Obj2,+SList,-DList)` builds Vertex/Arc descriptions of the CSP solutions, subobjects of Obj2, given in SList and unifies the result with DList.
- `create_solutions(+Obj1,+Obj2,+DList,-OList)` creates the solution objects from the descriptions given in DList and stores their identifiers in OList.
- `encode_solution(+Obj1,+Obj2,+Num,?ID)` creates an identifier ID for the Num'th solution of the SUBGRAPH-ISOMORPHISM problem for Obj1 and Obj2.

- `decode_solution(+ID,?Obj1,?Obj2,?Num)` reconstructs the object identifiers Obj1 and Obj2 as well as the solution number Num from the solution object identifier ID.
- `is_domain(?Obj,?ID,?Dom)` succeeds if there exists a domain of type ID defined for object Obj and with domain Dom.
- `is_precomputed_constraint(?Obj,?ID,?C1,?C2)` succeeds if there exists a precomputed binary constraint of type ID formed by C1 and C2 defined for object Obj.
- `set_opt_oldSolutions` activates the optimization of using stored CSP solutions if present.
- `is_opt_oldSolutions` tests for the optimization of using stored CSP solutions if present.
- `unset_opt_oldSolutions` deactivates the optimization of using stored CSP solutions if present.

Module PREDS The predicate component

Uses modules: MISC

Exports predicates:

- `predicate(+Pred,+Lab)` succeeds if Pred(Lab) holds.
- `predicate_contain(+Pred1,+Pred2)` succeeds if the predicate Pred1 is contained in the predicate Pred2.

Module SMAIN The schema maintainer

Uses modules: OBJMAIN, MISC

Exports predicates:

- `begin_schema(+Schema)` must be called before defining the schema Schema.
- `end_schema(+Schema)` finishes up the definition of the schema Schema.
- `destroy_schema(+Schema)` destroys the schema Schema.
- `is_schema(?Schema)` succeeds if there exists a schema Schema.
- `detect_types(+Schema)` detects for all vertices and arcs in the schema whether a label is a constant, a predicate, a variable definition or a multiple element (path).
- `is_type(?Schema,?Elem,?Type)` succeeds if Type is the type of the vertex or arc Elem in schema Schema.
- `is_binding(?Schema,?Elem,?Var)` succeeds if Var is a variable associated to the vertex or arc Elem in schema Schema.

- `is_size_min(?Schema,?Elem,?Min)` succeeds if Min is the minimum length associated to the (vertex or) arc Elem in schema Schema.
- `is_size_max(?Schema,?Elem,?Max)` succeeds if Max is the maximum length associated to the (vertex or) arc Elem in schema Schema.
- `extract_predicate(+Type,+Lab,-Pred)` extracts the predicate from label Lab of type Type and unifies it with Pred.
- `extract_predicate(+Schema,+Elem,+Lab,-Pred)` extracts the predicate from label Lab of vertex or arc Elem in schema Schema and unifies it with Pred.
- `is_constant(+Pred)` succeeds if the predicate Pred is a constant predicate.

Module OBJMAIN The object maintainer

Uses modules: MISC

Exports predicates:

- `objmain_init` initializes the object maintainer. All existing objects, vertices, arcs and labels are removed.
- `begin_object(+Obj)` must be called before actually defining an object Obj. Then the system automatically keeps track of "open" arcs where source or target vertices are still missing.
- `create_vertex(+Obj,+Vert,+Lab)` defines a vertex Vert in object Obj labeled Lab.
- `create_arc(+Obj,+Arc,+Src,+Tar,+Lab)` defines an arc Arc in object Obj going from Src to Tar and labeled Lab. Src and Tar either already exist or are remembered as "must be defined later".
- `end_object(+Obj)` finishes the definition of the object Obj. This will only succeed if all arcs have their source and target vertices defined.
- `is_object(?Obj)` succeeds if there exists an object Obj.
- `is_vertex(?Obj,?Vert)` succeeds if there exists a vertex Vert in object Obj.
- `is_arc(?Obj,?Arc,?Src,?Tar)` succeeds if there exists an arc Arc from Src to Tar in object Obj.
- `is_label(?Obj,?Elem,?Lab)` succeeds if there exists a vertex or an arc Elem in object Obj labeled Lab.
- `print_object(+Obj)` writes information on the object Obj to stdout.
- `destroy_object(+Obj)` destroys the object Obj.

- `simple_path(?Obj,?Path)` succeeds if Path is a (nonempty) trail in object Obj. Can also be used to generate all trails of an object.
- `source(?Obj,?Path,?Vert)` succeeds if Vert is the source of the arc or trail Path in object Obj.
- `target(?Obj,?Path,?Vert)` succeeds if Vert is the target of the arc or trail Path in object Obj.
- `sourcecheck(+Obj,+Path,?Vert)` succeeds if Vert is the source of the arc or trail Path in object Obj.
- `targetcheck(+Obj,+Path,?Vert)` succeeds if Vert is the target of the arc or trail Path in object Obj.
- `induced_subobject(+Obj,+EList,?VList,?AList)` computes the by the list of vertices and arcs EList induced subobject of Obj and unifies the resulting vertex list with VList and the resulting arc list with AList. The induced subobject is computed by adding the source and target vertices of the arcs in EList.

Module MISC Miscellaneous supporting predicates

Uses modules: –

Exports predicates:

- `atom_list(?Atom,?List)` converts between the atom representation and a list of the ASCII codes of the characters. At least one of the arguments must be instantiated.
- `between(+Lower,+Upper,+Num)` succeeds if Num is a number between Lower and Upper.
- `deleteall(+List1,+Elem,?List2)` deletes all occurrences of Elem from List1 and unifies the result with List2.
- `indexlist(+NestedList,+Index,-List)` collects the Index'th elements from every sublist of NestedList and unifies the resulting list with List.
- `is_list(+Arg)` succeeds if Arg is a list.
- `is_set(+Arg)` succeeds if Arg is a list and does not contain multiple occurrences of the same element.
- `last(?Elem,?List)` succeeds if Elem unifies with the last element of List. At least one of the arguments should be instantiated.
- `listcaseequal(+List1,+List2)` succeeds is List1 and List2 are lists of ASCII codes representing upper and lower case letters and the words they are representing are the same ignoring the case of the letters.

- `listprefix(?List1,+List2)` succeeds if List1 unifies with an arbitrary prefix of List2. Can also be used to generate all prefixes of List2.
- `nth1(+Index,+List,?Elem)` succeeds if Elem unfies with the Index'th element of List. The first element of List has index 1.
- `pairlist(?PairList,?List1,?List2)` succeeds if PairList unifies with the list of pairs of corresponding elements (i.e.,elements at the same position) from List1 and List2. This predicate can also be used to split a PairList.
- `setequal(+Set1,+Set2)` succeeds if Set1 and Set2 are both sets and are the same.
- `split(+List,+Elem,-List1,-List2)` splits List into List1 and List2, guided by the first occurrence of Elem. Elem will not be a member of either List1 or List2. If Elem is not a member of List the predicate fails.
- `sublist(+List1,+List2)` succeeds if List1 is a continuous sublist of List2 starting at an arbitrary element of List2. Uses listprefix/2.
- `term_atom(?Term,?Atom)` converts between the term and the atom representation. At least one of the arguments should be instantiated.

C.2 The user interface

Two files have to be added to the ECLiPSe-based schema matcher to provide a graphical user interface.

kshow.tcl	The actual interface implemented in Tcl/Tk
tkiface.pl	A set of supporting Prolog predicates

The interface can be invoked by calling the predicate `tk([file('kshow.tcl')])`. This opens a window similar to the one shown in Figure 8.2 on Page 118. The top half of the screen is used to show details on the open databases and schemata, whereas the bottom half presents matches of schemata in databases. In addition there is a menu situated at the very top of the window. The list box at the top left allows the user to switch between open databases. Schemata are selected by clicking once on their name. For both, databases and schemata, the details are updated automatically. Double-clicking on a schema name produces the list of matches of that schema in the current database. The names of the matches are presented at the bottom left. Again, clicking once on a name produces the details of the match. In the following we describe the individual menu entries:

Menu File Interaction with files, open and close databases and schemata

- `Open Database` opens a dialog window that allows the user to open a database from a `.pl`-file. Note that the module `OBJMAIN` must be loaded.
- `Open Schema` opens a dialog window that allows the user to open a schema from a `.pl`-file. Note that the module `SMAIN` must be loaded.
- `Open XML-File` opens a dialog window that allows the user to open a database from an `.xml`-file. A separate window is opened for setting a database name. Note that the module `OBJMAIN` must be loaded.
- `Close Database` closes the current database.
- `Close Schema` closes the selected schema.
- `Quit` quits the graphical user interface.

Menu Database Everything that has to do with the databases

- `Display` displays details of the current database. A graphical representation is planned, but not yet implemented.

Menu Schema Everything that has to do with the schemata

- `Display` displays details of the selected schema. A graphical representation is planned, but not yet implemented.
- `Match` invokes the computation of the matches of the selected schema in the current database. Note that the module `SMATCH` must be loaded.
- `Statistics` displays some statistics of the selected schema and its matches in the current database. This is not implemented yet.

Menu Optimization Set optimization switches

- `Use old solutions` Do not solve a CSP again when solution objects of the schema match problem already exist.
- `Use old matches` Do not solve a CSP again when solutions of the CSP already exist.
- `Reduce search space` Use schema containment to reduce the search space of CSPs.
- `Present first few matches` Use schema containment to present the first matches of the schema immediately.

Menu Help Overview of the tool

- `Introduction` provides an online help for the tool.
- `About` displays the name and the current version number of the tool.

The Tcl/Tk-script that implements the graphical user interface is based on an ECLiPSe module. This module exports the functionality of the schema matcher.

Module TKIFACE Provides functionality to the graphical user interface

Uses modules: SMATCH, SMAIN, OBJMAIN

Exports predicates:

- `all_objects(?List)` unifies the list of ids of all currently existing objects with List.
- `all_schemata(?List)` unifies the list of ids of all currently existing schemata with List.
- `all_databases(?List)` unifies the list of ids of all currently existing databases with List. Databases are objects that are neither schemata nor matches.
- `all_vertices(+Obj,?List)` unifies the list of ids of all vertices of object Obj (the id given as a string) with List.
- `all_arcs(+Obj,?List)` unifies the list of ids of all arcs of object Obj (the id given as a string) with List.
- `database_destroy(+Obj)` destroys the database with id Obj (given as a string).
- `schema_destroy(+Obj)` destroys the schema with id Obj (given as a string).
- `find_schema_matches(+Schema,+DB,-List)` gives a list List of newly created objects that are the minimal matches of schema Schema in database DB. Both Schema and DB have to be given as strings.

C.3 The XML support

The package XML2PL is a parser that transform an XML document into a database that can be used in our system. The package consists of the following files:

COPYRIGHT	The copyright note.
Makefile	The Makefile.
README	A short description.
simple.xml	A simple sample.
xml2pl.c	The main program.
xmllex.l	A lexer for XML documents.
xmlparse.y	The parser that generates Prolog output.

The program can be compiled by typing `make`. Possibly, some library flags must be adapted in the Makefile. The Makefile contains default values that should work for

Linux and Solaris operating systems. An executable called `xml2pl` should be created. In the simplemost case the program can be called like this:

```
./xml2pl <XML-file>
```

This will write the generated Prolog code to stdout. The way the document is transformed into a database is described in Section 8.2. The database generated is named 'XML_document' by default. The name can be changed by giving a second argument.

```
./xml2pl <XML-file> <name>
```

Of course, the output can be redirected into a file like this:

```
./xml2pl <XML-file> <name> > <output-file>
```

This output file can be read in from the ECLiPSe system by using the `compile` predicate. The only prerequisite is that the module `objmain`, the object maintainer, must already be loaded.

Appendix D

List of Mathematical Symbols

Symbol	Description	Section
G, G_1, G_2, G', H	(Labeled) total directed graph	2.1
$\mathcal{L}$	Set of labels	2.1
o, o_1, o_2, o'	Object	2.1
$V, A, V^{(o)}, A^{(o)}$	Set of vertices, arcs (of object o)	2.1
$s, t, l, s^{(o)}, t^{(o)}, l^{(o)}$	Source, target, label function (of object o)	2.1
p, p_1, p_2	Walk, trail, path	2.1
$P, P^+, P^{(o)}$	Set of (nonempty) trails (of object o)	2.1
$p_1 \circ p_2$	Concatenation of walks, trails, paths	2.1
$o_1 \subseteq o_2, o_1 \supseteq o_2$	Subobject relationship	2.1
$\mathfrak{P}(o)$	Set of all subobjects of o	2.1
m, m_1, m_2, m'	Graph morphism	2.2
Σ	Signature	2.3
s, s_1, s_2	Sort	2.3
S	Set of sorts	2.3
$\omega, \omega_1, \omega_2$	Operation symbol	2.3
Ω	Set of operation symbols	2.3
A, A_1, A_2, A', B	Algebra	2.3
f	Σ-Homomorphism (Σ-Isomorphism)	2.3
S, T	Source, target incidence	2.3
$R \circ S, RS$	Product of relations	2.3
I	Identity relation	2.3
R^T	Transpose of a relation	2.3
$(M_V, M_A), M$	Graph homomorphism (isomorphism)	2.3
$\mathcal{P}$	Set of predicates	3.1
s, s_1, s_2, s'	Predicate schema	3.1

Symbol	Description	Section
m, m_1, m_2, m'	(Naive) match	3.1
$\mathfrak{M}^{(s)}(o), \mathfrak{M}^{(s)}_{min}(o)$	Set of (minimal) matches of s in o	3.1
$\mathcal{V}$	Set of variables	3.2
s_P, t_P	Source, target for nonempty trails	3.3
G_P, o_P	Corresponding trail graph	3.3
s, s_1, s_2, s'	Schema	3.3
$v, v^{(s)}$	Variable mapping (of schema s)	3.3
$q_{min}, q_{max}, q^{(s)}_{min}, q^{(s)}_{max}$	Length restrictions (of schema s)	3.3
m, m_1, m_2, m'	Match (function)	3.3
$\mathfrak{M}^{(s)}(o), \mathfrak{M}^{(s)}_{min}(o)$	Set of (minimal) matches of s in o	3.3
q, q_1, q_2, q'	Schema, focus, transformation query	4.1 - 4.3
t	term-labeled graph	4.3
a	aggregation graph	4.3
r	Graph rule	4.5
L	Left-hand side of a graph rule	4.5
R	Right-hand side of a graph rule	4.5
m	Redex of a left-hand side	4.5
(X, D, C)	Constraint Satisfaction Problem	5.1
x, x_1, x_2, y	Variable in a CSP	5.1
X	Set of variables in a CSP	5.1
D, D_1, D_2	Domain in a CSP	5.1
D	Set of domains in a CSP	5.1
$C, C_1, C_2, C_S, C_{(x,y)}$	Constraint in a CSP	5.1
S, S_1, S_2	Tuple of variables in a CSP	5.1
C	Set of constraints in a CSP	5.1
w	Width of a constraint graph	6.4
$s_1 \subseteq s_2, s_1 \supseteq s_2$	Schema containment relationship	7.1

Bibliography

[ABGVG87] S. Abiteboul, C. Beeri, M. Gyssens, and D. Van Gucht. An introduction to the completeness of languages for complex objects and nested relations. In *Proceedings of the International Workshop on Theory and Applications of Nested Relations and Complex Objects in Databases*, pages 117–138, Darmstadt, Germany, April 1987.

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 1–18, Delphi, Greece, January 1997.

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 73–84, Dublin, Ireland, August 1993.

[Agg] The AGG-Homepage, http://tfs.cs.tu-berlin.de/projekte/gragra/agg/.

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. M. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[AGM+97] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. In *Proceedings of the Workshop on Management of Semistructured Data (in conjunction with SIGMOD/PODS)*, Tucson, AZ, USA, May 1997.

[AKW88] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The awk Programming Language*. Addison-Wesley, Reading, MA, USA, 1988.

[Alt] AltaVista HOME, http://www.altavista.com/.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To weave the web. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 206–215, Athens, Greece, August 1997.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, 1997.

[ASU79] A. Aho, Y. Sagiv, and J. D. Ullman. Equivalence of relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[AV97] S. Abiteboul and V. Vianu. Queries and computation on the web. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 262–275, Delphi, Greece, January 1997.

[Bar98] R. Bartak. Guide to constraint programming, 1998. http://kti.ms.mff.cuni.cz/ bartak/constraints/.

[Bar99] R. Bartak. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS)*, Prague, Czech Republic, June 1999.

[BB89] D. Balfanz and A. Bergholz. Digitale kodierung von alphabeten. *Nachrichtentechnik / Elektronik*, 39(6):220–222, 1989.

[BC99] A. Bonifati and S. Ceri. Comparitive analysis of five XML query languages. Submitted for publication, 1999.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 336–350, Delphi, Greece, January 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.

[BDK92a] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Francisco, CA, USA, 1992.

[BDK92b] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 152–167, Vienna, Austria, March 1992.

[BDS95] P. Buneman, S. B. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of the International Workshop*

on Database Programming Languages (DBPL), Gubbio, Italy, September 1995.

[Ber93] A. Bergholz. A general diagnostic engine based on an assumption-based truth maintenence system, February 1993. Midterm thesis at University of Edinburgh.

[Ber95] A. Bergholz. Begleitung: Rezension des Buches "Der LaTeX-Begleiter". *alpha*, 29(10):27, 1995.

[Ber96] A. Bergholz. Suche nach genetisch korrelierten Proteinabschnitten unter Verwendung einer relationalen Datenbank. Master's thesis, Humboldt-University Berlin, August 1996. German Diploma thesis.

[Ber99] A. Bergholz. Rich schemata for semistructured data: Thesis proposal. In *Conference on Advanced Information Systems Engineering (CAiSE*99), 6th Doctoral Consortium*, Heidelberg, Germany, June 1999.

[Ber00] A. Bergholz. *Querying Semistructured Data Based On Schema Matching.* PhD thesis, Humboldt-University Berlin, 2000.

[BF98] A. Bergholz and J. C. Freytag. A three-layer approach to semistructured data. In *Proceedings of the International Workshop on Theory and Application of Graph Transformations (TAGT)*, Paderborn, Germany, November 1998.

[BF99a] A. Bergholz and J. C. Freytag. Matching schemata by utilizing constraint satisfaction techniques. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats (in conjunction with ICDT'99)*, Jerusalem, Israel, January 1999.

[BF99b] A. Bergholz and J. C. Freytag. Querying semistructured data based on schema matching. In *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, Kinloch Rannoch, Scotland, UK, September 1999.

[BF00a] A. Bergholz and J. C. Freytag. Integration biologischer Sequenzdaten zum Aufbau eines Genomwarehouses. In *Proceedings of the Workshop "Informationssysteme in der Biotechnologie"*, Magdeburg, Germany, February 2000.

[BF00b] A. Bergholz and J. C. Freytag. Managing schemata for semistructured databases using constraints. In *Proceedings of the East-European Con-*

ference on Advances in Databases and Information Systems (ADBIS), Prague, Czech Republic, September 2000.

[BHSF97] A. Bergholz, S. Heymann, J. A. Schenk, and J. C. Freytag. Sequence comparison using a relational database approach. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 126–131, Montreal, Canada, August 1997.

[BR75] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[BSMM93] I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Harri Deutsch, Thun, Frankfurt, Germany, 1993.

[BTBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, pages 9–19, Nafplion, Greece, August 1991.

[BTBW92] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 140–154, Berlin, Germany, October 1992.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 117–121, Tucson, AZ, USA, May 1997.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scoll. From structured documents to novel query facilities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 313–323, Minneapolis, MN, USA, May 1994.

[CAW98] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 4–13, Orlando, FL, USA, February 1998.

[CDAR97] S. Chakrabarti, B. Dom, R. Agrawal, and P. Raghavan. Using taxonomy, discriminants, and signatures for navigating in text databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 446–455, Athens, Greece, August 1997.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogenous information sources. In *Proceedings of the Information Processing Society of Japan (IPSJ) Conference*, pages 7–18, Tokyo, Japan, October 1994.

[Che76] P. P.-S. Chen. The Entity-Relatioship-Model – toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[CHMW96] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO: An integrated query/browser for object databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 203–214, Bombay, India, September 1996.

[CHN+95] W. F. Cody, L. M. Haas, W. Niblack, M. Arya, M. J. Carey, R. Fagin, D. Lee, D. Petkovic, P. M. Schwarz, J. Thomas, M. Tork Roth, J. H. Williams, and E. L. Wimmers. Querying multimedia data from multiple repositories by content: The Garlic project. In *Proceedings of the International Conference on Visual Database Systems (VDB)*, pages 17–35, Lausanne, Switzerland, March 1995.

[CHY96] M. S. Chen, J. Han, and P. S. Yu. Data mining: An overview from a database perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, 1996.

[CK97] M. J. Carey and D. Kossmann. On saying "Enough Already!" in SQL. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 219–230, Tucson, AZ, USA, May 1997.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 190–200, Taipei, Taiwan, March 1995.

[CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 77–90, Boulder, CO, USA, May 1977.

[CN91] B. J. Cox and A. J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, MA, USA, 2nd edition, 1991.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, number 6 in Courant Computer Science Symposia Series, pages 65–98. Prentice-Hall, San Jose, CA, USA, 1972.

[Cod80] E. F. Codd. Data models in database management. In *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling*, pages 112–114, Pingree Park, CO, USA, June 1980.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, OH, USA, May 1971.

[Coo89] M. C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[Dat95] C. J. Date. *An Introduction To Database Systems*. The System Programming Series. Addison-Wesley, Reading, MA, USA, 6th edition, 1995.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the International World Wide Web Conference*, pages 1155–1169, Toronto, Canada, May 1999.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 431–442, Philadelphia, PA, USA, June 1999.

[DJ88] R. Dechter and Pearl J. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[DKA+86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch. A DBMS prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchiesa. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 356–367, Washington, DC, USA, May 1986.

[Ecl] ECLiPSe - The ECRC Constraint Logic Parallel System, http://www.ecrc.de/eclipse/.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *Proceedings of the IEEE Symposium on Switching and Automata Theory*, pages 167–180, Iowa City, IA, USA, October 1973.

[FFK+98] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 414–425, Seattle, WA, USA, June 1998.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[FLS98] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 139–148, Seattle, WA, USA, June 1998.

[Fou92] L. R. Foulds. *Graph Theory Applications*. Springer-Verlag, Berlin – Heidelberg – New York, 1992.

[Fre82] E. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[FS98] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 14–23, Orlando, FL, USA, February 1998.

[Gas77] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Cambridge, MA, USA, 1977.

[Gas78] J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisfying assignment problems. In *Proceedings of the Conference of the Canadian Society for Computational Studies of Intelligence*, 1978.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, USA, 1979.

[GMW99] R. Goldman, , J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, Philadelphia, PA, USA, June 1999.

[Gol91] C. F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, UK, 1991.

[Got82] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[GSVGM98] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 26–37, New York, USA, August 1998.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 436–445, Athens, Greece, August 1997.

[GW99] R. Goldman and J. Widom. Approximate DataGuides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats (in conjunction with ICDT'99)*, Jerusalem, Israel, January 1999.

[HE80] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[HHK95] M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 453–462, Milwaukee, WI, USA, October 1995.

[HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 562–573, Zürich, Switzerland, September 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[HSBM95] S. Heymann, J. A. Schenk, A. Bergholz, and B. Micheel. What else is in a genome - minimum information, a new concept. in preparation, 1995.

[JGJ+95] M. Jarke, R. Gallersdoerfer, M. A. Jeusfeld, M. Staudt, and S. Eherer. ConceptBase - a deductive object base for meta data management. *Journal of Intelligent Information Systems*, 4(2):167–192, 1995.

[Jun90] D. Jungnickel. *Graphen, Netzwerke und Algorithmen.* Wissenschaftsverlag, Mannheim – Wien – Zürich, 1990.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the World-Wide Web. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 54–65, Zürich, Switzerland, September 1995.

[Kum92] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[KV98] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 205–213, Seattle, WA, USA, June 1998.

[KvB97] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[KWP+91] E. A. Kabat, T. T. Wu, H. M. Perry, K. S. Gottesman, and C. Foeller. *Sequences of Proteins of Immunological Interest.* Number 91-3242. National Institutes of Health Publications, Bethesda, MD, USA, 1991.

[Lam94] L. Lamport. *LaTeX.* Addison-Wesley, Reading, MA, USA, 1994.

[LL95] S. M. Lang and P. C. Lockemann. *Datenbankeinsatz.* Springer-Verlag, Berlin – Heidelberg – New York, 1995.

[LMSS95] A. Y. Levy, A. M. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 95–104, San Jose, CA, USA, May 1995.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.

[LP85] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.

[LS93] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 171–181, Dublin, Ireland, August 1993.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A declarative language for querying and restructuring the web. In *Proceedings of the International Workshop on Research Issues in Data Engineering (RIDE)*, pages 12–21, New Orleans, LA, USA, February 1996.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[MEB88] J. Moore, A. Engelberg, and A. Bairoch. Using PC/GENE for protein and nucleic acid analysis. *Biotechniques*, 6(6):566–572, 1988.

[MH86] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MMM96] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World-Wide Web. In *Proceedings of the Conference on Parallel and Distributed Information Systems (PDIS)*, pages 80–91, Miami Beach, FL, USA, December 1996.

[MMM97] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World-Wide Web. *Journal of Digital Libraries*, 1(1):68–88, 1997.

[MS99] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 277–295, Jerusalem, Israel, January 1999.

[MW97] J. McHugh and J. Widom. Integrating dynamically-fetched external information into a DBMS for semistructured data. *SIGMOD Record*, 26(4):24–31, 1997.

[MW+98] J. McHugh, , J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Stanford University, Stanford, CA, USA, 1998.

[MW99] J. McHugh, , and J. Widom. Query optimization for XML. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 315–326, Edinburgh, Scotland, UK, September 1999.

[Nad89] B. A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[NUWC97] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 79–90, Birmingham, UK, April 1997.

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogenous information sources. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 251–260, Taipei, Taiwan, March 1995.

[PL88] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparision. *Proceedings of the National Academy of Sciences (PNAS)*, 85:2444–2448, 1988.

[PR69] J. L. Pfaltz and A. Rosenfeld. Web grammars. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 609–619, Washington, DC, USA, 1969.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[PV99] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 455–466, Philadelphia, PA, USA, June 1999.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 319–344, Singapore, December 1995.

[Rit94] O. Ritter. The integrated genomic database. *Computational Methods in Genome Research*, pages 57–73, 1994.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *Proceedings of the W3C Query Languages Workshop*, Boston, MA, USA, December 1998.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1. World Scientific, London, UK, 1997.

[RPD89] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. Technical Report ACT-AI-222-89, Microelectronics and Computer Technology Corporation, Austin, TX, USA, 1989.

[Rud98] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *Proceedings of the International Workshop on Theory and Application of Graph Transformations (TAGT)*, Paderborn, Germany, November 1998.

[Sch97] A. Schuerr. Programmed graph replacement systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1. World Scientific, London, UK, 1997.

[SF94] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the European Conference on Artificial Intelligence*, pages 125–129, Amsterdam, The Netherlands, August 1994.

[SGRS95] M. Senger, K. H. Glatting, O. Ritter, and S. Suhai. X-husar, an x-based graphical interface for the analysis of genomic sequences. *Computational Methods and Programs in Biomedicine*, 46(2):131–141, 1995.

[Shm93] O. Shmueli. Equivalence of datalog queries is undecidable. *Journal of Logic Programming*, 15(3):231–241, 1993.

[SLR96] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 99–110, Bombay, India, September 1996.

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[Sto96] M. Stonebraker. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, San Francisco, CA, USA, 1996.

[Swi] SWI Prolog, http://www.swi.psy.uva.nl/projects/SWI-Prolog/.

[SY81] Y. Sagiv and M. Yannakakis. Equivalence among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1981.

[TMD92] J. Thierry-Mieg and R. Durbin. Syntactic definitions for the ACEDB data base. Technical report, MRC Laboratory for Molecular Biology, Cambridge, UK, 1992.

[Tro92] W. T. Trotter. *Combinatorics and Partially Ordered Sets.* The John Hopkins University Press, Baltimore, MD, USA, 1992.

[TYF86] T. J. Teorey, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, 1986.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[Xml] Extensible Markup Language (XML), http://w3c.org/XML/.

[Yah] Yahoo!, http://www.yahoo.com/.

[Zlo77] M. Zloof. Query by example. *IBM Systems Journal*, 16(4):324–343, 1977.

[Zue93] A. Zuendorf. A heuristic for the subgraph isomorphism problem in executing PROGRES. Technical Report AIB 93-5, RWTH Aachen, Aachen, Germany, 1993.

Index

Erklärung

Ich erkläre hiermit, daß

1. ich die vorliegende Dissertationsschrift "Querying Semistructured Data Based On Schema Matching" selbständig und ohne unerlaubte Hilfe angefertigt habe;
2. ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze;
3. mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist.

Berlin, den 26. Mai 2000

André Bergholz

Curriculum Vitae

Personal Information

Name:	André Bergholz
Email:	bergholz@dbis.informatik.hu-berlin.de
Day of birth:	March 20th, 1971
Place of birth:	Berlin
Marital status:	single

Professional Experience

06/98-01/00	Project work and consulting in the human-genome area (Kelman GmbH)
12/94-11/96	Freelance work as an IT-consultant (ICG mbH)
10/93-07/94	Instructor for a student's project on Computer Algebra Systems (Humboldt-University Berlin)
09/91-07/92	Teacher for Computer Science (Pupil's Society for Mathematics, Berlin)

Education

01/98-03/98	Research stay at the database group of Stanford University (Lore project)
12/96-01/00	Graduate study of Computer Science (Graduate School in "Distributed Information Systems", Database group of Humboldt-University Berlin)
08/96	Diplom Computer Science (diploma degree; very good) from Humboldt-University Berlin; Diploma thesis: "Search for Genetically Correlated Protein Segments by Using a Relational Database"
07/93-09/93	Internship Bioinformatics (Max Delbrück Center for Molecular Medicine Berlin Buch)
10/92-06/93	Undergraduate study of Artificial Intelligence Computer Science (University of Edinburgh, Scotland, UK); Midterm thesis: "A General Diagnostic Engine based on an Assumption-Based Truth-Maintenance System"

03/92	Vordiplom Computer Science (intermediate examination; good) from Humboldt-University Berlin
11/89-04/90	Military service
09/89-08/96	Undergraduate study of Mathematics and Computer Science (Humboldt-University Berlin)
07/89	Abitur (high school graduation; very good) from Heinrich-Hertz-Oberschule
09/85-08/89	High school (Heinrich-Hertz-Oberschule, extended secondary school with mathematical, scientific, and technical profile), Berlin
09/77-08/85	Elementary school (Paul-Zobel-Oberschule), Berlin

Berlin, 26th May 2000

André Bergholz

Lebenslauf

Persönliche Daten

Name:	André Bergholz
Email:	bergholz@dbis.informatik.hu-berlin.de
Geburtsdatum:	20. März 1971
Geburtsort:	Berlin
Familienstand:	ledig

Berufserfahrung

06/98-01/00	Industrieprojekt und Firmenberatung im Humangenom-Bereich (Kelman GmbH)
12/94-11/96	freiberufliche Tätigkeit als IT-Berater (ICG mbH)
10/93-07/94	Leiter eines Projekttutoriums Computeralgebrasysteme (Humboldt-Universität Berlin)
09/91-07/92	Kursleiter Informatik (Mathematische Schülergesellschaft Berlin)

Ausbildung

01/98-03/98	Forschungsaufenthalt in der Datenbankgruppe der Stanford Universität (Lore Projekt)
12/96-01/00	Promotionsstudium Informatik (Graduiertenkolleg „Verteilte Informationssysteme“, Datenbankgruppe der Humboldt-Universität Berlin)
08/96	Diplom Informatik (sehr gut) an der Humboldt-Universität Berlin; Diplomarbeit: „Suche nach genetisch korrelierten Proteinabschnitten unter Verwendung einer relationalen Datenbank“
07/93-09/93	Praktikum in Bioinformatik (Max-Delbrück-Centrum für Molekulare Medizin Berlin-Buch)

10/92-06/93	Studium der Informatik und Künstlichen Intelligenz (University of Edinburgh, Scotland, UK); Projektarbeit: "A General Diagnostic Engine based on an Assumption-Based Truth-Maintenance System"
03/92	Vordiplom Informatik (gut) an der Humboldt-Universität Berlin
11/89-04/90	Grundwehrdienst
09/89-08/96	Studium der Mathematik und Informatik (Humboldt-Universität Berlin)
07/89	Abitur (sehr gut) an der Heinrich-Hertz-Oberschule
09/85-08/89	Gymnasium Heinrich-Hertz-Oberschule, Berlin (Spezialschule mathematisch-naturwissenschaftlich-technischer Richtung)
09/77-08/85	Grundschule Paul-Zobel-Oberschule, Berlin

Berlin, den 26. Mai 2000

André Bergholz