ALGORITHMS AND IMPLEMENTATION OF FUNCTIONAL DEPENDENCY DISCOVERY IN XML

A thesis presented in partial fulfilment of the requirements for the degree of

MASTER OF INFORMATION SCIENCES

IN

INFORMATION SYSTEMS

at Massey University, Palmerston North, New Zealand

Zheng Zhou

May 2006

Acknowledgement

I wish to take this opportunity to give my sincere thanks to Associate Professor, Dr. Sven Hartmann for his kind and enlightening guidance and constructive suggestions throughout this research and valuable recommendations on improvement of the thesis.

It has been my pleasure to meet those wonderful people at the University and I have developed a solid friendship with many of them; without their support in varied forms, it would have been an arduous year for me. Particularly, I am thankful to Madre Chrystall for her inspiring encouragement, valuable technical advice on MySQL and kind proofreading. I also benefited from discussions with Thu Trinh who shared with me her knowledge on Latex. Furthermore, the Massey Masterate Scholarship which I was awarded in 2005 academic year has made my life financially manageable.

Finally, I owe my entire life and all my success to my parents for their incomparable enormous *love*; they are appreciated, respected and treasured by me forever and a day. My heart is always warmed also by the fondness from my aunt and my sister. They light up my life and are my everything.

Zheng Zhou
9 May 2006

Contents

1 Introduction **6**

1.1 Background 6

1.2 Motivations 6

1.3 Related Work 9

1.3.1 Hypothesis Research 9

1.3.2 Schema Derivation Algorithms 10

1.3.3 Schema Tree Inference Algorithms 11

1.3.4 XML-RDB Mapping 12

1.4 Our Contribution 14

1.5 Thesis Outline 14

2 XML Essentials **15**

2.1 XML Basics 15

2.2 XML Schema Schemes 16

2.2.1 The DTD (Document Type Definition) Family 16

2.2.2 The W3C XSD (XML Schema Definition) Family 16

2.2.3 The RELAX NG Family 17

2.2.4 The DataGuide Family 17

2.3 Functional Dependencies in XML 18

3 XML Schema Extraction **19**

3.1 Element Relationship Model for XML (ER-XML) 19

3.2 ER-XML Extraction 23

3.2.1 Depth-First Search (DFS) vs. Breadth-First Search (BFS) 23

3.2.2 ER-XML Extraction (EXE) Algorithm 25

4 XML-Relation Data Transformation **29**

4.1 Preliminary Definitions 29

4.1.1 Rooted Tree and XML Tree 29

4.1.2 XML Schema Tree 30

4.1.3 XML Schema Tree Features 30

4.1.4 Mappings between XML Trees 31

4.1.5 XML Data Tree 33

4.1.6 Functional Dependencies for XML 33

4.1.7 Schema Vertex Table 35

4.2 Transformation Methodology 36

4.3 The Algorithm SVT-Trans 39

4.3.1 SVT-Trans Overview 39

4.3.2 Handling NULL Values 42

5 Prototype System **45**

5.1 System Overview 45
5.1.1 Functionality 45
5.1.2 System Architecture 46
5.2 Functional Dependency Inference Algorithms 47
5.2.1 The Naive Algorithm 48
5.2.2 The Enhanced Algorithm Using Transversals 49
5.2.3 The Enhanced Algorithm FastFDs 50
5.3 Implementation Considerations 52
5.3.1 Programming language — Java 52
5.3.2 Data Definition and Manipulation Language — MySQL 54
5.3.3 XML Parser — DOM 54
5.3.4 Data Structure 55

6 Case Study **59**

6.1 Walking Through XFD-Miner 59
6.2 Performance Testing 61

7 Conclusion and Future Work **65**

7.1 Conclusion 65
7.2 Future Work 66

A XFD-Miner Guide **68**

A.1 Environment Configuration 68
A.2 Run XFD-Miner 69

A.2.1 Installed Directory Tree 69

A.2.2 Execute XFD-Miner 70

A.3 Miscellaneous 71

B Sample XML document: *warehouse.xml* **72**

Bibliography **77**

Chapter 1

Introduction

1.1 Background

Following the advent of the web, there has been a great demand for data interchange between applications using internet infrastructure. XML (eXtensible Markup Language) provides a structured representation of data empowered by broad adoption and easy deployment. As a subset of SGML (Standard Generalized MarkupLanguage), XML has been standardized by the World Wide Web Consortium (W3C) [Bray et al., 2004]. XML is becoming the prevalent data exchange format on the World Wide Web and increasingly significant in storing semi-structured data. After its initial release in 1996, it has evolved and been applied extensively in all fields where the exchange of structured documents in electronic form is required.

As with the growing popularity of XML, the issue of functional dependency in XML has recently received well deserved attention. The driving force for the study of dependencies in XML is it is as crucial to XML schema design, as to relational database(RDB) design [Abiteboul et al., 1995].

1.2 Motivations

As semi-structured data has become prevalent with the growth of the Internet and other on-line information repositories, many organisational databases are presented on the web as semi-structured data. Designing a 'good' semi-structured database is increasingly cru-

cial to sustain data integrity and prevent data redundancy, inconsistency and updating anomalies. Redundant information caused by functional dependencies in XML may give rise to such problems. Therefore, identifying XML functional dependencies and thus achieving normalisation becomes vital in good XML design.

Often in design practice, we are facing a task of finding all possible functional dependencies satisfied by a given XML document, which may imply business rules. Thus emerges a new research direction: the *XML dependency discovery* problem, on which however, little investigation has been conducted so far though a breakthrough would be of prominent value in practice.

XML schema plays a substantial role in discovering functional dependencies of XML data, since they are defined on top of schematic information, as with relational databases.

In addition, it is well-known that XML schema information specifies the internal structure of an XML document, which realises the promise of XML as the universal data representation format enabling free electronic data interchange (EDI) and integration of disparate data sources. It is also critical in the efficient storage of XML data as well as formulation, optimisation and query processing [Garofalakis et al., 2000]. Unfortunately, in practice many XML documents are not associated with schema definitions, giving rise to the task of inferring the schematic information from XML documents.

Our preliminary feasibility studies on XML dependency discovery have suggested the 'divide-and-conquer' strategy, leading to the following problem decomposition:

1. **XML Schema Extraction**

   This is determined by the fact explained previously that XML schema information is essential but absent in most cases. Certain generalisation of input data is often required in schema extraction; ideally the extracted schema should, on one hand, tightly represent the data, and be concise and compact on the other hand. As the two requirements essentially contradict each other, finding an optimal tradeoff is a difficult and challenging task [Chidlovskii, 2001].

2. **XML-Relation Data Transformation**

   Next, semantic data held in the original XML instance is extracted into a tabular

format with the help of its schema. The inspiration for such a transformation comes from appreciation of over 20 years of work invested in relational database technology. Relational functional dependencies have been well explored and some inference algorithms with satisfactory performance are already in existence, which we can leverage to assist in discovering functional dependencies in XML. There have been some research endeavours on mapping XML documents to relational tables, as further illustrated in the section 'Related Work'.

3. **Relational FD Inference**

The final step is to apply some well-developed relational functional dependency inference algorithms to the data in relational format after the transformation to achieve our ultimate goal.

Figure 1.1 shows the entire work flow:

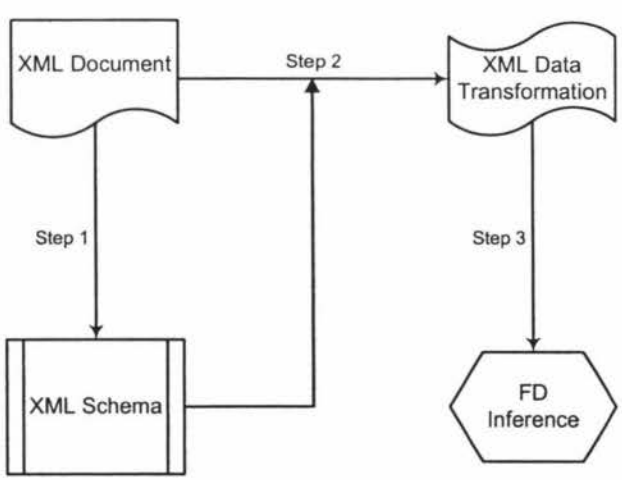

Figure 1.1: High Level Work Flow

1.3 Related Work

1.3.1 Hypothesis Research

The inference of structure out of semi-structured data has been long-standing in the XML research area [Sakakibara, 1997]. Some approaches investigated possible solutions derived from theoretical grammatical inference and were very powerful at the conceptual level. [Ahonen, 1996] presented a technique based on machine learning, with the help of finite-state automata describing the given instances completely. These automata were modified by considering certain context conditions, corresponding to generalisation of the underlying language, which were then converted into regular expressions to construct the grammar. Although traditional grammatical inference methods for DTD generation stated in [Ahonen, 1996] are theoretically appealing (as they guarantee to infer languages falling within certain language classes), it is not clear whether the structure within the limited context is valid in practice, i.e., its theoretical appeal may not necessarily translate into practical applicability.

[Young-Lai, 1996] discussed a grammatical inference method generating stochastic finite automata using an adapted stochastic method and attempted to improve it by isolating low frequency data components and allowing adjustment at the generalisation level. This approach was derived from more recent work in grammatical inference, with the base algorithm known as Alergia [Carrasco and Oncina, 1994]. As with the methods of Ahonen, Alergia has strong theoretical significance. Again, though, our interest lies in practical performance. Moreover, none of them even touched the problem of how to present schema information with high understandability to the user.

[Garofalakis et al., 2000] proposed XTRACT, a specialized DTD induction system consisting of a generation module, a factoring module and an MDL[1] (Minimum Description Length) module. XTRACT employed generalisation and factorisation of regular expressions, to derive a pool of candidate DTDs, and then used the MDL principle as the basis to make a final selection. Still, XTRACT requires human intervention and judgement in making a choice out of all candidates.

[1]The reader of interest in MDL is referred to [Rissanen, 1978, Rissanen, 1989] for further details.

1.3.2 Schema Derivation Algorithms

Some research focusing on XML practice has also been going on, mainly centring on DTD and schema tree extraction. [Chen, 1991] talked about generation of '*de-facto grammar*', which simply aggregated structures of all XML instances to be the DTD. The *de-facto grammar* is obviously far too simple and limited.

[Chidlovskii, 2002] modelled the XML schema as extended context-free grammars and developed an extraction algorithm inspired by methods of grammatical inference. The algorithm was also said to cope with the schema determinism requirement imposed by XML DTDs and XML Schema languages. He defined *(range) Extended Context Free Grammar* (ECFG) as a 5-tuple $G = (T, N, D, \delta, Start)$, where T, N and D are disjoints set of terminals, non-terminals and datatypes; $Start$ is an initial non-terminal and δ is a finite set of production rules. The rules take the form $A \rightarrow a$ for $A \in N$, where a is a range regular expression over terms, and one term is a terminal-nonterminal-terminal sequence like $t\ B\ t'$, briefly $t : B$, where $t, t' \in T$ and $B \in N \cup D$. The extraction algorithm firstly generalized ECFG from XML content, which was then transformed to an XML schema definition. Details of the algorithm are shown in Figure 1.2:

0. Represent XML documents as set I of structured examples.
1. Induce an extended context-free grammar G from I:
   - 1.1 Create the initial set of nonterminals N:
   - 1.2 Merge nonterminals in N with the similar content and context;
   - 1.3 Determine tight datatypes for terminals in G;
   - 1.4 Generalize contents in nonterminals into range REs.
2. Transform the result ECFG G into an XML Schema definition S.

Figure 1.2: ECFG Extraction Algorithm (Adapted from [Chidlovskii 2002, p. 292])

In addition to work concerned with the problem of DTD inference, there have also been many papers published on related topics. Most notable amongst these is work within the Lore semistructured database project to infer DataGuides [Goldman and Widom, 1997]. This included the MakeDataGuide algorithm to construct a strong DataGuide over a source database as shown in Figure 1.3 — A DataGuide is strong *iff* it shares exactly the same set of label paths as in the source, nothing more or less. Despite its simplicity and high understandability at the conceptual level, the algorithm does not even mention

any technical aspects, such as data structure, i.e., how the schematic information can be actually stored and utilised.

```
// Input: o, the oid of the root of a source database
// Effect: dg is set to be the root of a strong DataGuide for o
targetHash = global empty hash table, to map source target sets to DataGuide
objects
dg = global oid
MakeDataGuide(o) {
    dg = NewObject()
    targetHash.Insert(o, dg)
    RecursiveMake(o, dg)
}
RecursiveMake(t1, d1) {
    p = set of <label, oid> children pairs of each object in t1
    foreach (unique label l in p) {
        t2 = set of oids paired with l in p
        d2 = targetHash.Lookup(t2)
        if (d2 != nil) {
            add an edge from d1 to d2 with label l
        } else {
            d2 = NewObject()
            targetHash.Insert(t2, d2)
            add an edge from d1 to d2 with label l
            RecursiveMake(t2, d2)
        }
    }
}
```

Figure 1.3: Algorithm MakeDataGuide (Adapted from [Goldman et al. 1997 , Figure 4, p. 8])

1.3.3 Schema Tree Inference Algorithms

The subject of schema tree and related issues, such as tree extraction, are not recent in research area. A labelled tree specifying nesting relationships between labelled vertices was referred to as XML schema tree elements [Cruz et al., 2004]. A schema tree was also defined as an ordered tree representing the XML schema in terms of a set of constructors:

sequence (','), *repetition* ('*'), *union* ('|'), < *tagname* > (corresponding to a tag) and < *simpletype* > (corresponding to base types) [Ramanath et al., 2003].

[Chen et al., 2002] stated a schema tree generation algorithm as displayed below:

ALGORITHM 3: Generate schema tree
INPUT: Node N of the tree T' constructed at Step 7 in Algorithm 1
OUTPUT: Schema tree

Step 1: if N is a leaf node then return;
Step 2: for all child node C of node N do
Step 3: if name of edge E which connect C and N existed at the same level then{
Step 4: find node C'and corresponding edge E'holding same name with E, which connects C' and N;
Step 5: all subtrees of C is moved to be subtrees of C';
Step 6: delete node C and edge E;}
Step 7: for all child node C of node N do
Step 8: recursively applying algorithm 3 from node C;

Fig.9: Generate schema tree algorithm

Figure 1.4: Schema Tree Generation (Adapted from [Chen et al. 2002 , Figure 9, p.84])

There are at least three deficits in this algorithm: firstly, it only considers, compares and processes identical elements appearing at the same level (in Step 3), exclusive of the scenario with one element occurring at different levels. Secondly, it just simply aggregates subtrees of all occurrences of an element (node) (in Step 5), which will merely give a document tree at most, instead of a schema tree as supposed. Third, structural information captured is rather poor; only a collection of possible sub-element names, without any knowledge of element order, whether they are optional, compulsory, or iterating.

1.3.4 XML-RDB Mapping

Researchers have already shown their interest in transforming data in XML format into relational database. [Christophides et al., 1994] proposed a one-to-one mapping from each element declaration in the DTD to a relation. It is apparently a simple way of generating corresponding relational schema but likely leads to excessive fragmentation.

[Shanmugasundaram et al., 1999] suggested analyse a DTD and automatically convert it to a set of relational schemata. To do this, the original DTD should be firstly simplified by discarding element order information before generating the final relation schema:

- **Basic approach** Generate a DTD graph after grouping or flattening element frequency specifications and the respective element graph on which the relational schemata are decided;

- **Shared approach** Create a separate relation for each element node represented by multiple relations in the basic approach, and share this relation; or

- **Hybrid approach** Same as the shared approach except for some variance in element processing.

Their work will also result in excessive fragmentation of DTDs, causing unnecessary data scatter, which incurs unaffordable cost from joins when multiple relations need to be accessed.

A new inlining algorithm was put forward by [Lu et al., 2003], featuring modeling XML attributes as XML elements since they can be treated as elements without further nesting structure. It comprises similar steps as the others: Create a DTD graph after DTD simplification and inline as many descendant elements as possible to an XML element to eliminate redundancy caused by shared elements in the generated schema, which is to be eventually generated based on the inlined DTD graph. Such an inlining algorithm can relatively reduce redundancy in comparison to the shared approach introduced previously, though data scatter is still present.

[Yan and Fu, 2001] described construction of schema prototype trees representing the structure of a simplified DTD and subsequent generation of relational schema prototypes. They also briefly mentioned functional dependency and candidate key detection and relational schema prototype normalisation techniques.

In a summary of relevant literature, most of them are of little practical significance and applicability as their studies are no more than academic research, although their complexity varies. Furthermore, none of them even addressed how to render the schema information to the user. Nearly all research on XML-RDB mapping is somewhat a schema-awared approach, requiring the existence of the DTD to operate, which most XML documents in practice lack. Excessive data fragmentation is another common awkwardness; we end up with an unmanageably large number of relations, complicating the situation since many functional dependencies may span several relations.

1.4 Our Contribution

In our research, we delved into both schema extraction and the XML-Relation data transformation problem. A novel data representation model, ER-XML (Element Relationship model for XML) was devised, utilising an implementation-focused algorithm capable of being directly applied to XML practice. ER-XML can also help to extract and identify cardinality constraints. The data structure invented was properly designed to facilitate graphical representation generation as well as compatibility validation. As for XML-Relation transformation, we have developed an entire set of algorithms, *SVT-Trans* with the help of ER-XML and the concept of 'Almost Copy' in XML tree, which retrieves semantic data from an original XML document and places them into a relational format using recursion computation. The output of SVT-Trans can be directly exploited by relational functional dependency discovery algorithms. A prototype system was also successfully implemented and a case study was provided which demonstrated correctness and soundness of our work.

1.5 Thesis Outline

Following the introduction and review of related work, Chapter 2 of the thesis presents some essential XML and affiliated technology. We investigate XML schema extraction and illustrate the ER-XML model and the ER-XML Extraction (EXE) algorithm in Chapter 3. Chapter 4 discusses in detail SVT-Trans designed to convert XML data into a relation. Some preliminary definitions are also covered. Design decisions and architecture of the prototype system, XFD-Miner, unfold in Chapter 5, with a case study in Chapter 6. Finally, Chapter 7 summarises our work and points out future research directions.

Chapter 2

XML Essentials

This chapter provides background knowledge on XML standards and related technology. We start with an overview of XML, then its assorted schema languages and finally functional dependency issues in XML.

2.1 XML Basics

As a metalanguage used to define new markup languages, XML was developed by the W3C[1] (World Wide Web Consortium). It has been well acknowledged as a standard for data representation and exchange on the Internet. Unlike HTML, which was designed to display data, XML is intended to describe structured data [Sun Microsystems Inc., 2002]; it allows designers to create their own customised tags, enabling the definition, transmission, validation and interpretation of data between applications and between organisations[Gaskin, 2000]. Besides, XML will enable a new generation of web-based data viewing and manipulation applications.

XML documents are composed of content and markup by way of nested pairwise start- and end-tags. There are some kinds of markup that can occur in an XML document, for instance elements, attributes, entity references and document type declarations, with the first two particularly pertinent to our work[2].

A decent XML document must be well-formed. The well-formedness of an XML doc-

[1]It first became a W3C Recommendation on 10 February 1998.

[2]Relevant fundamental knowledge on XML is assumed for the reader and thus not reiterated here.

ument is ensured by enforcing proper nesting as for its logical and physical structure [Bray et al., 2004]. Those grammatical features need to be captured as schema and will be used in the validation process of an arbitrary XML document.

2.2 XML Schema Schemes

At present, several XML schema schemes are widely used, including DTD, XML schema, Relax NG, DataGuide and so forth. It is worthy of note that the first three are prescriptive schema clearly defining and imposing legitimate components and structure, whereas DataGuide serves as a descriptive schema only depicting and inferring structural information from an existing XML document.

2.2.1 The DTD (Document Type Definition) Family

Inherited from SGML, DTD is the most widely deployed means of defining an XML schema. Its purpose is to specify document structure and legal building blocks, such as elements and attributes of an XML document (Bray et al. 2000). The W3C Recommendation on XML defines a DTD as containing or pointing to markup declarations that provide a grammar for a class of documents, such as entity declaration, element type declaration and attribute-list declaration. A DTD, to the author, is indeed a formal prescription in XML declaration syntax of a desired document, which sets out what names are to be used for the different types of elements and associated attributes, where they may occur, and how they all fit together.

2.2.2 The W3C XSD (XML Schema Definition) Family

W3C has developed XSD to provide an alternative to XML DTD that supports namespaces, facilitates the design of open and extensible vocabularies, and meets the requirement of data-oriented applications for a richer datatyping system. It does so by borrowing many features from object-oriented programming languages, and hence is generally considered complex, partly because of additional features, and partly because of the style of the

recommendation which describes the validation process more than the modelling features. W3C XSD is a strongly typed schema language that eliminates any non-deterministic design from the described markup to ensure that there is no ambiguity in the determination of the datatypes and that the validation can be made by a finite state machine [Fallside and Walmsley, 2004].

2.2.3 The RELAX NG Family

RELAX NG is a schema language rooted in finite tree automata (FTA) that can be used to validate an XML document against a particular schema. It is formed by the unification of TREX [Clark, 2001] with RELAX [Murata, 2001], adopting XML for writing schemas. Its syntax appears similar to a description of the instance document in ordinary English and simpler than W3C XML Schema. Some constraints, especially those on the fringe of non-deterministic models and combinations in document structure can be expressed by RELAX NG but not W3C XML Schema. RELAX NG arranges markup declarations as XML syntax itself, such as <element name="">, <attribute name="">. Additional integrity constraints are also expressed in pairwise tags, <zeroOrMore>, <oneOrMore> and <optional> for instance, compared to '*', '+' and '?' in DTD. Despite its plausible technical superiority to W3C XML Schema, at present it is short of support from software vendors and XML.

2.2.4 The DataGuide Family

DataGuide was released as a partial outcome of the Lore (Lightweight Object Repository) project at Stanford University in the late 1990s, which extends the theoretical foundation of Representative Objects (ROs) [Nestorov et al., 1997] and the Object Exchange Model (OEM) [Papakonstantinou et al., 1995]. It is a concept of dynamically generated structural summaries of graph-structured databases and an object data model, respectively [Goldman and Widom, 1997]. Goldman et al. [1997, p. 5] defined that 'DataGuide for an OEM source object s is an OEM object d such that every *label path* of s has exactly one data path instance in d, and every label path of d is a label path of s'. Its intention is to dynamically generate and maintain 'concise' and 'accurate' structural summaries of semi-structured databases, which, yet undesirably, are not associated with an explicit

schema. Therefore DataGuide, as a descriptive schema, reflects data and its changes, instead of forcing data to adhere to a predefined schema as relational and object-oriented databases. In fact, DataGuide should be categorised under XML schema graph, which was established as 'an XML graph where no two successors of the same vertex have the same name and the same kind' [Hartmann et al., 2003]. Goldman et al. also proposed a path index, strong DataGuide, with the building algorithm analogous to conversion from a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA). [Milo and Suciu, 1999] believed that it was just a simple labelled path and unsuitable for complex path queries with regular expressions. In addition, there is certainly a substantial overhead for extracting and maintaining DataGuide repositories of unmanageably large size.

2.3 Functional Dependencies in XML

Normalisation as a design technique has long been widely used as a guide in designing relational databases. It is essentially to create a set of relational tables that are free of redundant data and that can be consistently and correctly modified through a two step process of putting data into tabular form by removing repeating groups and then removing duplicated data from the relational tables. The underlying concept is that of functional dependencies. The functional dependency X determines Y, written as $X \rightarrow Y$, holds if and only if all occurrences of a certain value presented in column X determines a single value for column Y without exception, with X, Y belonging to a relational table schema R. Following up functional dependencies drives relational tables into normal form and eliminates redundant data which in turn saves space and reduces data manipulation anomalies. Unfortunately functional dependencies can also strike XML; a poor XML design may well incur subsequent data redundancy and inconsistency problems.

Chapter 3

XML Schema Extraction

In this chapter we elaborate the ER-XML model which has been devised to capture schematic information of an XML document, its graphical representation, and the ER-XML Extraction (EXE) algorithm detailing the specific steps in building up an ER-XML model out of an XML document.

3.1 Element Relationship Model for XML (ER-XML)

DTD declaration in XML documents and no such DTD association in existence drives XML schema extraction, deriving schema information from an XML data source based on only the XML data source. The rationale is that in semistructured data, the information normally associated with a schema is contained within the data, so-called "self-describing" [Buneman, 1997], on which we have contrived a comprehensive data representation model ER-XML.

Without loss of generality, the following assumptions are taken in our research:

1. Usage of name space is not of our interest.

   It is not our everyday encounter and support for name space can always be incorporated afterwards.

2. 'ID', 'IDREF' and 'IDREFS' are also beyond our consideration.

ID, IDREF, and IDREFS are analogous to PK/FK (primary key/foreign key) relationships in relational databases, with few differences.

ER-XML captures essential structural information out of an XML instance, still with complementary desirable features, such as simplicity, high usability and applicability. Nevertheless, it has only three basic concepts: element, relationship and cardinality.

1. **Element**

   An element corresponds to a vertex in an XML schema tree. There is some similarity between element and entity type in traditional data modelling. An entity type has a name and is characterized by the set of attributes that belongs to that entity, whereas an element in ER-XML is described only by its name (tag name) in that semistructured data is normally less structured, i.e. the set of sub-elements and attributes associated with each instance of a certain element varies. As a result, an element is identified by its name and represented as a labelled node in ER-XML. There exist two types of element:

   - An element serves as a *base element* if there is at least one occurrence of it in the XML source.
   - An element serves as a *non-base element* if at least one occurrence of it in the XML source appears as a descendent of a base element.

   The categorisation as 'base' or 'non-base', only refers to the role that an element can take. Obviously, it does *not* exclude that an element can be both, which is often the case, as in Figure 3.1 'State' should appear in the ER-XML model twice, with the role of 'non-base' and 'base', respectively:

   The dual-role property of an element is indeed determined by the fact that an internal node in an XML tree structure can be a 'parent' node and a 'child' node at the same time[1]. Each base element has a non-base element set, which turns out to be an empty set for a leaf node.

2. **Relationship**

[1] Clearly, the root can only have children, but no parent.

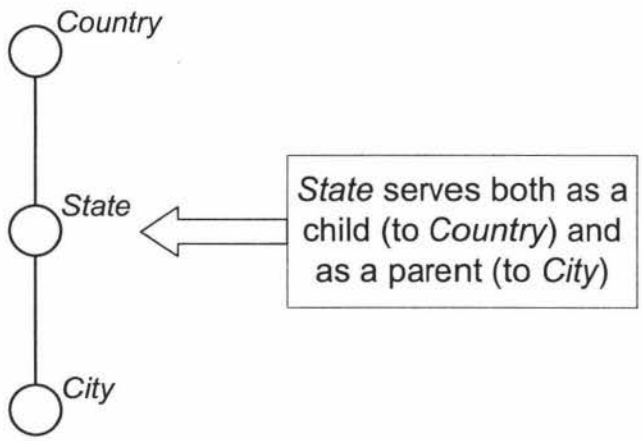

Figure 3.1: Base and Non-Base Role Example

Due to the observation that an element may well take two roles, XML schema tree structure can be constructed by putting together all 'parent-child' element pairs. Therefore in ER-XML we concentrate on the 'parent-child' relationship of elements, representing 'parent-child' in an XML schema tree. Such a relationship in an ER-XML diagram is represented by an arc connecting a parent and its child with an arrowhead pointing to the latter.

3. **Cardinality**

As part of the work on establishing requirements for W3C XML Schema, the Working Group are seeking information about usage patterns of numerical occurrence constraints in content models, i.e. 'minOccurs' or 'maxOccurs' with a more specific numeric value other than 0, 1, or wildcards on elements [Fallside and Walmsley, 2004]. Also for that reason, it becomes essential to extract such occurrence information from a source document.

Capability of recording element cardinality details is another facility of ER-XML. It is well known that occurrences of an element in an XML document can be stipulated in its DTD using frequency constraints, such as '?'(optional), '*'(optional and repeatable). Unfortunately an XML document itself in the absence of DTD is not fully expressive in terms of element cardinality and optionality. For instance, a single presence of an element in an XML file indicates it is either a mandatory or an optional occurrence since a particular XML instance does not exhaust all legitimate options. Instead, the data representation model automatically counts and tracks

the minimal and maximum number of occurrences of every element, despite many possible cases of it. In an ER-XML diagram, cardinality information of each element is displayed next to the corresponding relationship/arc in the format '([min. count], [max. count])'.

Let's look at the following XML document, *reference.xml*.

```
<monographs>
    <book>
        <title>Forever Xanadu</title>
        <author>
            <name>Hengren</name>
            <address>1, Sesame Street</address>
        </author>
        <author>
            <name>Gai</name>
            <address>1, Sesame Street</address>
        </author>
        <editor>
            <name>
                <firstname>Lan</firstname>
                <lastname>Zhou</lastname>
            </name>
            <address>2, Sesame Street</address>
            <address>3, Sesame Street</address>
        </editor>
    </book>
    <book>
        <title>Eternal Bliss</title>
        <author>
            <name>li</name>
        </author>
    </book>
</monographs>
```

Figure 3.2 shows the coloured ER-XML model diagram of *reference.xml*. 'monographs' is the root element with 'book' as its sub-element, which in turn consists of 'title', 'author' and 'editor'. Both 'author' and 'editor' have sub-elements, 'name' and 'address'. 'name' is the parent of 'firstname' and 'lastname'. It is explicit that 'author' is both a base element, to 'name' and 'address' and non-base one, to 'book'. The cardinality information tells us that 'editor', 'address', 'firstname' and 'lastname' are optional (they may be unknown), indicated by the minimal occurrence of zero.

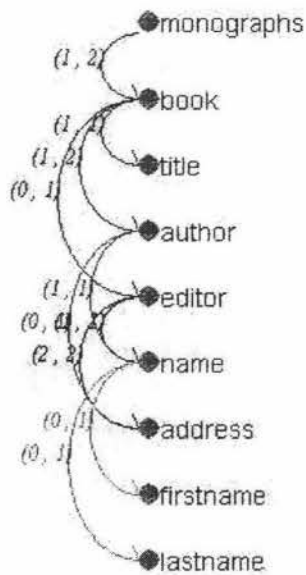

Figure 3.2: ER-XML Diagram Example

3.2 ER-XML Extraction

3.2.1 Depth-First Search (DFS) vs. Breadth-First Search (BFS)

DFS and BFS are the two core approaches to analyzing and processing connected graphs of vertices in computer science and other engineering disciplines.

DFS extends a maze-exploration strategy by always selecting a frontier edge incident on the most recently acquired vertex of a tree graph. If not possible, it then backtracks to the next most recently labelled vertex, i.e. the parent, and tries another edge

[Gross and Yellen, 1999]. DFS examines a frontier edge of the current vertex all the time until the end is reached and certain requirement is not met. Therefore, the algorithm, unless not possible, tends to go deeper in the graph. It is central to algorithms for solving connectivity and numerous other graph-processing problems.
BFS traverses the tree by always selecting frontier edges incident on vertices as close to the starting vertex as possible, i.e. all frontier edges from the starting vertex need to be considered in turn before looking at the next closest vertex. BFS considers *all* frontier edges of the current vertices prior to moving down the graph. It is mainly used for finding the shortest path between vertices.
A comparison example of DFS and BFS is more revealing:

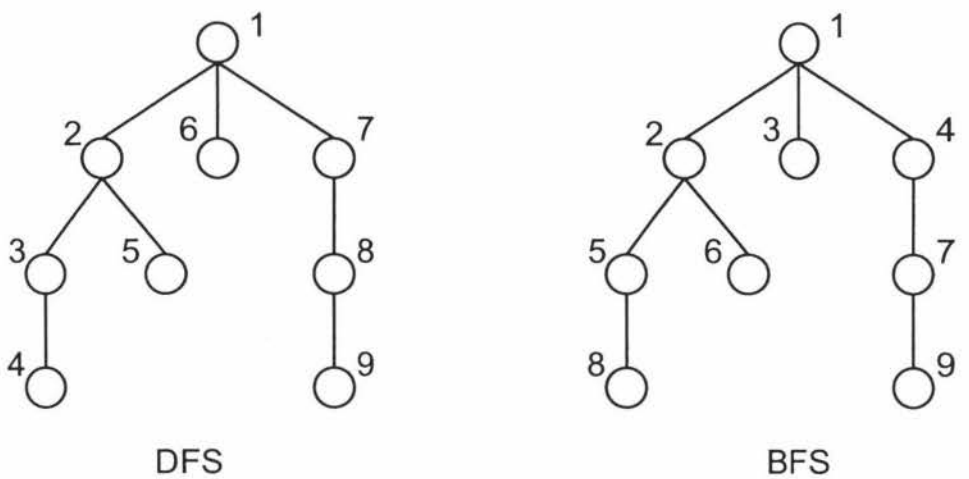

Figure 3.3: DFS and BFS Example(The number associated with each node indicates the sequence in which it is processed)

Both search algorithms maintain a list of edges to be recursively explored until it becomes empty. while both algorithms add items to the end of the list, the only difference is that BFS removes them from the beginning, which results in maintaining the list as a FIFO (First-in-First-out) queue, while DFS removes them from the end, maintaining the list as a stack.

The ER-XML Extraction algorithm follows the BFS approach in that immediate edges between nodes at adjacent levels, 'parent-child' relationship, are of our interest, namely the fan-out edges of a particular node in an XML schema tree are considered in the first place, rather than its further descendants.

3.2.2 ER-XML Extraction (EXE) Algorithm

Required by distinct processing of elements and attributes, EXE is mainly composed of *eleProc* invoking *attrProc* accordingly. Figure 3.4 gives an overview of function calls between those methods:

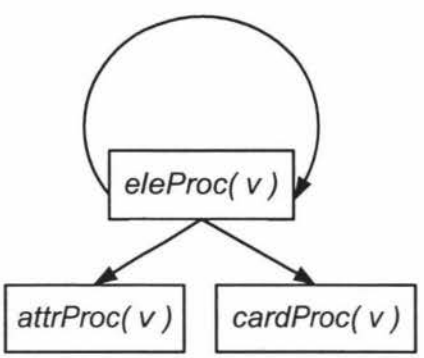

Figure 3.4: EXE Function Call Diagram

EXE is initialised by simply calling *eleProc* for the root:

Input: An XML Data tree T_D with root r
Output: The corresponding $ER-XML$ model for T_D
/* EXE starting point */
(1) $eleProc(r)$;

Figure 3.5: Algorithm: EXE

```
Input: An XML Data tree T_D and v ∈ T_D
Output: Schematic information of v and all its descendants
(1)  attrProc(v);                    //process attributes
(2)  if v is not a leaf then
(3)    if not exist v' already processed of the same type as v then
(4)      Add v as a base element;
(5)      Create its non-base element set NB_v;
(6)      Filled NB_v with all v's child nodes CN_i(1 ≤ i ≤ m, m ≥ 1);
(7)      for each CN_i ∈ NB_v do
(8)        Record Card(CN_i) as for v;
(9)      endfor
(10)   else do
(11)     cardProc(v);                //process cardinality information
(12)   endif
       /* Recursive case */
(13)   for each CN_i of v do
(14)     eleProc(CN_i);
(15)   endfor
(16) else do                         //v is a leaf
(17)   if v previously processed as a base element then
(18)     for each CN_i ∈ NB_v do
(19)       Set lower bound of Card(CN_i) to 0;
(20)     endfor
(21)   else do
(22)     Record v as a base element with empty NB_v;
(23)   endif
(24) endif
```

Figure 3.6: Method: eleProc(v)

Input: An XML Data tree T_D, $v \in T_D$

Output: Attribute information of v, if applicable

```
(1)  if v has attributes present then
(2)    if not exist v' already processed of the same type as v then
(3)      newV = true;
(4)      Create its recorded attribute list AL_v and put in all its attributes;
(5)    else do
(6)      for each attribute a_i of v do
(7)        if a_i ∉ AL_v then
(8)          Add a_i into AL_v;
(9)        endif
(10)     endfor
(11)   endif
(12) endif
```

Figure 3.7: Method: attrProc(v)

Input: An XML Data tree T_D, $v \in T_D$ and its child nodes, $CN_i (1 \leq i \leq m, m \geq 1)$
Output: cardinality information of child nodes of v

```
(1)  for each CN_i do
(2)    if exist its counter C(CN_i) then
(3)      C(CN_i) = C(CN_i) + 1;
(4)    else do
(5)      Initilise C(CN_i) to 1;
(6)    endif
(7)  endfor
     /* Reflect(adjust or add) cardinality of CN_i */
(8)  for each CN_i do
(9)    if CN_i recorded as v's child before then
(10)     Compare C(CN_i) with Card(CN_i) & adjust if needed;
(11)   else do
(12)     Add CN_i into NB_v;
(13)     Set its cardinality: (0, C(CN_i));
(14)   endif
(15) endfor
     /* Check for the reappearance of each element in NB_v */
(16) for each CN_i ∈ NB_v do
(17)   if not exist C(CN_i) then
(18)     Set it's cardinality lower bound to 0;      //The CN_i is optional
(19)   endif
(20) endfor
```

Figure 3.8: Method: cardProc(v)

Chapter 4

XML-Relation Data Transformation

There has been some research on transformation of XML data into relational structure prompted by the growing demand from the industry, which however, due to the complexity and difficulty, has not been satisfactorily catered for. In this section, we contrive a novel data transformation algorithm known as *SVT-Trans* to tackle the problem. We start off with some preliminary definitions upon which our algorithm is rooted [Hartmann and Link, 2003, Hartmann et al., 2003].

4.1 Preliminary Definitions

4.1.1 Rooted Tree and XML Tree

A *rooted tree* is a tree T with one distinguished vertex R_T as the root, such that every vertex can be reached via a directed path from R_T and T contains neither directed nor non-directed cycles. It is a common practice to illustrate the structure of an XML document with the help of trees. For every tree T, let V_T denote its vertex set and A_T its arc set.

Definition 4.1.1. *An XML tree is a rooted tree T together with mappings $name : V_T \rightarrow Names$, $kind : V_T \rightarrow \{E, A\}$ to assign every vertex its name and kind, respectively. We suppose that $Names$ is a fixed set of vertex names, whilst kind E and A reveal whether a vertex actually represents either an element or an attribute.* □

In particular, N_v will be used wherever we refer to the name of vertex v.

4.1.2 XML Schema Tree

Definition 4.1.2. *An XML schema tree is an XML tree T in which no vertex has more than one successor of the same name and the same kind.* □

An XML schema tree can be interpreted as a graphical representation of the structure of a native XML document, which may be easily generated either from the document itself or its associated DTD. We refer to vertices in an XML schema tree as *schema vertices.*

4.1.3 XML Schema Tree Features

In a rooted tree, vertices of either kind A or E without successors are said to be *leaves.* For simplicity, we focus on E-vertices in the rest of the chapter without loss of generality in that A-vertices can be easily accommodated with slight adaptation. Moreover, textual data attached to a non-leaf vertex is also not of interest to us.

Definition 4.1.3. *Given a rooted tree T and any v, $v' \in V_T$, a (v, v')-walk in terms of T is a directed walk from v to v'.* □

It is evident that at most one (v, v')-walk can be found for a vertex pair (v, v') in T. Throughout the paper, a (v, v')-walk is denoted by $v...v'$ due to the absence of confusion. In particular, in the case where v becomes the root of T, i.e. R_T, this can be further eased to $\sim v'$.

In line with the structural flexibility of XML, i.e. different vertices may well have a child vertex of the same name and type, we can pinpoint a vertex v by virtue of $\sim v$ for an XML schema tree T with root R_T and $v \in T$. Such an inspiration gives us the *Universal Naming Scheme* to uniquely represent one vertex. The Universal Naming Scheme utilises the name of all vertices on $\sim v$ partitioned by some delimiters. Take Figure 3.2 for instance, there exist two different 'name' vertices, *monographs$book$author$name*[1] and *monographs$book$editor$name*. This differentiation is vital since they represent distinct semantics; 'author name' and 'editor name'. Where there is no such naming collision, we, for conciseness, instead refer to vertices directly by their element name.

[1]The choice of delimiter $ is partially because it is one of few special characters fully supported by MySQL employed in our case tool to be introduced in the next chapter.

Definition 4.1.4. *Let L_T be the set of all leaves in a rooted tree T. The v-subtree T_v for vertex v in T is the union of all (v, w)-walks in T where $w \in L_T$. In addition, a c-subtree with c as an immediate successor of v is an offspring subtree of v-subtree.* □

Note that any v-subtree of an XML schema tree is again an XML schema tree, which may well be just a single leaf vertex. We also refer to the complete leaf set of T_v as *v-leafset* L_v.

It is obvious that the union of two subtrees in T is again a subtree, which can be used to show that v-subtree is the union of all c_i-subtrees corresponding to its N immediate successors c_i for $1 \leq i \leq N$, which is also justified by an observation that all elements in v-leafset are distributed within some c_i-subtree with $1 \leq i \leq N$.

Example 4.1.1. *Take for instance the XML schema tree T with root R presented in Figure 4.1. (X, P)-walk $= X\$P$, (Z, N)-walk $= Z\$M\$N$ and (R, M)-walk $= R\$Z\$M$, i.e. $\sim M$. X-subtree T_X and Z-subtree T_Z are enclosed by dashed line. In particular, T_X has two offspring subtrees T_P and T_Q, i.e. leaf vertex P and Q.*

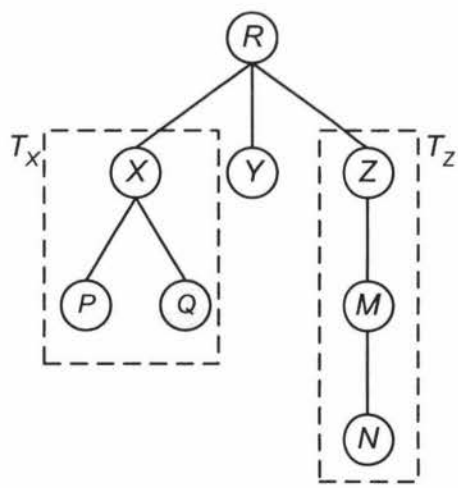

Figure 4.1: Schema tree, (v, v')-walk and v-subtree example

4.1.4 Mappings between XML Trees

Consider a mapping $\phi : V_{T'} \rightarrow V_T$ between two XML trees T and T' with root R and R' respectively. ϕ is *kind-preserving* if $kind(v') = kind(\phi(v'))$ for all $v' \in V_{T'}$. Furthermore,

we regard ϕ *name-preserving* if $name(v') = name(\phi(v'))$ for all $v' \in V_{T'}$. Such a mapping ϕ becomes a *homomorphism* between T and T' if the following conditions hold:

- R' is mapped to R, that is, $\phi(R') = R$,
- every arc in T' is mapped to an arc of T, that is, $(u', v') \in A_{T'}$ implies $(\phi(u'), \phi(v')) \in A_T$,
- ϕ is kind-preserving and name-preserving.

Definition 4.1.5. *A homomorphic $\phi : V_{T'} \rightarrow V_T$ is an isomorphism if ϕ is bijective and ϕ^{-1} is also a homomorphism. In this case, we say T' is isomorphic to T or a copy of T.*

Example 4.1.2. *Consider XML tree T_1, T_2 and T_3 in Figure 4.2. Clearly, the kind-preserving and name-preserving mapping $\phi : V_{T2} \rightarrow V_{T1}$ between T_2 and T_1 is homomorphic in that the root and every arc in T_2 are mapped to the correspondence in T_1. So is the mapping $\phi' : V_{T3} \rightarrow V_{T1}$. Moreover, T_3 is isomorphic to T_1 whilst it is not true in terms of T_2 and T_1; ϕ is not bijective and nor is ϕ^{-1} homomorphic.*

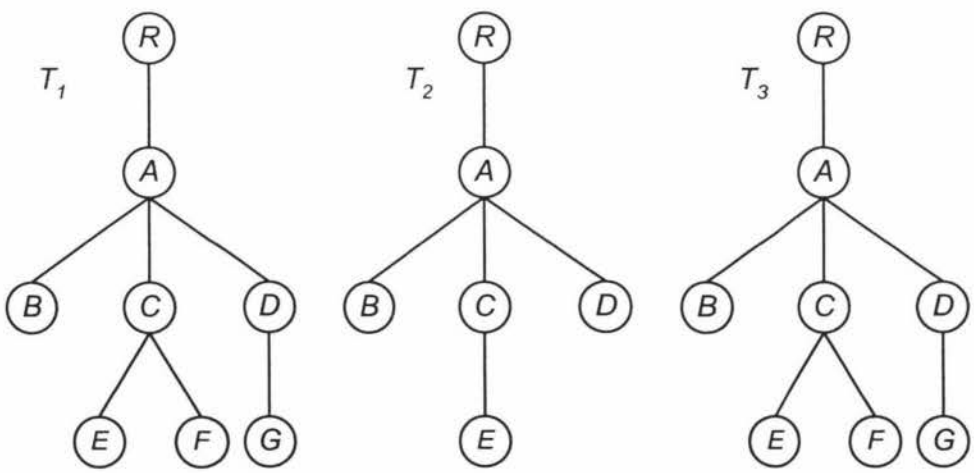

Figure 4.2: Homomorphism and Isomorphism example

Definition 4.1.6. *Let T and T' be two XML trees. An H'-subtree $T'_{H'}$ in T' is a subcopy of T if it is a copy of some H-subtree T_H in T. An almost copy of T, denoted by O_T, is the maximal subcopy of T such that any other H''-subtree $T'_{H''}$ in T', with $O_T \subset T'_{H''}$, is no longer a subcopy of T.* □

Example 4.1.3. *Figure 4.3 shows two XML trees T_1 and T_2. It is easy to identify two almost copies of T_1 in T_2, each of which contains exactly one of the two D-subtrees in T_2 and all other vertices.*

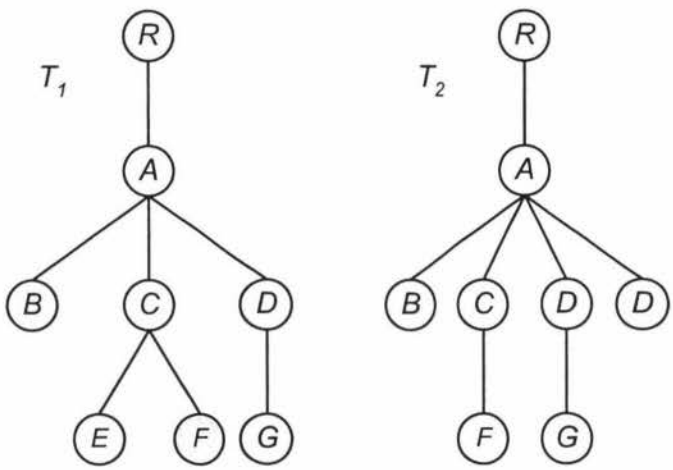

Figure 4.3: Almost copy example

4.1.5 XML Data Tree

Definition 4.1.7. *An XML data tree is an XML tree T together with an evaluation $val : V_T \rightarrow STRING$ assigning every leaf a (possibly empty) string $val(v)$.* □

An XML data tree T_D can be derived from the associated XML schema tree T by instantiating each schema vertex within. We refer to vertices in T_D as *instance vertices* corresponding to some schema vertices in T correspondingly.

Definition 4.1.8. *An XML data tree T' is said to be compatible with an XML schema tree T if the mapping $\phi : V_{T'} \rightarrow V_T$ is homomorphic.*

4.1.6 Functional Dependencies for XML

Functional dependency in XML(or XFD for short) is introduced by the following definition:

Definition 4.1.9. *Let T be an XML schema tree with root R, a functional dependency is an expression $v : X \rightarrow Y$ where X and Y are R-subtrees of T. Let T_D be an XML data tree compatible with T. Then T_D satisfies the XFD $v : X \rightarrow Y$ if for any two almost copies O_1 and O_2 of T in T_D, the projections $O_1|_Y$ and $O_2|_Y$ are equivalent whenever the projections $O_1|_X$ and $O_2|_X$ are equivalent and copies of X, i.e. $O_1|_X = O_2|_X \Rightarrow O_1|_Y = O_2|_Y$.* □

Example 4.1.4. *Let us consider an XML schema tree T and its compatible XML data tree T_D[2] shown in Figure 4.4. Based on the total four almost copies of T, $T_1 \sim T_4$ (Figure 4.5) identified in T_D , T_D satisfies $XFD : C \rightarrow D$ because a fixed D_1 presents for all occurrences of C_1. In contrast, neither $C \rightarrow E$ nor $D \rightarrow E$ holds.*

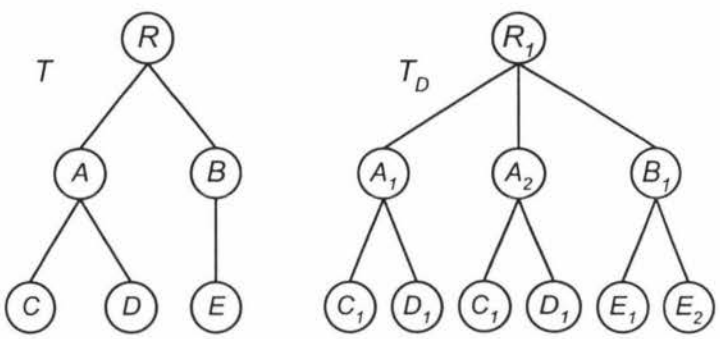

Figure 4.4: XFD example

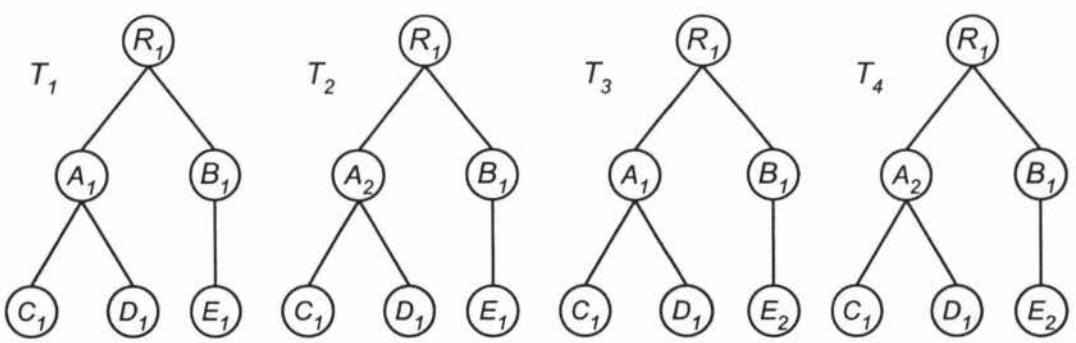

Figure 4.5: XFD example (contd.) - identified almost copies

[2]We make it a convention to always label a schema vertex with an upper-case letter, say A, and its instance vertices A_i accordingly, with $i \in \mathbb{N}$.

4.1.7 Schema Vertex Table

Now we are ready to launch a crucial concept, *schema vertex table* (*SVT* for short), utilised as the basic building block in SVT-trans.

Definition 4.1.10. *Consider an XML schema tree T, its entire leaf set L_T and an XML data tree T' compatible with T, a schema vertex table SVT_T for T is the consequence of mappings schema : $L_T \rightarrow \#SVT_T$ and, for each almost copy of T presented in T', value : $val(\ell) \rightarrow SVT_T$ for each leaf $\ell \in L_T$, to allocate to SVT_T schema exactly one column representing each leaf in L_T and to accommodate their associated textual values from all almost copies of T, O_T identified in T'.* □

In particular, S_T refers to the entire set of SVTs for a schema tree T and all subtrees in T.

Example 4.1.5. *Let us revisit the pair of XML schema and data tree in Figure 4.4. SVT_T contains 5 tables as shown in Figure 4.6 (I), whilst subgraph (II) exhibits the result of SVT_T population with data from T_D.*

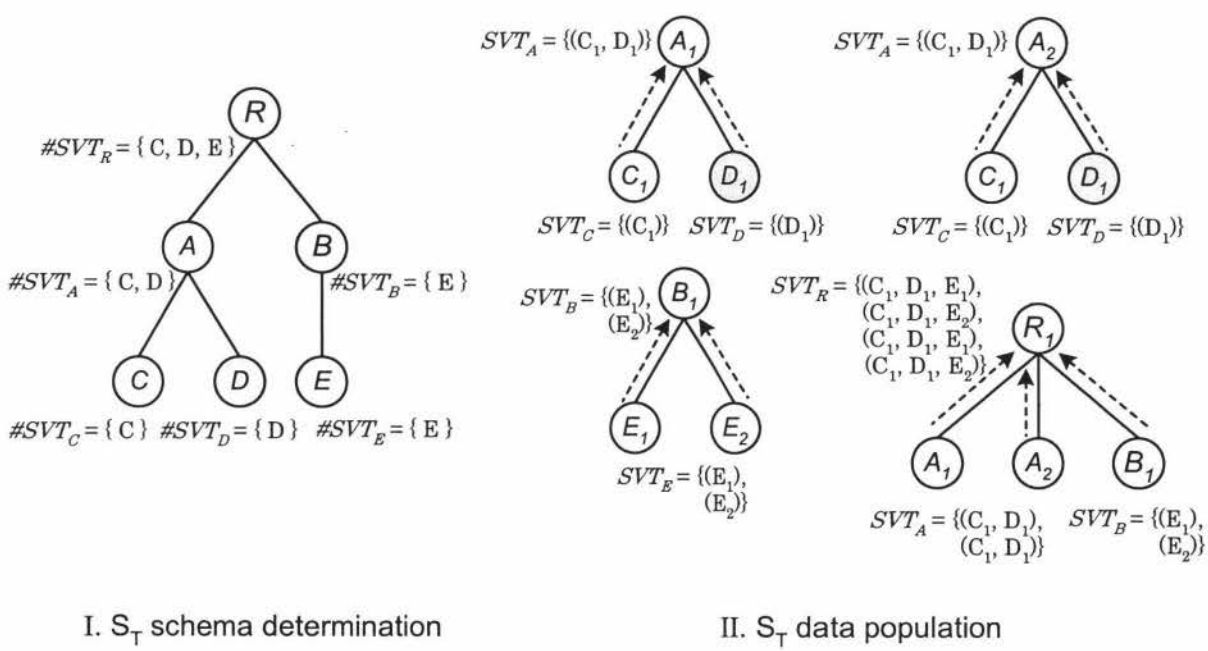

Figure 4.6: SVT example

We can see from the above example that each v-subtree is represented by a schema vertex table SVT_v, which is essentially in a two-dimension tabular format with each column representing a leaf in v-subtree. In particular, SVT_ℓ consists of one column only, with column name as the element name of ℓ, $N_\ell \in Names$ for any $\ell \in L_G$, as no leaf has children but only textual data, which populates SVT_ℓ. It is also obvious that the semantic information in each almost copy O_v of v-subtree in T_D is captured by one single relation in SVT_v due to the way the schemata of SVT, denoted by $\#SVT$, is decided. For instance, 4 relations of SVT_R in Example 4.1.5 correspond to those 4 almost copies in Figure 4.5 respectively.

Subsequently emerges an intuitive question — how their SVTs correlate for a P-subtree and its offspring subtrees if any, the answer to which leads to:

Theorem 4.1.1. *Let P-subtree be in a schema tree $T(P \in V_T)$ and C_i-subtrees $(1 \leq i \leq N)$ be its offspring subtrees, then $\#SVT_P$ is the union of schema of all SVT_{Ci}, i.e. $\#SVT_P = \bigcup_{i=1}^{N} \#SVT_{Ci}$. In particular, SVT_P is populated with the Cartesian product of all SVT_{Ci}, i.e. $SVT_P = SVT_{C1} \times SVT_{C2} \times ... \times SVT_{CN}$.*

Proof. As pointed out, the collaboration of all C_i-subtrees $(1 \leq i \leq N)$ constitutes P-subtree, whilst we know that a SVT_v retains all structural and semantic information presented in v-subtree from Definition 4.1.10. Thus substituting SVT_v for v-subtree $(v \in \{P, C1, C2, ..., CN\})$ justifies this proposition in two aspects: union operation in the structural perspective and Cartesian join in the semantic perspective. The latter is determined by the inference that no SVT_{Ci}, SVT_{Cj} $(i \neq j)$ exist with $\#SVT_{Ci} \cap \#SVT_{Cj} \neq \emptyset$ originating from Definition 4.1.2. □

4.2 Transformation Methodology

By now, we are ready for the methodology in SVT-Trans, the *Schema Vertex Table Manipulation* approach by means of recursion developed based on Theorem 4.1.1, with the major processes of *Initialisation* and *Population*, and the operations *Aggregation* and *Propagation*. It applies to $\#SVT$ determination as well as SVT population. We will scrutinize them in detail:

- **Initialisation process**

  Initialisation is conducted right at the beginning to decide the structure for S_T for an XML schema tree T, depending on type(v):

  - For a leaf ℓ, the leaf vertex name N_ℓ constitutes $\#SVT_\ell$ as previously demonstrated.
  - For a non-leaf v-subtree, $\#SVT_v$ can be determined by recursively applying Theorem 4.1.1 from the leaf level up, i.e. a bottom-up approach, with the help of aggregation and propagation.

- **Population process**

  After being set up, those SVTs then need to be populated with the semantic data contained in all almost copies of corresponding subtrees identified in the data tree of the original XML document. Again, we start with SVTs at the leaf level, each of which in most cases holds only a single row[3] — the textual data directly attached to the leaf itself. Sometimes, population of a leaf-level SVT_ℓ may trigger *aggregation* if ℓ is the last (or only) child of its parent.

- **Aggregation operation**

  Aggregation is a process in which a parent SVT is subsequently populated with the Cartesian product of all its child SVTs once they are done (a *join* operation), or a parent $\#SVT$ is set by combining all its child $\#SVT$s once they are done (a *union* operation). This is justified by the second half of Theorem 4.1.1. Aggregation always occurs at leaf level.

  Example 4.2.1. *Let us look at when aggregation happens to S_T of the schema tree T in Figure 4.7 during its initialisation process. Schema determination for leaf vertex table SVT_F (as the last child of C) or SVT_G (as the only child of D) will result in aggregation leading to similar action on SVT_C or SVT_D correspondingly.*

  Under the subsequent circumstance, propagation comes into play:

  - The parent vertex with its SVT populated or $\#SVT$ determined as a consequence of aggregation again is the last (or only) child of its parent.

[3] An exception is when a leaf instance vertex has some siblings of the same type as itself.

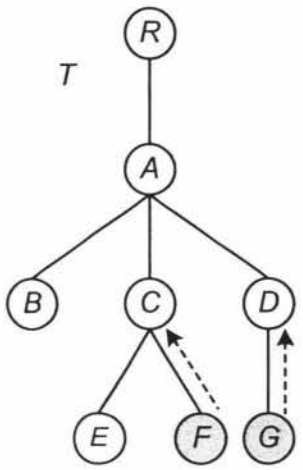

Figure 4.7: Aggregation example

Example 4.2.2. *The aggregation from G in Example 4.2.1 will give birth to a propagation, indicated by an empty arrow in Figure 4.8, upward to A in that its parent D is also the last child of A.*

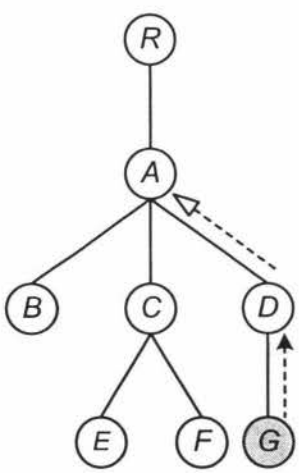

Figure 4.8: Propagation example

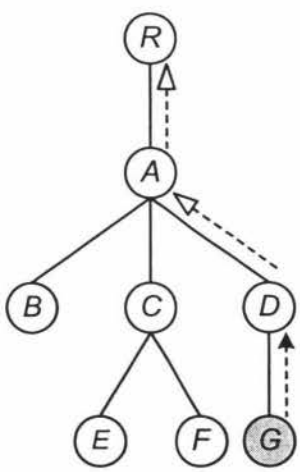

Figure 4.9: Greedy propagation example

- **Propagation operation**

  As seen above, propagation takes place when some aggregation leads to further aggregation. It bears the utmost significance as it 'glues' together aggregations at multiple levels and pushes the whole SVT-processing up the XML schema tree finally to its root. So propagation is considered 'greedy' since the propagation

condition is again verified at the end of a propagation and, if satisfied, it is performed repeatedly; the process persists until the propagation condition fails to hold, which is now relaxed to:

- The parent vertex with its SVT populated or $\#SVT$ determined as a consequence of aggregation (*or propagation*) again is the last (or only) child of its parent.

Example 4.2.3. *The XML tree from the previous example also demonstrates propagation being 'greedy'. Figure 4.9 shows how a series of propagations boost the processing up eventually to the root R.*

4.3 The Algorithm SVT-Trans

The algorithm SVT-Trans is introduced in full detail here. Figure 4.10 firstly presents an overview of SVT-Trans covering its structure and all methods which we will have a closer look at in turn.

4.3.1 SVT-Trans Overview

Given an XML document X, its data tree T_D[4] with root r, its schema tree T with root R and the entire leaf set L_T as a result of applying the EXE algorithm, SVT-Trans is composed of two primary methods, $setSVT(v)$ and $populateSVT(v, \nu)$ with both utilising recursion by essentially starting with R(and r):

- $setSVT(v)$

  This method corresponding to the *initialisation* process determines and establishes a SVT_v for every schema vertex $v \in T$. It then recursively calls itself for *every* child of v in sequence until reaching a leaf ℓ, at which $\#SVT_\ell$ is set to $N_\ell \in Names$. Aggregation or propagation takes place whenever their condition is satisfied. $setSVT(v)$ terminates when $\#SVT_v$ is eventually settled(Figure 4.11 & Figure 4.12).

[4] T_D can be conveniently acquired with the help of an XML parser. Detail is discussed at an appropriate later point.

Input: T, R, T_D, r, L_T
Output: the SVT_R fully populated with all semantic data from X
/* Establish all SVTs */
(1) $setSVT(R)$;
/* Set up all SVTs */
(2) Create all SVTs using Database Definition Language(DDL);
/* Populate SVTs */
(3) $populateSVT(R, r)$;

Figure 4.10: Algorithm: SVT-Trans

- $populateSVT(v, \nu)$
  This method corresponding to the *population* process retrieves and transforms the semantic data bound in X into the SVT_R to be used in the XFD discovery analysis later. It recursively calls itself for v's *every* child c, together with *every* almost copy of c in T_D, represented by an instance vertex c_i, i.e. for (c, c_i) in sequence until reaching the leaf level (ℓ, l), at which SVT_ℓ is packed with the textual data attached to l. Aggregation or propagation takes place whenever their condition is satisfied. $populateSVT(v, \nu)$ terminates when SVT_v is eventually populated.
  A schema vertex v and its every instance vertex $\nu \in T_D$ is paired up as the parameters here since SVT_v need to convene the semantic information held in every almost copy of v-subtree, i.e. the corresponding ν-subtree in T_D. Hence it is essential to keep track of the schema-instance vertex correspondence(Figure 4.13, Figure 4.14 & Figure 4.15).

As seen from Figure 4.16, recursion plays a significant role in both methods, partially empowered by XML schema tree and data tree, both a rooted tree. With the root as the starting point, recursion pushes the processing down and then up the tree to determine and populate SVTs — a 'top-down-up' fashion.

Compactness and high reusability is another feature of SVT-Trans attributed to the innovative concept of SVT; an SVT_v corresponds to the v-subtree in an XML schema tree and deals with *all* its almost copies in a compatible XML data tree. This also explains why all previously populated SVTs need to be evacuated for future use, i.e. for other

```
Input: T, T_D, R, v ∈ T
Output: #SVT_v
        /* Base Case */
  (1)   if v ∈ L_T then                    //v is a leaf
  (2)     set #SVT_v = N_ℓ;                //| #SVT | = 1
  (3)     check against aggregation condition;
  (4)     if satisfied then
  (5)       aggregateSetSVT(v);
  (6)     endif
        /* Recursive Case */
  (7)   else do                            //v is an internal vertex
  (8)     for each child c of v do
  (9)       setSVT(c);                     //recursion
 (10)     endfor
 (11)   endif
```

Figure 4.11: Method: setSVT(v)

```
Input: T, R, T_D, r, L_T
Output: the SVT_R fully populated with all semantic data from X
  (1)  Union #SVT_v (and all its sibling #SVTs) into #SVT_p;  //p is v's parent
  (2)  Check against propagation condition;
  (3)  if satisfied then
  (4)    aggregateSetSVT(p);                                  //recursion
  (5)  endif
```

Figure 4.12: Method: aggregateSetSVT(v)

almost copies to be processed, after the population of their parent SVT as an outcome of aggregation or propagation.

```
Input: T, T_D, R, v ∈ T, ν ∈ T_D of type v
Output: Populated SVT_v
      /* Base Case */
 (1)  if v ∈ L_T then                    //v is a leaf
 (2)    insert val(ℓ) into SVT_v;
 (3)    check against aggregation condition;
 (4)    if satisfied then
 (5)      aggregatePopSVT(v, ν);
 (6)    endif
      /* Recursive Case */
 (7)  else do                            //v is an internal vertex
 (8)    for each child c of v do
 (9)      check for all c_D ∈ T_D of type c attached directly to ν;
(10)      if any c_D then
(11)        for each c_D do
(12)          populateSVT(c, c_D);   //recursion
(13)        endfor
(14)      else                          //c is not presented under ν ∈ T_D
(15)        populate SVT_c with NULL;
(16)        check against aggregation conditions;
(17)        if satisfied then
(18)          aggregatePopNullSVT(c, ν);
(19)        endif
(20)      endif
(21)    endfor
(22)  endif
```

Figure 4.13: Method: populateSVT(v, ν)

4.3.2 Handling NULL Values

As a highly functional subset of SGML, the popularity XML enjoys is to a large extent ascribed to its great structural flexibility. Rather frequently, optional elements are encountered, which are defined in an XML schema but may not always be presented in a

(1) Join SVT_v (with all its sibling SVTs) into SVT_p; //p is the parent of v
(2) Empty SVT_x for every $x \in p$-subtree and $x \neq p$;
(3) Check against propagation condition;
(4) **if** satisfied **then**
(5) **for** v's parent $p_D \in T_D$ **do**
(6) $aggregatePopSVT(p, p_D)$; //recursion
(7) **endfor**
(8) **endif**

Figure 4.14: Method: aggregatePopSVT(v, ν)

(1) Join SVT_c (with all its sibling SVTs) into SVT_v; //v is the parent of c
(2) Empty SVT_x for every $x \in v$-subtree and $x \neq v$;
(3) Check against propagation condition;
(4) **if** satisfied **then**
(5) $aggregatePopSVT(v, \nu)$;
(6) **endif**

Figure 4.15: Method: aggregatePopNullSVT(c, ν)

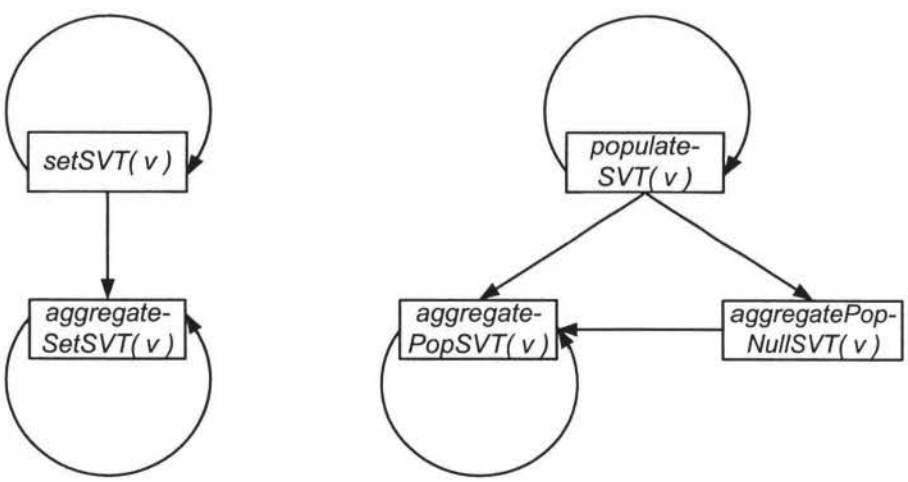

Figure 4.16: SVT-Trans Function Call Diagram

compatible XML instance. One may assign to such vertex disparate frequency constraints, such as '+' and '*', apart from the default 'exactly one', to suit specific needs. Let's take 'student' and its child 'qualification' for example. A student s may have attained some qualifications, or not at all[5]. In the latter case, there will be no qualification information for s. As far as our work is concerned, $SVT_{qualification}$ will reflect the information absence by placing a NULL value for s, which method $aggregatePopNullSVT()$ in SVT-Trans takes care of.

It is also essential to take NULL values into account when interpreting the previously introduced aggregation and propagation conditions for the SVT initialisation and population processes since special care is needed for processing data trees where element optionality makes a difference. Then the phrase '...the last (or only) child...' in aggregation and propagation conditions should be interpreted accordingly as '...the last (or only) *presented* child...'

Example 4.3.1. *Figure 4.17 includes a schema tree T and a compliant data tree T_D. w in T_D does not possess any children of type Z as W does in T. Hence, on the absence of Z, $y2$ becomes the last presented child.*

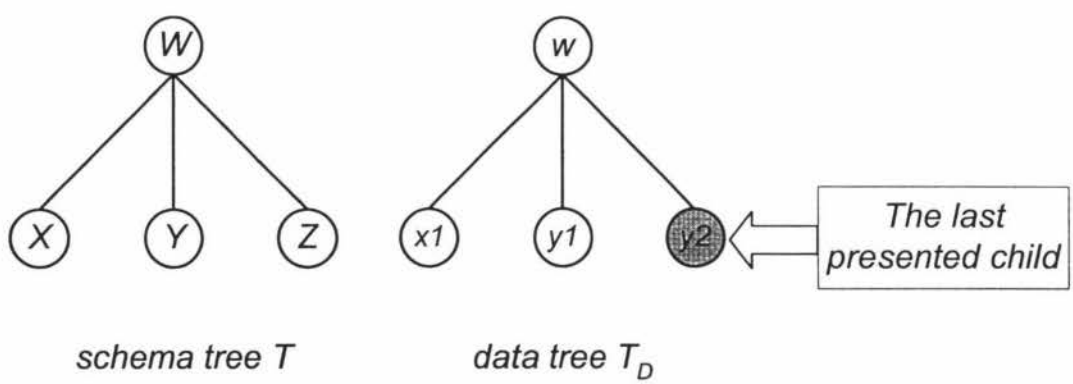

Figure 4.17: Example on the *last presented* child

[5]An similar scenario is that qualification data are known to exist but with unknown values.

Chapter 5

Prototype System

This section unfolds *XFD-Miner*, a prototype system implemented in Java by virtue of the proposed framework as described in Chapter 2 and 3; it demonstrates the correctness and soundness of our research. An overview of XFD-Miner is supplied in Section 5.1. Section 5.2 illustrates three functional dependency inference algorithms and some design decisions are justified in Section 5.3.

5.1 System Overview

5.1.1 Functionality

XFD-Miner is conceived to comply with the EXE and SVT-Trans algorithms illustrated previously and fulfils the following tasks with high automation and usability:

- XML Schema Generation

  The ER-XML algorithm is applied to extract the ER-XML model information from a user specified XML document X.

- Schema Visualisation

  The ER-XML model is then rendered in a graphical representation.

- Data Transformation

This is the core function; the extracted ER-XML model with root r is applied back to the XML document itself, the semantic data embedded in the XML document is then retrieved in terms of almost copies and transformed into SVT_r using SVT-Trans.

- XFD Discovery

  XFD-Miner provides users with options in discovering XFD satisfied by SVT_r; it realises three FD discovery algorithms in disparate approaches to be discussed subsequently.

The use case diagram follows:

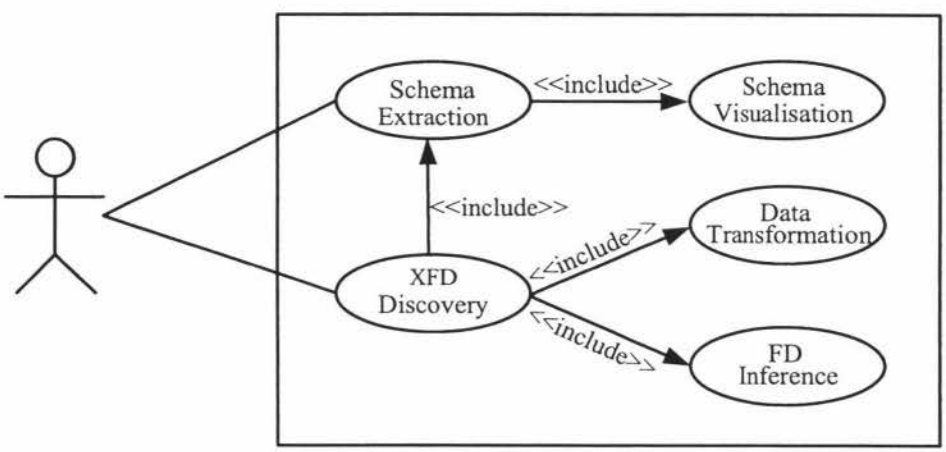

Figure 5.1: XFD-Miner Use Case Diagram

Furthermore, execution of all algorithms is timed to enable time-wise efficiency comparisons among them.

5.1.2 System Architecture

The entire XFD mining process comprises the following steps as illustrated in Figure 5.2:

1. Generate a DOM tree for user specified XML Document X with the help of DOM Parser;

2. Extract XML schema from the DOM Tree using the ER-XML algorithm;

3. Display the graphical representation of the ER-XML to the user;
4. Run SVT-Trans to perform data transformation;
5. Compute XFDs according to the user's selection of FD inference algorithm and then render the result.

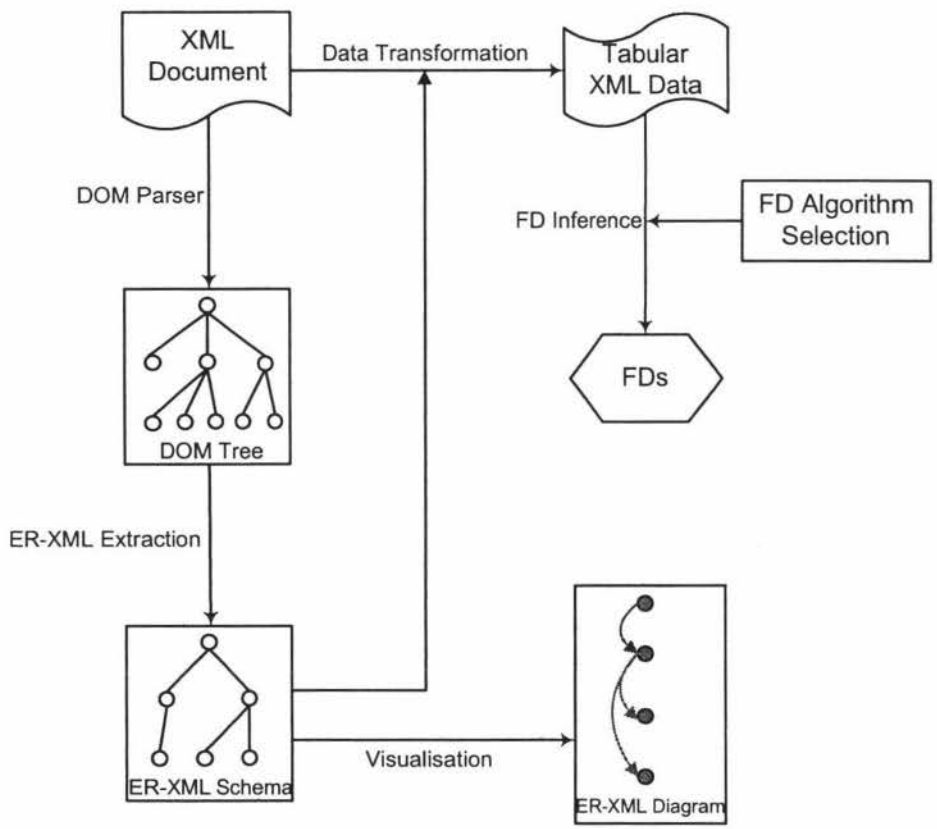

Figure 5.2: System Architecture

5.2 Functional Dependency Inference Algorithms

Functional dependency inference is the task of computing the functional dependency constraints that hold in an existing database or relation. Research on this regard has been long going on with some full-grown algorithms already in existence. We have carefully selected and successfully implemented in XFD-Miner some outstanding ones to compute functional dependencies held in the root SVT after the data transformation process, providing the user with the subsequent options:

- A naive algorithm;
- An algorithm using transversal approach;
- FastFDs.

NULL Values

As explained before, null values do occur even frequently although it is not mentioned in many FD discovery algorithms. We face and act on this issue; special care has been taken to amend the original algorithms in our implementation to handle NULL values.

5.2.1 The Naive Algorithm

Naive computation [Mannila and Räihä, 1992] is to check all possible dependencies(Figure 5.3).

Input: A relation r over R
Output: F, the set of FDs holding in r
(1) $F := \emptyset$;
(2) **for each** subset $X \subseteq R$ **do**
(3) **for each** attribute $A \in R \setminus X$ **do**
(4) **if** $r \models X \rightarrow A$ **then**
(5) $F := F \cup \{X \rightarrow A\}$;
(6) **endif**
(7) **endfor**
(8) **endfor**

Figure 5.3: The naive algorithm

Verification of $r \models X \rightarrow A$ does *not* consider the case in which NULL values appear in X, i.e. $val(Y) =$ NULL for some $Y \subseteq X$.

Input: A hypergraph $\mathcal{H}$ on a set R

Output: The edges of the hypergraph, $Tr(\mathcal{H})$

(1) $Tr := \{\emptyset\}$;

(2) **for each** $E \in \mathcal{H}$ **do**

(3) $\quad Tr := \{X \cup \{e\} \mid X \in Tr \wedge e \in E\}$;

(4) $\quad Tr := \{X \in Tr \mid$ there is no set $Y \in Tr$ with $Y \subset X\}$;

(5) **endfor**

Figure 5.4: $Tr(\mathcal{H})$ Computation

5.2.2 The Enhanced Algorithm Using Transversals

The naive algorithm is rather simple; it suffers from the drawback of exponential computation time no matter how regular the relation r is. The next algorithm tries to be more adaptive to the input taking advantage of *transversal*, a concept defined on top of *hypergraph*.

Given a relation schema R, a *hypergraph* on R is a collection $\mathcal{H}$ of subsets of R such that $\emptyset \in \mathcal{H}$ and if $X, Y \in H$ and $X \subseteq Y$ imply $X = Y$. In particular, the elements of $\mathcal{H}$ are the *edges* of the hypergraph. A transversal T of $\mathcal{H}$ is a subset of R intersecting all the edges of $\mathcal{H}$, i.e. $T \cap E \neq \emptyset$ for all $E \in \mathcal{H}$. The transversal computation algorithm is shown in Figure 5.5.

It is now about time to unveil the algorithm using the transversal approach (enhanced, originally from [Mannila and Räihä, 1992]):

Taking NULL value comparison into consideration, we have defined two versions of equality:

- $=_L$ — *left-hand-side equality* where NULL $\neq_L$ NULL
- $=_R$ — *right-hand-side equality* where NULL $=_R$ NULL

Null values are not regarded as equal in the left-hand-side(LHS) equality, whereas in the right-hand-side(RHS) equality they are. For instance, at step 2) in the above algorithm,

Input: A relation r over R

Output: F, the set of FDs holding in r

(1) **for each** attribute $A \in R$ **do**

(2) $J_A := \{R \setminus ag_L(t,t') \mid t,t' \in r \text{ and } t[A] \neq_R t'[A]\}$;

(3) $K_A := \{W \in J_A \mid \text{there is no } V \in J_A \text{ such that } V \subset W\}$;

/* now $K_A = cmax(A)$, for each $A \in R$ */

(4) $L_A := Tr(K_A)$;

(5) **endfor**

(6) $F := \{X \rightarrow B \mid B \in R, X \in L_B\}$;

Figure 5.5: The algorithm using transversals

$t[A], t'[A]$ are treated as equal if $t[A], t'[A]$ both are NULL during the pairwise tuple selection $t[A] \neq_R t'[A]$, nevertheless the agree set computation $ag_L(t,t')$ will not include B provided $t[B], t'[B]$ both are NULL($t[B] \neq_L t'[B]$).

5.2.3 The Enhanced Algorithm FastFDs

Based on [Mannila and Räihä, 1987] [Mannila and Räihä, 1994], [Wyss et al., 2001] put forward FastFDs, a heuristic-driven, depth-first search to infer FDs from a relation instance, with improved performance[1]. Presented below is an enhanced version of FastFDs with one flaw[2] remedied and NULL value handling mechanism added, consisting of *genDiffSets*() and *findCover*():

Follows the enhanced FastFDs amalgamating the above two methods:

[1]This judgement is claimed by the authors based on their experiments.

[2]Original FastFDs may yield more FDs with $\emptyset$ at the left hand side than it should.

Input: A relation instance r over schema R

Output: Difference sets for $r, \mathcal{D}_r$

```
    //Initialise:
(1) resDS := ∅;
(2) strips := ∅;
(3) tmpAS := ∅;
(4) oneVal := ∅;
    //Compute stripped partitions for all attributes:
(5) for each A ∈ R do
(6)   Compute stripped partitions for A except for val(A) = NULL
      and add to strips;
(7)   if (| π_A |= 1) ∧ (| ⟨t_A⟩ |=| t |) then
(8)     Add A to oneVal;
(9)   endif
(10) endfor
    //Compute agree sets(using LHS equality) from stripped partitions:
(11) for each π ∈ strips do
(12)   for each t_i ∈ π do
(13)     for each t_j ∈ π, j > i do
(14)       Add A_L(t_i, t_j) to tmpAS;
(15)     endfor
(16)   endfor
(17) endfor
(18) for each X ∈ tmpAS do
(19)   Add R − X to resDS;
(20) endfor
```

Figure 5.6: Method: genDiffSets()(enhanced version)

```
Input: Attribute A ∈ R (RHS)
       Original difference sets, D_r^A
       Difference sets not thus far covered, D_curr
       The current path in the search tree, X ⊆ R
       The current partial ordering of the attributes, >_curr
Output: Minimal FDs of the form Y → A
       Base Cases:
(1)    if >_curr= ∅ and D_curr ≠ ∅ then
(2)      Return;          //no FDs here
(3)    endif
(4)    if D_curr = ∅ then
(5)      if no subset of size | X | −1 of X covers D_r^A then
(6)        Output X → A and return;
(7)      else do
(8)        Return;        //wasted effort, non-minimal result
(9)      endif
(10)   endif
       Recursive Case:
(11)   for each attribute B ∈ >_curr in order do
(12)     D_next := difference sets of D_curr not covered by B;
(13)     >_next is the total ordering of {B̂ ∈ R | B̂ >_curr B} according to D_next;
(14)     findCovers(A, D_r^A, D_next, X ∪ {B}, >_next);
(15)   endfor
```

Figure 5.7: Method: findCovers()(enhanced version)

5.3 Implementation Considerations

5.3.1 Programming language — Java

Java, a highly modular, object-oriented language, was selected to implement the system, due to its superior portability, reliability, robustness and security. Java has gained enormous popularity since it first appeared. Its rapid ascension and wide acceptance can be

Input: A relation r over R
Output: F, the set of FDs holding in r
(1) $\mathcal{D}_r$:= genDiffSets(R, r);
(2) **for each** attribute $A \in R$ **do**
(3) Compute $\underline{\mathcal{D}_r^A}$ from D_r;
(4) **if** $\underline{\mathcal{D}_r^A} = \emptyset$ **then**
(5) **if** $A \in oneVal$ **then**
(6) Output $\emptyset \rightarrow A$;
(7) **else do**
(8) Output $B \rightarrow A$ for all $B \in R \setminus \{A\}$;
(9) **endif**
else if $\emptyset \notin \underline{\mathcal{D}_r^A}$ **then**
(10) $>_{init}$ is the total ordering of $R \setminus \{A\}$ according to $\underline{\mathcal{D}_r^A}$;
(11) findCovers$(A, \underline{\mathcal{D}_r^A}, \underline{\mathcal{D}_r^A}, \emptyset, >_{init})$;
(12) **endif**
(13) **endfor**

Figure 5.8: Algorithm: FastFDs(enhanced version)

traced to its design and programming features, particularly in its promise of platform independence ('program once, run anywhere'™).

The Java platform shares many key synergies with the XML standard:

- The Java platform's portable code capability has been invaluable in fostering a collaborative environment for XML; Java promises "portable code" and XML assures "portable data". These two technologies can be used together to architect enterprise application that work cross-platform. XML applications written in Java can be reused on any tier in a multi-tiered client/server environment, offering an added level of reuse for XML documents. The same cannot be said of scripting environments or platform-specific binary executables.

- Java's object-oriented feature legitimately benefits the XML specification of a hierarchical representation of data.

- The Java technology binding to the W3C Document Object Model provides a highly

productive environment for processing and querying XML documents. The Java platform can become a ubiquitous runtime environment for processing XML documents.

The suitability of Java and XML with the intrinsic features of platform and application neutralisation explains their prevalent combination as a universal and portable solution. Sun Microsystems, Inc. provides core Java technologies for XML: the Java API for XML Parsing (JAXP), the Java API for XML Messaging (JAXM) and the Java API for XML Data Binding (JAXB). This powerful, yet easy-to-use tool set supports XML data manipulation. It can therefore be anticipated that Java will maintain its leading position in programming for XML in the future.

5.3.2 Data Definition and Manipulation Language — MySQL

MySQL, the most popular Open Source SQL database management system, is developed, distributed, and supported by MySQL AB[3] and has now grown up to a well-respected product capable of more than commercial operation. It supports most of the functionality that can be expected from a commercial RDBMS, such as transactions complying with the ACID Model, indexes, standard data type. Besides, MySQL is lightweight and can run rather fast. Its cross-platform compatibility and conditional free distribution also contribute to its ubiquity. It also provides a set of programming libraries including for Java. In brief, MySQL was evaluated highly suitable to store and manage XML data for XFD-Miner considering the nature of the research and features of MySQL. This has been verified by its successful fulfilment of expectations in practice.

5.3.3 XML Parser — DOM

Parsing an XML document is the essential process of interpreting and capturing its structural information. There are currently two standards for XML parsing APIs with varied purposes: Document Object Model (DOM) developed by the W3C consortium, and an unofficial standard called the Simple API for XML (SAX). A DOM parser treats XML

[3]MySQL AB is a commercial company, founded by the MySQL developers. More details available on http://www.mysql.com.

documents as a tree-structure of objects/nodes (mapped directly onto Java Objects), while a SAX parser handles XML data as a flat stream of elements ('on the fly'), requiring extra effort to keep track of their interrelated relationship [Geroimenko and Geroimenko, 2001]. *Overall Judgement*: a DOM parser is more appropriate since it preserves the original XML tree structure and usually gives better functionality, flexibility and usability and therefore outperforms a SAX parser as far as our requirements are concerned.

DOM Level 3(DOM3) released by W3C in 2004 as a full-fledged recommendation is finally employed in XFD-Miner. Among a number of new features, DOM3 has been tuned to empower additional common and useful functions, in particular its support for the *DOMUserData* type to store application data, which its previous versions lack. This can be utilised to attach application specific information to a DOM node, referred to as the user data system. If a developer wants to annotate a document with non-XML information, the user data mechanism may also be used. This turns out to be very advantageous in our case because of the fact that instance vertices at distinct levels in the DOM data tree of an XML document may well belong to the same schema vertex and it is thus essential to associate with each instance vertex (DOM node) its path information (from the root) for the purpose of differentiation. A new class *MyNode* was consequently created based on the original *Node* class after parsing to capture the addition path information.

5.3.4 Data Structure

Vector

In Java programming practice, vectors (the *java.util.Vector* class) are commonly used instead of arrays, because of their dynamic growth in comparison with the fixed length of arrays. The Vector class also enjoys high usability, for example its implementation of the Enumeration interface makes traversing the contents of a Vector extremely easy. Moreover, natural extensibility of XML fits and demands the compatible features offered by the Vector class.

The data structure behind ER-XML algorithm takes advantage of the Vector class and is composed of 6 vectors in total to store schematic information:

nodeList() Stores all parent node (base element) names and their child node (non-base element) names in sequential order. A base node is immediately followed by ALL its children.

isBase() Indicates whether or not a corresponding node is a parent node.

level() Indicates the level (in DOM tree) of each element in *nodeList*().

minCount() Keeps track of the minimal occurrence of a certain non-base element.

maxCount() Keeps track of the maximum occurrence of a certain non-base element.

attrList() Stores names of all element nodes with at least one attribute node in sequential order; such an element is immediately followed by ALL its attribute/s.

Their correspondence is better illustrated in Figure 5.9:

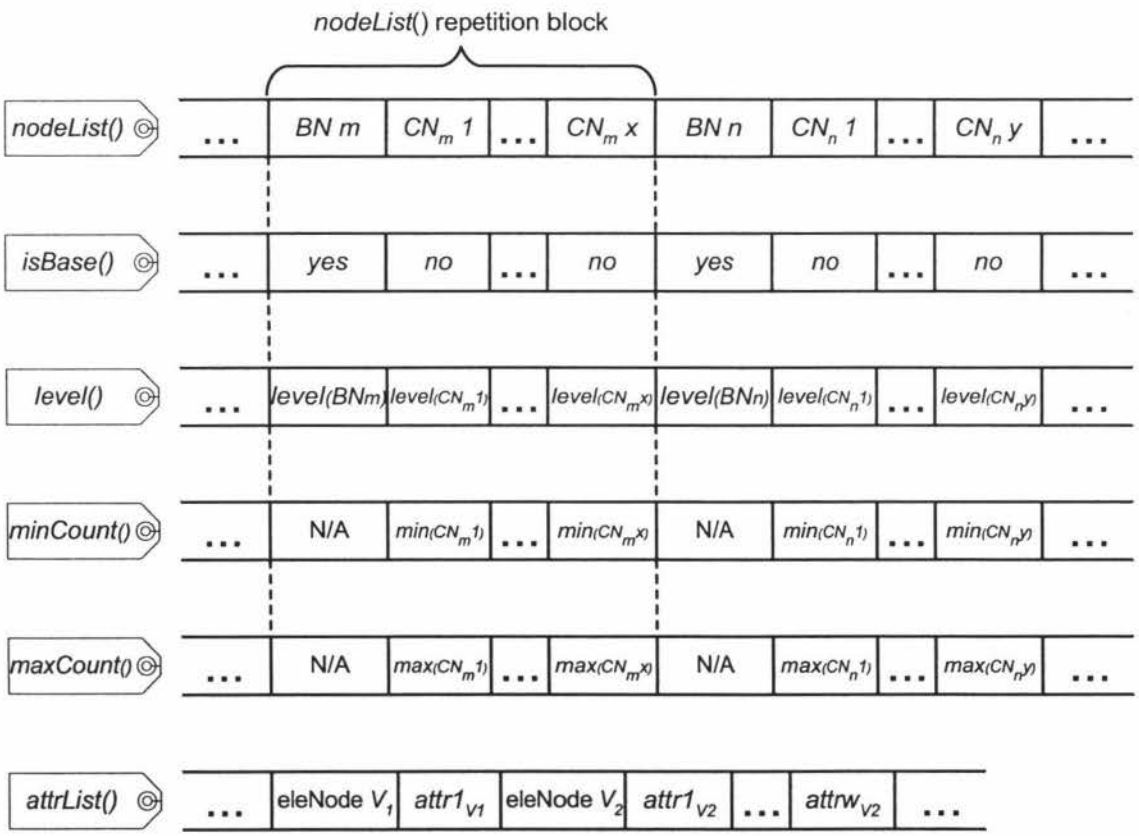

Figure 5.9: Vector structure

Clearly, there is no coupling between *attrList*() and the other 5 vectors as within them.

LinkedHashMap

While vectors form the static structural information base, LinkedHashMap creates a linked-list structure by chaining all these together. As a fresh addition to the Collections API in JDK 1.4, LinkedHashMap (the *java.util.LinkedHashMap* class) is a Map implementation based on *HashMap*. It differs in that it also maintains a linked list running through the map entries, and so can iterate through those entries in a predictable order. Usually, this order is the order in which entries are inserted into the map. Recalling the data transformation algorithm SVT-Trans, the order maintenance feature becomes valuable in keeping track of the order among sibling vertices, especially the *last* one which normally serves as the catalyst for further action.

MyNode class briefly mentioned before employs LinkedHashMap to reflect the XML parent-child relationship between *MyNode* objects; it includes a LinkedHashMap instance *childMap* to accommodate all its children also of type *MyNode*. Starting from the root, *MyNode* is instantiated recursively — from parent to child, and terminates at a leaf which does not possess any child to proceed with. After every instantiation process(except for the root), the new *myNode* object is linked back to its parent — it is inserted into the *childMap* of its parent.

Example 5.3.1. *Figure 5.10 shows how LinkedHashMap is utilised in* MyNode *class by way of an example of a* MyNode M *and its two children* $N1$ *and* $N2$ *also of type* MyNode. *Suppose the universal name for* M *is* UN_{M}.

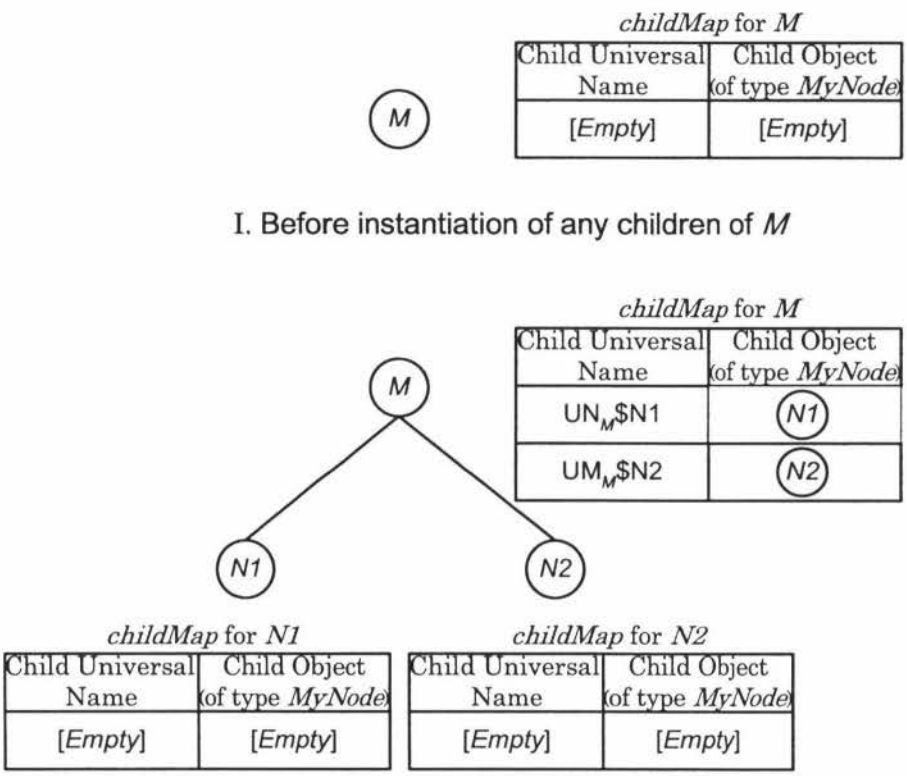

Figure 5.10: LinkedHashMap usage in *MyNode* class

Chapter 6

Case Study

As a tangible achievement of our work, XFD-Miner is a mature application framework for inferring XFDs from an independent XML document. This case study illustrates in detail how XFD-Miner performs on an XML instance and demonstrates its outcome. Finally, some scalability performance testing is carried out to estimate its resource consumption for XML documents with diverse sizes.

Good'N'Go is a specialised commodity warehousing organisation in New Zealand. With headquarters based in Auckland, it has storage establishments nationwide, such as Wellington and Christchurch. Their information system has been logging the stock storage information in XML format. Now the management stakeholder would like some insights into the functional dependencies (business rules) implied in these records. A sample file *warehouse.xml* is supplied in Appendix B for this purpose.

6.1 Walking Through XFD-Miner

Step 1: Run XFD-Miner

Figure 6.1 shows the initial GUI after the application is executed. It is composed of the top pane for choosing the source XML file and two parallel panels for visualising its schema graph and displaying the functional dependency output respectively.

Step 2: Locate the source XML document

After clicking the 'Browse' button for the source XML document in the top pane, a file

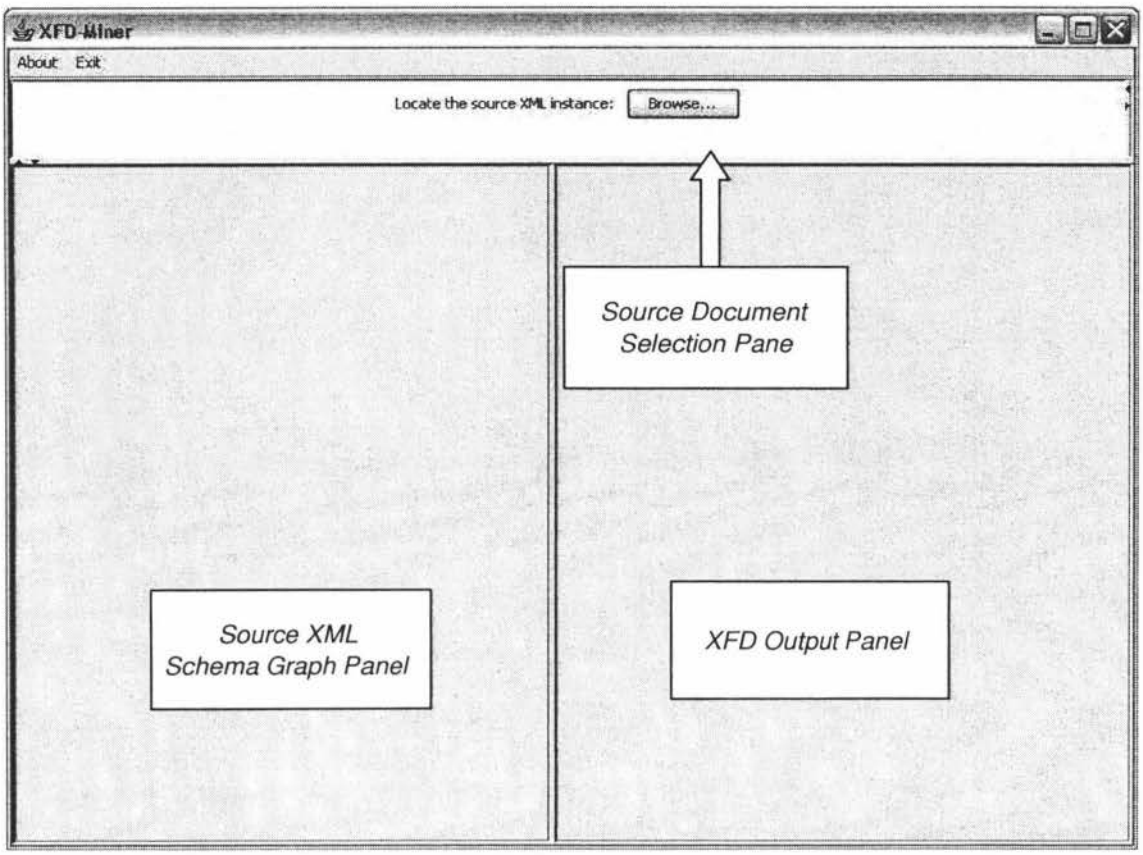

Figure 6.1: XFD-Miner Portal Interface

chooser window pops up for the user to navigate and locate the source XML document, which is carefully designed to allow only XML files to be selected (.xml).

Step 3: Display source XML schema graph

The schema graph of the source XML document is subsequently rendered, in which varied colours are used for high clarity. With the assumption that the source XML document does not possess a DTD (neither do most XML files in practice, where also partially comes our motivation), the DOM parser employed in the system is configured to switch off schema validation against the DOCTYPE declaration, if any; otherwise it would throw out an *IOException* error(Figure 6.3).

One may have already noticed that the algorithm selection pane now appears. The user is now eligible to make the choice out of the algorithm list to be used in discovering functional dependencies.

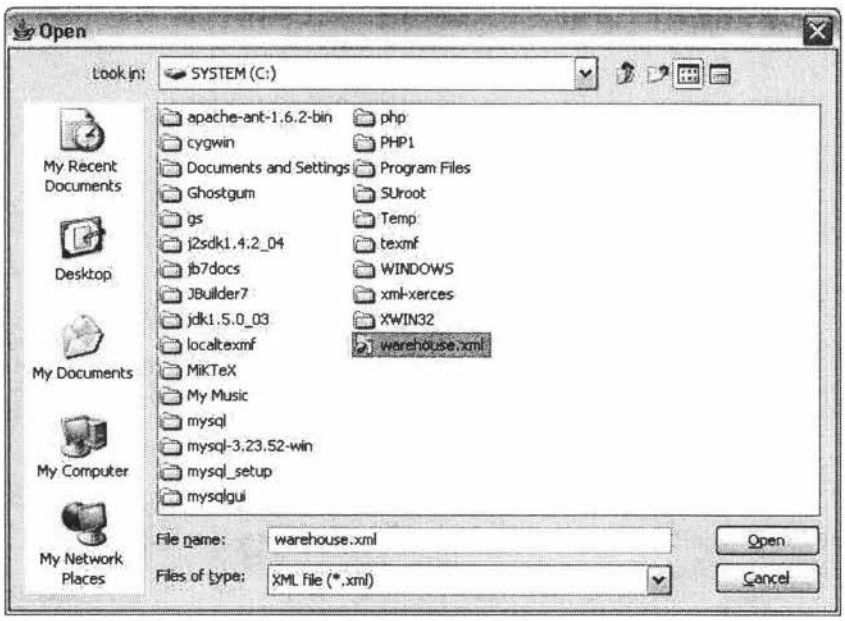

Figure 6.2: XFD-Miner File Chooser Window

Step 4: Make the algorithm selection

Here it is as simple as selecting an algorithm and then hitting the 'Run' button.

Step 5: Exhibit the final output

XFD-Miner automatically renders the entire set of functional dependencies satisfied by the source document and other benchmark information, for instance time cost on data transformation and on function dependency inference, number of almost copies and functional dependencies, as illustrated in Figure 6.4:

6.2 Performance Testing

XFD-Miner is designed to deliver relatively high level of performance for XFD discovery. However, as is true for most applications, its performance is also influenced by other factors such as XML data, JVM options specified at runtime. This section focuses on the results of running XFD-Miner against XML documents of varied size, with the intention of providing their respective time consumption estimates. The performance test was run

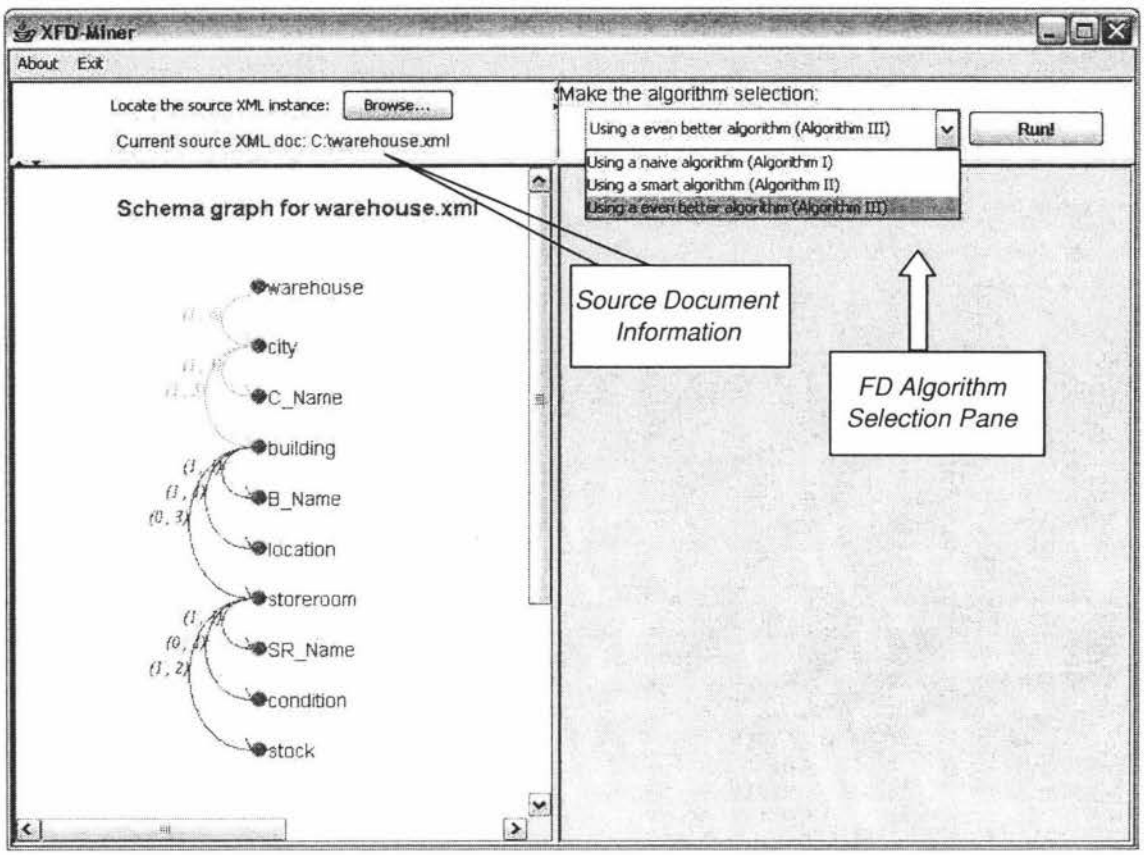

Figure 6.3: XFD-Miner GUI with the source XML schema graph

on the following hardware settings:

Processor: Sun Ultra UPA/PCI (4 X UltraSPARC-II 450 Mhz)
System clock frequency: 113 MHz
Memory size: 4096 Mbytes.

The results are presented in Table 6.1 while their graphic representations, Figure 6.5 $\sim$ 6.7, are more revealing:

As evident from those figures, we have observed:

- Time consumption in operating XFD-Miner is directly proportional to the number of almost copies in the XML document, rather than its size. It also confirms to what we have expected due to the fact that the concept of 'almost copy' is the rationale to *SVT-Trans*.

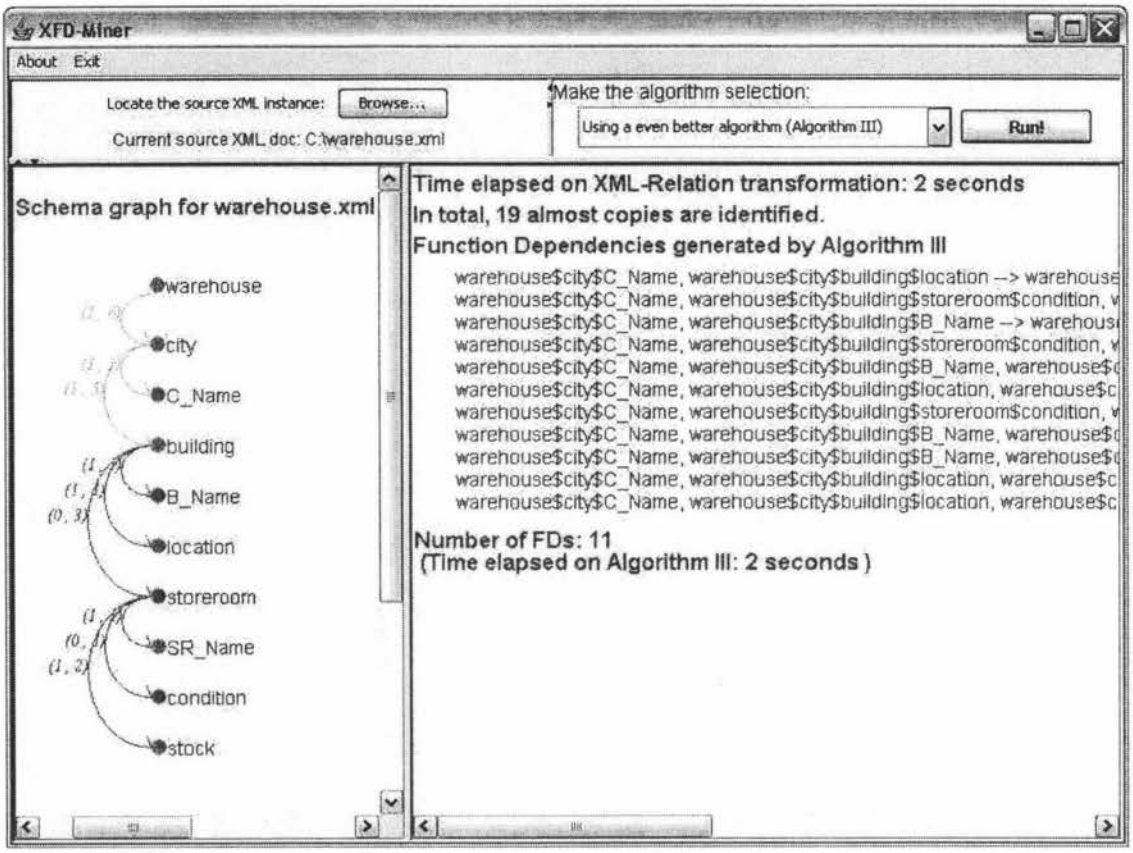

Figure 6.4: XFD-Miner Final Output

Table 6.1: Testing Results

XML File Size(KB)	*No. of Almost Copies*	*Time Consumption of FD Algorithms(min.)*			*Time Consumption of XFD-Miner(sec.)*
		Algorithm I	*Algorithm II*	*Algorithm II*	
5	52	0.68	0.02	0.02	9
10	660	10.6	1.02	0.47	16
25	3200	301.17	30.82	9.67	14
50	8704	397.17	157.63	63.9	30
75	10624	515.77	319.62	98.8	42
100	10912	870.42	340.7	105.93	46
150	19744	6027.7	2370.32	657.45	69
200	9400	551.32	267.55	85.25	61

- *SVT-Trans* has demonstrated an overall satisfactory and stable processing time; it takes only about 1 minute to transform an XML input of size as large as 200KB with nearly 20,000 almost copies.

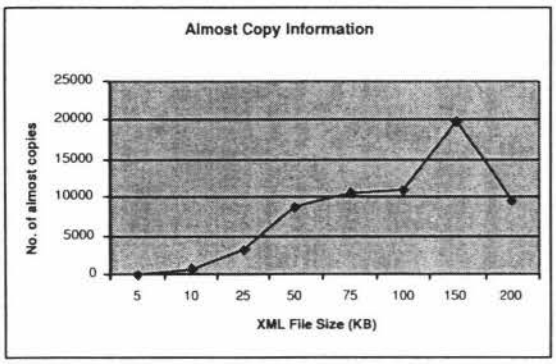

Figure 6.5: Almost Copy

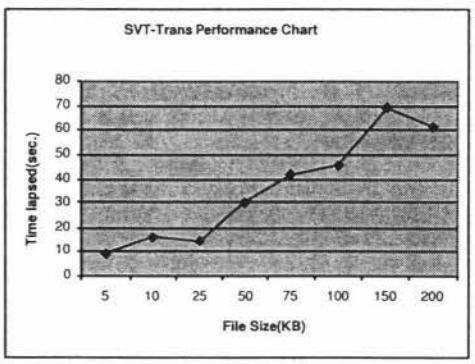

Figure 6.6: SVT-Trans Performance

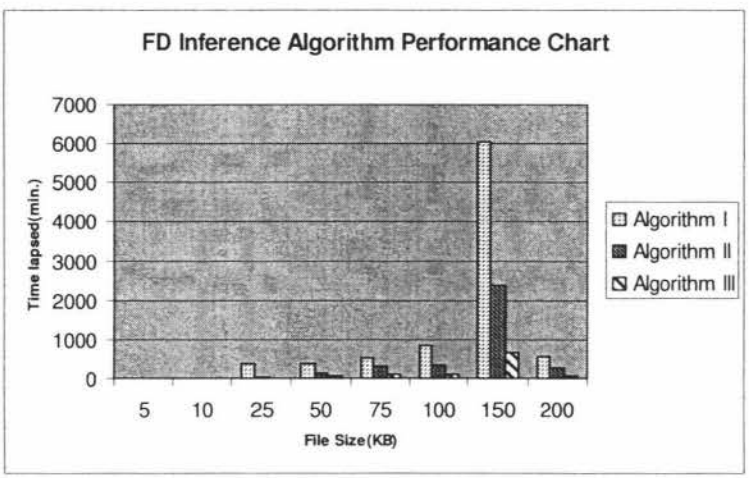

Figure 6.7: FD Algorithm Performance Chart

- Those 3 implemented algorithms have exhibited a substantial efficiency variance; algorithm III outperforms the other two whilst algorithm I takes the longest to complete, which can be explained by Section 5.2.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In contrast with long standing relational database technology, eXtensible Markup Language (XML) is rapidly emerging as the dominant standard for representing data on the Internet. Along with its popularity, research attention has been drawn on XML normalisation, often complicated by schema information absence problem. No satisfactory achievements have yet come into being, albeit some peer work has been going on in XML Functional Dependency(XFD) discovery.

In the light of the 'divide-and-conquer' approach suggested by our preliminary studies, we firstly target the schema scarcity problem in XML practice caused by the fact that XML schematic information is essential, yet unfortunately often unavailable. As a suitable solution of practical significance, a powerful back-end data representation model with richer semantics, ER-XML, is devised together with a corresponding schema extraction algorithm EXE, based on which schema visualisation is also realised.

With the help of such schema information, subsequent conversion of XML data into a relational structure constitutes the essence of this thesis. We have conceived a novel concept of Schema Vertex Table and contrived a data transformation algorithm known as SVT-Trans to extract entire XML semantic data into a single relation powered by recursion. This innovative notion is free from the awkwardness confronted by most related work on XML-RDB mapping caused by their unmanageably large number of relations (many functional dependencies may then span quite a few relations).

Finally, three existing relational functional dependency inference algorithms are carefully chosen and leveraged to extend our work to the final result.

In addition, a Java-based prototype system XFD-Miner has been successfully developed integrating our entire work[1] to generate a set of functional dependencies satisfied by an XML instance, also the only input, and relevant performance statistical information. We have conducted experimental testing to gauge scalability performance of XFD-Miner as for varied XML document size.

7.2 Future Work

This research is not a panacea to all aspects of the XML functional dependency discovery problem, yet still shed some light on some further research directions:

- Name space and element reference

  Name space has been employed to cluster elements with names of the same semantics arising as electronic data exchange and integration nowadays happens frequently and from heterogeneous, likely distributed, data repositories. ID, IDREF, and IDREFS are also widely used in practice. They however are not of our concern for the time being.

- Efficiency

  Utilisation of DOM parser is a feature of the implementation of the prototype system XFD-Miner. But DOM parser is relatively more expensive to operate in terms of resource consumption, as it builds up the entire tree structure in memory. Further improvement on efficiency is worthy of attention.

- Graphics

  Despite its gratifying performance with XML documents, as adequate justification for our work, XFD-Miner may still be considered plain in terms of graphical representation, such as the arrangement of cardinality information display. Schema graph layout optimisation remains a challenging task.

[1] Also inclusive of the three relational FD inference algorithms, providing the user choice.

- Limitations from implementation tools

  Regardless of its satisfactory fulfilment of expectations and even at no cost, MySQL has also unfurled its capability limits, for instance at most 31 tables can participate in a join operation. Even Java, known as the perfect match for XML technology, also gets its power bounded by configurations of Java Virtual Machine(JVM) and other restrictions from the underlying platform. Although we have explored possible countermeasures[2], they may still become problematic especially for XML documents of excess large size. Endeavours to unleash and exploit their full potential are therefore envisaged to be highly promising and rewarding.

[2]Details are held in the attached technical report.

Appendix A

XFD-Miner Guide

A.1 Environment Configuration

XFD-Miner was developed in Java with the J2SE Development Kit™(JDK) 5.0 embracing MySQL and DOM3 technology.

- Establish JDK 5.0

  JDK of any earlier version than 5.0 does not support DOM3 and thus will lead to performance failure.

  1. Verifying current JDK version

     To find out whether JDK 5.0 or above has already been installed, simply run the following command in MS-DOS environment:

     ```
     C:\>java -version
     ```

     If the answer is lower than '1.5.0'[1], JDK 5.0 shall be installed with steps set out as follows; skip to the MySQL installation portion otherwise.

  2. Installing JDK 5.0

     JDK 5.0 is in free distribution by Sun Microsystems, Inc., available at http://java.sun.com/j2se/1.5.0/download.jsp as of 8 December 2005). It is a self-extracting executable file and installation afterwards is rather evident.

[1]It is also possible that JDK 5.0 has already been installed without proper configuration — see item 3 below

3. Configuring newly installed JDK

   System variable PATH needs to be amended to make the newly installed JDK tool set fully function. This can be done from the command prompt:

   C:\>SET PATH=%path%;*<full path to \bin of JDK home directory>*

Re-check the java version at the end to ensure successful installation.

- Establish MySQL

  XFD-Miner was implemented on top of MySQL v4.0.12, which version (or above) is recommended, though any MySQL previous release may also support. Skip this section if such an appropriate version of MySQL has already been installed; otherwise go to http://dev.mysql.com/downloads/.

  To ensure universal deployment, XFD-Miner adopts anonymous user access which by default may not be accepted by recent release of MySQL, with fortified security measures to thwart unauthorised attempts. Post-installation configuration of user account authorisation hence becomes essential. Make sure the subsequent commands be successfully executed in MySQL environment:

```
mysql> CREATE USER ''@'localhost';
Query OK, 0 rows affected (0.00 sec)
mysql> GRANT ALL ON *.* TO ''@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

A.2 Run XFD-Miner

A.2.1 Installed Directory Tree

XFD-Miner has the directory structure shown below:

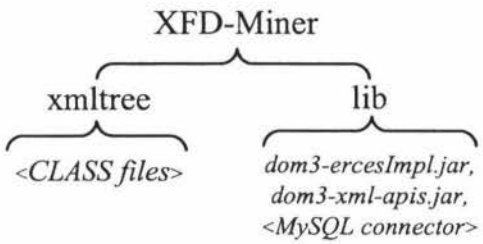

Figure A.1: XFD-Miner Directory Tree

MySQL connector in *lib* needs attention. It is a standard database driver connectivity to enable incorporation between MySQL and Java via JDBC. Since MySQL is version-specific, the user is responsible for an appropriate connector in a JAR file format compatible with the version of MySQL being used. Such a connector will be referred to by a generic expression *<MySQL connector>* in the future.

A.2.2 Execute XFD-Miner

Suppose XFD-Miner is located on D: drive. It should be activated in a command shell using:

```
D:\XFD-Miner>java -classpath .;lib/dom3-xml-apis.jar;lib/dom3-xercesImpl.jar;
lib/<MySQL connector>  xmltree.Door
```

Depending on operating system and Java Virtual Machine (JVM) settings, the above command may not be sufficient for processing XML documents of enormous size[2]. In this case, a variation of the command should be used instead:

```
D:\XFD-Miner>java -Xmx1200M -Xoss20M -Xss20M -classpath
.;lib/dom3-xml-apis.jar;lib/dom3-xercesImpl.jar;
lib/<MySQL connector>  xmltree.Door
```

It tweaks maximum Java heap size(-Xmx), Java stack size(-Xoss) and native stack size (-Xss). But be aware that these parameters set in the sample command are for illustration purpose only and thus may well not apply; the user shall scrutinise and, if necessary,

[2]Some unexpected runtime errors may be thrown out, for example, `Exception in thread "main" java.lang.OutOfMemoryError: Java heap space` and `java.lang.StackOverflowError`.

redefine taking account of the operating system settings.

A.3 Miscellaneous

DOM parser does not tolerate any invalid XML documents, i.e. not well-formed. Moreover, it is also particular about the header — a header is optional, however once presented must be in a correct format; an incorrect XML header will lead to '*not well-formed*' error during the parsing phase. It is therefore a safe practice to remove its entire header before processing an XML document with XFD-Miner.

Certain values have been reserved for application use as illustrated in Table A.1:

Table A.1: System reservations

Reservation	*Situation*	*Purpose*	*How to avoid conflicts*
xfdminertree	name of a database working area in MySQL	allocated for XFD-Miner use only	make sure no existing databases in MySQL share the same name
NULL	in SVT_T after data transformation	indicate NULL values	No textual values in the XML file shall be equal to '*NULL*' literally

Appendix B

Sample XML document: *warehouse.xml*

```
<warehouse>
    <city>
        <C_Name>Auckland</C_Name>
        <building>
            <B_Name>BP House</B_Name>
            <location>20 Customhouse Quay</location>
            <storeroom>
                <SR_Name>Pederson</SR_Name>
                <condition>air isolation</condition>
                <stock>Eta</stock>
            </storeroom>
            <storeroom>
                <SR_Name>Wilkinson</SR_Name>
                <condition>air isolation</condition>
                <stock>Hafele</stock>
            </storeroom>
            <storeroom>
                <SR_Name>Greenfield</SR_Name>
                <condition>warm</condition>
                <stock>Huston</stock>
            </storeroom>
        </building>
```

```
    <building>
        <B_Name>Peppertree</B_Name>
        <location>16 Cathy street</location>
        <storeroom>
            <SR_Name>Greenfield</SR_Name>
            <condition>warm</condition>
            <stock>Dixon</stock>
        </storeroom>
    </building>
    <building>
        <B_Name>Timona</B_Name>
        <location>12 Featherston street</location>
        <storeroom>
            <SR_Name>Greenfield</SR_Name>
            <condition>cool</condition>
            <stock>Dixon</stock>
        </storeroom>
    </building>
    <building>
        <B_Name>Tararua</B_Name>
        <location>223 Ashford avenue</location>
    </building>
    <building>
        <B_Name>Waterdale</B_Name>
        <location>1 King street</location>
        <storeroom>
            <SR_Name>Emerson</SR_Name>
            <condition>icy</condition>
            <stock>Dixon</stock>
        </storeroom>
    </building>
</city>
<city>
    <C_Name>Christchurch</C_Name>
```

```
    <building>
        <B_Name>Waterdale</B_Name>
        <location>1 King street</location>
        <storeroom>
            <SR_Name>Emerson</SR_Name>
            <condition>icy</condition>
            <stock>Dixon</stock>
        </storeroom>
        <storeroom>
            <SR_Name>Gemini</SR_Name>
            <stock>Blue Bird</stock>
            <stock>Uncle Toby</stock>
        </storeroom>
    </building>
    <building>
        <B_Name>Ruahine</B_Name>
        <location>56 Worcester street</location>
    </building>
</city>
<city>
    <C_Name>Dunedin</C_Name>
    <building>
        <B_Name>Waterdale</B_Name>
        <location>1 King street</location>
        <storeroom>
            <SR_Name>Emerson</SR_Name>
            <condition>dark</condition>
            <stock>Dixon</stock>
        </storeroom>
        <storeroom>
            <SR_Name>Hartley</SR_Name>
            <stock>Pam's</stock>
            <stock>Silverbell</stock>
        </storeroom>
```

```
    </building>
</city>
<city>
    <C_Name>Hamilton</C_Name>
    <building>
        <B_Name>Waterdale</B_Name>
        <location>1 King street</location>
        <storeroom>
            <SR_Name>Woodville</SR_Name>
            <condition>dark</condition>
            <stock>Dixon</stock>
        </storeroom>
    </building>
</city>
<city>
    <C_Name>Palmerston North</C_Name>
    <building>
        <B_Name>Waterdale</B_Name>
        <location>11 Queen street</location>
        <storeroom>
            <SR_Name>Woodville</SR_Name>
            <condition>dark</condition>
            <stock>Dixon</stock>
        </storeroom>
    </building>
</city>
<city>
    <C_Name>Wellington</C_Name>
    <building>
        <B_Name>Daladale</B_Name>
        <location>11 Queen street</location>
        <storeroom>
            <SR_Name>Woodville</SR_Name>
            <condition>dark</condition>
```

```
                <stock>Dixon</stock>
                <stock>Healtheries</stock>
            </storeroom>
            <storeroom>
                <SR_Name>Taonui</SR_Name>
                <stock>Symphony</stock>
            </storeroom>
        </building>
    </city>
</warehouse>
```

Bibliography

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[Ahonen, 1996] Ahonen, H. (1996). *Generating Grammars for Structured Documents Using Grammatical Inference Methods*. PhD Thesis, Series of Publications A, Report A-1996-4, Department of Computer Science, University of Finland.

[Bray et al., 2004] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2004). W3c recommendation: Extensible Markup Language (XML) 1.0 (3rd Ed.). World Wide Web Consortium (W3C). Retrieved 1 November 2005 from http://www.w3.org/TR/REC-xml.

[Buneman, 1997] Buneman, P. (1997). Semistructured data. In *PODS*, pages 117–121. ACM Press.

[Carrasco and Oncina, 1994] Carrasco, R. C. and Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method. In Carrasco, R. C. and Oncina, J., editors, *ICGI*, volume 862 of *Lecture Notes in Computer Science*, pages 139–152. Springer.

[Chen et al., 2002] Chen, E., Wu, G., Lindemann, G., and Minor, M. (2002). Semistructured data store mapping with XML and its reconstruction. In Manolopoulos, Y. and Návrat, P., editors, *ADBIS Research Communications*, pages 78–87. Slovak University of Technology, Bratislava.

[Chen, 1991] Chen, J. (1991). Grammar generation and query processing for text databases. Research proposal, University of Waterloo, Canada.

[Chidlovskii, 2001] Chidlovskii, B. (2001). Schema extraction from XML: A grammatical inference approach. In [Lenzerini et al., 2001].

[Chidlovskii, 2002] Chidlovskii, B. (2002). Schema extraction from xml collections. In *JCDL*, pages 291–292. ACM.

[Christophides et al., 1994] Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. (1994). From structured documents to novel query facilities. In Snodgrass, R. T. and Winslett, M., editors, *SIGMOD Conference*, pages 313–324. ACM Press.

[Clark, 2001] Clark, J. (2001). TREX - tree regular expressions for XML. Thai Open Source Software Center. Latest version available online: http://www.thaiopensource.com/trex/.

[Cruz et al., 2004] Cruz, I. F., Xiao, H., and Hsu, F. (2004). Peer-to-Peer semantic integration of XML and RDF data sources. In *Third International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2004)*.

[Fallside and Walmsley, 2004] Fallside, D. C. and Walmsley, P. (2004). W3C recommendation: XML schema part 0: Primer second edition. World Wide Web Consortium (W3C). Retrieved 1 November 2005 from http://www.w3.org/TR/REC-xml.

[Garofalakis et al., 2000] Garofalakis, M. N., Gionis, A., Rastogi, R., Seshadri, S., and Shim, K. (2000). XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In Chen, W., Naughton, J. F., and Bernstein, P. A., editors, *SIGMOD Conference*, pages 165–176. ACM.

[Gaskin, 2000] Gaskin, J. E. (2000). XML comes of age. *InternetWeek, CMP Media LLC.*

[Geroimenko and Geroimenko, 2001] Geroimenko, V. and Geroimenko, L. (2001). Visual interaction with XML metadata. In Banissi E, Khosrowshahi F, S. M. and A, U., editors, *IV2001*, pages 539–545. IEEE Computer Society: Los Alamitos, CA, USA.

[Goldman and Widom, 1997] Goldman, R. and Widom, J. (1997). DataGuides: Enabling query formulation and optimization in semistructured databases. In Jarke, M., Carey, M. J., Dittrich, K. R., Lochovsky, F. H., Loucopoulos, P., and Jeusfeld, M. A., editors, *VLDB*, pages 436–445. Morgan Kaufmann.

[Gross and Yellen, 1999] Gross, J. and Yellen, J. (1999). *Graph theory and its applications*. Boca Raton, Florida: CRC Press.

[Hartmann and Link, 2003] Hartmann, S. and Link, S. (2003). More functional dependencies for XML. In Kalinichenko, L. A., Manthey, R., Thalheim, B., and Wloka, U., editors, *ADBIS*, volume 2798 of *Lecture Notes in Computer Science*, pages 355–369. Springer.

[Hartmann et al., 2003] Hartmann, S., Link, S., and Kirchberg, M. (2003). A subgraph-based approach towards functional dependencies for XML. In Callaos, N., Lesso, W., Rahimi, S., Boonjiing, V., Mohamard, J., Liu, T. K., and Schewe, K. D., editors, *Proceedings of the 7th World Multicoference on Systemics, Cybernetics and Informatics(SCI2003)*, Computer Science and Engineering: II, pages 200–211. International Institute of Informatics and Systemics(IIIS).

[Lenzerini et al., 2001] Lenzerini, M., Nardi, D., Nutt, W., and Suciu, D., editors (2001). *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001), Rome, Italy, September 15, 2001*, volume 45 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Lu et al., 2003] Lu, S., Sun, Y., Atay, M., and Fotouhi, F. (2003). A new inlining algorithm for mapping xml dtds to relational schemas. In Jeusfeld, M. A. and Pastor, O., editors, *ER (Workshops)*, volume 2814 of *Lecture Notes in Computer Science*, pages 366–377. Springer.

[Mannila and Räihä, 1987] Mannila, H. and Räihä, K.-J. (1987). Dependency inference(extended abstract). In Stocker, P. M., Kent, W., and Hammersley, P., editors, *VLDB*, pages 155–158. Morgan Kaufmann.

[Mannila and Räihä, 1992] Mannila, H. and Räihä, K. J. (1992). *The Design of Relational Databases*. Addison-Wesley.

[Mannila and Räihä, 1994] Mannila, H. and Räihä, K.-J. (1994). Algorithms for inferring functional dependencies from relations. *Data Knowl. Eng.*, 12(1):83–99.

[Milo and Suciu, 1999] Milo, T. and Suciu, D. (1999). Index structures for path expressions. In Beeri, C. and Buneman, P., editors, *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer.

[Murata, 2001] Murata, M. (2001). RELAX (REgular LAnguage description for XML). Latest version available online: http://www.xml.gr.jp/relax/.

[Nestorov et al., 1997] Nestorov, S., Ullman, J. D., Wiener, J. L., and Chawathe, S. S. (1997). Representative objects: Concise representations of semistructured, hierarchial data. In Gray, W. A. and Larson, P.-Å., editors, *ICDE*, pages 79–90. IEEE Computer Society.

[Papakonstantinou et al., 1995] Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. (1995). Object exchange across heterogeneous information sources. In Yu, P. S. and Chen, A. L. P., editors, *ICDE*, pages 251–260. IEEE Computer Society.

[Ramanath et al., 2003] Ramanath, M., Freire, J., Haritsa, J. R., and Roy, P. (2003). Searching for efficient XML-to-Relational mappings. In Bellahsene, Z., Chaudhri, A. B., Rahm, E., Rys, M., and Unland, R., editors, *Xsym*, volume 2824 of *Lecture Notes in Computer Science*, pages 19–36. Springer.

[Rissanen, 1978] Rissanen, J. (1978). Modeling by shortest data description. Automatica 14: 465-471.

[Rissanen, 1989] Rissanen, J. (1989). Stochastic complexity in statistical inquiry. Volume 15 of Computer Science, World Scientific Publishing Co Pte Ltd.

[Sakakibara, 1997] Sakakibara, Y. (1997). Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185(1):15–45.

[Shanmugasundaram et al., 1999] Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J., and Naughton, J. F. (1999). Relational databases for querying xml documents: Limitations and opportunities. In Atkinson, M. P., Orlowska, M. E., Valduriez, P., Zdonik, S. B., and Brodie, M. L., editors, *VLDB*, pages 302–314. Morgan Kaufmann.

[Sun Microsystems Inc., 2002] Sun Microsystems Inc. (2002). Web services made easier: The Java APIs and architectures for XML. Java™Technical White Paper.

[Wyss et al., 2001] Wyss, C. M., Giannella, C., and Robertson, E. L. (2001). FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In Kambayashi, Y., Winiwarter, W., and Arikawa, M.,

editors, *DaWaK*, volume 2114 of *Lecture Notes in Computer Science*, pages 101–110. Springer.

[Yan and Fu, 2001] Yan, M. H. and Fu, A. W.-C. (2001). From xml to relational databases. In [Lenzerini et al., 2001].

[Young-Lai, 1996] Young-Lai, M. D. (1996). Application of a stochastic grammatical inference method to text structure. Master's thesis, Computer Science Department, University of Waterloo.