

THE UNIVERSITY OF HULL

XML Documents Schema Design

Being a thesis submitted for the degree of

Doctor of Philosophy

in The University of Hull

By

ZURINAHNI ZAINOL

**B.Sc (Hons), Universiti Kebangsaan Malaysia, 1989
Master of Computer Science, Universiti Sains Malaysia, 1995**

Department of Computer Science

January 2012

ABSTRACT

The eXtensible Markup Language (XML) is fast emerging as the dominant standard for storing, describing and interchanging data among various systems and databases on the internet. It offers schema such as Document Type Definition (DTD) or XML Schema Definition (XSD) for defining the syntax and structure of XML documents. To enable efficient usage of XML documents in any application in large scale electronic environment, it is necessary to avoid data redundancies and update anomalies. Redundancy and anomalies in XML documents can lead not only to higher data storage cost but also to increased costs for data transfer and data manipulation.

To overcome this problem, this thesis proposes to establish a formal framework of XML document schema design. To achieve this aim, we propose a method to improve and simplify XML schema design by incorporating a conceptual model of the DTD with a theory of database normalization. A conceptual diagram, Graph-Document Type Definition (G-DTD) is proposed to describe the structure of XML documents at the schema level. For G-DTD itself, we define a structure which incorporates attributes, simple elements, complex elements, and relationship types among them. Furthermore, semantic constraints are also precisely defined in order to capture semantic meanings among the defined XML objects.

In addition, to provide a guideline to a well-designed schema for XML documents, we propose a set of normal forms for G-DTD on the basis of rules proposed by Arenas and Libkin and Lv. et al. The corresponding normalization rules to transform from a G-DTD into a normal form schema are also discussed. A case study is given to illustrate the applicability of the concept. As a result, we found that the new normal forms are more concise and practical, in particular as they allow the user to find an 'optimal' structure of XML elements/attributes at the schema level. To prove that our approach is applicable for the database designer, we develop a prototype of XML document schema design using a Z formal specification language. Finally, using the same case study, this formal specification is tested to check for correctness and consistency of the specification. Thus, this gives a confidence that our prototype can be implemented successfully to generate an automatic XML schema design.

To my beloved husband: Meor Azli Ayub

*To my wonderful children: Arief, Nabilah, Nabihah, Najihah, Amer, Ahmad and
Nadhirah*

To my parents: Bashah and Zainol

ACKNOWLEDGEMENTS

“IN THE NAME OF ALLAH, THE BENEFICENT AND THE MERCIFUL”

I would like to acknowledge the help and support of many people, who made the completion of this thesis possible.

First and foremost, I would like to express my deep and sincere gratitude to my supervisor, Dr Bing Wang, for everything he taught me. His enthusiasm, insightful ideas, moral support and encouragement have made working with him a great pleasure.

I would like to thank the members of my PhD advisory committee, Dr Leonardo Bottaci and Dr David Grey for their helpful comments and advice over the years.

I would like to acknowledge Universiti Sains Malaysia, Ministry of Higher Education Malaysia and Department of Computer Science, University of Hull for giving me this opportunity and for financial support during my study.

My deepest thanks go to my dearest husband, Meor Azli, for his prayers, support, patience and understanding during hard times in my work over the years, and more importantly, for being my best friend and companion. Infinite thanks to my lovely children and wonderful parents for their endless love, sacrifices and encouragement in all aspects of my life. Indeed, they meant the whole world to me and I am nothing without their love.

Last but not least, massive thanks to all my friends especially Waakil, Joshua, Mayur, Julius, Nongnuch, Nabil, Rahman, Amer, Shawulu and Sebta for being so supportive. I feel very lucky to have had the opportunity of working and learning among a great group of people like you all.

DECLARATION

Parts of this thesis were published as research papers with Dr Bing Wang in the following sources:

1. Zainol, Z. and Wang, B. (2010), G-DTD: Graphical Notation for Describing XML Documents, *In Proceeding of 2nd International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDA, IEEE*, pp. 214-221.
2. Zainol, Z. and Wang, B. (2010), XML Document Normalization using G-DTD, *In Proceeding 8th International Conference of Internet Computing, ICOMP 2010, 12- 15 July Las Vegas, USA*, pp. 232-239.
3. Zainol, Z. and Wang, B. (2010), XML Document Design via G-DTD, *European Journal of Scientific Research*, Vol. 44 (2), pp. 314-336.
4. Zainol, Z. and Wang, B. (2011), XML Documents Normalization using G-DTD, *International Journal of Information Retrieval Research*, Vol. 1(1), pp. 53-76.
5. Zainol, Z. and Wang, B. (2011), A Formal Specification of G-DTD: A conceptual Model to Describe XML Documents, *In Proceeding of 6th International Conference on Software Engineering Advances, ICSEA, IEEE*, 23-28 October, Barcelona, Spain. 338-347.
6. Zainol, Z. and Wang, B. (2012), A Formal Framework of XML Documents Schema Design, *Journal of Computing*, Vol 4 (2). In Print

ABBREVIATIONS

1NF	First Normal Form
1XNF	First XML Normal Form
2NF	Second Normal Form
2XNF	Second XML Normal Form
3NF	Third Normal Form
3XNF	Third XML Normal Form
4XNF	Fourth XML Normal Form
ATT	Attribute Node
BCNF	Boyce Codd Normal Form
CE	Complex Element Node
DTD	Document Type Definition
FD	Functional Dependency
G-DTD	Graph Notation - Document Type Definition
GFP	Global Functional Dependency
LHS	Left Hand Side
MVD	Multivalued Dependency
NF-NR	Normal Form for Nested Relation
NNF	Nested Normal Form
O-NF	Object Normal Form
ORA-SS	Object Relational Attribute –Semi Structured
PFD	Partial Functional Dependency
RD	Relational Dependency
RHS	Right Hand Side
R-NF	Relational Normal Form
SE	Simple Element Node
TFD	Transitive Functional Dependency
X3NF	Third XML Normal Form
XFD	XML Functional Dependency
XML_DM	XML_Design Model
XNF	XML Normal Form
XSD	XML Schema

TABLE OF CONTENTS

ABSTRACT	i
ACKNOWLEDGEMENTS	iii
DECLARATION.....	iv
ABBREVIATIONS	v
TABLE OF CONTENTS	vi
LIST OF FIGURES.....	ix
Chapter 1	1
Introduction	1
1.1 Motivation of the Research.....	1
1.2 Outline of Research Problems and Hypothesis	3
1.3 Thesis Aim and Objectives	5
1.4 Thesis Contributions.....	6
1.5 Thesis Structure	7
Chapter 2	10
Background and Literature Review.....	10
2.1 Introduction.....	10
2.2 Relational Database	11
2.2.1 Introduction	11
2.2.2 Data Dependencies in Relational Databases	12
2.2.3 Normal Forms for Relational Database.....	16
2.2.4 Normalization Process	20
2.3 XML Documents	23
2.3.1 Introduction	23
2.3.2 Schema Languages for Markup Languages Based Documents	24
2.3.3 Basic notations	28
2.3.4 Functional Dependencies for XML (XFD).....	34
2.3.5 Inference Rules for XML Functional Dependencies	47
2.3.6 Data Redundancies and Anomalies for XML Documents	48
2.3.7 Normal Forms for XML Documents.....	50
2.3.8 Other Definitions of Normal Forms.....	63
2.3.9 Discussion of Current XML Normal Forms	65
2.4 Summary.....	67
Chapter 3	68
A G-DTD: A Graph Model for Describing XML Documents	68
3.1 Introduction.....	68
3.2 XML Model Review.....	70
3.2.1 Models for Schema level.....	71
3.2.2 Models for Instance Level.....	73
3.2.3 Models for Mix of Instance and Schema Levels.....	74
3.2.4 Other data models	75

3.3	Document Type Definition (DTD) – Its Basic and Rationale	76
3.3.1	Introduction	76
3.3.2	Overview of DTD syntax	77
3.4	G-DTD Data Model	83
3.4.1	Objectives of the Model	83
3.4.2	Overall view of G-DTD	83
3.4.3	G-DTD Components	84
3.4.3.1	Simple Element Nodes	85
3.4.3.2	Complex Element Nodes	86
3.4.3.3	Attributes nodes	87
3.4.3.4	Root Node	88
3.4.3.5	Relationships	88
3.4.3.6	Semantic Constraint Between set Relationship	92
3.5	Example of G-DTD	95
3.6	G-DTD Operations	99
3.6.1	Query Operations	99
3.6.2	Insert Operations	100
3.6.3	Delete Operations	101
3.6.4	Replicate Operations	102
3.6.5	Determine the root node and last node	102
3.7	Rules for converting G-DTD to DTD	103
3.7.1	General Rule	103
3.7.2	Semantic Rule	105
3.8	Summary	109
Chapter 4	110
Normal Forms for XML Documents	110
4.1	Introduction	110
4.2	Data Dependency of G-DTD	111
4.2.1	Key Dependency (KD)	115
4.2.2	Functional Dependency (FD)	115
4.3	Normal Forms for G-DTD	119
4.3.1	First XML Normal Form (1XNF)	119
4.3.2	Second XML Normal Form (2XNF)	120
4.3.3	Third XML Normal Form (3XNF)	121
4.3.4	Fourth XML Normal Form (4XNF)	121
4.4	The Process of Normalization	122
4.4.1	Normalization Rules	122
4.4.2	Normalization Algorithms	125
4.5	Case study	132
4.5.1	1XNF G-DTD	134
4.5.2	2XNF G-DTD	134
4.5.3	3XNF G-DTD	138
4.5.4	4XNF G-DTD	142
4.6	Comparisons of Proposed Approach with Existing Approaches	146
4.7	Discussion	154
4.8	Summary	155

Chapter 5	156
Formal Specification of XML Document Design Model.....	156
5.1 Introduction.....	156
5.2 The Formal Specification and Its Importance	157
5.3 Z notations	158
5.4 The overview of XML Document Design Model (XML_DM).....	159
5.5 Formal Specification of a Conceptual G-DTD Model.....	161
5.5.1 Basic Type Definitions.....	161
5.5.2 The Data Structure of G-DTD.....	161
5.5.3 Abstract state of Environment XML_DM	166
5.5.4 Initial state of Environment XML_DM	168
5.5.8 Manipulating the G-DTD in an Environment	168
5.5.9 Operations of G-DTD in an Environment.....	177
5.6 Formal Specification of G-DTD Normalizer.....	188
5.6.1 Determine Type Functional Dependency Operation.....	189
5.6.2 Normalization Operations	190
5.6.3 Basic Functions of G-DTD Normaliser	191
5.6.4 Restructure IXNF Schema Operations.....	197
5.6.5 Restructure 2XNF Schema Operations	200
5.6.6 Restructure 3XNF Schema Operations	202
5.7 Formal Specification of G-DTD Translator	204
5.8 Summary	206
Chapter 6	208
Specification Testing – A Case Study	208
6.1 Introduction.....	208
6.2 Representing G-DTD Diagram in a Z Specification	209
6.3 Consistency of the Operations in G-DTD Normalizer	211
6.3.1 IXNF	212
6.3.2 Normalize 1XNF to 2XNF.....	213
6.3.3 Normalize 2XNF to 3XNF.....	222
6.3.4 Normalize 3XNF to 4XNF.....	230
6.4 Summary.....	239
Chapter 7	240
Conclusion and Future Work	240
7.1 Contributions of the Research	240
7.2 Limitations and Future Research	246
Bibliography	248
Appendix A	263

LIST OF FIGURES

Figure 2.1: Relation <i>Student</i>	11
Figure 2.2: Relation <i>BranchStaffOwner</i>	14
Figure 2.3: Level of Normalization (Date, 2000).....	16
Figure 2.4: Relation <i>Patient</i>	18
Figure 2.5: An Algorithm to Check the Correctness of 3NF (Abiteboul et al., 1995).....	19
Figure 2.6: Relation <i>Appointment</i>	21
Figure 2.7: Relation <i>Dr_room</i>	21
Figure 2.8: A BCNF Algorithm (Abiteboul et al., 1995)	22
Figure 2.9: An XML Document	24
Figure 2.10: A DTD Describing the XML Document (Arenas and Libkin, 2004).....	25
Figure 2.11: A Fragment of an XSD for XML Documents	26
Figure 2.12: Tree Representation of XML Document	30
Figure 2.13: Tree representation of an XML Document.....	41
Figure 2.14: XSD Model	43
Figure 2.15: Scheme Graph.....	45
Figure 2.16: Four Pre-Images of the $v_{lecturer}: X \rightarrow Y$	47
Figure 2.17: DTD for DBLP Database.....	49
Figure 2.18: XNF Decomposition Algorithm (Arenas and Libkin, 2004).....	52
Figure 2.19: Moving Attribute	53
Figure 2.20: Create New Element	54
Figure 2.21: Algorithm of Schema Normalization (Yu and Jagadish, 2008).....	59
Figure 2.22: Normalized XSD.....	60
Figure 2.23: An XML Document in X3NF	62
Figure 2.24: ORA-SS Schema Diagram.....	64
Figure 2.25: Normalized ORA-SS Schema Diagram.....	65
Figure 3.1: XML Document Design Process	69
Figure 3.2: DTD	81
Figure 3.3: Types of Simple Element Nodes	86
Figure 3.4: Complex Element Node <i>Student</i>	87
Figure 3.5: Many-to-Many Binary Relationship	91
Figure 3.6: <i>Part_of</i> Link and <i>Has_A</i> Link.....	91

Figure 3.7: Relationship Between Complex Element, Attribute And Simple Element Nodes	92
Figure 3.8: Sequence of Simple Elements.....	93
Figure 3.9: Sequence of Attribute and Simple Elements	93
Figure 3.10: Binary Disjunction of Simple Elements	94
Figure 3.11: Disjunction of Several Simple Elements	95
Figure 3.12: G-DTD's Notations	97
Figure 3.13: G-DTD	98
Figure 3.14: Transformation Rules.....	109
Figure 4.1: XML Document Conforming to G-DTD in Figure 4.2	113
Figure 4.2: G-DTD	114
Figure 4.3: Algorithm 2XNF	126
Figure 4.4: Create a New Complex Element Node	127
Figure 4.5: Algorithm 3XNF	128
Figure 4.6: Moving Up a Complex Element Node	129
Figure 4.7: Algorithm 4XNF	130
Figure 4.8: Moving a Complex Element Node under a Root Node	131
Figure 4.9: G-DTD in a 2XNF	136
Figure 4.10: An XML Document That Conforms to 2XNF.....	137
Figure 4.11: G-DTD in a 3XNF	140
Figure 4.12: An XML Documents Conform to 3XNF	141
Figure 4.13: G-DTD in a 4XNF	143
Figure 4.14: An XML Document That Conforms to 4XNF.....	145
Figure 5.1: Layers of XML Document Design Model	160
Figure 6.1: Schema GDTD.....	210
Figure 6.2: Set of KD and FDs.....	211
Figure 6.3: Schema GDTD in 2XNF.....	221
Figure 6.4: Set of KDs and FDs	222
Figure 6.5: SchemaGDTD in 3XNF.....	229
Figure 6.6: Set of FDs	230
Figure 6.7: SchemaGDTD in 4XNF.....	238
Figure 6.8: Set of KDs and FDs	239

Chapter 1

Introduction

1.1 Motivation of the Research

With the wide utilization of the web and the availability of a huge amount of electronic data, XML (eXtensible Markup Language) has been used as a standard means of information representation and exchange over the Web. Its usage has increased extensively in many commercial applications with complex data structures such as Manufacturing, Bioinformatics, B2B (Business to Business), Medicine and Geographical data (Powell, 2007; Ma and Yan, 2007; Pankowski, 2009). Thus, effective means of the management of XML documents as databases are needed for query, consistent and efficient storage. Various databases, including relational, object-oriented, and object-relational databases have been used for mapping to and from XML documents (Florescu and Kossmann, 1999; Runapongsa and Patel, 2002). Among this kind of database, most researchers use a relational database as a persistent storage since it is a more promising alternative, because of its maturity.

However, this approach has disadvantages, since it does not support well complex data structures such as scientific data because it cannot retain the original of XML documents (Bourret, 2007). With such problem, has led to the development of native XML database system for a number of applications and its use is increasingly rapidly because its ability to hold and manage highly complex data structures (Bourret, 2007; Philippi and Kohler, 2004; Lee et al., 2010). Such applications may use native XML database facilities (Kanne and Moerkotte, 2000) to store and update XML data (Tatarinov et al., 2001). The native XML database stores XML documents directly without performing any conversion or shredding the XML documents into another format thus reduce processing time and provide better performance. However, native XML database is still in its infancy and not as mature as traditional databases (e.g. relational database), hence many important problems and questions remain unanswered, especially on the principles of XML database design (Arenas, 2006; Schewe, 2005; Libkin, 2007).

Essentially, an XML document can be regarded as a native XML database because every XML document contains both metadata and data (Powell, 2007). It is important to design *non-redundant* XML document for the sake of readability and manageability. The non-redundant design means there are no duplicate information, store correct and complete information. This is because duplicate information will waste space and increases the likelihood of errors and inconsistencies.

Like managing traditional database, the management of XML documents requires capabilities to handle with integrity, consistency, data dependency, redundancy, views, access rights, integration, and normal forms (Yu and Jagadish, 2008; Libkin, 2007; Arenas and Libkin, 2004; Dobbie, 2001; Feng et al., 2002). Amongst the important problem related to XML database design are data redundancies and update anomalies. Similar to relational database, reducing data redundancy is an important step in XML design because it cause update anomalies, which lead to an efficient use of the database.

In XML documents, redundancies and update anomalies occur when the *schema* such as Document Type Definition (DTD) allows addition of redundant values (Arenas and Libkin 2004). The redundancy of data in XML documents highly depends on how the *schema* is designed. Thus, we can say that efficient use of XML documents depends on the quality of schema design.

However, to formulate the criteria for non-redundant of XML document schema design is very challenging for the following reasons (Arenas, 2006; Libkin, 2007):

- The structure of XML documents is different from that of relational databases which contain a complicated path structure, so it is difficult to see whether it contains redundancies.
- Expression of semantic constraints of XML documents imposed by *schema* such as DTD and XML Schema Definition (XSD) is limited.

- There are problems of ensuring that data and semantic constraint of designed *schema* are not lost and preserved after the process of normalization of the *schema*.
- No acceptable notion of an XML updates as yet exists, comparable to the notion of the relational updates, which makes it hard to say what makes an update anomaly.
- There is no standard query language for XML document compared to relational algebra for relational databases.

Thus, the above challenges and issues motivate us to investigate further the needs and requirements to achieve a non-redundant XML document design, particularly one that is free from data redundancy.

1.2 Outline of Research Problems and Hypothesis

XML can be classified into two main types (Bourret, 2007; Wang and Topor, 2005; Vincet et al., 2007). The first type of application is called *document centric* XML and the second type is called *data centric* XML. Document centric XML is used as a mark up language for semi-structured text documents with mixed-content elements, where the content and order of sibling elements is significant, for instance a user's manual, webpage, etc. Data centric XML consist of more regular structured data for automated processing and there are few or no elements with mixed content, comment and processing instruction, such as geographic and scientific databases which contain complex semi-structured data, e.g molecular biology, protein data being the most prevalent example. In this thesis, we will focus on data centric applications and we will refer to data centric XML as XML documents.

To date, several normal forms and normalization algorithms for XML documents have been proposed. The reason for this is to give a guideline to the user to eliminate data redundancy in XML documents. For example, different notions of normal forms for

XML documents have been proposed by Arenas and Libkin (2004), Lv et al., (2004), Wang and Topor (2005), Kolahi (2006), and Yu and Jagadish (2008). Generally these normal forms are based on functional dependency (Yu and Jagadish, 2008; Arenas and Libkin, 2004; Vincent et al., 2004; Wong and Topor, 2005; Emberly and Mok, 2001; Ling, 2001, Lee et al., 1999; Mani et al., 2001; Wu et al., 2001) or multivalued dependencies (Vincent, 2003; Emberly and Mok, 2001). Many notions of XML functional dependencies (XFD) have been defined by them to represent the semantic constraints in XML as well. However, the best approach so far to defining XML functional dependencies is that based on tree tuples, proposed by Arenas and Libkin (2004), since they used a well-developed concept from the relational model (Codd, 1970). Arenas and Libkin proposed a notion of tree tuple based on the idea from relational schema (Codd, 1972) and nested relational schema (Mok et al., 1996).

However there are problems with Arenas and Libkin's definition of normal forms and normalization algorithms, as follows:

- The current definitions of XML normal forms are presented in term that are difficult to be understood by non technical users or practitioners. Thus, the approach did not show the tremendous benefit to practitioners because proposed XML normalization concepts have been very complicated for designers to apply effectively in real world applications (Bourret, 2007). In particular, the information system academics or practitioners might be interested in finding out some means of normalization without formulas, interpretation, theorems etc. XML normal form needs to be defined in a simple way that is easy to be implemented.
- The normalization algorithms only work for the existing normal forms, which have limited semantic expressiveness (Yu and Jagadish, 2008; Pankowski, 2009).

- Even though the proposed normalization algorithms can eliminate data redundancy, the semantics of the original data (dependency preserving) for initial schema could not be preserved during the construction of the new schema and the original information might be lost (Kolahi, 2007). XML normalization needs to be improved to be more precise and understandable.
- The cost of restructuring the original XML documents schema will be very expensive if it involves a huge XML document, since decomposition is an expensive operation (Arenas and Libkin, 2006).

Therefore, in this work, we are looking into the first two issues. For these issues, such XML normal forms need to be redefined in an easy and more practical way. We believe that defining a simple definition of XML normal form will make XML document schema design easier. Under this assumption, we argue that to produce a non-redundant schema of an XML document for application A , we should first produce a conceptual model, S at schema level and then apply a normalization rule to transform S into a normal form S' and finally convert the conceptual model S' back to the XML schema.

1.3 Thesis Aim and Objectives

The research aim is to establish a formal framework of XML document schema design by incorporating a conceptual model of XML schema, specifically DTD, with a theory of database normalization.

To achieve this aim the following research objectives were defined:

- (1) To investigate how design guidelines for relational schema are applied to XML database schema design using normalization theory. This involves examining of XML functional dependency (XFD) concepts and discussing various definitions of XML normal forms based on these XFDs and highlights their strengths and limitations.

- (2) To propose a systematic approach to simplify XML document schema design by first proposing a graphical XML schema based on DTD called Graph Document Type Definition (G-DTD) at the schema level. We believe having the G-GTD model as a tool could describe the structure of XML documents at the schema level clearly and precisely.
- (3) To redefine a set of normal form for G-DTD on the basis of Arenas and Libkin's rule (2004) and Lv et al.'s rules (2004) which is easy to understand and implement programmatically. To achieve this, a basic property of XML normal form, which is functional dependency, is proposed, such as relationship dependency, partial functional dependency, transitive functional dependency and global functional dependency. In the context of XML document normalization, it is important to develop normalization rules to transform an XML document schema into a normalised one.
- (4) To develop a prototype of an XML document schema design using a formal approach. More specifically, to propose a formal framework of XML document normalization using Z formal specification language in order to give a precise and a clearer understanding of the whole system requirement.
- (5) The final research objective is to test the specification constructed to show the consistency of the specification using a simple case study.

1.4 Thesis Contributions

The thesis contributes to the research literature, by proposing a specific solution to the problem mentioned previously, using conceptual model and database normalization theory. More specifically, these significant contributions can be measured along four dimensions:

- (1) This thesis has proposed G-DTD, as a conceptual model to describe XML documents at the schema level in a precise and simple way by adopting some of ER diagram (Chen, 1976) and ORA-SS diagram (Dobbie et al., 2000) notations. The structure, semantic and conceptual operations of G-DTD model were

introduced and developed to describe the XML document at the schema level and the dynamic properties of D-DTD model. This ultimately helps contribute to understanding the DTD and DTD design.

- (2) The thesis has refined a set of normal forms for G-DTD: First Normal Form (1XNF), Second Normal Form (2XNF), Third Normal Form (3XNF) and Fourth Normal Form (4XNF). The set of normal forms for G-DTD have been generalised from Arenas and Libkin (2004) and Lv et al.'s (2004) normal forms. This set of normal form can be used as a guideline to the user to design non-redundant XML document schema on the basis of their application.
- (3) The thesis proposed a novel prototype of an XML Document Schema Design. This is to support the claim that users (designers) can profitably bring formal specification to bear in development of a real XML Document Schema Design tool. The complete framework and formal specification of the XML Document Design model was presented in Chapter 5. The full specification of the model using Z notation was constructed, which gives the precise and clear meaning of the model.
- (4) The thesis developed a case study to test the XML document design specification to enable us to demonstrate that the specification constructed in Chapter 5 is satisfied and consistent with certain fundamental criteria of XML document design. This will increase the confidence in the implementation of an automatic XML document design.

1.5 Thesis Structure

The thesis is organised as follows.

Chapter 2 presents an overview of relational database design. We discuss how the concept of relational database design is applied to XML database design by the XML database researchers. The current work of related to XML document design, which includes definition of functional dependencies and normal forms, is thoroughly reviewed.

Chapter 3 proposes a conceptual model to describe an XML document. The current conceptual model for XML document is reviewed with particularly focus on the model of schema level. The informal definition of a G-DTD is presented. We precisely define what a G-DTD looks like.

Chapter 4 defines a set of normal forms for G-DTD such as First Normal Form (1XNF), Second Normal Form (2XNF), Third Normal Form (3XNF) and Fourth Normal Form (4XNF). In order to implement XML document normalization, we propose normalization rules and algorithm to obtain a normalised G-DTD up to fourth normal form. Then a case study is provided to illustrate the application of these normal forms and normalization algorithms. This chapter also evaluates the proposed work with the existing approach. A comparison between our work and existing approaches is based on a number of criteria, specifically on expression of DTD structure, XML normal forms and normalization algorithms.

Chapter 5 presents a formal specification of the XML document design system called XML design model (XML_DM) which comprises the conceptual model G-DTD and normalization procedure, discussed in chapters 3 and 4. This formal specification is used to describe a fundamental framework of what the XML_DM can do and also as an abstraction of a full complete system which can serve as a reliable reference blueprint for those who want implement the prototype later. In this chapter, we describe the specification into three layers: a formal specification of a G-DTD model, G-DTD normalizer and G-DTD mapping tool. All these specifications are described using Z notation style, which gives precise, mathematical meaning and provide a deeper understanding of the modelled syntax, structure, and semantics of model properties. The use of formal specification techniques contributes to the clarity and conciseness of the model, and enables formal derivation of model properties to be performed easily.

Chapter 6 demonstrates precisely how we test the XML document design in order to illustrate the consistency of the specification, using the same case study as in Chapter 4. G-DTD normaliser is chosen as an example because it contains the important properties of XML document design, such as 1XNF, 2XNF, 3XNF and 4XNF designs.

Chapter 7 concludes the thesis with a summary of the main contributions of the thesis and gives some suggestions for future work.

Chapter 2

Background and Literature Review

2.1 Introduction

There are two approaches to the design process to achieve a non-redundant relational schema. The first is the logical (or conceptual) level which interprets the relational schema and the meaning of attributes. The second is the implementation (or storage level) which describes how the tuples in a relation are stored and updated. In this thesis we focus on the conceptual level of database design and how this theory is applied to XML database design.

We start in section 2.2 by presenting a review of conceptual modelling of database design and discuss theory that has been developed to design non-redundant schema related to relational databases in general. This includes basic concepts like data dependency such as functional dependency, key dependency and multi valued dependency. These data dependencies are formal constraints among attributes that are used as the main tool for formally measuring the semantic relation among attributes. We also describe how functional dependencies, key dependencies and multi valued dependencies can be used to group attributes into relational schema that are in a normal form. To address the normalization process, we present an algorithm for 3NF and BCNF design based on functional dependency and measure the correctness using two properties: information lossless and data dependency preservation.

In section 2.3, we describe how design guidelines for relational schemas are applied to XML database design using normalization theory. Schema for XML such as DTD and XSD are discussed. Different definitions of XML functional dependencies based on a path-based approach and subtree approach are presented and compared thoroughly. Lastly, we discuss various definitions of XML normal forms based on both

normalization theory and a conceptual approach and finally highlight the problems of existing XML normalization algorithms.

2.2 Relational Database

2.2.1 Introduction

Designing a relational database means selection of an appropriate relational schema for the particular data. The relational schema, which describes an overall description of the relational database, consists of a set of relations, or tables and a set of constraints over these relations. For instance, a relational database storing information about students in a library branch is shown in Figure 2.1. Each row of this table contains the student number (SNO), its title, author and branch.

<i>SNO</i>	<i>Title</i>	<i>Author</i>	<i>Branch</i>
0201385902	Database System	Date, C.J	BJL
0301456101	Data Structure	Berztiess	BJL
0501652111	Conceptual Database	Batini	KDL
0201385902	Database System	Date, C.J	KDL

Figure 2.1: Relation *Student*

The relation shown in Figure 2.1 consists of the data about the student and the *schema* of the relation. These two parts are the main component of the relational model. Formally, the relational schema is an expression of the form $R[U]$, where R is the name of the relation and $U = \{A_1, \dots, A_n\}$ is a set of attributes. For example, the schema of the relation in Figure 2.1 is *Student*[U], where $U = \{SNO, Title, Author, Branch\}$ and $domain(SNO)$ is the set of numbers. A tuple t is a mapping to associate value to each attribute of U . An *instance* I of a relational schema $R[U]$ is a set of U -tuples. For example, the instance shown in Figure 2.1 contains four tuples; the first is defined as $t_1(SNO) = 0201385902$, $t_1(Title) = Database\ System$, $t_1(Author) = Date, C.J$ and t_1

(*Branch*) = BJL. Thus, tuple t_1 is represented as (0201385902 Database System, Date, C.J, BJL). A *database schema* is a set of relational schemas $S = \{R_1[U_1], \dots, R_n[U_n]\}$.

Constraints or semantic constraints must be satisfied in the database. For example, in the relation shown in Figure 2.1, each SNO is associated to each Title. This semantic constraint is called data dependency. Thus, given a relational schema $R[U]$ and a set of data dependencies Σ over R , $(R[U], \Sigma)$ is also called a relational schema.

2.2.2 Data Dependencies in Relational Databases

The usefulness of data dependency theory for designing a well-formed relational database has been successfully proven over the past 30 years (Abiteboul et al., 1995). The theory concerns the question of non-redundant database design in terms of syntax and semantic properties (integrity constraints). Integrity constraints are the constraints that are imposed in order to protect the database from becoming inconsistent. Furthermore, integrity constraints are important for schema specification and query optimization because if the schema can satisfy these constraints, the problems of data redundancy and update anomalies in the database can be eliminated. Data dependency in a relational database can be classified into functional dependency, inclusion dependency, join dependency, key dependency, domain dependency and multi valued dependency. However, functional dependency (FD) and key dependency are the most useful.

Functional Dependency

A *functional dependency* (FD) over a relational schema $R[U]$ is written as $FD: X \rightarrow Y$, where X, Y is a subset or equal set of attributes in R . The set of attributes X is called the **left hand side** (LHS) of FD, and Y is called the **right-hand side** (RHS). Thus this means that X is a set of attributes that determine sets of attributes Y in a relation if and only if, in every possible value of R , whenever two tuples t_1, t_2 in I agree on their X value, they also agree on their Y value. This can be denoted as $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$. For example, the relation shown in Figure 2.1 satisfies the functional

dependency $SNO \rightarrow Title$, since each SNO number determines the $Title$ because the two tuples of this relation have the same value of attribute SNO and attribute $Title$. However the functional dependency $SNO \rightarrow Branch$ is not satisfied because the values on the attribute $Branch$ are different.

The number of functional dependencies will depend on the size of the set of relational schema. One obvious way to reduce the size of the set of FDs is to eliminate *trivial* dependencies. A dependency is trivial if it cannot be satisfied. This happens if and only if the right-hand side(RHS) is a subset of the left- hand side(LHS) (Date, 2000). For instance, the following FD for relational schema Figure 2.1 was trivial: $\{SNO, Title\} \rightarrow SNO$. We are more interested in practice in nontrivial dependencies because they show the real integrity constraint.

Key Dependency

Key dependency (KD) over relational schema $R[U]$ is written as $KD: X \rightarrow U$ where X is a *primary key* as all attributes (U) of the relation R are functionally dependent on X . However X is a *key* or *candidate key* if there are many attributes that can determine attributes in relation R . For instance, there is no primary key but $\{SNO, Title, Branch\}$ is a key for the relation shown in Figure 2.1.

Multivalued Dependency

Multivalued dependency (MVD) is a generalization of functional dependency, such that every FD is an MVD, but the converse is not true (i.e. there exist MVDs that are not FDs). For example let $R[U] = \{X, Y, Z\}$. A multivalued dependency over relational schema $R[U]$ is written as $X \twoheadrightarrow Y$, where X and Y are subsets of attribute U . It reads as X *multi-determines* Y . An instance I of $R[U]$ satisfies a multivalued dependency, written as $I \models X \twoheadrightarrow Y$, if every possible value set of Y values matching a given value (X value, Z value) pair depends only on the X value and is independent of the Z value (Fagin, 1977).

<i>branchno</i>	<i>staffname</i>	<i>ownername</i>
B003	Ann Beech	Carol Farrel
B003	David Ford	Carol Farrel
B003	Ann Beech	Tina Murphy
B003	David Ford	Tina Murphy

Figure 2.2: Relation *BranchStaffOwner*

For example, consider a relational schema *BranchStaffOwner* (*branchno*, *staffname*, *ownername*) (Connolly and Begg, 2002) of Figure 2.2, which multivalued dependency $branchno \twoheadrightarrow staffname$ holds for *BranchStaffOwner* because there is no direct relation between member of *staffname* and *ownername* at a given *branchno*. Hence a tuple for every combination of member of *staffname* and *ownername* must be created to ensure the relation is consistent. For example, if a new *ownername* for *B003* needs to be added to the relation, two new tuples for *staffname* have to be created as well to ensure the relation remains consistent.

Inference Rules for Functional and Multi-Valued Dependencies

The set of functional dependencies Σ over $R[U]$ that are implied by a given set functional dependencies X is called the *closure* of X written X^+ . A set of inference rules called Armstrong's axiom specifies how the closure of set of functional dependencies X can be inferred from given a set of functional dependencies Σ (Abiteboul et al., 1995). These inference rules or dependency implications have been studied for a relational database as it is an issue in a normalization theory. The following is a *sound* and *complete* set of inference rules for functional dependency:

Reflexibility : If Y is a subset of A , then $X \rightarrow Y$, where A is a set of Attributes.

Augmentation: If $X \rightarrow Y$, then $X, Z \rightarrow Y, Z$.

Transitivity : If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

The following is a sound and complete set of inference rules for multi-valued dependencies (Beeri et al., 1977) :

Complementation : If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow (U - Y)$.

Reflexivity : If Y subset or equal X , then $X \twoheadrightarrow Y$.

Augmentation : If $X \twoheadrightarrow Y$, then $XZ \twoheadrightarrow YZ$.

Transitivity : If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$.

Two rules have to be added to this set of inference rules in order to have a sound and complete set of rules for functional and multi-valued dependencies (Beeri et al., 1977):

Conversion : If $X \rightarrow Y$, then $X \twoheadrightarrow Y$.

Interaction : If $X \twoheadrightarrow Y$ and $XY \rightarrow Z$, then $X \rightarrow (Z - Y)$.

By *sound* is meant that any dependency that can be inferred from Σ holds in every relational schema on R that satisfies the dependencies in Σ . *Complete* means functional dependencies can be inferred repeatedly until no more dependencies can be inferred from Σ (Abiteboul et al., 1995). These inference rules are very important in normalization theory since they can be used to check and verify the correctness of a normalization algorithm, i.e. whether the generated relational schema is semantically equivalent to the original relational schema.

2.2.3 Normal Forms for Relational Database

A normal form in a relational database consists of 1NF, 2NF, 3NF, BCNF, 4NF and 5NF. The first three (1NF, 2NF, 3NF) were defined by Codd (1972). Figure 2.3 shows a level of normalization which defines that all normalised relational schema are in 1NF; some 1NF are also in 2NF; and some 2NF are also in 3NF. Generally, a non-redundant database design should have 3NF relation since it is more desirable than 2NF and 1NF. A revised normal form called the Boyce-Codd normal form (BCNF) was defined in Codd(1974) to replace from 3NF. Subsequently, Fagin (1977) defined new fourth normal form (4NF) and *projection-join normal form* (PJ/NF) also known as the *fifth normal form* or 5NF.

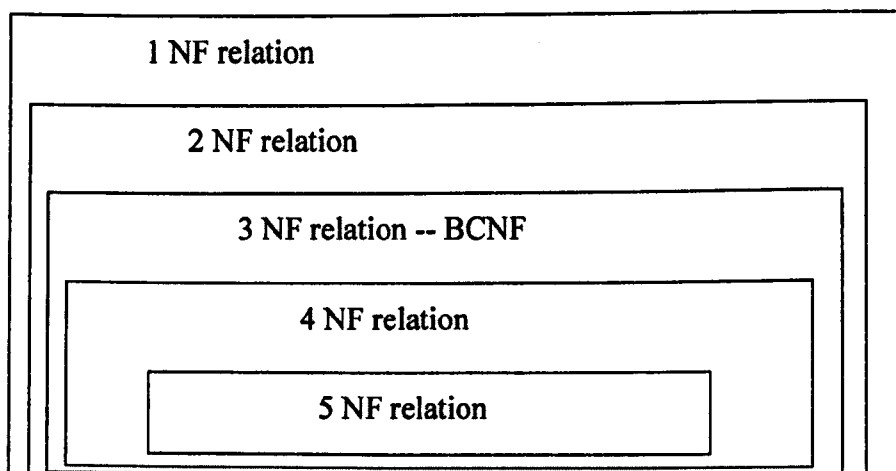


Figure 2.3: Level of Normalization (Date, 2000)

However in this work, we are more interested in 3NF and BCNF since they are most used in practice as they are well designed. BCNF decomposition guarantees to produce relations that do not cause any redundant data while 3NF decomposition may not produce non-redundant relations, but guarantees to preserve all the FDs (Abiteboul et al., 1995).

First Normal Form (1NF)

A relational schema is in first normal form (1NF) if and only if all attributes contain only atomic value; that is, there is no repeated group or attribute within a row. A relation in 1NF often suffers from data duplication, update performance and update integrity problems. These issues are related to concepts of key such as *superkey*, *candidate key* and *primary key*. A *superkey* is a set of one or more attributes which can uniquely identify an entity. Any subset of *superkey* is called a candidate key. A *primary key* is selected from the set of candidate keys to be used as index for the relation.

Second Normal Form (2NF)

A relational schema $(R[U], \Sigma)$ is in 2NF if and only if it is in 1NF and every nonkey attribute is fully dependent on the primary key. An attribute is fully dependent on the primary key if it is on the RHS of an FD for which the LHS is either the primary key itself or something that can be derived from the primary key using transitivity of FD.

Third Normal Form (3NF)

A relational schema $(R[U], \Sigma)$ is in 3NF if and only if for every nontrivial FD $X \rightarrow A \in \Sigma^+$, where X is superkey or A is a prime attribute. A database schema S is in 3NF if every relation schema in S is in 3NF. For example consider a relation *Patient* (*Patient No*, *Appointment Date*, *Appointment time*, *Dr_ID*, *Room_No*) with the following FDs:

FD1 $\{Room_No, Appointment_Date, Appointment_Time\} \rightarrow Dr_ID, Patient_No$ and
FD2 $\{Dr_ID, Appointment_Date\} \rightarrow Room_No, Appointment_Time$. The instance of a relational schema shown in Figure 2.4 is in 3NF since for both of RHS FD1: $\{DrID, PatientNo\}$ and FD2: $\{Room_No, Appointment_Time\}$ are prime attributes.

<i>Patient No</i>	<i>Appointment Date</i>	<i>Appointment Time</i>	<i>Dr_ID</i>	<i>Room No</i>
P34	1 Sept 08	10.30	JP_2	112
P14	1 Sept 08	12.00	JP_2	112
P35	1 Sept 08	12.00	Ale_3	102
P15	10 Sept 08	10.30	JP_2	102

Figure 2.4: Relation Patient

Maniila and Raiha (1989) has developed an algorithm to check whether the relation schema is in 3NF by testing if an attribute is a prime attribute. Given relational schema $R [U]$, and Σ a set of FDs over U ($R[U], \Sigma$), for every $A \in U$, $max(A) = \{ Y \subseteq U, \text{ where } Y \text{ is a maximal set such that } Y \rightarrow A \notin \Sigma^+ \}$ where max refers to a prime attribute. $X \in max(A)$ if and only if $X \rightarrow A \notin \Sigma^+$ and $XB \rightarrow A \in \Sigma^+$, for every $B \in U - XA$ such that XA is a superkey.

When transforming a database schema into a new one which is normalised it must be tested whether the transformation is correct or not. Two basic properties have been used to test their correctness: *information lossless* and *dependency preservation* (Abiteboul et al., 1995). To check this property, Bernstein (1976) developed an algorithm for producing dependency preserving 3NF which was extended later on by Biskup et al. (1979) to produce 3NF with information lossless. Figure 2.5 shows the algorithm to check the correctness of 3NF.

As presented in Figure 2.5, minimal cover Γ is a set of functional dependencies Σ that satisfies the following:

1. Every functional dependency in Σ has a single attribute for its right hand side (RHS).

2. Any functional dependency $FD X \rightarrow A \in \Sigma$ cannot be replaced with $FD Y \rightarrow A$, where Y is a proper subset of X , and there is a set of FD that is equivalent to Σ .
3. Any FD from Σ can be removed and still have a set of FD that is equivalent to Σ .

Generally we can simplify the above condition as a set of FD in a standard or canonical form with no redundancies. A partition $\Sigma_1, \dots, \Sigma_n$ of Σ is a LHS partition of Σ if no two set of Σ have the same LHS.

```

Set  $S' := \emptyset$ 

Find a minimal cover  $\Gamma$  from the set of FD

Find a LHS partition  $\Gamma_1, \dots, \Gamma_n$  of each FD in set FD

 $S' := \{(R_i[U_i], \Gamma_i) \mid U_i \text{ is the set of all attributes in } \Gamma_i\}$ 

If there is  $(R_i[U_i], \Gamma_i)$  such that  $U_i$  is a superkey

Then output  $S'$ 

Else

Determine a key  $X$  of  $U$ 

Output  $S' \cup \{R_{n+1}[X], \emptyset\}$ 

```

Figure 2.5: An Algorithm to Check the Correctness of 3NF (Abiteboul et al., 1995)

Boyce-Codd Normal Forms (BCNF)

A relational schema $(R[U], \Sigma)$ is in BCNF if and only if for every nontrivial FD $X \rightarrow Y \in \Sigma$, where X is superkey attribute, $X \rightarrow U \in \Sigma^+$. A database schema S is in BCNF if every relational schema in S is in BCNF. Using the same relational schema and FDs as in Figure 2.4, however it is not in BCNF due to presence of LHS of FD2 with $\{DR_ID, Appointment_Date\}$ attribute, which is not a superkey for the relation. BCNF requires that every LHS attributes in FD is a superkey for the relation. For example, if instance of DR_ID which is JP_2 is assigned a new room number on 1 Sept 08, two tuples have to be updated. As a consequence, the *Patient* Relation may suffer from update anomalies.

2.2.4 Normalization Process

The process of normalization was first developed by Codd (1972). Normalization is often performed by a decomposition of a relational schema so that it satisfies the requirement of a given normal form such as BCNF. The process of normalization is a formal method that identifies relations based on functional dependency among their attributes. This process is applied to each relation so that a relational schema can be normalised to a specific schema that prevents data redundancy and update anomalies in the database, and hence, reduces file storage space required.

An example of redundancy in a relational schema is shown Figure 2.4, where JP_2 appears redundantly on 1 Sept 08. On the other hand, update anomalies can be classified as insertion, deletion, or modification anomalies. In Figure 2.4, for example, to change the room number for JP_2 on 1 Sept 08, we must update two tuples. If only one tuple is updated with a new room number, this results in inconsistency of the database. As a consequence, the *Patient* Relation may suffer from update anomalies or more specifically modification anomalies. To overcome this problem, Codd (1972) informally showed how to transform a relation to generate a schema that satisfies BCNF. For instance, the relation *patient* schema shown in Figure 2.4 should be split into two new relation schemas called *Appointment* and *Dr_room* to avoid the anomalies presented above. Figures 2.6 and 2.7 show the *Appointment* and *Dr_room* relations respectively.

Patient No	Appointment Date	Appointment Time	Dr_ID
P34	1 Sept 08	10.30	JP_2
P14	1 Sept 08	12.00	JP_2
P35	1 Sept 08	12.00	Alex_3
P15	10 Sept 08	10.30	JP_2

Figure 2.6: Relation *Appointment*

Dr_ID	Appointment Date	Room No
JP_2	1 Sept 08	112
Alex_3	1 Sept 08	102
JP_2	10 Sept 08	102

Figure 2.7: Relation *Dr_room*

To test whether a given database schema satisfies a BCNF, a normalization algorithm has been developed by Beeri and Berstein (1979) as shown in Figure 2.8. This algorithm shows how to transform a given database schema into a BCNF form.

Like 3NF, the transformation of a database schema in BCNF is semantically correct if it satisfies *information lossless* and *dependency preservation* properties (Abiteboul et al., 1995). To check these properties consider the following examples. Let S_1, S_2 be two database schemas. Two instances I_1 of S_1 and I_2 of S_2 contain the same information if it is possible to retrieve the same information from them; for every query Q_1 over I_1 there exists query Q_2 over I_2 such that $Q_1(I_1) = Q_2(I_2)$, and vice versa.

Set $S' := \{(R[U], \Sigma)\}$

Repeat until S' is in BCNF

Choose a relation schema $((R'[U'], \Sigma')) \in S'$ that is not in BCNF

Choose nonempty disjoint set of attributes X, Y, Z such that

$XYZ = U'$, Σ' satisfies $X \rightarrow Y$ and Σ' does not satisfy $X \rightarrow A$, for every $A \in Z$

Replace $((R'[U'], \Sigma'))$ by $(R_1[XY], \pi_{xy}(\Sigma'))$ and $(R_2[XZ], \pi_{xz}(\Sigma'))$,

Where R_1 and R_2 are new relation names and π is a projection of set of FD over the related attributes

Figure 2.8: A BCNF Algorithm (Abiteboul et al., 1995)

Normalization algorithm tries to achieve the goal of information losslessness; if any of them transform a database schema S into a database S' , then S' should be semantically equivalent with S (Abiteboul et al., 1995). The normalization algorithm presented in Figure 2.8 takes a relational schema $S = (R[U], \Sigma)$ as input and uses a *projection* operator to transform it into a new database schema $S' = (R'[U'], \Sigma')$ in BCNF. Then, S' is a lossless decomposition of S if every instance of I of S can be transformed into an instance I' of S' by using a *projection operator*, and I can be constructed from I' by using a *join operator*. It is proved that S' is a lossless decomposition of S if and only if $\Sigma \models \bowtie[U_1, \dots, U_n]$.

Projection operator and join operator are among the core operators for query language operator used in a relational database. Normally in relational algebra, they are presented in the form π and \bowtie respectively (Abiteboul et al., 1995).

2.3 XML Documents

In this section, we review basic notions of XML documents, such as XML trees, DTDs, XML Schema and present some proposals for XML integrity constraints as well as the existing design principles for XML documents.

2.3.1 Introduction

XML is a simple and flexible text format. It allows us to model information systems in a natural and intuitive way. It was originally designed for publishing electronic data. However, today it has emerged as the standard language for storing and interchanging data on the web. XML has a number of powerful capabilities to model information (Chaudhri et al., 2003):

- *Heterogeneity*: Since in the real world, data is not actually organised into tables, rows and columns, there is an advantage for XML to express information, as it exists without restrictions, where each “record” can contain different data fields.
- *Extensibility*: New data types of data can be added when it is necessary, with no need for them to be determined in advance.
- *Flexibility*: data fields can vary in size and can be configured from time to time without any restriction on the data.

Information is modelled and designed into an XML document using for basic components: *tags*, *data elements*, *attributes*, and *hierarchy*. For example, as shown in Figure 2.9, this document contains two different types of tag: *start-tags*, such as `<course>` and `<student>`, and *end-tags*, such as `</course>` and `</student>`. These tags must be balanced and they are used to delimit elements. Every element can contain raw text, other elements, or a mixture of them. For instance, the element `<firstname> David </firstname>` contains raw text while the element `<student>` contains three sub elements:

<firstname>, <lastname> and <lecturer>. Elements can also contain attributes, such as element <student sno = "112344">. Student element contains one attribute: *sno* with the value "112344". XML documents have a nested structure. This gives a lot of flexibility when storing information. The document shown in Figure 2.9 is part of a database storing information about students.

```
<!DOCTYPE department [  
  <course>  
    <course cno = "csc101">  
    < title > XML database </title>  
    < student >  
      <student sno = "112344">  
        <firstname> David</firstname>  
        <lastname> Grey</lastname>  
        <lecturer>  
          <lecturer tno = "123">  
            <name>Bing </name>  
          </lecturer>  
        </student>  
      < student >  
        <student sno = "112345">  
          <firstname>Helen </firstname>  
          <lecturer>  
            <lecturer tno = "123">  
              <name> Bing </name>  
            </lecturer>  
          </student>  
        </course>  
      </Department>]
```

Figure 2.9: An XML Document

2.3.2 Schema Languages for Markup Languages Based Documents

Like relational database, XML also has a schema to specify the structure of XML documents such Document Type Definition (DTD) (W3C, 1998), RELAX (Murata et al., 2003), TREX (Clack, 2001), and XML Schema (XSD) (W3C, 2001). However

compared to the number of schemas we can find on the web, DTD and XSD seem the most accepted ones and they are standard schema being used currently to validate the structure of XML documents (Arenas et al., 2002; Schwentick, 2007). Hence in this section, we provide briefly the background on the DTD and XSD.

DTD

DTD was the first form of schema for XML documents that the W3C recommended in 1998 when XML was first released. DTD is a means for defining constraints on the syntax and structure of valid XML documents. An example of DTD for the XML document in Figure 2.9 is shown in Figure 2.10.

```
<!DOCTYPE department[
  <!ELEMENT department(course*)>
  <!ELEMENT course(title, student*)>
    <!ATTLIST course cno ID #REQUIRED>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT student (firstname|lastname?, lecturer)>
    <!ATTLIST student Sno ID #REQUIRED
  <!ELEMENT firstname(#PCDATA) >
  <!ELEMENT lastname(#PCDATA) >
  <!ELEMENT lecturer (name)>
    <!ATTLIST lecturer tno ID #REQUIRED>
  <!ELEMENT name (#PCDATA) >
]>
```

Figure 2.10: A DTD Describing the XML Document (Arenas and Libkin, 2004)

This DTD specifies the elements allowed in XML documents by means of ELEMENT declaration. For example, <course> is an element since <!ELEMENT course (title, student*)> appears in the DTD. An ELEMENT declaration also specifies the sub-elements of an element by means of a regular expression over the alphabet of elements. One element name is designated as the start symbol *course*. The keyword #PCDATA is

indicated as text data; it derives its name from “Parsed Character Data” for instance `<! ELEMENT first_name (#PCDATA)>`. The keyword `CDATA #REQUIRED` indicates that the attribute of an element contains character data and the value must be specified for that attribute. For instance attribute `<! ATTLIST Sno ID #REQUIRED>`. The details of DTD specification will be presented in Chapter 3, Section 3.4.

XSD

XSD defines both a type system and class of integrity constraints. Its type system subsumes DTDs. It supports a variety of simple data types (e.g., string, integer, float, double, byte), complex data types (e.g., sequence, choice) and path mechanisms (e.g., extension, restriction). For example Figure 2.11 represents the XML Schema for the XML document in Figure 2.9.

```
< xsd: element name = "course">
  <xsd: Complex Type>
    <xsd: sequence>
      <xsd: element name= "Student" type = "StudentType" minoccurs = "1"
maxOccurs = "unbounded"/>
    </xsd: sequence >
      <xsd: attribute name = "sno" type "xsd:integer"/>
    </xsd: Complex Type >
  </xsd: element>
  <xsd: Simple Type name = "studentType">
    <xsd: sequence>
      <xsd:element name = "first_name" type "xsd:string"/>
      <xsd:element name = "last_name" type "xsd:string"/>
    </xsd: sequence>
  </xsd: Simple Type name>
</xsd:schema>
```

Figure 2.11: A Fragment of an XSD for XML Documents

The above XSD consists of two kinds of types elements: simple and complex types. Simple type, indicated by `<xsd: Simple Type >` describes an element that can contain data type as *string*, *integer*, or *float* for instance element *first_name* and *last_name*. A

complex type element indicated as `<xsd: Complex Type>` describes that the element can have multiple data elements sharing the same parent in a given element. XSD permits the occurrence of the elements through cardinality, which a DTD is unable to provide. For example, in the XML document shown in Figure 2.9, the element *student* contains two elements indicated by attributes `minOccurs = "1"` `maxOccurs = "unbounded"`. A *sequence* of child elements that appear within the content of a parent attribute, for instance element *student* which consist of child elements (*first_name* and *last_name*) is indicated by `<xsd: sequence>`.

DTD verses XSD

As presented in the above section, DTD and XSD are different in syntax expressive power on attribute class, element class and data-value class. For example DTD has many drawbacks such as no modularity, no XML syntax, limited basic data types, restricted referencing mechanism and limited expressiveness. Most of these concerns have been addressed by XSD, which it can provide a rich set of data types that can be used to define the values of elements, provide much richer means for defining nested tags for instance tag with sub-tag and provide the namespace mechanism to combine XML documents with heterogeneous vocabulary (Arenas et al., 2002; Wyke and Watt, 2002).

XSD also supports namespaces and richer and more complex structures than DTDs. In addition, stronger typing constraint on the data enclosed by a tag can be described because a range of primitive data types such as string, decimals and integer are supported. This makes XSD highly suitable for defining data-centric documents. Another significant advantage is that XSD definitions can be exploited the same data management mechanism, as XSD is an XML document itself. This is in direct contrast with DTDs, which require specific support to build into an XML data management system. The descriptive power of XSD makes it suitable for XML database schema. For

instance Tamino XML server uses this concept and supports the schema description of documents via XSD.

However, XSD do have problem in consistency (Arenas et al., 2002). This problem has been studied in their research work and they proved that the semantics of XSD constraint makes the consistency analysis of a schema rather intricate. They proved that checking consistency for XSD is very hard and expensive (NP-hard and PSPACE-hard) (Arenas et al., 2002). This indicates that the current semantic of XSD is inconsistent and fail to validate documents. Even though XSD has been improved from DTD, some researchers found that it is still too complicated and not well defined (Wyke and Watt, 2002; Moller and Schwartzback, 2006). In terms of structure, DTD and XSD, are very similar. The structure of both DTD and XSD can be represented as tree models as described in the literature (Lv and Yan, 2006; Arenas and Libkin, 2004). Functional Dependencies(FDs) over XSD can also be presented as relationships between paths, the same as in DTDs (Yu and Jagadish, 2008). However, DTDs have been of more interest to the database research community, and thus we consider only DTDs since they have been used by other reserchers to address integrity constraints and design issues for XML data.

2.3.3 Basic notations

We present a formal model for XML documents and DTDs adapted from Arenas and Libkin (2004) and review some basic concept such as conformity, path in DTDs and XML documents. These notations are very important since they will be used in this thesis.

Assume that we have the following disjoint sets: *El* represent element names, *Att* represents attribute name, *str* represents attribute values and text, and *Vert* represents node identifiers. All attribute names start with the symbol @, *S* and \perp (null) are reserved symbols.

XML documents

Arenas and Libkin (2004) represented XML documents as trees. An *XML tree* is defined to be a rooted tree $T = (V, lab, ele, att, root)$, where

- (1) V is a finite set of nodes
- (2) $lab : V \rightarrow El$ assigns a label to each node of the tree
- (3) $ele : V \rightarrow str \cup V^*$ assign to each node a string or an ordered set of nodes as its children
- (4) att is a partial function of type $V \times Att \rightarrow Str$. For each $v \in V$, the set $\{@l \in Att \mid att(v, @l) \text{ is defined}\}$ is finite.
- (5) $root \in V$ is the root node tree T .

The parent-child edge relation on V , $\{v_1, v_2\} \in V \times V \mid v_2 \text{ occurs in } ele(v_1)\}$, is required to form a rooted tree. For each $v \in V$, the set of all $v' \in V$ that occur in $ele(v)$ are called subelements or children of v , and the set $\{@l \in Att \mid att(v, @l) \text{ is defined}\}$ is called attributes of node v .

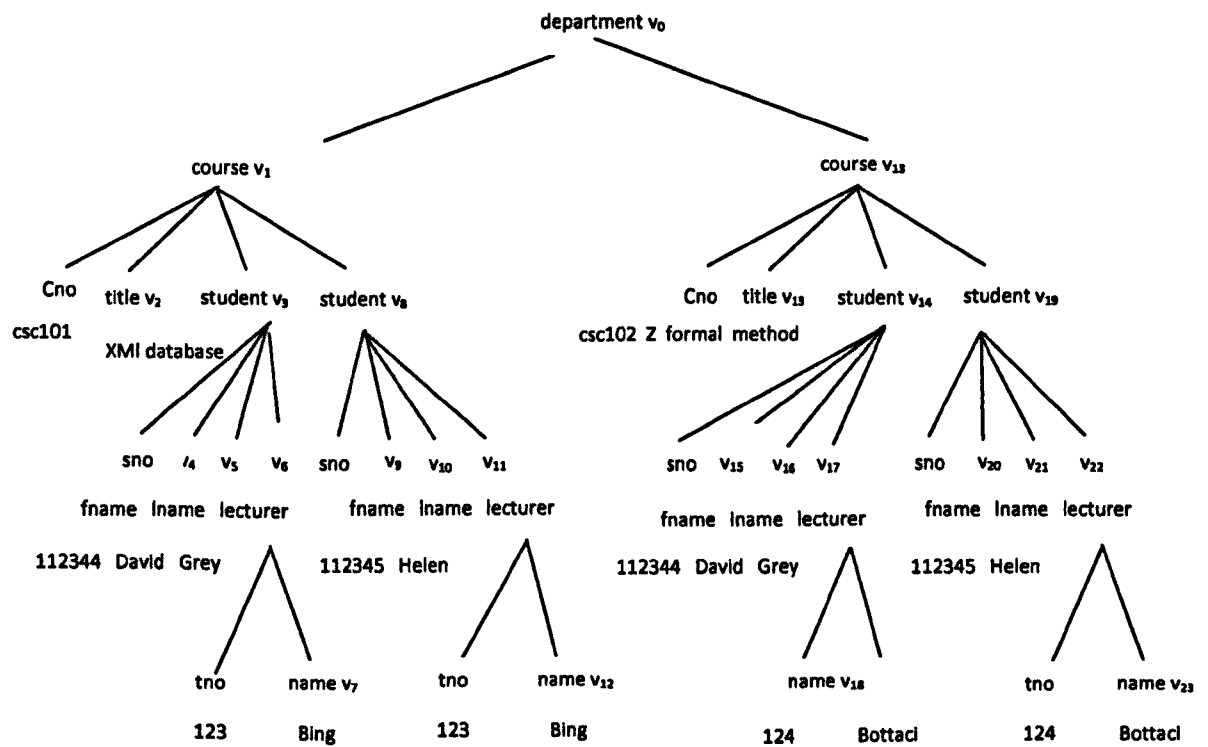


Figure 2.12: Tree Representation of XML Document

The XML tree in Figure 2.12 shows the tree representation of the XML documents shown in Figure 2.9. Note that, for simplicity we abbreviate both *first_name* and *last_name* as *fname* and *lname* respectively.

This tree contains a set of nodes $V = \{v_i \mid i \in [0, n]\}$ and v_0 is the root. Part of the functions *lab*, *ele*, and *att* is shown below:

$$lab(v_0) = \text{department}$$

$$ele(v_0) = [v_1, v_{12}]$$

$$lab(v_1) = \text{course}$$

$$ele(v_1) = [v_2, v_3, v_8]$$

$$lab(v_2) = \text{title}$$

$$ele(v_2) = \text{XML database}$$

$lab(v_3) = \text{student}$	$ele(v_3) = [v_4, v_5, v_6]$
$lab(v_4) = \text{fname}$	$ele(v_4) = \text{David}$
$lab(v_5) = \text{lname}$	$ele(v_5) = \text{Grey}$
$lab(v_6) = \text{lecturer}$	$ele(v_6) = [v_7]$
$lab(v_7) = \text{name}$	

For function *att* are defined as follows:

$att(v_1, @cno) = \text{csc101}$

$att(v_3, @sno) = 112344$

$att(v_6, @tno) = 123$

DTD

Arenas and Libkin (2004) defined a *DTD* to be $D = (E, A, P, R, r)$, where:

(1) $E \subseteq El$ is a finite set of element types.

(2) $A \subseteq Att$ is a finite set of attributes.

(3) P is a mapping from E to element type definition defined in a regular expression

$\alpha = \varepsilon | \tau' | \alpha | \alpha \cup \alpha | \alpha^*$ where ε is the empty sequence, $\tau' \in E$, and “,” , “ \cup ” and “ $*$ ” denote concatenation, union, and Kleene star, respectively.

(4) R is a mapping from E to the power set of value of $R : \mathbb{P}(A)$

(5) $r \in E$ and is called the element type of the root.

The symbols ε and S represent element type declaration EMPTY and #PCDATA, respectively.

For example, the DTD shown in Figure 2.10 is represented as follows:

$E = \{\text{department, course, student, lecturer, title, fname, lname, name}\}$

$A = \{\text{@cno, @sno, @tno}\}$

$r = \text{Department}$

Furthermore, P and R are defined as follows:

$P(\text{department}) = \text{course}^*$

$P(\text{lecturer}) = \text{name}$

$P(\text{course}) = \text{title, student}^*$

$P(\text{lname}) = S$

$P(\text{student}) = \text{fname, lname, lecturer}$

$P(\text{fname}) = S$

$P(\text{name}) = S$

$R(\text{department}) = \emptyset$

$R(\text{course}) = \{\text{@cno}\}$

$R(\text{student}) = \{\text{@Sno}\}$

$R(\text{lecturer}) = \{\text{@tno}\}$

$R(\text{fname}) = \emptyset$

$R(\text{lname}) = \emptyset$

Conformity

The notion of conformity of a DTD with XML tree is defined as follows (Arenas and Libkin, 2004):

Given a DTD $D = (E, A, P, R, r)$ and XML tree $T = (V, \text{lab}, \text{ele}, \text{att}, \text{root})$, T is said to conform to D , denoted $T \models D$, if

(1) *lab maps every node in V to E .*

(2) *for every element node $v \in V$, if $\text{ele}(v) = (v_1, \dots, v_n)$ then the sequence $\text{lab}(v_1), \dots, \text{lab}(v_n)$ is regarded as string S which is defined by $P(\text{lab}(v)) = S$*

(3) *att is a partial function from $V \times \text{Att}$ to Str . For each $v \in V$, and $\text{@}l \in A$, $\text{att}(v, \text{@}l)$ is defined iff $\text{@}l \in R(\text{lab}(v))$.*

(4) *lab(root) = r*

For example, the XML tree in Figure 2.12 conforms to the DTD shown in Figure 2.10.

Paths in XML Documents and DTDs

The notion of *path* is used to navigate and query XML trees and is also used to define constraints for XML. Given an XML tree $T = (V, lab, ele, att, root)$, a path in T is a string $w = w_1 \dots w_n$, with $w_1 \dots w_n \in El$ and $w_n \in El \cup Att \cup \{s\}$, such that there are nodes v_1, \dots, v_{n-1} in V with labels $w_1 \dots w_{n-1}$, respectively, such that (Arenas and Libkin, 2004):

(1) v_{i+1} is a child of v_i , $i \in [1, n-2]$,

(2) If $w_n \in El$, then there is a child v_n with label w_n . If $w_n = @l$ is an attribute in Att , then $att(v_{n-1}, @l)$ is defined. If $w_n = S$, then v_{n-1} has a child in Str .

The set of all paths in a tree T that start from the root is denoted by $path(T)$. For example, *course.student*, *course.student.@sno*, *course.student.lname*, *course.student.lname.S* all path in XML T is shown in Figure 2.12 and DTD in Figure 2.10.

Paths can also be defined for DTDs. Given a DTD $D = (E, A, P, R, r)$, a path in D is a string $w = w_1 \dots w_n$ such that w_i is in the alphabet of $P(w_{1-i})$ for $i \in [2, n-1]$, and w_n is either an attribute ($@l \in R(w_{n-1})$) or is in the alphabet of $P(w_{n-1})$.

Paths are an important component of XML, as they have been used as one of the basic languages for navigating and querying XML documents (Arenas and Libkin, 2004). The available query languages that use path and are used currently are Xpath (Clark and Derose, 1999) and XQuery (W3C, 2007). Generally, the semantic to retrieve a particular data in a tree using a path is as follows. Given an XML tree T , a node v of T , and a path w in T , *retrieve* (v, w) is defined to be the set of all nodes and values in T reached by following w from v in this tree. Thus, for example the result of retrieving a particular node in a XML tree in Figure 2.12, is as follows.

retrieve (v_n , department, course, student, lname) = { $v_5, v_{10}, v_{16}, v_{21}$ }

retrieve (v_n , department, course, student, lname.S) = { Grey, \emptyset , Grey, \emptyset }

retrieve (v_n , department, course, student) = { v_3, v_8, v_{14}, v_{19} }

2.3.4 Functional Dependencies for XML (XFD)

As in the case of a relational database, the design of an XML database is also guided by integrity constraints or data dependencies. Several classes of integrity constraints have been extended and defined for XML including key dependencies, inclusion dependencies, equality-generating dependencies, path constraints, join dependencies, functional dependencies and multi-valued dependencies. To the best of our knowledge, however, the most prominent form of data dependency in XML is functional dependency since it has been most widely studied in the literature on the XML database design. This is because of its ubiquity, simplicity and application to database design as well as to maintaining data integrity (Wang and Topor, 2005; Arenas and Libkin, 2004).

Although functional dependency (FD) has matured and proved to be a useful class of integrity constraint for relational database, however the principles and systematic theory for XML data are still in the infancy stage. This is because XML is new compared to relational database, and there are many differences between relational schemas and XML schemas in their structure: relational models are flat and structured while XML data are nested and unstructured. Due to this, several different definitions of XML functional dependency (XFD) have been put forward by Lee et al., (2002), Arenas and Libkin (2004), Vincent et al., (2004), Hartmann and Link (2003), Wang and Topor (2005), and Yu and Jagadish (2008). Generally, these definitions differ from each other in term of the method of choosing sub-trees, path identifier or value equality to describe the relationship between elements and attributes in XML database. Most of them define a functional dependency as an expression of the form $p_1, \dots, p_2 \rightarrow q$, where p_1, \dots, p_2, q are path expressions.

Here, to demonstrate the XFDs definition, reconsider Figure 2.12 which illustrates an example of XML document. The example shows a university department which offers different courses. Each course consists of course number (cno), its title and the list of students taking the course. The XML tree of Figure 2.12 satisfies the following constraints.

Constraint 1: Any two lecturers, with the same *tno* values will have the same name

(2.1)

Constraint 2: For any specific course, no two students can have the same value of attribute *@sno*

(2.2)

Constraint 1 involves a *single element* such as a lecturer with the same *tno* value (e.g. value *tno* = '123') will have the same name (e.g. node element v_7 and v_{12}). Meanwhile constraint 2 involves a *multiple hierarchies* which involve a node *course* element (v_1) that is an ancestor for a node *student* element (v_3). For instance, for *course* with 'title = XML database' it cannot have a *student* with same *sno* (e.g. *sno* = 112344 and *sno* = 112345).

Two main approaches have been followed in order to define these XML functional dependencies (XFDs). In the first approach, XFDs are defined based on path identifiers based on schema (DTD or XSD) (Lee et al., 2002; Arenas and Libkin, 2002; Vincent et al., 2004; Schewe, 2005; Yu and Jagadish, 2008), while in the second one XFDs are defined based on sub-trees (Hartmann and Link, 2003).

In this section, we compare XFDs definition using both approaches and discover their strengths and limitations. In the following section, for brevity, we abbreviate the XFDs defined by Lee et al., (2002), Arenas and Libkin (2004), Vincent et al. (2004), Hartmann and Link (2003), Wang and Topor (2005), and Yu and Jagadish (2008) as XFD 02, XFD 04(a), XFD 03, XFD 04(b), XFD 05, and XFD 08 respectively. We present both approaches next.

Path Based Approach

XFD 02

Lee et al. (2002) informally proposed XFD notation and DTD for XML by considering the path nature of an XML database. They used the notation of Xpath to define XFD. We present their definition next.

Definition XFD 02: *is an expression of the form $(Q, [P_1, \dots, P_n \rightarrow P_{n+1}])$, where Q is a path starting from the XML document root which defines the scope in which the constraint holds, and P_i for $i \in (1, \dots, n+1)$ is either an element or an element followed by dot and a set of key attributes of the element.*

An XML tree is to satisfies the XFD 02 if for any two sub trees rooted at a node in $\text{root}(Q)$, if they agree on the value of P_1, \dots, P_n , they also agree on the value P_{n+1} , provided these values exist (Lee et al., 2002).

Based on the above definition, constraint 1 in (2.1) can be expressed by XFD-02 as follows:

XFD1: $(\text{department.course.student}.[\text{lecturer.tno} \rightarrow \text{name}])$ (2.3)

Constraint 2 in (2.2) cannot be expressed and satisfied by XFD 02 because relative constraints (that only hold in a part of the document) are not considered. Thus the semantics only work properly if some syntactic restrictions are imposed on the XFD (Lee et al., 2002). Furthermore, no exact definitions for the value of an element, set elements and null values are provided.

XFD 04(a)

To overcome the above limitation, Arenas and Libkin(2004) formally defined XFD 04(a) by considering a relational representation of XML documents. They proposed two types of constraint: *relative constraint* and *absolute constraint*, that hold in the entire document. They extend the notion of tuple for relational databases and use the concept

of path particularly *tree-tuples* to the definition. The *tree-tuples* map between a set of paths on a *DTD* to a set of node (values) in an XML tree. They assumed a *DTD* as a single relation schema, a set of all paths in *DTD Paths(D)* as an attribute, and a *tree tuple* is all tuples in that relation. This concept is adopted from the concept of total unnesting of a nested relation (Atzeni and DeAntonellis, 1993). To present this definition, we need to introduce an example of a *tree-tuple*. Consider the *DTD* in Figure 2.10 and XML tree in Figure 2.12. This tree contains four tree tuples. One of tree tuple can also be represented as a function as follows.

$t(\text{department}) = v_0$

$t(\text{department.course}) = v_1$

$t(\text{department.course.@cno}) = \text{csc101}$

$t(\text{department.course.title}) = v_2$

$t(\text{department.course.title.S}) = \text{XML database}$

$t(\text{department.course.student}) = v_3$

$t(\text{department.course.student.@sno}) = 112344$

$t(\text{department.course.student.fname}) = v_4$

$t(\text{department.course.student.fname.S}) = \text{David}$

$t(\text{department.course.student.lname}) = v_5$

$t(\text{department.course.student.lname.S}) = \text{Grey}$

$t(\text{department.course.student.lecturer}) = v_6$

$t(\text{department.course.student.lecturer.@tno}) = 123$

$t(\text{department.course.student.lecturer.name}) = v_7$

$t(\text{department.course.student.lecturer.tname.S}) = \text{Bing}$

In a *tree tuple*, each path which ends with an element name is mapped to a distinct node or the null value (\perp), and every other path ending with an attribute name or S is mapped to either a string (PCDATA) or \perp .

Definition XFD 04(a): *XFD over D is an expression in the form $P_1 \rightarrow P_2$ where P_1, P_2 are finite non-empty of path in $Paths(D)$. An XML documents conforming to the DTD is said to be satisfy the XFD 04(a) if for every two tree-tuples t_1 and t_2 , $t_1.P_1 = t_2.P_1$, and $t_1.P_1$ is not null and imply $t_1.P_2 = t_2.P_2$ (Arenas and Libkin 2004).*

Generally, the semantic of this XFD is defined as the following: for any two tree tuples, which consist of a set of paths on LHS of the XFD and a single path on the RHS, they are defined to be satisfied if whenever two tree tuples agree on the LHS path, they must also agree on the path of the RHS.

Using the above definition, the constraints 1 and constraint 2 can be expressed as follows:

XFD1 : department.course.student.lecturer.@tmo \rightarrow

department.course.student.lecturer.name.S (2.4)

XFD2: department.course.deparment.course.student.@sno \rightarrow *department.course.student*

(2.5)

XFD 04(b)

Alternatively, Vincent et al. (2004) defines an XFD 04(b) close to XFD 04(a) definition with a few significant differences. Firstly, they did not use either DTD or XSD but consider a scheme file. Secondly, they used the concept of “*closest path*” to link values in both sides of XFD. *Path* is of the same as path in XFD 04(a) and *closed* means that if a path is in the set, then all prefixes of the path are also in the set. Lastly, they labelled every node in V (not just element nodes) and allow for mixed context in XML document. Mixed context is an element node that can contain both text and element nodes as children.

Definition XFD 04(b): An XFD 04 is an expression of the form $p_1, \dots, p_n \rightarrow q$, where p_1, \dots, q are paths in P and $n \geq 1$. (Vincent et al., 2004)

XFD1: department.course.student.lecturer.@tno →
department.course.student.lecturer.name.S
(2.6)

XFD2: department.course, deparment.course.student.@sno → *department.course.student*
(2.7)

The notion of strong satisfaction of an XFD by an XML tree is defined using the agreement path instances in the tree on the paths in the XFD. The major difference between XFD 04(a) and XFD 04(b) lies in the treatment of null values or missing nodes. When null values exist, the satisfaction of XFD 04(b) over XML tree is stronger than XFD 04(a). However, when there is no missing information in the XML document, the definition of Vincent et al. (2004) coincides with the definition of Arenas and Libkin (2004).

XFD 05

On the other hand, Wang and Topor (2005) defined XML tree differently from Lee et. al. (2002), Arenas and Libkin (2004) and Vincent et al. (2004) since they explicitly distinguished *complex and simple element* so that special text (S) node under simple element is not required such that $E = E_1 \cup E_2$. Simple element (E_1) is an element that only has a single value that is the same as an attribute and complex element (E_2) is an element that has sub elements and/or attributes. Furthermore they also made the ordering of child elements insignificant by treating them as a set rather than a sequence. For example, in Figure 2.12, title and name are simple elements while department, course and student are complex elements.

Wang and Topor also defined a path in an XML tree more precisely by distinguishing paths into four types: *simple path*, *downward path*, *upward path* and *composite path*. These four types of path are subclasses of Xpath as well.

A *simple path* is of the form $l_1 \dots l_m$, where l_i is a simple element and l_m can be a simple element, complex element or attribute. The number of *labels* (element names and attribute names) in a simple path is called the length of the path. A simple path with length 0 is called an *empty path*, denoted ϵ .

A *downward path* is a simple path which can be expressed as $l_1 \dots l_n$ where ($l_i \in E_1 \cup E_2 \cup A \cup \{_, -^*\}$ for $i \in [1, n-1]$, $l_n \in E_1 \cup E_2 \cup A$). The symbol $_$ represents wildcard (which can be matched with any label) and $-^*$ represent Kleene closure of the wildcard.

An *upward path* is of the form $\uparrow \dots \uparrow$ and *composite path* is of the form $\xi.p$, where ξ is an upward path, and p is a simple path.

In order to provide XFD 05 definition, we presented *path equality* based on agreement of two nodes. This agreement can be defined as (Wang and Topor, 2005):

- n_1 and n_2 node agree (*N-agree*) on path p if p is a *simple path* or p is *upward path*
- n_1 and n_2 is set agree (*S-agree*) on path p if for every node v_1 in $n_1[p]$, there is a node v_2 in $n_2[p]$ such that $v_1 = v_2$ (value equality) and vice versa.
- n_1 and n_2 is intersect agree (*I-agree*) on path p if there exist nodes $v_1 \in n_1[p]$ and $v_2 \in n_2[p]$ such that $v_1 = v_2$

For example, in Figure 2.12, v_3 and v_8 are *N-agree* on upward path.

Definition XFD 05: A XFD is an expression of the form $Q: p_1(c_1), \dots, p_n(c_n) \rightarrow p_{n+1}(c_{n+1})$ where Q is a downward path, p_1, p_2, \dots, p_n are simple paths, p_{n+1} is a simple path of length 1 or 0, and c_i ($i = 1, \dots, n+1$) is one of *N*, *S* and *I*. (Wang and Topor, 2005).

XFD 05 satisfies an XML tree if for any two nodes $n_1, n_2 \in \text{root}(Q)$, if every path agrees with of *I-agree*, *N-agree* and *S-agree* for all other paths.

For example, constraint 1 and constraint 2 are satisfied and expressed as follows:

$$\text{department.course.student.lecturer} : @tno \rightarrow \text{name (N)} \quad (2.8)$$

$$\text{department.course, department.course.student.@sno} \rightarrow \text{department.course.student (N, I)} \quad (2.9)$$

XFD 05 overcomes the limitation of XFD 02(a), XFD 02(b) and XFD 04 by unifying and generalising them by defining more semantic constraints such as *set elements*. For example reconsider again the same example but a set of address is added for a lecturer as in Figure 2.13.

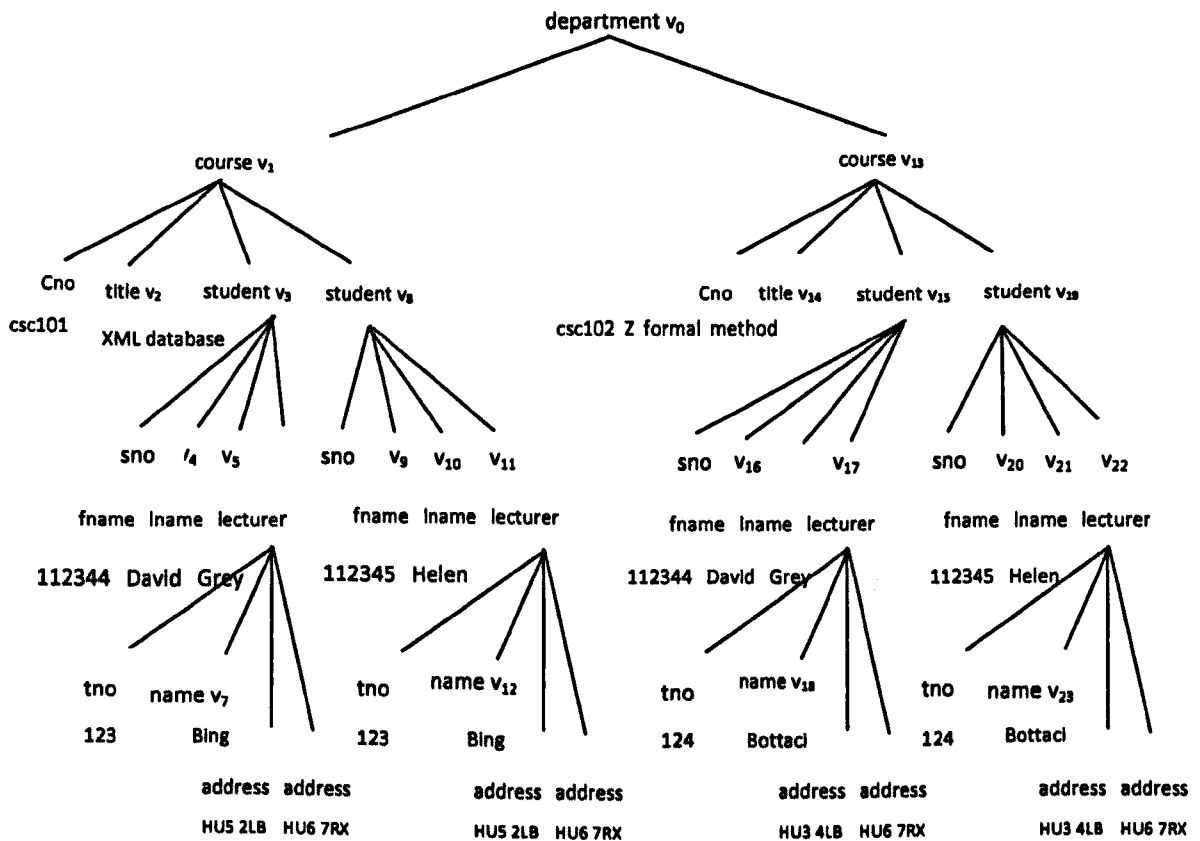


Figure 2.13: Tree representation of an XML Document

For constraint XFD3:

Lecturer number (tno) determines the lecturer's set of addresses satisfied by the XML tree in Figure 2.13 and it can be represented using XFD 05 as below.

department.course.student.lecturer : @tno → address(S) (2.10)

This is because XFD 05 determines and defines the path of the element node specifically. This is an advantage of XFD 05 since it can be used to capture more semantic constraints, hence it can detect more data redundancies in XML documents. However in defining the XFD 05, Wang and Topor did not use any specific XML schema to associate with XML documents. We believe it is very important to use DTD or XSD in defining XFD since it can determine whether the XFD is significant or consistent with an XML document.

XFD 08

Recently, Yu and Jagadish (2008) have taken the same approach with Wang and Topor (2005) by incorporating set of elements to define XFD 08. They named the notion as Generalized Tree tuple (GTT FD). GTT FD is similar to the original *tree tuple* (Arenas and Libkin 2004); however it has an extra parameter called *pivot* node. The *pivot* node can be used to prevent separation between sibling nodes in the same path and preserve the ancestor node and descendant nodes of that pivot node. For instance, in Figure 2.13, node *student* (v3, v8, v14, v19) are *pivot* nodes, node *course* (v1, v13) are an ancestor and *sno*, *fname*, *lname* and *lecturer* are descendant nodes for the node *student*.

Yu and Jagadish used XSD instead of DTD as a schema for XML documents and used Xpath notation to express the path in XML Schema. Figure 2.14 illustrates the schema of the XML Document in Figure 2.13.

```

department :Rcd
  course : setOfRcd
    cno: Str
    title : str
    student: SetOfRcd
      sno:str
      fname:str
      lname : str
      lecturer: setOfRcd
        tno:str
        name: str
        address: setOfRcd

```

Figure 2.14: XSD Model

Apart from that, the author also associated the notion of XML key with XFD. The notion of XML key has common similarities with the notion proposed by Buneman et al. (2003) which contains a target path (which identifies a set of nodes) and a set of key paths. We present the notion of GTT and key adopted from Yu and Jagadish (2008) before we present the definition of XFD 08.

A GTT of XML Tree $T = (N, P, V, n_r)$, with regard to a particular data element n_p (called pivot node), is a tree $t_{n_p}^T = (N_t, P_t, V_t, n_r)$, where:

1. $N_t \subseteq N$ is the set of nodes
2. $P_t \subseteq P$ is the set of parent-child edges;
3. $V_t \subseteq V$ is the set of value assignment
4. n_r is the same root node in both $t_{n_p}^T$ and T ;
5. $n \in N_t$ if and only if i) n is a descendant or ancestor of n_p in T or n is a non – repeatable direct descendant of an ancestor of n_p in T

Definition XFD 08: An XFD 08 is a triple (C_p, LHS, RHS) , is an expression of the form: $LHS \rightarrow RHS$ w.r.t C_p , where C_p denotes a pivot tuple class, LHS is a set of paths for left hand side relative to p and RHS is a single path of RHS relative to p (Yu and Jagadish, 2008).

For an XML tree, T satisfies an XFD-08 if and only if for any two generalised tree tuples $t_1, t_2 \in C_p$ if:

1. exist $t_1.P_{ii} = \perp$ or $t_2.P_{ii} = \perp$ where $i \in [1, n]$
2. all path values are equal between $t_1.P_{ii} = t_2.P_{ii}$, then $t_1.P_r \neq \perp$ or $t_2.P_r \neq \perp$, $t_1.P_r = t_2.P_r$

Based on the above definition, constraint 1 and constraint 2 are satisfied and can be expressed as follows:

XFD1: $department.course.student.lecturer.tno \rightarrow$

$$department.course.student.lecturer.name \text{ w.r.t } C_{lecturer} \quad (2.11)$$

XFD2: $department.course, department.course.studnet.sno$

$$\rightarrow department.course \text{ w.r.t } C_{student} \quad (2.13)$$

XFD3: $department.course.student.lecturer.tno \rightarrow$

$$department.course.student.lecturer.name.address \text{ w.r.t } C_{lecturer} \quad (2.14)$$

Generally the advantage of XFD 08 compared to XFD 02 XFD 04(a), XFD 04(b) is, it allows more flexible notions to express the multiple set of *tree tuples* within each set instead of a single element. This is because the elements are only separated if they are not descendants of the pivot nodes.

Sub Tree Based Approach

XFD 03

Hartmann and Link (2003) took a different approach to define XFD. They defined XFD 03 based on *sub trees* of the XML data tree. For the schema for XML documents, they used a *schema graph* instead of DTD or XSD. The schema graph is an XML tree T together with an edge assigned a frequency of either 1 or *. In a schema graph, no two descendants of a node can have the same type and label while in a data tree, every leaf node is assigned a value. Figure 2.15 shows the schema graph corresponding to the XSD in Figure 2.14.

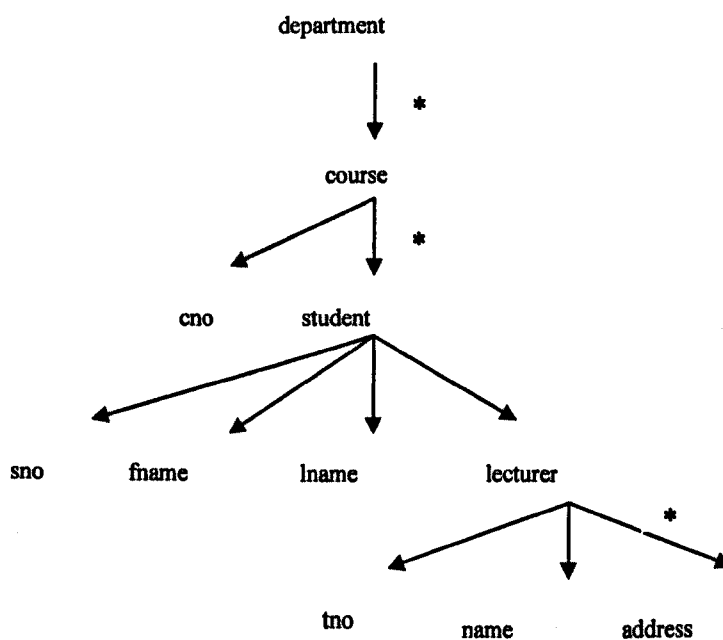


Figure 2.15: Scheme Graph

XFD 03 is defined using *homomorphism*, *v-subtrees* and *isomorphism* of XML trees (Hartmann and Link, 2003). A homomorphism between two trees T and G is a mapping \emptyset from nodes of T to the nodes of G such that

- (1) the root of T is mapped to the root of G , $\emptyset(\text{root}_T) = \text{root}_G$

- (2) image node v in T is the same kind as the node itself, $kind(v) = kind(\phi(v))$ for all nodes in T
- (3) image of node carries the same name as the node itself, $name(v) = name(\phi(v))$ for nodes in T
- (4) every edge of T is mapped to edge of G , $(v, w) \in V_G$ implies $(\phi(v), \phi(w)) \in V_G$

A v -subgraph is a sub graph which roots at node v and it is determined by the paths from v to a given subset of leaves of the original tree. The *isomorphism* between the two subtrees is a one to one mapping between the two sets of nodes which is homomorphism in both directions. Two XML trees are equivalent if there is an isomorphism between them and the $value(v) = value(\phi(v))$ for the leaf nodes.

Definition XFD 03: An XFD 03 is defined to be in the form of an expression $v: X \rightarrow Y$, where v is a node of T , and X and Y are v -subtrees of T . An XML tree conforms to the schema graph if it satisfies the following (Hartmann and Link, 2003):

- (1) Every two pre- images W_1 and W_2 of subtree $T(v)$ rooted at v , are equivalent for each image of the leaves of X and Y
- (2) A maximal subcopy of $U(v)$ is a sub tree of T which is isomorphic to a subtree of $U(v)$

For example constraint 1 and constraint 3 are expressed as follows:

XFD1: XFD $v_{lecturer}$: $X \rightarrow Y$, where X is the $v_{lecturer}$ - subgraph with the single leaf v_{tno} , while Y is the $v_{lecturer}$ -subgraph with the single leaf v_{name} (2.14)

XFD3: XFD $v_{lecturer}$: $X \rightarrow Y$, where X is the $v_{lecturer}$ - subgraph with the single leaf v_{tno} , while Y is the $v_{lecturer}$ -subgraph with the set of leaves $v_{address}$

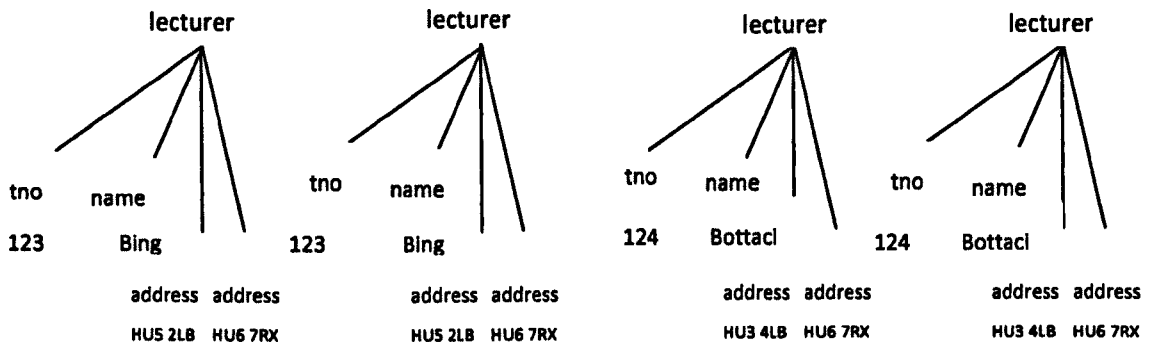


Figure 2.16: Four Pre-Images of the $\nu_{lecturer}: X \rightarrow Y$

As shown in Figure 2.16, the first and second, third and fourth sub graphs satisfy the XFD 03 since their pre-images are equivalent respectively.

Generally, the approaches taken by Yu and Jagadish (2008) and Hartmann and Link (2003) are almost similar, as they preserve all the element nodes under descendant nodes. However, the limitation of XFD 03 is that, it cannot define constraint 3 which involves multiple hierarchies' constraint. Since the previous definitions of XML FDs (Arenas and Libkin 2004; Vincent et al., 2004) use path expression, they are not able to express functional dependencies of this kind.

2.3.5 Inference Rules for XML Functional Dependencies

Inference Rules or Implication problems have been studied thoroughly in regards to relational databases. In section 2.2.1.4, we have presented inference rules for FDs and MVD for relational database (Beeri et al., 1977). In XML database context, to best of our knowledge, only Arena and Libkin, Vincent et al. and Yu and Jagadish studied this problem for XFD. This problem is very important in the normalization process because it will determine a non-redundant database schema (Arenas and Libkin, 2004; Vincent et al., 2004; Schewe, 2005; Hartmann and Link, 2004). Recently, Yu and Jagadish have

derived the following inference rules to compute the closure of XFD which is similar to the Armstrong Rules in the relational case (Yu and Jagadish, 2008). The only difference is the way it was presented. Yu and Jagadish(2008) used a path notation. We present them next.

Rule 1 (Reflexivity) $LHS \rightarrow P_1$ w.r.t. C_p is satisfied if $P_1 \subseteq LHS$.

Rule 2 (Augmentation) $LHS \rightarrow P_1$ w.r.t. C_p then $\{LHS, P_2\} \rightarrow P_1$ w.r.t. C_p .

Rule 3 (Transitivity) $LHS \rightarrow P_1$ w.r.t. $C_p \wedge \dots \wedge LHS \rightarrow P_n$ w.r.t. $C_p \wedge \{P_1, \dots, P_n\} \rightarrow P$ $C_p \Rightarrow LHS \rightarrow P$ w.r.t. C_p .

Yu and Jagadish (2008) have argued that deriving XFD inference rules formally from existing set of XFDs is very difficult and thus they have proposed an algorithm called *DiscoveryXFD* to detect all XFDs automatically.

Arenas and Libkin also investigated implication problems for XFD; in fact the XNF decomposition algorithm in Figure 2.18 involves XFD implication to test the membership in $(D, \Sigma)^+$. These XFDs implication have been tested on two classes of DTD: Simple Regular Expression DTD and Disjunctions DTD. Thus, they proved that the implication problem for a simple DTD can be solved in quadratime time and the implication problem for disjunctive DTDs can be solved in polynomial time (Arenas and Libkin, 2004). However they did not state clearly the inference rules for XFD in their work. In this work, we will not consider this problem.

2.3.6 Data Redundancies and Anomalies for XML Documents

Given a schema and a set of data dependencies, the goal is to refine the schema into a better schema so that update, insertion, or deletion anomalies are eliminated. We have examined many DTDs and XSDs on the Web and found many data redundancies appeared in the schema. As a first example, consider Figure 2.12 which illustrates the example DTD for an XML document (Arenan and Libkin, 2004). The example of constraint 1 in section 2.3.4 caused the information in the XML document to become

redundant. For instance, for *tno* = '123', the lecturer named *Bing* who teaches the *course number* (*cno*) *csc101* is stored twice, causing redundancy in XML document. As with a relational database, if we would like to update the name of the lecturer to full name "Bing Wang" in the XML document, this name needs to be updated twice. This type of update is called update anomalies.

Consider another real example for the DTD below. This is a part of a DBLP database for storing data about conferences (Ley, 2002)

```
<!DOCTYPE db [  
  <!ELEMENT db (conf*)>  
  <!ELEMENT conf (author, issue+)>  
  <!ELEMENT author(#PCDATA)  
  <!ELEMENT issue (inproceedings+)>  
  <!ELEMENT inproceedings (pages+, year)>  
  <!ATTLIST inproceedings  
    Key ID #REQUIRED  
    pages CDATA #REQUIRED  
    year CDATA #REQUIRED >  
>
```

Figure 2.17: DTD for DBLP Database

The DTD shown in Figure 2.17 describes that each conference has an author, and one or more issue. Papers are stored in the in proceedings element which consists of key, pages and year as its attributes. The document contains the following constraint: Any two **in proceedings** children of the same **issue** must have the same value of **year**. This constraint is considered relative to the **issue** element only. The constraint leads to redundancy since the value of **year** is stored redundantly for a conference.

As with a relational database, such data redundancies and anomalies presented above can be avoided by designing a non-redundant XML database schema. Those redundancies could be eliminated if we could refine the original schema to a new

schema by eliminating some XFDs from XML documents. Thus the concept of normalization theory is applied to improve the XML schema design for an XML database in order to use it efficiently. In this section, we present the XML normal form that has been defined currently.

2.3.7 Normal Forms for XML Documents

In section 2.2, 3NF and BCNF were presented for designing a relational database. If a relational schema satisfies these normal forms, then the relations on the schema are well designed. It is well known that BCNF is able to remove data redundancy caused by FDs; however, it is not dependency preserving. On the other hand, 3NF is dependency preserving but it not able to remove redundancy caused by FDs in all cases. This information is very important for the database designer to design a non-redundant database. In the context of an XML database, the same knowledge is needed to guide the designer for producing well designed XML schema (DTD or XSD). Due to this, several normal forms for XML documents have been defined in order to provide a well designed schema (Arenas and Libkin, 2004; Vincent et. al., 2004; Wang and Topor, 2005; Kolahi and Libkin, 2006; Yu and Jagadish, 2008). These normal form definitions differed in terms of schema used and constraints description, but most of them are based on Arenas and Libkin's proposal, because it is the most fundamental.

Hence, the main purpose of this section is to compare these XML normal forms with respect to their definitions and normalization algorithm to reduce data redundancy. To distinguish between them we will use the following notation: XNF 04(a), XNF 04(b), X3NF and XNF 08 introduced by Arenas and Libkin (2004), Vincent et al., (2004), Kolahi (2007) and Yu and Jagadish (2008) respectively. In the following discussion, we will the use constraints 1, constraint 2 and constraint 3 presented in section 2.3.4 and expresse them using XFD1, XFD2 and XFD3 respectively. We present them next.

XNF 04(a)

Arenas and Libkin (2004) followed the standard BCNF and a nested form NNF-96 (Mok et al., 1996) to define XNF 04. A nested form NNF-96 is normal form defined for a nested relational database.

We now present XNF 04(a)

Definition XNF 04(a): Given a DTD D and a set of Σ of FDs over D , $(D, \Sigma)^+$ is in XML normal form (XNF) if and only if for every nontrivial XFD $S \rightarrow p.@l$ or $S \rightarrow p.s$ or, the XFD $S \rightarrow p$ is also in $(D, \Sigma)^+$. Where S is a subset of path (D), p is a set of path (D) (Arenas and Libkin, 2004).

This normal form generalises BCNF for XML documents and disallows any redundancy caused by XFD. For example, the DTD shown in Figure 2.10 is not in XNF since the following XFD1 is not XNF.

department.course.student.lecturer.@tno \rightarrow *department.course.student.lecturer.name.S*

This XFD1 is not in XNF because it does not imply the functional dependency

department.course.student.lecturer.@tno \rightarrow *department.course.student.lecturer.name*

and is not in the set of closure $(D, \Sigma)^+$. The functional dependency XFD1 here leads to redundancy where *name* occurs multiple times for a lecturer. This kind of functional dependency is called *anomalous* XFD over (D, Σ) . Arenas and Libkin (2004) proposed an XNF decomposition algorithm to eliminate the above *anomalous* XFD and transform a DTD D and set of XFDs Σ into a new specification (D', Σ') that is XNF. We present this algorithm using the illustrated examples.

(1) IF (D, Σ) is in XNF then return (D, Σ) , otherwise go to step (2).

(2) / * moving attribute*/

If there is an anomalous XFD $q \rightarrow p.@l$ and $q \in \text{paths}(D)$ such that $q \rightarrow X \in (D, \Sigma)^+$ then:

2.1 Choose an attribute $@m$

2.2 $D := D[p.@l := q.@m]$

2.3 $\Sigma := \Sigma[p.@l := q.@m]$

2.4 Go to step (1)

(3) / * create new element types*/

Choose a (D, Σ) – minimal anomalous XFD $\{q, p_1.@l_1, \dots, p_n.@l_n\} \rightarrow p.@l$

(3.1) Create fresh element types $\tau, \tau_1, \dots, \tau_n$

(3.2) $D := D[p.@l := q. \tau[\tau_1.@l_1, \dots, \tau_n.@l_n @l]]$

(3.3) $\Sigma := \Sigma[p.@l := q. \tau[\tau_1.@l_1, \dots, \tau_n.@l_n @l]]$

(3.4) Go to step (1)

Figure 2.18: XNF Decomposition Algorithm (Arenas and Libkin, 2004)

Figure 2.18 shows the XNF Decomposition Algorithm. The input of this algorithm is a set schema D contains an anomalous XFD corresponds to relative constraint or minimal anomalous XFD corresponds to absolute constraint. The definition of minimal anomalous XFD is similar to relational database context (see section 2.1.2.1) but is more complex because of the used path in XFD. The output of this algorithm is a new XML schema specification (D', Σ') that is in XNF which contains the same information.

Moving Attribute

Consider the following anomalous XFD satisfied for the DTD in Figure 2.18 that causes data redundancy.

$$db.conf.issue \rightarrow db.conf.issue.inproceeding.@year \quad (2.11)$$

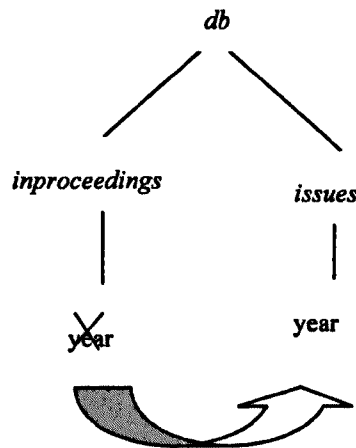


Figure 2.19: Moving Attribute

As illustrated in Figure 2.19, to eliminate the anomalous XFD, the attribute $@year$ from the set of attributes of **inproceedings** is moved to the new attribute **issue**. After transforming DTD (D) into a new DTD(D'), a new set of XFD (Σ') is generated and the set of XFDs does not include this anomalous XFD (2.11) and thus, contains only reduced anomalous path (Arenas and Libkin, 2004). However, this algorithm only restructures the DTD but it does not decompose the original DTD.

Create New Element Type

Consider the university database shown in Figure 2.13, which contains the following minimal anomalous XFD1.

$$department.course.student.lecturer.@tno \rightarrow department.course.student.lecturer.name.S \quad (2.12)$$

To remove the above XFD1 in (2.12), the following steps are needed:

- (i) a new element type *lecturer_info* as a child of *Department* is created.

- (ii) Then *name.S* from lecturer element is made an attribute for *lecturer_info*
- (iii) Make *tno* as a child of *lecturer_info* as its attribute where the original @tno attribute of lecturer remains the same.

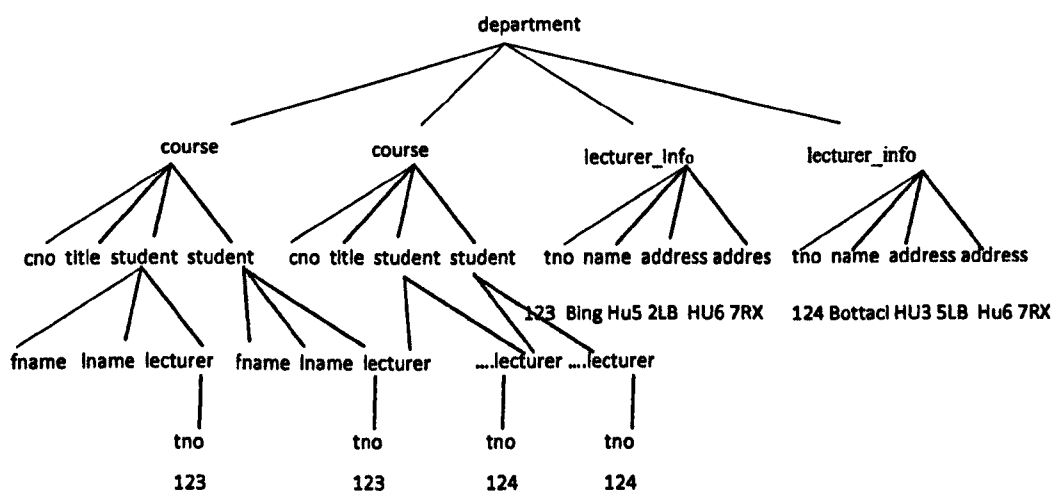


Figure 2.20: Create New Element

This rule will be repeatedly applied until all anomalous XFDs are removed from the DTD. The restructured version of the XML document that reflects this decomposition is illustrated in Figure 2.20. Arenas and Libkin (2004) tested the complexity of XNF for simple DTD and relational DTDs which are in cubic time and are coNP-complete respectively. This XNF decomposition algorithm can transform any DTD to the XNF normal form without loss of any information from the original XML documents. This is achieved if there is a mapping f from paths in DTD D' to paths in the DTD D such that every tree T conforms to schema specification (D, Σ) , there is a tree T' that conforms to schema specification (D', Σ') such that T and T' agree on all paths with respect to this mapping f (Arenas and Libkin, 2004). They used a classical information theory approach (Cover and Thomas, 1991) namely entropy to measure the effectiveness of a XML database design and proved that their XNF is equivalent to BCNF in the relational environment and XML documents at the instance level. In this approach the authors

measured the amount of redundancy for a schema regardless of any query/update language (Arenas and Libkin, 2005).

XNF 04(b)

XNF 04(b) is an extension from XNF 04(a) where the condition of $last(q) \notin S$ has been added to the definition to guarantee that the redundancy can be eliminated.

Definition XNF 04(b) : Let P be a closed set of paths and let Σ be a set of XFDs such that $P\Sigma \subseteq P$. Σ of XFDs is in XML normal form (XNF) if for every nontrivial XFD $p_1, \dots, p_m \rightarrow q \in \Sigma^+$, $Last(q) \notin S$ and if $Last(q) \in A$ then $p_1, \dots, p_m \rightarrow Prefix(q) \in \Sigma^+$ where Σ^+ denotes the set of XFDs logically implied by Σ . (Vincent et al., 2004).

For example, consider the DTD in Figure 2.10 and the set of XFDs

XFD1: $department.course.student.lecturer.tno$
 $\rightarrow department.course.student.lecturer.name.S$

This is not in XNF because the *last* label in the path $department.course.student.lecturer.name.S$ is a text node.

Vincent et al., (2004) have formally proved that XNF 04(b) can eliminate data redundancy using formal justification adopted from (Vincent and Levene, 2000). They showed that the implication problem is decidable for simple XFDs and provide an algorithm to compute the closure for a unary XFD (only one path in LHS) with linear run time. To the best of our knowledge, they did not provide any normalization algorithm to convert un-normalized XML documents to normalized ones.

XNF 05

Like BCNF, Wang and Topor (2005) defined normal form for XML by using key dependencies where LHS of nontrivial functional dependency is a super key. To present normal form XNF 05, we define a terminology for key. We present them next.

If there is a XFD, $Q: p_1(c_1), \dots, p_n(c_n) \rightarrow \varepsilon$ over the DTD D , then we call $p_1(c_1), \dots, p_n(c_n)$ a key of Q . (Wang and Topor, 2005)

$p_1(c_1), \dots, p_n(c_n)$ is a key of Q (of T) means that, for every XML tree T conforming to D , and for any two nodes n_1 and n_2 in XML tree, if n_1 and n_2 agree on p_i , then n_1 and n_2 must be the same node. This definition is different from (Buneman et al., 2003) since they define several different interpretations of path agreements (see XFD 05 definition in section 2.3.4.).

Definition XNF 05: *A DTD D is said to be in XNF with respect to a set of XFD, if for every non-trivial XFD the following conditions hold.*

1. *The LHS is a key*

or

1. $Q: p_1(c_1), \dots, p_n(c_n) \rightarrow p_{n+1}(c_{n+1})$ in $(D, \Sigma)^+$
2. $Q: p_1(c_1), \dots, p_n(c_n) \rightarrow p_{n+1}(N)$ is also in $(D, \Sigma)^+$

Q is a downward path, p_1, p_2, \dots, p_n are simple paths, p_{n+1} is a simple path of length 1 or 0, and $c_i \in \{N, S \text{ and } I\}$ for $(i \in 1, n+1)$. (Wang and Topor, 2005)

For example, the DTD in Figure 2.10 is not in normal form with respect to the following XFD1 (2.8) *Department.course.student.lecturer : tno \rightarrow name* because the XFD1 is non trivial and *: Department.course.student.lecturer : tno $\rightarrow \varepsilon$* cannot be derived from the DTD. This means that *Department.course.student.lecturer : tno* is not a key since there are two different nodes *tno* label with '123' in XML tree shown in Figure 2.12.

However, to date there is no justification of this XNF 05 and no algorithm has been developed to transform an XML document to a normal form.

XNF 08

Similar to Wang and Topor's definition, Yu and Jagadish (2008) also used key to define XML normal form. An XSD is in *XNF08*, if each XML FD (Cp, LHS, RHS) , (Cp, LHS) is an XML key, where LHS is a set of paths and RHS is a single path.

An XML key is a pair (Cp, LHS) , where T satisfies the XML FD $(Cp, LHS, .@key)$ (Yu and Jagadish, 2008). This notion of XML key has similarities with the key notion proposed by Buneman et al. (2003).

For example XSD in Figure 2.14 that satisfies the XFD1 (2.11), is not in a normal form since $(C_{lecturers}, \{Department.course.student.lecturer.@tno\})$ is not a key; hence it cannot be used to uniquely identify an individual lecturer in the set of lecturers.

Yu and Jagadish (2008) proposed a normalization algorithm to eliminate such XFD and refine XSD into a XNF. This algorithm is actually an extension of the normalization algorithm proposed Arenas and Libkin but their algorithm is more comprehensive since it can be used to eliminate XFD caused by a set of elements. As shown in Figure 2.20 generally, the input of this algorithm will be a schema with a set non trivial of XFD and XML keys while the output is XSD with no redundancy.

Particularly, in this algorithm, the authors classified XFD into two categories: *Local XFD* and *global XFD*: *local XFD* means the XFD satisfies within a relative subtree while *global XFD* means the XFD satisfies the absolute XML tree. For example XFD1 and XFD3 are global XFD while XFD2 is a local XFD. Similar to Arenas and Libkin (2004), Yu and Jagadish (2008) also include two rules in the algorithm: creating new element and moving an element. For example to eliminate global XFD such as XFD1 and XFD3, procedure 1 is applied where a new element containing both its LHS element (e.g. tno) and RHS element (e.g. name) is created and put under the root. The RHS element is then removed from its original position. Meanwhile, procedure 2 is applied

to eliminate a local XFD such as XFD2, where a new element containing the subset of its LHS elements is created (e.g. sno) that are not part of the key for the ancestor tuple class (e.g. course) and RHS element (e.g. lname), and this new element is put under the schema element corresponding to the pivot path of the ancestor tuple class. Figure 2.21 illustrate the XSD after eliminating all XFDs.

Furthermore, to eliminate redundancy, the structure of the XML tree needs to restructure by moving the attribute/element or creating a new element. For instance, procedures 1 and 2 have similarities with the procedure *create element* and *moving element* in the XNF decomposition algorithm (Arenas and Libkin, 2004) but the differences are where the location of element will be created and the type of this new element. We simplified the process of this normalization algorithm as follows:

1. XFDs are grouped into *local XFD* and *global XFD* according to LHS (for example XFD1 and XFD3 are grouped together since both have the same LHS {../tno}).
2. XFDs are processed according to number of paths in their LHS as a strategy to reduce storage cost.
3. XFDs are processed according to the hierarchy depth of their tuple class by using a bottom-up approach
4. The algorithm is terminated after procedure 1 or procedure 2 removes at least one redundancy indicated by XFD.

We note that this algorithm has been analysed and been tested on a real data set namely PIR schema by normalizing the existing schema to **GTT-XNF** normal form (Yu and Jagadish, 2008). However, to the best of our knowledge, no justification has been given to verify the correctness of this algorithm and this algorithm can only be applied if XML data is stored in a relational database. Hence, there is a need for further enhancement if the XML data needs to be stored in an XML database.

Input: Schema S , a set Σ of redundancy – indicating FDs, a set γ of XML keys

1. Group FDs in Σ based on tuples class C_p and LHS , order them according to decreasing depth of C_p (lowest first) and increasing number of paths in LHS second (fewest first);
2. While Σ is not empty
 - 2.1 let \mathcal{F} be the first set of FDs in Σ with the same LHS and C_p ;
 - 2.2 Let F be the first FD in \mathcal{F} ;
 - 2.3 If F is local; Modify S by applying procedure 2;
 - 2.4 Else (if F is global) Modify S by applying procedure 1
 - 2.5 For each additional $F' \in \mathcal{F}$:
 - 2.5.1 Modify S in the same way by applying procedure 1 or 2, but using the new schema element already created in dealing with F'
 - 2.6 remove all Fds in \mathcal{F} from Σ ;
 - 2.7 For each $F \in \Sigma$:
 - 2.7.1 if F no longer valid: remove F from Σ ;
 - 2.7.2 if F is now structurally – redundant;
Convert F into its equivalent F' that is not structurally redundant and add F' to Σ

Output: schema S' , the modified redundancy-free schema

Figure 2.21: Algorithm of Schema Normalization (Yu and Jagadish, 2008)

```

department :Rcd
  course : setOfRcd
    cno : str
    title: str
    student :setOfRcd
      sno:str
      fname:str
      lname:str
      lecturer
        tno: str
    lecturer_info :SetOfRcd
      tno:str
      name : str
      address: setOfStr

```

Figure 2.22: Normalized XSD

A Third Normal Form for XML (X3NF)

Kolahi and Libkin (2006) have proposed a third normal form for XML by extending 3NF to XML. Kolahi adopted the notions of XML tree and DTD from Arenas and Libkin (2004) but extended the notion of prime attribute from relational database to the case of paths to XML tree. We present here the definition of a prime attribute path in order to present X3NF:

A path attribute $p.@l$ is a prime path if there exist a nontrivial FD: $S \rightarrow q \in (D, \Sigma)^+$ such that

1. *q is an element path,*
2. *$p.@l \in S$*
3. *S is minimal (not implied by other XFDs)*

Definition X3NF: XML specification (D, Σ) is in X3NF if and only if for every nontrivial XFD $S \rightarrow p.@l \in (D, \Sigma)^+$, we have that $S \rightarrow p \in (D, \Sigma)^+$ or $p.@l$ is a prime path (Kolahi and Libkin, 2006).

As a relational counterpart, a prime path is a path that uniquely determines path elements of a tree tuple from the root. Like the 3NF, X3NF tries to achieve a schema that can preserve functional dependency and at the same time reduce data redundancy in XML documents. For example reconsider the XML document in Figure 2.12 which illustrates a university database. This document satisfies the following constraints.

XFD4: department.course.student.lecturer.@tno

→ department.course.student.lecturer.@name

means any two lecturers with the same tno value must have the same name.

XFD5: department.course.title, department.course.student.lecturer.@name

→ department.course

means if two lecturers with the same name and teaching based on course title, they are teaching the same course.

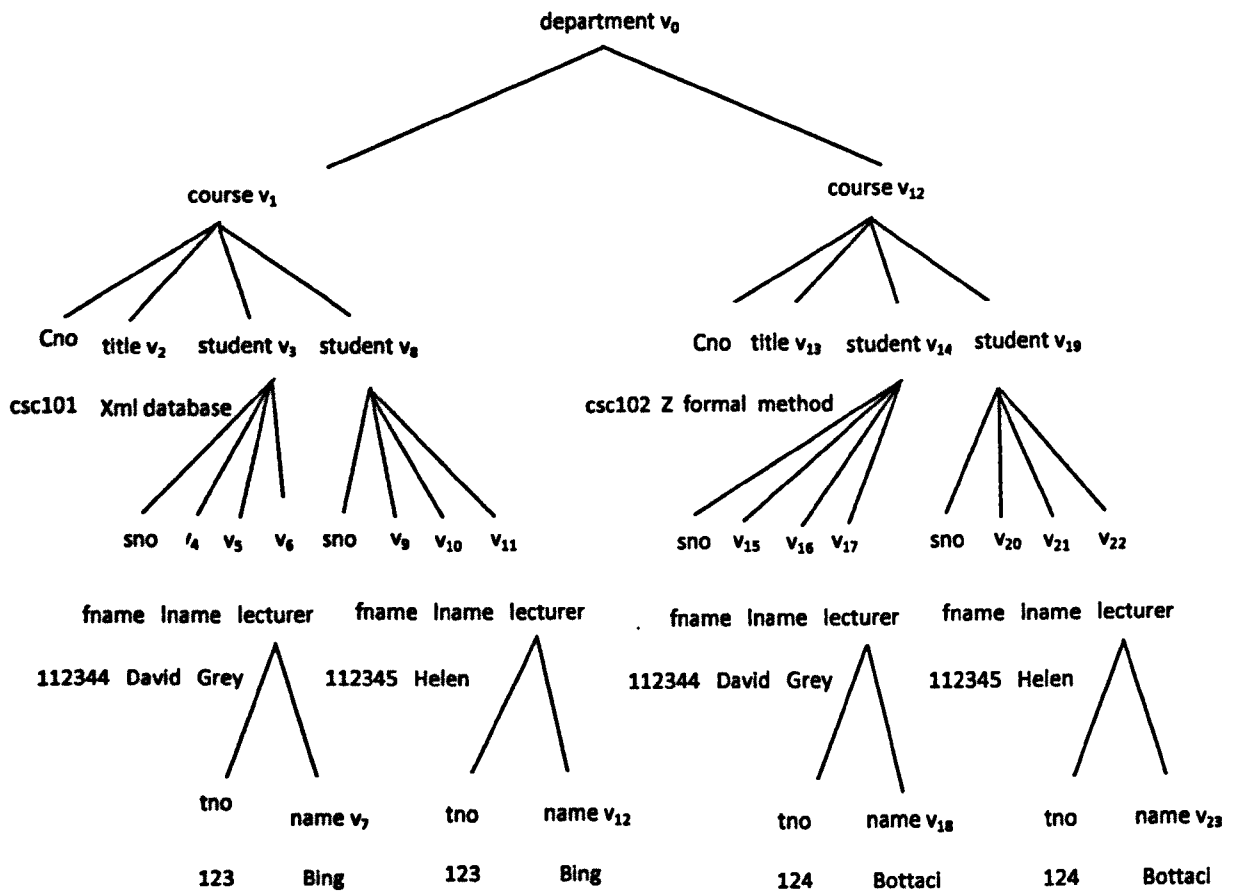


Figure 2.23: An XML Document in X3NF

According to this example, XFD4 and XFD5 satisfy the X3NF because *department.course.student.lecturer.name* is a prime path since it can uniquely determine the path element *department.course*.

Like 3NF, Kolahi claimed that this X3NF can preserve functional dependency and reduce data redundancy to a certain extent. However, to date no normalization algorithm has been developed and it remains to be proved that the decomposition algorithm based on X3NF definition is dependency preserving for any XML document.

2.3.8 Other Definitions of Normal Forms

Another common way to design a relational database is to *model* the requirements using ER diagrams (Chen, 1976). In order to combine requirement modelling and normalization, Ling (1985) proposed normal form for ER diagrams, which ensured that all relations mapped from ER diagram are in a normal form such as in 3NF or 5NF. The concept of normalization has been extended to the nested relational data model, where normal forms such as NNF (Nested Normal Form) (Ozsoyoglu and Yuan, 1987) and NF-NR (Normal Form for Nested Relation) (Ling and Yan, 1994) have been proposed to guarantee non-redundant properties for underlying nested relational databases. Many XML database researchers have applied this approach, i.e. conceptual data modelling approach to design non-redundant XML documents (Embley and Mok, 2001; Lee et al., 1999; Ling et al., 2005; Mani et al., 2001; Yuliana and Chittayasothorn, 2005). The one most related to our work is proposed by Ling et al. (2005) which defined a semi-structured data model, Object Relational Attribute-Semi-Structured (ORA-SS) (Dobbie et al., 2000) to represent data conceptually. The details of the ORA-SS schema diagram will be explained in Chapter 3.

The concept of normal form ORA-SS depends on the twin concepts of an object class normal form (O-NF) and a relationship type normal form (R-NF) which is an extension to the NF-NR for nested relation (Ling and Yan, 1994). Object and relationship of the ORA-SS schema diagram are similar to entity and relation in an ER diagram. This approach differs from Arenas and Libkin's approach, because they take constraints from the conceptual model rather than from a specified XML functional dependency (XFD). The nature of the definition for the normal form ORA-SS depends on a number of conditions. First, none of the attributes of the object class have multi-value or transitive dependency on the key of object class and relationship type. Second, every nested object class and relationship within the parent object class must be non-redundant.

Definition (NF): An ORA-SS schema diagram D is said to be in normal form (NF), if and only if it satisfies the following four conditions (Ling et al., 2005):

1. Every object class O in D is in O-NF.
2. Every relationship type R is in R-NF
3. No attributes or relationship types can be derived from other attributes or relationship types in D
4. The following two cases are satisfied:
 - (a) The attributes of object class and relationship types are connected to correct object class
 - (b) The relationship type is connected to the correct object class.

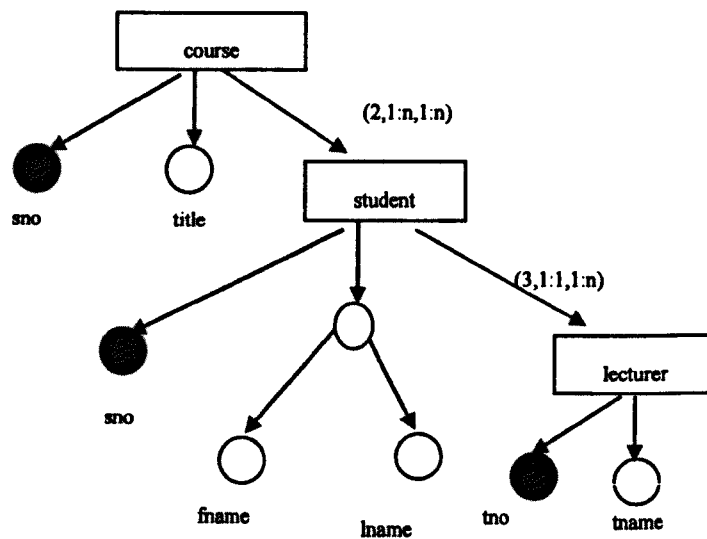


Figure 2.24: ORA-SS Schema Diagram

Ling et al. (2005) have proposed an algorithm to convert the ORA-SS schema diagram to a normal form ORA-SS schema diagram. Using their normalization algorithm (Ling et al., 2005) a normal form ORA-SS as shown in Figure 2.25 is derived.

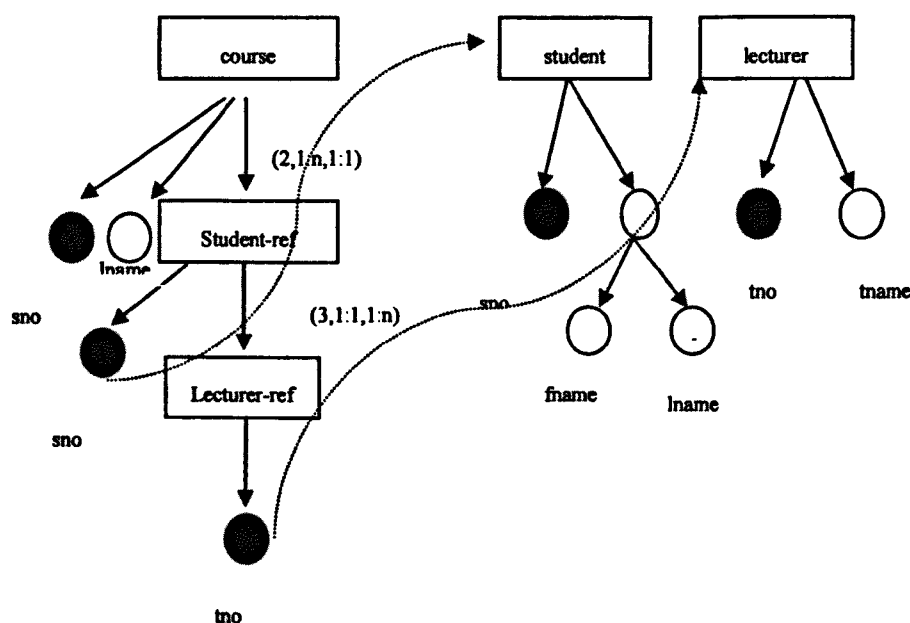


Figure 2.25: Normalized ORA-SS Schema Diagram

2.3.9 Discussion of Current XML Normal Forms

We have presented definitions of XML normal forms proposed by Arenas and Libkin (2004), Vincent et al. (2004), Wang and Topor(2005), Kolahi(2007) and Yu and Jagadish(2008). Most of them proposed the XNF, except Kolahi proposed XML third normal form (X3NF). However, XML normal form XNF, proposed by Arenas and Libkin (2004) achieves the best possible design from the point of view of eliminating redundancies in XML documents (Kolahi, 2007). Arenas and Libkin (2004) have defined XFDs and XML normal forms (XNF) entirely within the context of the XML document. XFD is formally defined based on the concept of 'tree tuple'. Arenas and Libkin (2004) proved that their XNF can avoid redundancies and update anomalies using information theory measure (Arenas and Libkin, 2005) at the schema and instance levels. They also showed that XNF is generalised from BCNF if the XML schema is converted into a relational presentation. However, the problem with this approach is that the way they express the semantic constraint (functional dependency) is very complicated due to the textual presentation of a schema. As we know, functional

dependency is already the area where designers have the most problem specifying in relational models, so making them more complicated and unfamiliar to designers makes XML document design more difficult. Moreover, a common problem with this approach is that the whole schema has to be redesigned when requirements change and information is added or withdrawn. In addition, the functional dependencies defined by Arenas and Libkin (2004) are dependent on the XML labelled tree (a model for XML documents) where paths are defined through the tree. Therefore when paths change, the functional dependency is adjusted as well. For this reason XNF can never be dependency preserving (Kolahi, 2007). Another shortcoming is that both DTD and XML tree are represented in textual representation, and so as a result it is difficult to visualise the data and their relationship. Furthermore, as discussed in section 2.3.7, the XML normal form proposed by Arenas and Libkin (2004) is limited in its ability to capture certain semantic constraints. The notion of XML normal forms is presented in a term difficult to understand because of the lack of graphical interpretations for the proposed theories. As a consequence, the current approach to XML normal form does not have the tremendous benefit for practitioners (Bourret, 2007). As shown in section 2.3.4, the definition of XFD and normal forms are very difficult for the normal users to understand if they do not have a theoretical background. Thus, these limitations will directly affect the application of XML normal form in practice by the end user. We believe that by simplifying the current definitions of XML normal form with a simple presentation will help end users to apply it and design XML documents in an easily and simple way.

In contrast, an ORA-SS data model is proposed to assist in XML document design (Dobbie et al., 2004) at the conceptual level. The tree structure of the object class is clearly shown in the ORA-SS model. In Dobbie et al.'s work, it assumes that the starting point for the design of an XML document is at a conceptual model ORA-SS. Using the algorithm, then an XML document is derived from a normal form ORA-SS. This approach follows from the ER normal form (Ling and Yan, 1994). Ling and Yan (1994) have shown that this approach is guaranteed to produce a redundancy-free and compact relational database. In traditional database design, practitioners routinely use an ER model and convert the ER diagram to a relational model. Another advantage of this

approach is that it is easier and simpler for designers compared to normalization theory (Ling et al., 2005; Halpin, 2010). However, the normal form ORA-SS relies upon definition of NF-NR (Ling and Yan, 1994). In order to use and understand the normal ORA-SS the user must understand normal form for nested relations first. Another disadvantage, in Ling et al.'s approach, is that they assume the XML document is not associated with DTD; hence extraction of the schema from the XML document is required.

2.4 Summary

In this chapter, we gave a brief background of relational database design and discussed thoroughly an XML database design through the normalization approach. We described some criteria for non-redundant and bad relational schemas. Based on such approaches, functional dependency was considered to represent the semantic constraint of XML data. This is because functional dependency forms the important basis for the normalization process in the database design. However, we noticed that the definitions of functional dependencies, the notion of XML data tree, definition of element and path in XML database are very difficult in terms of their presentation and use a lot of theoretical or mathematical notions. From the discussion of existing XFDs and XML normal forms, we are aware of their advantages and limitations, the latter of which will influence XML document design in real world practice. To overcome these limitations is one main task of this thesis. Furthermore through our investigation of some XML normal forms, we have found that this XML normal forms notion could be defined in a simpler way. We noticed also that, a non-redundant graphical data model for XML needs to be developed to support XML design. These issues are very important for XML database design. Therefore, we will propose such a model to support the XML design process. This will be examined in more detail in the next chapter.

Chapter 3

A G-DTD: A Graph Model for Describing XML Documents

3.1 Introduction

In database design theory, conceptual data modelling is an important part of database designs which deals with structure, organization and effective use of the information. Moreover, the purpose of any data model is to allow us to describe constraints, manipulate objects and relationships among objects in the real world that we intend to reflect in the database (Beeri and Bernstein, 1979). As pointed out by Biskup (1995), finding a unifying data model and extending achievement of database design theory to advanced databases consisting of complex object types such as XML is a very challenging task. This is because, first, XML is hierarchically structured and requires to be conformed to its schema such as DTD (Mani et al., 2001). Second, the expression of dependency constraints such as functional dependency will be different from the conventional model due to XML structure (Arenas and Libkin, 2004). Third, mapping an XML document into well-defined and highly-structured schemas, such as those in relational and object oriented models, often requires a lot of effort and frequent schema modification. These difficulties have prevented the use of relational and object oriented approaches to XML data modelling. Therefore an appropriate conceptual model for XML documents has become important.

In this chapter, we propose a conceptual model of DTD called Graph-DTD (G-DTD). The G-DTD helps to give a better understanding of DTD structures, to improve XML design and also the normalization process as well. G-DTD has a richer syntax and structure which incorporate attribute identity, simple data types, complex data types and relationship types between the elements. Furthermore, the semantic constraints that are important in XML documents are defined clearly and precisely to express the semantic expressiveness. We believe G-DTD can be used to represent and support XML structure

explicitly and capture more semantics of XML documents in order to solve some difficult issues, for example, query processing, information losslessness and normalization.

A G-DTD model is proposed to assist in XML document design at the conceptual level. In this work, it is assumed that the starting point for the design of an XML document is at a conceptual model G-DTD. Using the algorithm, then an XML document is derived from a normal form G-DTD. As shown in Figure 3.1, the process of designing an XML document has several steps. First, we take DTD as input and represent it into G-DTD. Second, we transform G-DTD to normal form G-DTD. During this step, normalization is carried out automatically based on the number of data dependencies provided by the user in the conceptual model. Third, we map the normal form G-DTD to DTD and finally, the XML document is generated on the basis of the normalized DTD.

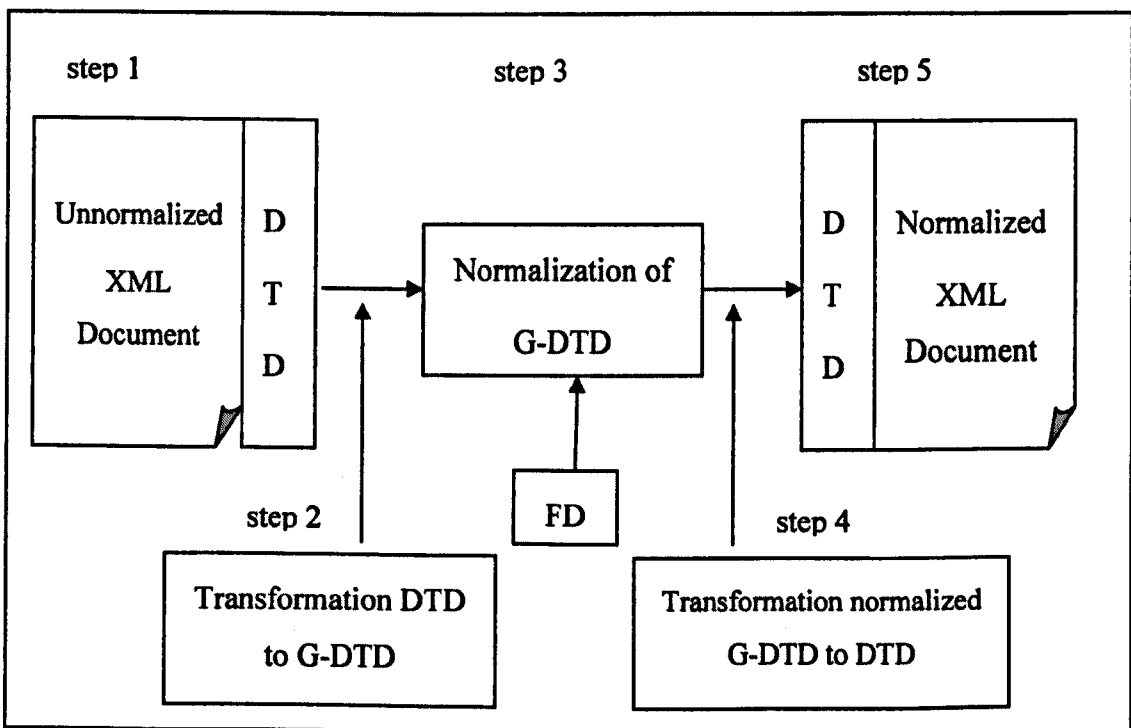


Figure 3.1: XML Document Design Process

Our contributions in this chapter are as follows:

- We will present a graphical representation of G-DTD using a directed and labelled tree. The important features of G-DTD such as element and sub-element relationships and attributes are presented clearly. This G-DTD has richer syntax and structure which incorporate element/attribute identity, simple and complex data types and relationship types between the elements.
- We proposed a transformation rule to transform from G-DTD to DTD structure.

This chapter is organized as follows. In section 3.2 we review other related XML data models. In section 3.3 the basic notation, structure of DTD and rationale for G-DTD are described. We propose the G-DTD's notations, component, structure and semantics in section 3.4 and in section 3.5 discuss the operations of G-DTD. Lastly section 3.6 proposes the transformation rule to show that G-DTD can also be transformed back to DTD structure.

3.2 XML Model Review

Major current data models that are commonly used to represent instances of XML documents and their schemas are based on a directed graph model. This model consists of nodes and directed edges which, respectively, represent XML elements in the document and relationships among the elements, e.g. element-sub-element and relationships between elements. The current XML models can be categorized into three types. There are XML models to represent instances of XML documents, XML models to represent XML schema and XML models to represent both XML documents and XML schema. Some common data models used to represent an instance of XML documents are the Document Object Model (DOM) (Apparao and Byrne, 1998) and Object Exchange Model (OEM)(McHugh et al., 1997). On the other hand, the Semistructure-graph (S3-graph) (Lee et al., 1999), Semantic network (Feng et al., 2002), Concept Model Hypergraph (CM-Hypergraph)(Mok and Embley, 2006), Dataguide (Goldman and Widom, 1997) and Extended Entity Relational(EER) models (Mani et al., 2001) are models to represent only the XML schema while the XML Tree (Arenas and

Libkin, 2004) and Object Relational Attribute-SemiStructured (ORA-SS)(Dobbie et al., 2000) models can be used to model both instances and XML schema. We also noticed that there are other methods found in the literature used to represent XML documents and their schema, such as Hedge automaton theory, Declarative Description (XDD) theory and the Functional programming approach. In the following section, we will present the above data models by classifying these models into three types: models for instance level, models for schema level and models for a mix of schema and instance levels. We describe each of these models as follows.

3.2.1 Models for Schema level

The following models can only be used represent the schema level but cannot represent an instance of an XML document.

Semi- Structured Schema Graph

A Semi-structured schema graph (S3-graph)(Lee et al., 1999) is a directed graph model for schema where each node in a graph can be classified as an entity node or a reference node. An entity node represents an entity which can be a basic atomic data type such as string, or complex object type. A reference node is a node which references to another entity node. Each directed edge in the graph is associated with a tag. The tag represents the relationship between the source node and destination node. The tag may be suffixed with a “*” to indicate that it can have one or many children. S3-Graph can represent the hierarchical structure of the element sets and represent one to many relationships. However, it does not distinguish between elements and attributes. Semantic constraints such as cardinality of the element, relationship between elements and attributes are not presented precisely.

Semantic Network

The semantic network model for XML was introduced by Feng et al. (2002) to model a schema level. This model is based on hierarchical graph model and adopts entity type and attribute notation from the Entity-Relationship (E-R) model. Nodes in semantic

networks are used for modelling objects from the real world and their attributes, and edges are used for modelling relationships between the objects. Only binary relationship between nodes can be represented in this model. Many constraints can be specified in the semantic network model, such as constraints on edge and constraints on nodes. The details of this model can be found in Feng et al. (2002).

Concept Model Hypergraph

A data model Concept Model Hypergraph (CM Hypergraph) was proposed by Mok and Embley (2006). The model is used to represent object sets, relationship sets and some semantic constraints conceptually. Object sets that refer to element sets represented as labelled rectangles, relationship sets are represented by edges and constraints are depicted using arrows. The graphical type of arrow is used to distinguish between the types of relationship between the object set. For example, an edge with no arrow heads represents a many-to-many relationship set, an edge with one arrow head represents a many-to-one relationship, and an edge with an arrow head at both ends represents a one-to-one relationship. The symbol “o” on the arrow indicates that the object is optional. CM hypergraph can model both binary and n-ary relationships but it cannot represent the hierarchical structure of the schema. The sequence and the disjunction of objects set cannot be represented in this model either.

Extended Entity Relational diagram

An Extended Entity Relational (EER) diagram for modelling XML schemas was defined in Mani et al. (2001). This model is extended from the Entity Relational (ER) diagram and can be used to capture the hierarchical link and ordering of entity sets. The hierarchical link or element-subelement relationship is represented using a ‘has’ relationship. The ordering of the entity sets is presented as a solid line between the relationship set and entity sets. Similar to ER notation, entity sets are represented as rectangles and relationship sets are represented by diamonds on the edge. The concept of attributes here is the same as in an ER diagram; however it not possible to show

whether the attributes are optional or required. Furthermore, in an EER diagram, it is not indicated which entity set is the root of the tree and the relationship between the entity sets is not presented clearly.

DataGuide

A DataGuide (Goldman and Widom, 1997) models the schema of an OEM instance graph. In the DataGuide, the complex objects are depicted by a triangle. Recently, Yu and Jagadish (2008) used the DataGuide to represent a schema called *Schema Tree* based on XSD. However, DataGuide is less expressive than DTD since it is not possible to represent relationships and semantics constraint among elements. DataGuide depicts only the hierarchical structure of the element sets using textual representation and as with the OEM there is no distinction between element set and attributes.

3.2.2 Models for Instance Level

Document Object Model

The Document Object Model (DOM) proposed by Apparao and Byrne (1998) represents the instance of a semi-structured data as a tree. Each node represents an object that contains one of the components from an XML structure. The three most common types of node are element nodes, attribute nodes and text nodes. A DOM represents the instance of a document, showing the hierarchical structure of the elements and the relationship between the elements. DOM can distinguish between elements and attributes. However DOM only represents the instance of a semi-structured data, it does not represent the schema information directly or the constraints of the elements.

Object Exchange Model

The Object Exchange Model (OEM) which was proposed by McHugh et al.(1997) also represents an instance of semi-structured data. An OEM model is a labelled directed graph where the vertices are object, and the edge is relationship. Each object has a unique object identifier (OID), a label and a value. There are two types of objects, atomic and complex. Both atomic and complex object are depicted as 3-tuples (OID, label, value). An atomic object contains a value with types e.g., integer, string, etc. A complex object consists of sub-objects. An OEM indicates the hierarchical structure of the objects. Similarly to DOM, OEM does not represent the schema and semantic constraints clearly in the model.

3.2.3 Models for Mix of Instance and Schema Levels

Object Relational Attribute- Semi Structured (ORA-SS)

ORA-SS is a rich hierarchical model of a semi-structured database proposed by Dobbie et al. (2000). An ORA-SS model can represent an instance of XML document and ORA-SS schema as well. ORA-SS has three basic components: object types, relationship types and attributes. The Object type's notation is similar to entity type from a conventional ER model. The ORA-SS schema diagram is like a CM hypergraph diagram and an ORA-SS instance diagram is similar to a DOM tree. Relationship type between object types represents hierarchical relationships. ORA-SS has the features of cardinality constraint, ordering concepts and disjunction between two or more attributes. The advantage of this model is that it can represent n-ary hierarchical relationships and it can distinguish between elements and attributes clearly. However the attributes described here are different from attributes defined in XML documents since it uses the same concept of attribute as an ER model. Even though the relationship between element set is defined precisely, but the presentation simple element type and complex element type is not well distinguished in terms of graphical notation. More details of this model can be found in Dobbie et al. (2000).

XML tree

As described in Chapter 2, XML tree proposed by Arenas and Libkin (2004) is used to represent the XML document graphically and define languages for describing DTD. Their approach is able to represent both instances and schemas of XML documents precisely. However, it has the disadvantage that it is not possible to represent semantic relationships between nodes and distinguish relationship between attributes and element sets. Furthermore, the DTD is described in this model using textual presentation, so it is very difficult to interpret the data and relationships between elements. Moreover, the notion of DTD defined by Arenas and Libkin (2004) is quite different from normal DTD since they incorporate the value of elements and attributes together in the notation.

3.2.4 Other data models

Hedge automaton theory was developed by Murata (1999) using the basic ideas of string automaton theory to formalize XML documents and their DTDs. A hedge is a sequence of trees or a sequence of XML elements. An XML document is represented by a hedge and a set of documents conforming to a DTD by a regular hedge language, which can be described by a regular hedge grammar. By using hedge automaton, one can validate whether a document conforms to a given regular hedge grammar. XGrammar (Mani et al., 2001) is an example of an XML model based on hedge automaton theory.

Wuwongse et al. (2003) have proposed data model for an XML database using XML Declarative Description (XDD) theory (Anutariya et al., 2000). In this model, XML elements are associated with variables called XML *expression* and the constraint and relationship is represented in terms of XML clauses. There is other related work using multimodal logic (Bidiot et al., 2004) and spatial tree logic (Conforti and Ghelli, 2003) to present and reason about semi structure data.

A functional programming approach to modelling XML documents and formalizing operation has been developed by Fernandez (1999) by incorporating the notion of node:

Algebra for XML queries, expressed in terms of list comprehensions in the functional programming paradigm. On the other hand, Conrad et al. (2000) proposed to conceptually model DTD using Unified Modeling Language (UML). They used important features of UML to model DTD. We are also aware that Bird et al. (2000), used Object Role Modeling (Halpin, 1999) as their conceptual model to describe XML schema.

In conclusion, data model such as OEM, DOM and DataGuide have been designed for the purpose of information or schema integration. The focus of these data models is on modelling the nested structure of semi-structured data but not modelling the constraints that hold in the data. In contrast, data models such as S3-Graph, CM Hyper graph, EER, XML Trees and ORA-SS have been defined specifically for data management. On the other hand, we noticed that a graph based model provides an effective and straightforward way to handle XML documents. Based on the above review, we proposed to adopt ORA-SS's (Dobbie et al., 2000) because of its capability and because the model uses established notations from traditional ER model (Chen, 1976) which is already well known to database designers. However some modification needs to be made. The rationale is, these models can only be used to represent semi-structured data. The semi-structured data model described above assumes that the collected of data or elements are unordered, whereas with XML documents, elements are ordered (Connolly and Begg, 2002) and the XML document must be associated with its schema, i.e. DTD.

3.3 Document Type Definition (DTD) – Its Basic and Rationale

3.3.1 Introduction

As presented in Chapter 2, Section 2.3.2, to define the structures for XML documents, we need to use a schema. We use a DTD in our work as it has been well accepted. Even though DTDs are less expressive than XML Schema, in general they are expressive enough for a large variety of applications (Arenas and Libkin, 2004). Moreover, from a theoretical point of view, DTD can be characterized in terms of

unranked tree automata (Nevan, 2002), which have been widely studied in automata theory and more recently in database theory (Arenas and Libkin 2004). Furthermore DTD is an early standard for XML, and many legacies XML document structures are defined by DTDs. Therefore in this thesis, we aim to represent the details of DTD syntax and structure for the purpose of XML data modelling and normalization.

3.3.2 Overview of DTD syntax

The DTD consists of the set of rules that each XML document must conform to. Normally, such rules are represented using context free grammar and instances of a DTD are seen as a syntax tree. DTD defined here is much like tree structure (Arenas and Libkin, 2004; Murata et al., 2003). The DTD can be used to map between metadata and instance and validate whether the structure of XML document is correct or not. The main criteria of DTD consist of root, element, attribute and element type definition, which is represented as regular expression.

A DTD document begins with a document type declaration in its simplest form as shown below:

```
<! DOCTYPE root < [all the elements in the document] >
```

the DOCTYPE declaration will followed by all element and attribute declaration. Similar to DOCTYPE declarations, the basic common syntax of each line in the DTD must start with the symbol start tag '<!' followed by element declaration and end with end-tags '>'. These tags must be balanced and they are used to delimit elements. To differentiate between elements and attributes, the keyword ELEMENT or ATTLIST is written after the start symbol '<'. For example the general syntax is written as follows.

```
<! ELEMENT Element declaration> and
```

```
<! ATTLIST attribute declaration >
```

DTD Elements

The more specific syntax for declaring DTD elements is as follows;

```
<! ELEMENT elementname { [content] } >
```

Elementname represents name of object and content of DTD can be categorized as follows:

EMPTY Keyword is applicable to element that does not require data content. For example `<!ELEMENT year EMPTY>`

ANY keyword indicates a combination of elements that can contain data of type #PCDATA or any other element defined in DTD.

```
<!ELEMENT year ANY>
```

```
<! ELEMENT lname (#PCDATA)>
```

```
<! ELEMENT conf (lname, issue) >
```

where *conf* is the name of the element and *lname* and *issue* are two sub elements of element Conf. The symbol “,” depicts that *lname* and *issue* must be in sequence.

- *Choice of subelements*

Content of subelement can also be a set of choice, represented by a “|” symbol (sometimes called OR operator). For example, consider in the following DTD syntax, where the student element can contain a sequence of sub-elements *sno*, *name* and optionally contain sub-element *hostel* or *home* element.

```
<! ELEMENT student (sno, name, (hostel| home) >
```

```
<! ELEMENT sno (#PCDATA) >
```

```
<!ELEMENT name (#PCDATA) >
```

```
<!ELEMENT hostel (address)>
```

```
<!ELEMENT address(#PCDATA) >
```

```
<!ELEMENT home (address)>
```

<!ELEMENT address(#PCDATA) >

DTD element cardinality

In a DTD, the cardinality determines how many times an element can occur within a specific content layer. There are four DTD cardinality constraint syntax specifiers:

In the following example, there can be zero or more course elements contained within each department element.

<!ELEMENT department (course)>*

DTD attributes

For XML documents, attributes are defined in DTDs using the ATTLIST declaration. An attribute can be tagged as an identifier, indicating that it is expected to have a unique value within an instance XML document. An attribute can have a string value or be a reference to the identifying attributes of an element sets. Attributes are defined using the following syntax:

<!ATTLIST [element][att name][attribute type] [default] >

element represents element name, *att name* represents attribute name, while *attribute types* can be defined using various different types as follows:

Types	Description
CDATA	characters
(p, q[,...])	originates from list of values
ID	unique identifier
IDREF(S)	identifier of a different element(s)
NMTOKEN(s)	XML name(s)

and *attribute defaults* categories as follows:

Default	Descriptions
Value	the initial setting for an attribute
#REQUIRED	An attribute must have a default value
#IMPLIED	An attribute does not have to have a default value
#FIXED value	An attribute value is predetermined

For example the following syntax;

<!ATTLIST course cno ID #REQUIRED>

represents that the element *course* has attribute name *cno* with type *ID* and an attribute default is required.

Consider again following example of DTD to summarise the above definition:

```
1 <!DOCTYPE department[
2   <!ELEMENT department(course*)>
3   <!ELEMENT course(title, student*)>
4   <!ATTLIST course cno ID #REQUIRED>
5   <!ELEMENT title (#PCDATA)>
6   <!ELEMENT student (firstname|lastname?, lecturer)>
7     <!ATTLIST student Sno ID #REQUIRED
8     <!ELEMENT firstname(#PCDATA) >
9     <!ELEMENT lastname(#PCDATA) >
10    <!ELEMENT lecturer (name)>
11    <!ATTLIST lecturer tno ID #REQUIRED>
12    <!ELEMENT name (#PCDATA)>]
```

Figure 3.2: DTD

The detailed specification of DTD is described as follows: The first line of DTD in Figure 3.2 depicts that department is the root of the DTD. The second line shows that department consists of the sub-element *course*. The semantic relationship between department and course is indicated by the symbol *, representing that department can consist of zero or many course for each department. The third line of the DTD shows that each element course has sub-element *title* and element *student*. The symbol “,” between them indicates that they must occur in sequence. The fourth line indicates that element course has an attribute cno. The keyword ‘#REQUIRED’ represents that the attribute cno must appear in every course while “ID” indicates that the value of cno is unique within the XML document. The fifth line of the DTD shows that the keyword “PCDATA” depicts that element lname has no sub-elements and it is a leaf element and has a string value. The same semantics are applied to DTD from line 6 to line 13.

More specifically, this DTD has the following information:

- Each department offers many courses indicated by the notation *.
- Every course is described by the attribute course no (cno), title and numbers of students taking the course.
- Each student has a student number (sno), first name or last name as optional and an assigned lecturer.
- Each lecturer has his/her number (tno), and name (tname).

As shown in Figure 3.2, DTD is commonly represented as textual representation in a hierarchical structure which is difficult to be analysed and understood by end users or practitioners. Normally even the design of a simple DTD may cause difficulties, partly due to the textual form of the grammar itself. Moreover, DTD lacks of clarity and readability, which may cause errors during the design process. DTD description using graphic interpretation is very important for the purpose of better understanding of the XML design because graphical notations are commonly regarded as more accessible than formal notation. Furthermore, the absence in current approaches of graphic notations of the DTD could cause difficulties in the normalization process. For example, the normalization process proposed by Arenas and Libkin (2004), Kolahi (2007), Wang and Topor (2005), Yu and Jagadish (2008) is very difficult to understand and hard to be implemented programmatically due to the many theoretical or mathematical terms defined.

Hence, a summary of the existing XML models in section 3.3 is necessary and important in particular to survey which approaches and notations are the best to be adopted and applied in our model. Our focus is to propose a graphical data model at a schema level. In our model we propose to differentiate between elements that contain sub-elements and elements with no sub-elements more explicitly. This is important for the normalization process because elements with sub-elements will normally cause data redundancy in XML documents. We called the former complex elements and the latter simple elements. Specifically, a simple element is an element associated with the keyword #PCDATA in DTD syntax. Most previous models do not distinguish precisely between complex elements and simple elements. Instead they define simple elements

similarly to attributes. As shown in Figure 3.2, the attributes DTD is used to define the property or an identifier for a complex element; hence it must be distinguished from a simple element. In the G-DTD model, various different notations are proposed to represent all the most important features of DTD in a very simple and practical way, thus providing a more flexible modelling approach. Furthermore, in the G-DTD model, the representation of semantic constraints between the complex elements, simple elements and attributes is emphasized.

3.4 G-DTD Data Model

3.4.1 Objectives of the Model

The conceptual data model represented here is called G-DTD. It has enhanced the current XML data models by including different structural node types and relationship types. The aim of this extended XML data model is to capture the syntax and semantics of XML documents in a simple but precise way. G-DTD has richer syntax and structure which incorporates attribute entity, simple data types, complex element data types, relationships types, tree structure, cardinality, sequence and disjunctions between elements or attributes. It is important that all these structures and semantic constraints in XML documents are defined clearly and precisely to express semantic expressiveness at the schema level. Having G-DTD as a tool, helps the user to arrange the content of XML documents in order to give a better understanding of DTD structures, to improve XML design and the normalization process.

3.4.2 Overall view of G-DTD

Generally, a G-DTD is a labelled and ordered tree consisting of a hierarchy of nodes that are connected to each other through directed unlabelled or labelled arrows. In addition, particularly, each node in a tree corresponds to a complex element, simple element or attribute, while the link or edge between each node denotes the type of relationship between nodes. Nodes that represent elements that have basic property types of

#PCDATA are considered as the leaves of the tree. The G-DTD represents the structure and the semantic constraints of the XML document at schema level. The G-DTD adopts graphic notation with some modification from conventional entity relational data model (Chen, 1976) and semi-structured data model (Dobbie et al., 2000) because of its clarity and simplicity.

The main features supported by this G-DTD are as follows:

- Supports structure of individual nodes by defining the type, level and cardinality of each node.
- Enhances the abilities of the current XML model by its complex element nodes and simple element nodes and attributes.
- Provides semantic relationship definition to allow users to define semantic relationship between node types. A semantic relationship between two node types such as *path link* and *part-of link* can be defined by the user. The path link can be a binary relationship or n-ary relationship. Each path link is assigned a unique name, type of relationship, parent constraint and child constraint.
- Allows the user to define the structure of nodes in an ordered and in a hierarchical way

3.4.3 G-DTD Components

G-DTD has six basic components:

- A set of complex element nodes representing the elements that have subelements.
- A set of simple element nodes representing the elements that have no subelements.
- A set of attribute nodes representing the attributes defined in ATTLIST.
- A set of relationships representing the semantic relationships between the complex elements, simple elements and attribute nodes.

- A root node representing the first element in the DTD and every node has a level, i.e. the distance from the root.
- A last node.

3.4.3.1 Simple Element Nodes

A simple element node is used to represent an element associated with #PCDATA or #CDATA. It is illustrated as a labelled rounded rectangular box with the form $\langle name, type \rangle$ where *name* is the name of simple element and *type* represents PCDATA or CDATA or string 'S'. A simple element node can be:

- Single -valued which has only one value
- Multi - valued which can have a set of values
- Required/mandatory, which must have a value for every instance
- Optional which may not have a value in some instances

All simple element nodes are assumed to be mandatory and single valued, unless the node contains an ?, which signifies it is single valued and optional, or + which signifies that it is optional and multi-valued. This notation is similar to ORA-SS (Dobbie et al., 2000). The symbol is written in front of the tuple $\langle name, type \rangle$ to differentiate among them accordingly.

<u>Notation</u>	<u>Meaning</u>
$\langle \text{firstname}, S \rangle$	Mandatory, single value, PCDATA
$+ \langle \text{tutor}, S \rangle$	Required, multi value, PCDATA

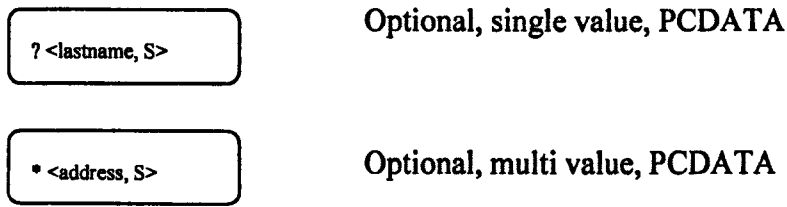


Figure 3.3: Types of Simple Element Nodes

Figure 3.3 shows the notation and semantics of each simple element, for example simple element *firstname* is mandatory, simple element *lastname* is optional, *tutor* is multi-valued and required while *address* is multi-valued and optional. All of them have type #PCDATA.

3.4.3.2 Complex Element Nodes

Complex element node is used to represent a set of elements which has other sub elements and attributes. The sub element node of complex element node can be classified as follows:

PCDATA, EMPTY, ANY, mixed context, complex element and simple element. Each complex element node has one or more labelled directed arrow going from it to another node. The complex element node is illustrated as a labelled rectangular box. This notation is adopted from the ER model (Chen, 1976) and is similar to entity. The label is written in a rectangular box as the tuple <name> where name represents the name of the node in the G-DTD. The name is mandatory. Figure 3.4 gives an example of a complex element node labelled as <student> which represents that complex element node *student* is located at level one of the G-DTD.

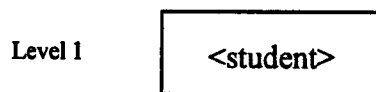






Figure 3.4: Complex Element Node Student

3.4.3.3 Attributes nodes

Attribute nodes are used to represent attributes defined in *ATTLIST*, which describe the property of a complex element node. For an attribute node, the *attribute name*, *attribute type* and *attribute default* must be presented clearly in the diagram. These criteria will be written as tuple $\langle name, type \rangle$. However the *attribute default* is presented using different notations to differentiate among them. The attribute node is represented by various notations of labelled oval diagrams corresponding to *attribute default* as follows:

<u>Notations</u>	<u>Attribute default</u>
	#REQUIRED(mandatory)
	#IMPLIED(optional)
	#FIXED(optinal)
	#IDREF(reference)

Normally an attribute is an identifier for a complex element, presented as ID #REQUIRED. This means that it is unique among the instances of complex element and mandatory, while an optional attribute is defined by the keyword #IMPLIED in the DTD. Attributes can be classified as single attribute or multi attributes. A single attribute has only an atomic value but a multi attribute for a complex element such as IDREF(s) has special meaning value(s). In addition, some attributes such as IDREFS contain one more data values. For example the following notation represents that attribute name SNO and it is required (unique) in DTD:

<sno, ID>

The above notation is represented in DTD as follows:

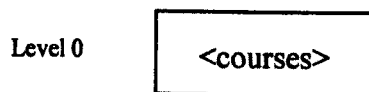
<! ATTLIST Student sno ID # REQUIRED >

3.4.3.4 Root Node

A root node is a member of a complex element node but the level of the root node always started at level 0. Root node notation similar to complex element notation where it is a special case of complex element node. The root node can be identified from DTD using the key keyword DOCTYPE. For example

<!DOCTYPE courses [content..]>

In G-DTD, the root node for the above example will be presented as follows:



3.4.3.5 Relationships

Links between nodes represent relationships which contains *type of cardinality constraint*. Three types of link exist in this model:

- Path link
- Part_of link
- Has_A link

Path Link

The path link is a relationship between a complex element node and another complex element node. This link shows the relation between parent nodes to child nodes or ancestor node to descendant node, where child node should not be in the same set as parent node.

Because the structure of G-DTD is a directed graph, a parent for each type of related node, i.e. complex element node, simple element node and attribute node is determined by the level position of the node; a parent node is always at the level position less than a child node by one level difference only.

In a hierachical link, the order between complex element nodes is important since it will determine the immediate parent and child of the complex element node. Another important feature of hierachical links is that they can be composed among themselves and can be repeated as many times as desired. The constraint relationship on the hierachical link must be a positive number and the link is cycle free, meaning that no complex element node is mapped to itself.

In the path link, a semantic meaning, which is indicated by the *connectivity* between complex element occurrences, is important. The connectivity of a relationship specifies the mapping of the associated complex element occurrence in a relationship. Basic constructs for connectivity are: *one-to-one* (unary or binary relationship), *one-to-many* (unary or binary relationship), and *many-to-one*, *many-to-many* (unary or binary relationship). All these types of relationship are indicated by directional arrows. To differentiate among them, both cardinality constraint and degree are attached to the arrow. The notation is presented as $(name, d, cp, cc)$ where *name* represents the name of the relationship, *d* is the degree of relationship, *cp* and *cc* are cardinality constraints for parent and child respectively. This notation is similar to ORA-SS (Dobbie et al., 2000).

(a) *Degree of relationship*

Degree of relationship is the number of complex elements associated with the relationship. An n-ary relationship is of degree n. Unary, binary and ternary relationships are special cases in which the degree is 1, 2, and 3, respectively.

(b) *Cardinality constraint for complex element*

To reveal more semantics in their relationship, the cardinality constraint is associated with the path link. As described in section 3.4, in a DTD declaration, there are four possible cardinality relationships between parents and children: This is illustrated as follows

`<! ELEMENT E (E1, E2+, E3*, E4?)>`

The above segment of DTD shows that complex element E has four children E1, E2, E3 and E4. The cardinality of the constraint is described as follows:

- Only (default): An element *E* must have one and only one child *E1*.
- Any (*): An element *E* can have *zero or more* child *E3*.
- Optional(?): An element *E* can have either zero or one child *E4*
- At least (+) : An element *E* can have one or more child *E2*.

The same rule applies for cardinality constraints for both parent node (cp) and child node (cc). Here cardinality of complex elements in a relationship is represented as a 2 tuple (min: max). The constraint (0: N) (0:1) and (1: N) is represented as the operators * ? and + respectively except cardinality constraint (1:1) is presented as 1. This relationship cardinality constraint is indicated using a directional arrow. For instance, the diagram in Figure 3.5 illustrates a binary path link between complex element *courses* and complex element *student*, where a student can take zero or many courses while many courses can be taken by zero or many students.

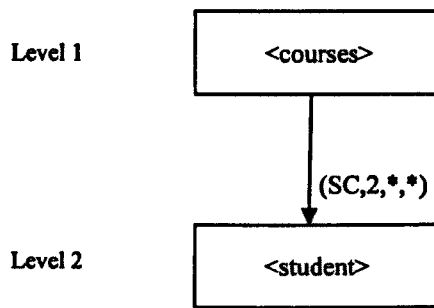


Figure 3.5: Many-to-Many Binary Relationship

Figure 3.5 also illustrates that two complex elements must be conceptually located at different levels between parent node and child node. In this example complex element *course* is a parent node and complex element *student* is a child node, which is located at level 1 and level 2 respectively. The label SC is a name relationship which refers to student and course complex elements node.

Part_of Link and Has_A Link

Part_of link is a relationship between complex element node and attribute node. Each complex element node has a unique attribute node which is mandatory or a set of attributes (optional). It is illustrated as bold double arrow. Has_A link is a relationship between a complex element node and a simple element node. It is illustrated as a double arrow. These types of relationship are illustrated using the following notation.

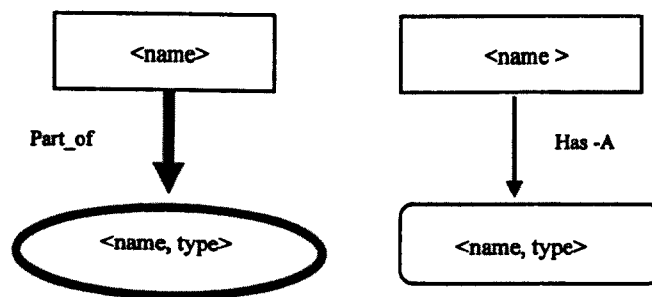


Figure 3.6: Part_of Link and Has_A Link

For example, Figure 3.7 illustrates that attributes *cno* and *sno* are identifiers and required for complex element *courses* and *student* respectively. The relationship between complex element *student* and attribute *sno* is defined as a Part-of link while the

relationship between complex element *student* and simple element *fname* and *lname* is shown as Has-A link. All simple element nodes have type #PCDATA. The depth of each node in this diagram is denoted by the level number.

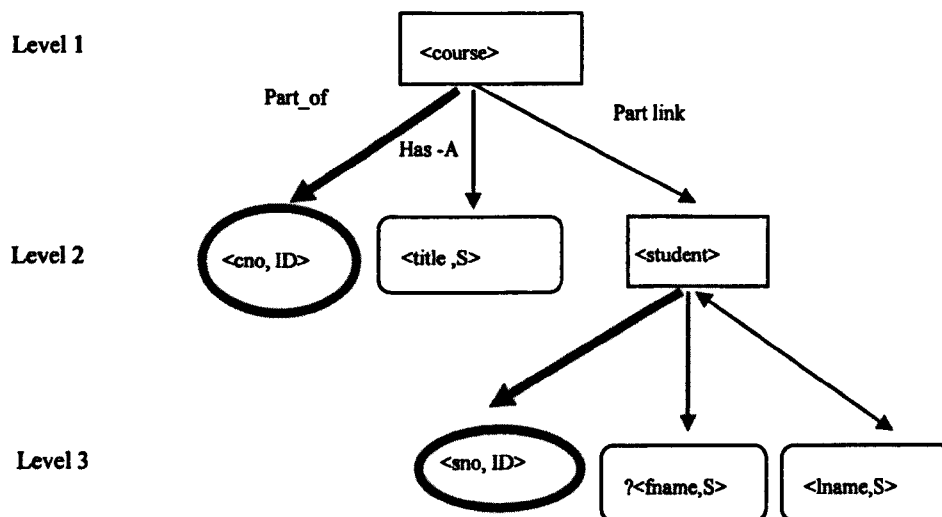


Figure 3.7: Relationship between Complex Element, Attribute and Simple Element Nodes

3.4.3.6 Semantic Constraint Between set Relationship

Sequence between set of child elements nodes

The complex element node may consist of child element nodes in a particular sequence. For example the complex element *student* consists of child elements first name, last name and grade. We say that all child elements of complex element *student* are in sequence: *fname* first, then *lname* and *grade* in the end. To illustrate this, we draw children of a complex element node in a sequence starting from the left position to right end of position in G-DTD.

```
<!ELEMENT student (fname,lname,grade)>
<!ELEMENT fname(#PCDATA) >
<!ELEMENT lname(#PCDATA) >
<!ELEMENT grade(#PCDATA) >
```

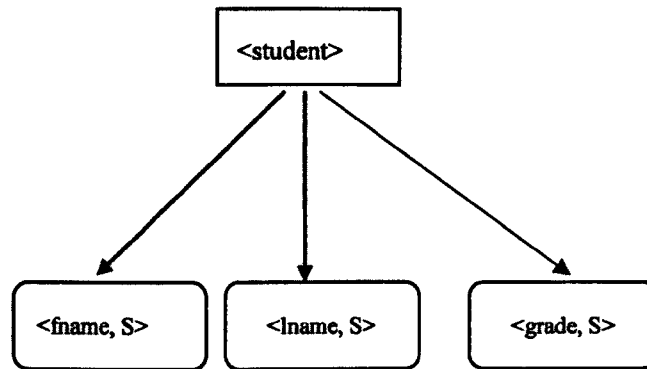


Figure 3.8: Sequence of Simple Elements

However, normally each complex element node consists of a single attribute node or multi attribute nodes. We emphasise in our notation that these attribute nodes must be located first in the sequence before including other simple or complex element nodes. Consider the following segment of DTD and its G-DTD where attribute *sno* is located in first position in the sequence of child elements.

```

<!ELEMENT student (fname, lname, grade )>
<!ATTLIST student
  Sno ID #REQUIRED>
<!ELEMENT fname(#PCDATA)>
<!ELEMENT lname(#PCDATA)>
<!ELEMENT grade(#PCDATA)>
  
```

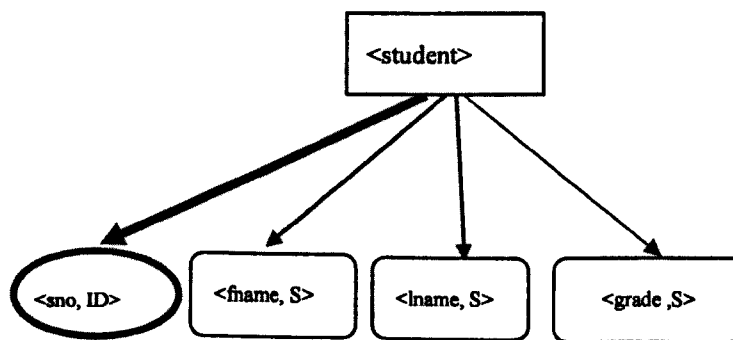


Figure 3.9: Sequence of Attribute and Simple Elements

Disjunction between the set of subelements

We have a set of sub-elements that are in an exclusive “OR” {XOR} relationship to represent notation “|” in DTD. For example, for the complex element node *student* only one of its sub-elements either *fname* or *lname*, appears as its sub-element in the XML document, which is represented in DTD as follows:

```
<!ELEMENT student (fname|lname, grade)
```

As shown in Figure 3.10, in G-DTD, we illustrate this as a line labelled with {XOR} across all the set of relationships involved.

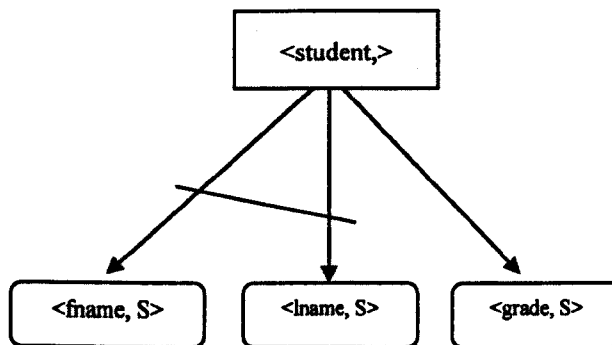


Figure 3.10: Binary Disjunction of Simple Elements

Following is another real example of an application taken from ETDML DTD (Powell, 2007).

```
<!ELEMENT chapter (page| citation| table)* > which is equivalent with  
<!ELEMENT chapter (page*| citation*| table*) >
```

can be represented in G-DTD as follows:

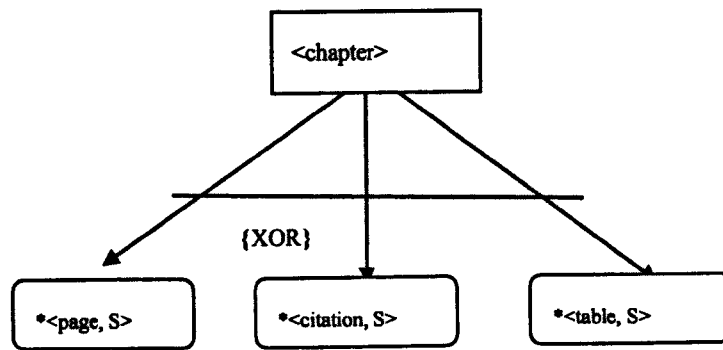


Figure 3.11: Disjunction of Several Simple Elements

Figure 3.11 shows that the line with label {XOR} indicates that more than two disjunction relationships are involved. Generally, this notation is flexible and corresponds to a number of disjunction sub-elements defined in the DTD. Sub-elements *page*, *citation* and *table* are simple multi-valued elements.

3.5 Example of G-DTD

Finally to illustrate the use of G-DTD's notation (summarised in Figure 3.12) let us consider a DTD in Figure 3.2 which describes a university database. Figure 3.13 shows the G-DTD describing the structure of an XML document corresponding to the DTD in Figure 3.2. Root node *Department* has a binary path link with the complex element node *course*.

The simple element node *title* is part-of the complex element *courses*. One *course* can be taken by many *students* while the complex element *student* consists of a sequence of attribute node *sno*, simple elements *fname*, *lname* and complex element *lecturer*. Attribute node *sno* is required for the complex element *student*. Complex element node *student* requires only one of its sub-elements, either *fname* or *lname*, to appear in the XML document while the simple element *lname* is optional.

The semantic relationship between *course*, *student* and *lecturer* is indicated as a ternary relationship since each student is assigned to a lecturer who is teaching the course. The

semantic relationship between them reveals that the *Department* can have one-to-many *courses* at one time.

The complex element *course* has a sequence of attribute *cno*, simple element node *title* and complex element node *student*. The part-of link attribute is a mandatory relationship where the attribute node *cno* is required and unique for every course in the XML document.

Attribute *tno* is required while simple element *tname* is mandatory and string S denotes that a node is a type PCDTA. Attribute key *tno* and simple element *tname* is the last node in G-DTD. The level of each node is indicated explicitly in the model.

<u>Notations</u>	<u>Meaning</u>
	Complex element
	Mandatory simple element, single value, CDATA
	Required simple element, multi value, CDATA
	Optional simple element, single value, CDATA
	Optional simple element, multi value, CDATA
	Composite Attribute
	Reference Attribute
	Required Attribute
	Has_A link simple element (complex element and simple element)
	Part_of link attribute (Complex element and attribute)
	2-ary many-to-one Path link
	3-ary many-to-many Path link
	Disjunction between set of relationships

Figure 3.12: G-DTD's Notations

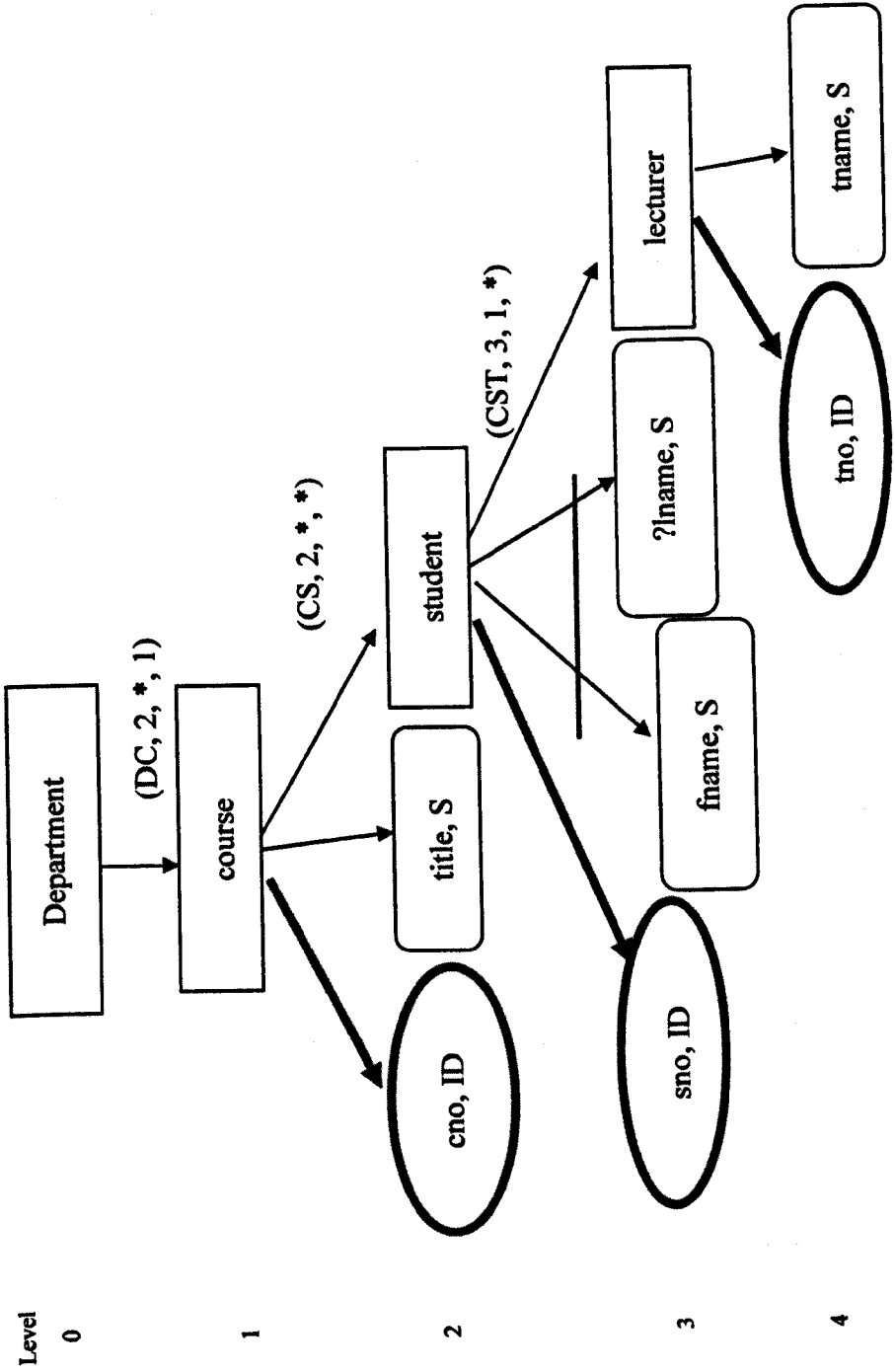


Figure 3.13: G-DTD

3.6 G-DTD Operations

The operations of the G-DTD model describe the dynamic properties of the model. G-DTD model operations are classified into five main parts: Query Operations, Insert operations, Delete Operation, Searching Operations and Update Operation. An operation to determine the root and leaves of the G-DTD is also required. Later, these operations will be used in normalizing the G-DTD into normal forms. In the following description we will conceptually discuss the semantic meaning of these operations according to this classification.

3.6.1 Query Operations

Query operations allow the user to query the node types and information, related nodes and links information defined in G-DTD.

(1) Query a Node Type and Information

The operations of querying node types allow the user to query different types of node stored in G-DTD such as complex element, simple element or attribute nodes. The user can also query the information of a particular node, such as name, level and node type. If the queried node does not exist, an error message is given.

(2) Query a Related Node

Since the structure of G-DTD is like a tree structure, the query operation allows the user to query the related node that links to a particular node using a path through existing link such as a *Path*, *Part_of* or *Has_A* link. For instance, the user can detect the parent of a complex element node by using the *path link* between two complex element nodes. As another example, the simple element for a particular complex element node can be determined through the *Has_A* link.

(3) Query a Path Link

Path links are the most important links in G-DTD. This operation allows the user to query the instance of a path link, such as name of link, degree of relations and parent and child constraint.

3.6.2 Insert Operations

Insert operations allow the user to insert a new complex element node to the G-DTD. When a new node is being inserted in the G-DTD model, the following situations are possible:

- A new complex element node, simple element node or attribute node is created
- A new path link is built between the complex element node and created complex element node
- A new Has_A link is built between the created complex element node and a simple element node
- A Part_of link is built between the created complex element node and an attribute node

To ensure the new node is not redundant with any node in the given G-DTD, it must be tested whether the node already exists. Then the proper location of the new node needs to be determined before it can be inserted into the G-DTD. More importantly, it must satisfy the data integrity constraint of the given G-DTD.

(1) Inserting a Node

In this case a new node is inserted into the G-DTD. Before the user can insert a new node, it must be created first. Whether the new node is a complex element, simple element or attribute node, the properties of the inserted node such as ID, level and types are inserted and stored together in the G-DTD. The operation of inserting a new node implies that when the node is inserted, related nodes such as parent node or child node should be reported to the user since the structure of the G-DTD is changed.

If the newly inserted node is a complex element node, the position of the new complex element node is based on the rules provided in the normalization procedure (see Chapter 4). In such a situation, a path link is created with its parent node. In this case, the parent node may be a root node or another complex element node based on the normalization rules provided. However if the created node is a simple element or an attribute node, a Part_of link or Has_A link is built between it and the parent node, which is a complex element node.

(2) Inserting an Instance of a path link

Inserting an instance of a path link means that the semantic relation between two complex element nodes has to be created. The user needs to know the semantic relationships before he/she can insert them to the G-DTD. The user can make links and insert the corresponding link information such as name, degree, parent constraint and child constraint. In contrast, for a part-of link and has-a link, the user is not required to put any instance for the links.

3.6.3 Delete Operations

Delete operations results in the corresponding data being removed from the G-DTD. Since the structure defined in the G-DTD is a tree structure, delete will affect the location of the existing nodes in the G-DTD, especially the parent node and child node. The delete operation in G-DTD must satisfy the conditions and constraints given in the normalization rules defined in Chapter 4. In the following, we will discuss the different situations of delete operations in the G-DTD

(1) Deleting a Complex Element Node

Deleting a complex element node is a complex deleting process in G-DTD. This is because every complex element node is related to its parent node and child node.

Before the process of deleting a complex element node is started, it is important for the user to find its related nodes such as parent node and child nodes.

Eventually, by deleting a complex element node, its attribute and simple element nodes with the relevant Part_of and Has_A links are automatically deleted as well. A new link is built up with its new parent node and child node.

(2) Deleting a Path Link type and its Instance

According to the path link type definition, each instance of a path link type represents a semantic relationship between two complex element nodes. When such an instance is deleted, the specific relationship between the two nodes has no further semantic link between them.

3.6.4 Replicate Operations

Replicate operations copy the name of the current node such as simple element node and attribute node and create a replicated node. However the new level and new ID of a copied node are depending on the current complex element node. The replicated node can be moved around from one location to another. In the process of replicating a node, all the related nodes including complex element nodes should be notified if the replicating node has a relationship with them. It may be necessary to move an attribute node and simple element node up to another level when there exists dependency between an attribute node and a simple element node. In this situation, it is not necessary to create a new complex element node but rather to restructure the G-DTD by moving up the node at level n (n_n) to level $n-1$ (n_{n-1}).

3.6.5 Determine the root node and last node

This operation will determine the root node and last node (last level) in the G-DTD. The last node may be a simple element node or attribute node. These operations are very

important because in order to avoid duplication, we need to move the corresponding node as near as possible to the root node.

3.7 Rules for converting G-DTD to DTD

Another feature of our approach is that instead of generating XML DTD directly, we first generate a conceptual schema, G-DTD. After obtaining a set of normal forms G-DTD, we can apply the transformation rules to generate a DTD of an XML document. This approach is adopted from Mok and Embley (2006) with some modification to suit the tree structure of G-DTD. In this section, we describe the mapping from the G-DTD consisting of nodes, links and constraints to the DTD, which is mainly concerned with elements/attributes declarations and simple/complex element type definitions. In principle, each basic node in the G-DTD can be mapped to either an element or an attribute. Each complex element node in the G-DTD can be transformed to an element, whose content may include embedded sub-elements. The transformation rules include both generic and semantic rules. This mapping simply represents G-DTD syntactically in this XML document schema in one-to-one correspondence, depth by depth starting from the lowest node depth. For this transformation, we must take into account the hierarchical characteristic of XML document structure. In the following, we describe our transformation rules which incorporate general rules and semantic rules. General rules consist of the following four categories: Root node; Complex Element node/ Simple element node and Attribute node. Semantic Rules are applied for element and sub-element relationship, cardinality constraint, sequence and disjoint.

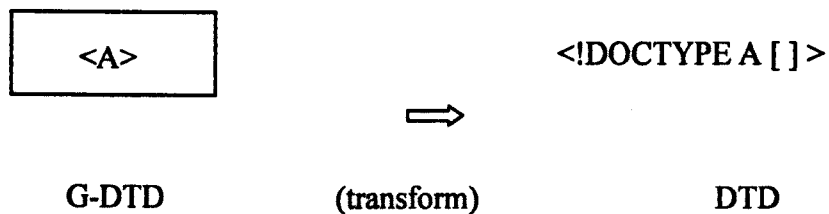
3.7.1 General Rule

In the following, we illustrate the G-DTD construct while explaining the transformation of this construct into the DTD fragments.

Root node

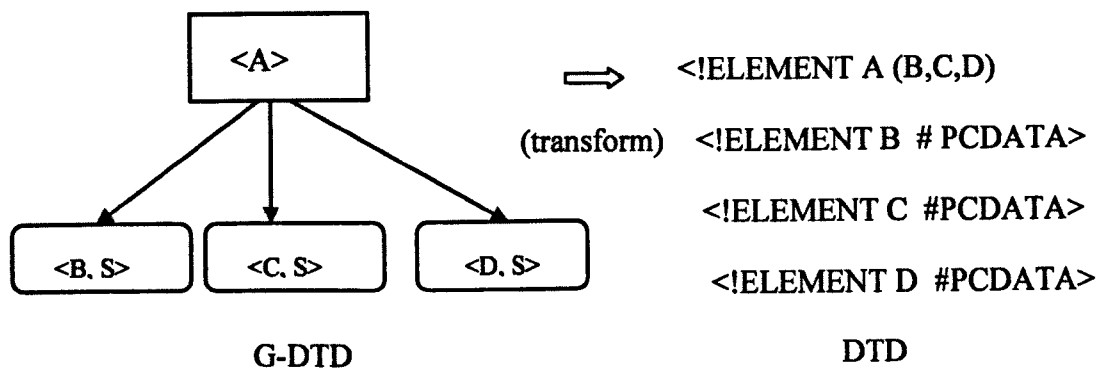
A special type of complex element node is a root node which has the same properties, consisting of node label and level. Thus, we transform the G-DTD root node into a DTD element type declaration. As indicated in step 1 of Figure 3.14, the root node name A will be mapped to the name of the element type starting with `<!DOCTYPE A[subelement] >`

The following graphical notation for root node will be transformed to the following syntax



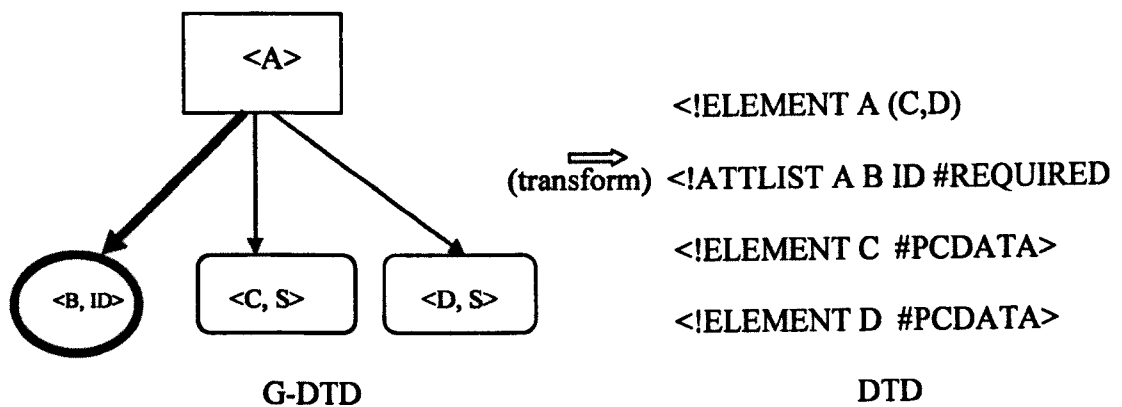
Complex Element node and Simple element node

The complex element node label becomes the name of the ELEMENT types. The simple elements are transformed into ELEMENT content description. This is motivated by considering a simple element node to be part of a complex element node. The name of simple element node provides the name for the simple element type in content specification. In G-DTD simple element names are mandatory. If there is no more sub element node representing a suitable declaration for the simple element node, the simple element node type is assumed to be a simple element type whose content type is #PCDATA. The following diagram depicts a complex element node and its simple element node with the transformation result.



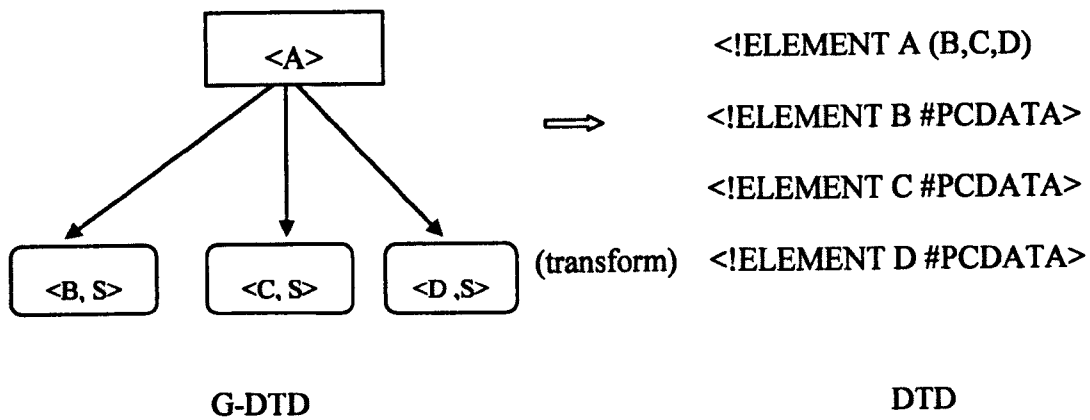
Attribute node

Similarly, each attribute node consists of label, level and type node. After the transformation, such attribute that has an additional ID attribute will be declared as its label followed by ID #REQUIRED

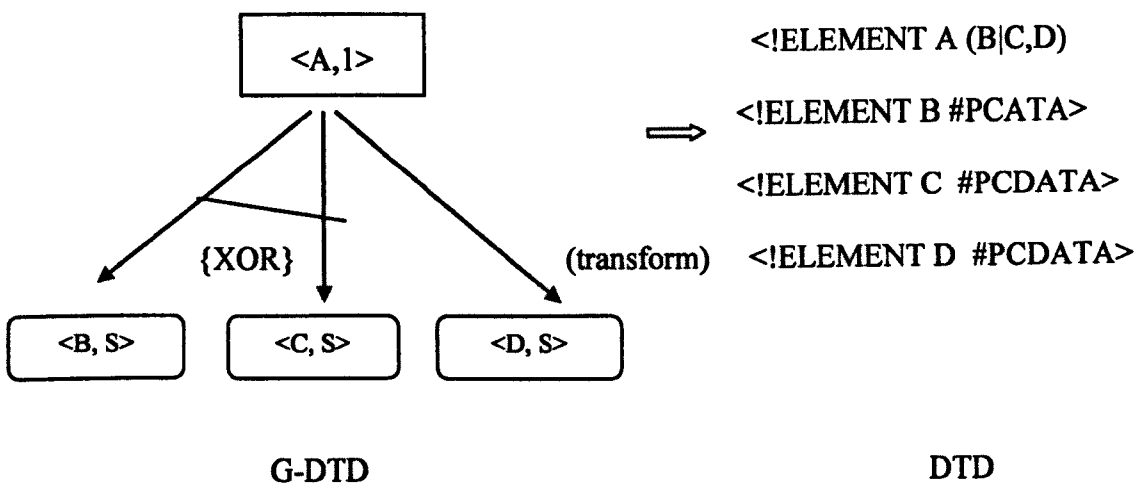


3.7.2 Semantic Rule

The sequence among nodes, especially the sequence among sibling simple elements within complex element, is significant. The sequence of sibling elements within parent element is represented left to right in the graph. The order in the sequence is explicitly given by the notation, an up-curved arrow. As shown in the following diagram, after mapping, DTD element types appear exactly in a sequence. The Has_A link specifies a relationship between complex element node and simple element node.

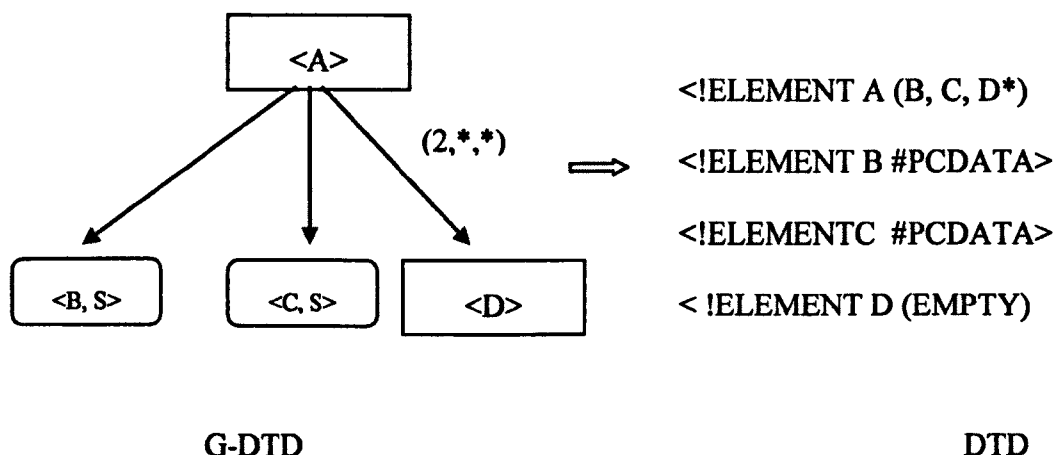


Disjunctive Relationship



Sub-elements of complex element A; simple element B and C are linked using has a link relationship and illustrated using {XOR}. This portion of the diagram is transformed as fragment DTD using notation “|” to represent a disjunctive relationship between complex element A and sub-elements B and C.

Cardinality Semantic



The cardinality specifications specified in a path link between complex element A and complex element D will be mapped into cardinality specification with operator ?, *, + . For example a many to many path link between complex elements A and D is mapped using operator *.

In Figure 3.14 we present a straightforward transformation rule which combines together the general and semantic rules to map from G-DTD back to DTD. This mapping rule requires the input to be a G-DTD and the output is DTD. This algorithm consists of 4 steps. Generally, step 1 is to select a root element node N of G-DTD and generate `<!DOCTYPE N [<complex element type definition> <simple element type definition>]>`. In step 2, the initial structure of G-DTD is determined to identify the number of node, types of nodes and relationship type. In Step 3, for each complex element name at the first level that appears in sequence in the G-DTD, replace `< complex element type definition >` by `<!ELEMENT N (<sub-element of Ni>)` where N_i is the list of sub-element nodes. The relationship type and cardinality semantic (semantic constraint) will be verified to map using the right symbols. In step 4, starting from level 2 of G-DTD, the nodes need to be traversed using depth first traversal, where the complex element from the left side will be first identified in the sequence. The process of step 3 is recursively applied until the last level of G-DTD. All attribute nodes with part-of link will be

replaced with `<! ATTLIST N_i attribute name ID # REQUIRED>` and a simple element node with has-a link will be replaced with `<!ELEMENT simple element name #PCDATA>`. The process in step 4 is continued for all nodes in sub tree G-DTD.

Step 1 Level 0, a root node is represented by `<!DOCTYPE root node name [<Complex element type definition> <simple element type definition>] >`

Step 2 Level 1, identify the subtree of G-DTD, check the number of nodes, type of nodes and relationship type

Step 3 If there is more than one node at level 1 and the relationship type between root and child node(s) is a binary one-to-many/many-to-one hierarchical link then generate

`<!ELEMENT root node name (N_i)>`

Where N_i is the list of subelements/child nodes

3.1 Certify the relationship set between *parent* nodes and *child* nodes,

3.1.1 If {XOR} means the relationship between node is a disjunction and will be represented using symbol '|'

Else

3.1.2 If {sequence} means the relationship is sequence and will be represented using symbol ','

3.2 Verify the semantic constraint between complex element nodes (*parent*) and *complex element* nodes (child) in each relationship set and map to the following operator:

3.2.1 if (m, 1, *) or (m, *, *) or (m, *, 1) map to operator *

3.2.2 if (m, 0, *) map to operator +

3.2.3 if (m, 0, 1) map to operator ?

Where m is n-ary relationship and $n > 1$

Step 4 If the list of subelements (N_i) is not empty, using depth first traversal, for each node in list subelement N_i

4.1 generate `<! ELEMENT N_i (subelement N_j)>`

4.2 repeat step 3.1 and 3.2

4.3 for each complex element (N_i), if the relationship between them is part-of link attribute (one-to-one) then generate

<! ATTLIST *N_i* attribute name ID # REQUIRED>

4.4 For subelement *N_j*

- 4.4.1 If *N_j* is a simple element has part of link simple element (many-to-one or one-to-many relationship) with *N_i* then generate

<! ELEMENT simple element name #PCDATA>

(Repeat for all simple element nodes)

- 4.4.2 If *N_j* is a complex element node has path link with complex element *N_i*

Repeat step 4

- 4.4.3 If *N_j* is a complex element node has part of link then generate

<! ELEMENT *N_j* (EMPTY) >

Step 5 Go to next subtree G-DTD and repeat step 4

Figure 3.14: Transformation Rules

3.8 Summary

In this chapter, we have presented several new features provided by G-DTD for describing XML documents. The most important feature of this model is that the complex elements; simple elements and attribute nodes are clearly distinguished and presented. In particular, we illustrate how binary or n-ary relationships through parent-child relationship can be represented using G-DTD. The sequence and disjunction between the nodes are presented as well and the path structures of the node are shown by level indicators. The relationships between complex element node, simple element node and attribute nodes are illustrated using Path link, Part_of link and Has_A link. We also developed transformation rules to generate new DTD.

We will propose a notion of normal forms for G-DTD on the basis of Arenas and Libkin's (2004) normalization theory. The normal forms of G-DTD will be used later as a guideline for the user to improve the quality of XML documents by reducing undesirable redundancies caused by functional, transitive, partial and local dependencies from the schema level. A normalization algorithm will be developed to convert from G-DTD to a normal form one. This will be presented in the next chapter.

Chapter 4

Normal Forms for XML Documents

4.1 Introduction

The concept of database design and normal forms are key components to achieve a high performance database design. Normal forms which have been introduced in normalization theory have been studied extensively for the relational model (Codd, 1970; Fagin, 1977). Three normal forms were initially proposed called first (1NF), second (2NF), and third (3NF) normal forms. Subsequently, Boyce and Codd introduced a stronger definition of the third normal form called the Boyce Codd Normal Form (BCNF) (Codd, 1974).

The purpose of normalization is primarily to remove redundancy that is, storing the same information several times in the database. If this happens, it will have impact whenever information is updated (i.e. update anomalies), and may use more space than needed. In the normalization process, the initial poorly designed relational schema is decomposed into an equivalent set of well-designed schemas, i.e. into schemas in desired normal forms (usually is 3NF and often also in BCNF). As presented in Chapter 2 (see Section 2.1), normalization involves the identification of the required attributes and their subsequent aggregation into normalized relation based on functional dependencies between attributes.

In this chapter, we will describe a systematic approach to designing a quality XML schema design in order to achieve a redundancy-free XML document. Similar to relational database theory, we propose a set of normal forms for XML documents. The set of normal forms is defined on the basis of G-DTD which has been presented in Chapter 3. Since G-DTD also utilizes association between complex elements, simple elements and attributes as a basis to identify the DTD, it is possible to apply data dependency principles in the conceptual data model. We believe the capability of G-DTD to capture more semantics will make the approach of normalising XML documents

more practical and speed its implementation. Normal forms of G-DTD are generalised and simplified from the normal form proposed by Arenas and Libkin (2004) and Lv et al., (2004).

Our contribution in this chapter is as follows:

- We apply the data dependencies concept to represent G-DTD semantic.
- We present a set of a normal forms for XML documents called first normal form, second normal form, third normal form and fourth normal form on the basis of the G-DTD model
- We propose algorithms to transforms G-DTD into the respective normal form G-DTD.
- We provide a case study to demonstrate the process of normalization of XML documents based on G-DTD.

The rest of the chapter is organized as follows. In section 4.2 we present various definition types of data dependencies for G-DTD. Section 4.3 defines multiple levels of normal forms for G-DTD. The normalization procedure of XML document based on G-DTD is presented in section 4.4. A case study is given in section 4.5 to illustrate the application of normal forms and normalization rules in designing a university database.

4.2 Data Dependency of G-DTD

In Chapter two (see Section 2.3.3) we have presented various definitions of data dependency, especially XML functional dependency in defining XML normal forms. In this chapter, we adopt the definition presented by Arenas and Libkin (2004) and Lv et al. (2004). We simplify this XML normal form and present them in a more practical way and provide a simple definition which can be easily understood by database designers.

It is well known that data dependencies are part of the real world semantics (Ling, 1985; Arenas and Libkin, 2004; Vincent et al., 2007; Lv et al., 2004; Wang and Topor, 2005). They represent the semantic information in the form of relationships between different attributes in the XML documents (Maier, 1983). In Ling (1985), it is stated that “*data dependencies should be modelled precisely early in the design stage for a correct and complete database representation of semantic.*” .

In line with the above statement, we provide various types of structural properties in the G-DTD model, which can be applied in the normalization process later. In our approach, first we consider two types of data dependency concept: functional dependency and key dependency and later we propose normal forms for G-DTD on the basis of these data dependencies.

To demonstrate the idea of data dependencies and normal forms for G-DTD, we represent again the same example of XML document with its G-DTD as presented previously in Chapter 3 section 3.5.

```
<!DOCTYPE department [
  <course>
    <course cno = "csc101">
      <title> XML database </title>
      <student >
        <student sno = "112344">
          <fname> David</fname>
          <lname> Grey</lname>
          <lecturer>
            <lecturer tno = "123">
              <tname>Bing </tname>
            </lecturer>
          </student>
        <student >
          <student sno = "112345">
            <fname>Helen </fname>
            <lecturer>
              <lecturer tno = "123">
                <tname> Bing </tname>
              </lecturer>
            </student>
          </student>
        </course>
      </course>
    </department>
  ]>
```

```

    </lecturer>
  </student>
</course>
<course>
  <course cno = "csc102">
    < title > Z formal Method </title>
    < student >
      <student sno = "112344">
        <fname> David</fname>
        <lname>Grey </lname>
        <lecturer>
          <lecturer tno = "124">
            <tname> Bottaci </tname>
          </lecturer>
        </student>
      < student >
        <student sno = "112345">
          <fname>Helen </fname>
          <lecturer>
            <lecturer tno = "124">
              <tname> Bottaci </tname>
            </lecturer>
          </student>
        </course>
      </courses>
    </Department>]

```

Figure 4.1: XML Document Conforming to G-DTD in Figure 4.2

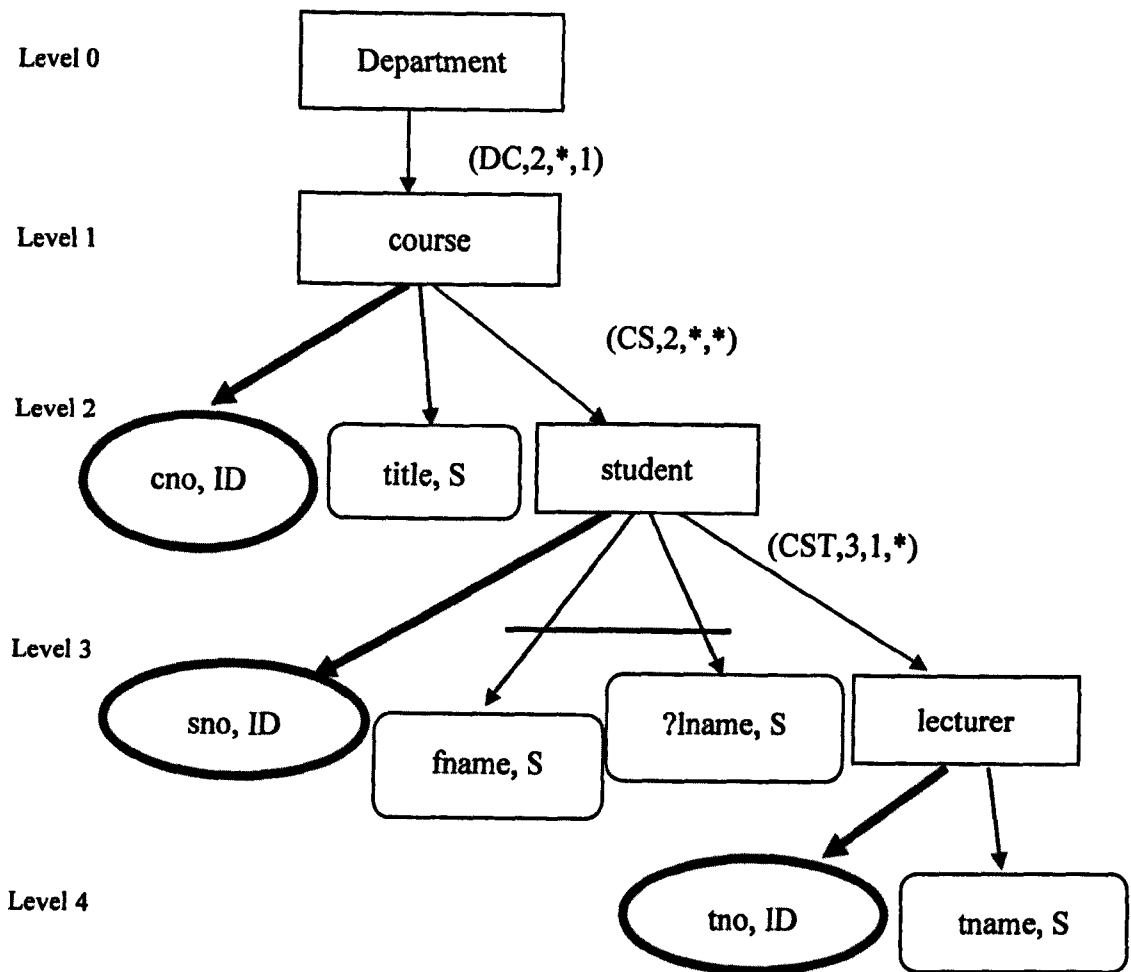


Figure 4.2: G-DTD

4.2.1 Key Dependency (KD)

We define a key dependency as a unique attribute that can determine uniquely other simple elements in the G-DTD. Referring to G-DTD in Figure 4.2, for instance, *course number (cno)*, *student number (sno)* and *lecturer number (tno)* are unique and mandatory, because they are represented as one-to-one relationships between complex elements and attribute nodes. We define this in G-DTD as a key attribute similar to Buneman et al., (2001). For instance, each *course* contains a unique *course number* or each *lecturer* has a unique number is written as follows:

$$\langle cno, 2 \rangle \rightarrow \langle course, 1 \rangle \text{ ----- (4.1)}$$

$$\langle tno, 4 \rangle \rightarrow \langle lecturer, 3 \rangle \text{ -----(4.2)}$$

4.2.2 Functional Dependency (FD)

Functional dependency models real world constraints, showing the dependencies among complex elements and simple elements/attributes. Based on the G-DTD model, we define three types of functional dependencies: global functional dependency, transitive functional dependency and partial functional dependency.

Global Functional Dependency (GFD)

GFD occurs based on a few constraints (dependencies) that a database designer may specify for his/her application. GFD holds in G-DTD if a dependency occurs between an attribute and simple element node of a particular complex element under n-ary many-to-many path link are leaves node of G-DTD. Both attribute and simple element are the children for a particular complex element node. We adopted this definition from (Connolly and Begg, 2002; Arenas and Libkin, 2004) with some modification to suit the tree structure of G-DTD.

Definition 1:

Let *CE* be a complex element node under binary or *n*-ary one-to-many/many-to-one/many-to-many path link. Let *ATT* be set identifier attribute node of *CE* and *SE* is set simple element node of *CE*. The GFD of *G-DTD* is defined as follows:

(1) For each attribute node $\langle ATT, l_{n-1} \rangle$ of $\langle CE, l_{n-2} \rangle$ and simple element node $\langle SE, l_{n-1} \rangle$ of $\langle CE, l_{n-2} \rangle$, where $\langle SE, l_{n-1} \rangle$ and $\langle ATT, l_{n-1} \rangle$ are leaves nodes in *G-DTD*.

$\langle ATT, l_{n-1} \rangle \rightarrow \langle SE, l_{n-1} \rangle$ is a GFD in *G-DTD* iff

(a) $\langle SE, l_{n-1} \rangle$ is fully functionally dependent on $\langle ATT, l_{n-1} \rangle$ but not on any proper subset of $\langle ATT, l_{n-1} \rangle$. $\langle SE, l_{n-1} \rangle$ can be a set of simple elements, list of simple elements or single value of a simple element.

(b) Both $\langle ATT, l_{n-1} \rangle$ and $\langle SE, l_{n-1} \rangle$ must be located at the same level in *G-DTD* and share the same parent node.

For instance, in Figure 4.2, the possible GFD of the complex element *lecturer* is as follows:

$$\langle tno, 4 \rangle \rightarrow \langle tname, 4 \rangle \text{ -----(4.3)}$$

This above GFD (4.3) represents the constraint, whenever two elements agree on the value of all attributes *tno*, they also agree on the value of all attributes in *tname*. This GFD can be viewed as a function from one set of attributes/simple elements to another set of attributes/simple elements. Similar to the relational model, the left hand side (LHS) of the arrow of GFD is called a determinant. For example, *tno* is the determinant of *tname*.

Transitive Functional Dependency (TFD)

We adopted the definition of transitive dependence from the relational model (Codd, 1972) and nested relation (Lv et al., 2004). TFD between complex elements occur if their attribute or simple element node has dependency with another simple element node from a different level.

Definition 2:

Case 1: Let ATT be a key identifier for complex element (CE) and $\{SE_a, SE_b\}$ are simple elements for CE

If there exist two constraints $\langle ATT, l \rangle \rightarrow \langle SE_a, l \rangle$ and $\langle SE_a, l \rangle \rightarrow \langle SE_b, l \rangle$, then we say that attribute $\langle SE_b, l \rangle$ is transitively dependent on $\langle ATT, l \rangle$.

Case 2: Let ATT_a be a key identifier for CE_a , ATT_b is a key identifier to CE_b and SE is a simple element for CE_c and they are located in different levels.

If there exist two constraints:

FD1: $\langle ATT_a, l \rangle \rightarrow \langle ATT_b, l_{+1} \rangle$

FD2: $\langle ATT_b, l_{+1} \rangle \rightarrow \langle SE, l_{+1} \rangle$,

then we say that

$\langle ATT_a, l \rangle \rightarrow \langle SE, l_{+1} \rangle$ is a TFD because $\langle SE, l_{+1} \rangle$ is transitively dependent on $\langle ATT, l \rangle$ and they are located at different levels.

Partial Functional dependency (PFD)

We adopted the definition of PFD from Lv et al. (2004).

Definition 3:

Let ATT_a, ATT_b, ATT_m be a key identifier for CE_a, CE_b and CE_m respectively. These CE have a binary or ternary relationship with each other and are located in different levels.

If there exist two constraints

FD1: $\{\langle ATT_a, l \rangle, \langle ATT_b, l_{+1} \rangle, \langle ATT_m, l_{+2} \rangle\} \rightarrow \langle ATT_m, l_{+3} \rangle$

FD2: $\langle ATT_a, l \rangle \rightarrow \langle ATT_m, l_{+3} \rangle$

Constraint FD2 is called PFD because it is a subset of FD1 where attribute $\langle ATT_a, l \rangle$ alone can be used to determine $\langle ATT_m, l_{+3} \rangle$

PFD involves composite attribute keys. The composite key attribute could be from the same level or from a different level. The subset of composite key attribute can functionally determine the simple element node.

Relationship Dependency (RD)

The relationship dependencies are presented clearly in the G-DTD diagram using a directional arrow. Our XML relationship dependency is defined in terms of structural constraints of *path link* in the relationship between complex element nodes in G-DTD. These types of relationships can cause data redundancy and must be eliminated from G-DTD. The categories are: *one-to-many n-ary path link*, *many-to-many n-ary path link*, and *many-to-one n-ary path link dependency where $n \geq 2$*

For instance, G-DTD in Figure 4.2 indicates the presence of a *binary many-to-one path link* between complex elements *department* and *course* represented as $\langle department, 0 \rangle \xrightarrow{R} \langle course, 1 \rangle$ where R is $(DC, 2, *, 1)$ DC is an abbreviation for both name of complex element department and course.

As summary, we defined data constraints into semantic constraints and structural constraints. Both KD and FD are categorized under semantic constraints while RD is categorized under structural constraints. KD is defined based on attribute node key for each of complex element node. FD such as a GFD, PFD and TFD is defined on the basis of attribute nodes and simple element nodes of a complex element node and presented as LHS and RHS of FDs. Every attribute node and simple element node is associated with its level and this indicates the depth of the nodes in the G-DTD model. As shown in the definitions, for the purpose of simplicity, we make an assumption that:

- (1) *Every complex element node in G-DTD has an attribute node.*
- (2) *Every node, whether attribute node, simple element node or complex element node, has a unique name. FDs such as GFD, TFD and PFD are in ' $\langle X, level \rangle \rightarrow \langle Y, level \rangle$ ' form where X and Y represent the LHS and RHS of the FDs respectively.*

- (3) *A set of FD of a G-DTD is presented as two element sets, one for LHS and the other for RHS set. Obviously, the order of attributes and element nodes in such a set is important.*

Compare with definitions defined by Arenas and Libkin(2004) and Lv et al.(2004), they use path expression based on “tree-tuples” to define the KD and FD (see definition in Chapter 2 (Section 2.3.4.) but we simplified their definition using unique node name and level as indicated in G-DTD model.

4.3 Normal Forms for G-DTD

A normal form specifies a set of syntactic conditions that a well-designed schema should satisfy (Codd, 1972). As mentioned in earlier section, normal form usually deals with removing redundancies from a database to avoid possible anomalies during update or insertion of data. Like the relational model (Codd, 1972), we next define a set of normal forms for G-DTD. These normal forms are on the basis of data dependencies and semantic relationship of G-DTD.

4.3.1 First XML Normal Form (1XNF)

The first normal form for G-DTD is about finding unique identifier attributes for the complex elements set, and checking that no node (complex element, simple element or attribute) actually represents multiple values. To be in first normal form, each attribute, complex element or simple element is not NULL and has a single label. Only one value for each simple element node or attribute node of G-DTD can be stored. If there is more than one value, we must add some new element nodes or attribute nodes to store them. For instance, consider G-DTD in Figure 4.2. If the complex element *course* has two titles, we need two *title* simple element nodes for each *course* to store the two *title* names. This is equivalent to having 'no repeating group' in relational schema (Codd, 1972). More importantly, the primary key (unique identifier) for the complex element must be defined. To be precise, we propose the following rules.

G-DTD is in first normal form if and only if:

- (a) For all attribute, complex element and simple element node in G-DTD must have exactly one unique label.
- (b) Each complex element node in the G-DTD has at least one key attribute node.
- (c) For all Path link relationship, the parent and child of the relationship must be a complex element node.
- (d) For all Part_of link relationship, the parent must be a complex element node and the child must be a simple element node.
- (e) For all HasA link relationship, the parent of the relationship must be a complex element node and the child must be is an attribute node.

4.3.2 Second XML Normal Form (2XNF)

If a relationship between a parent node (complex element node) and child node (complex element node) is many-to-one (such as *department* and *course*, as shown in Figure 4.2) the potential for data duplication might exist. Some nodes need to be restructured. However they can then still be in a single G-DTD. This is possible in XML because XML supports hierarchies in a single document, while relational databases do not support hierarchies in a single row. This is different from the relational second normal form (2NF), which requires many-to-one relationships to be in separate tables.

The G-DTD is in second normal form if and only if:

- (a) G-DTD is in 1XNF.
- (b) There is no nested binary *path link* or ternary *path link* under many-to-many/many-to-one or one-to-many *path links* with the following condition:
For each nested complex elements $\langle CE, l+1 \rangle$ of $\langle CE, l \rangle$, and any key attribute (ATT) of $\langle CE, l \rangle$, the key attribute and simple element node of $\langle CE, l+1 \rangle$ is not **partially functionally dependent** on ATT of complex element node $\langle CE, l \rangle$.

4.3.3 Third XML Normal Form (3XNF)

In the third normal form of G-DTD, making a change to one unique complex element node set would not affect the integrity of another complex element node set. If needed, a complex element node set would be divided into two separate complex element node sets.

G-DTD is in third normal form if and only if:

- (a) G-DTD is in 2XNF.
- (b) There exists no nested path link type of n -ary one-to-many or many-to-many under a many-to-one path link set in G-DTD and the following conditions are satisfied:
 - (i) For each nested set of complex elements $\langle CE_b, l_{+j} \rangle$ of set of complex elements $\langle CE_a, l \rangle$, any key attribute and simple element of $\langle CE_b, l_{+j} \rangle$ is **not transitively functionally dependent** on ATT of complex element $\langle CE_a, l \rangle$.
 - (ii) Any key attribute node of any complex element node located in a different level is disjoint ($ATT\langle CE, l \rangle \cap ATT\langle CE, l_{+j} \rangle \cap ATT\langle CE, n \rangle = 0$).

4.3.4 Fourth XML Normal Form (4XNF)

GN- DTD is in fourth normal form if and only if:

- (a) G-DTD is in 3XNF.
- (b) For nested one-to-many/many-to-one or many-to-many path links in G-DTD, the following condition is satisfied:
 - (i) There are no **GFD** between attributes and simple elements of complex element nodes under nested many-to-one or many-to-many *path links*.

As summary, the set of normal forms proposed above are based on FD and RD among attributes, simple element and complex elements of G-DTD. The higher normal form such as 4XNF is better than others because it can avoid redundant data in XML document which can causes update anomalies in implementation. In following section, we discuss the process of normalization, which is important for the XML document

schema design. In section 4.5 we present a case study to demonstrate the use of this set of normal forms in the normalization process in detail.

4.4 The Process of Normalization

The common feature of normalization procedure is to convert an initial schema into one in a normal form to reduce anomalies and redundancies in the XML document. In this section, we propose transformation rules to convert the un-normal form G-DTD into a normal form one. Two approaches to normalize G-DTD are given: First, the approach of restructuring the G-DTD which affects the relationship and location of the nodes in G-DTD. Second, we focus on normalization rules for G-DTD with data dependencies on the basis relationship dependency, GFD, PFD and TFD. Generally, in these rules, we first restructure the G-DTD by creating a new complex element node, moving up node and moving sub-tree node. We subsequently adjust the semantic relationship between simple element nodes or attribute nodes in G-DTD. We next define the normalization rules.

4.4.1 Normalization Rules

The following notations will be used in the following rules:-

r represents root element

ATT represents attribute

SE represents simple element

CE represents set of complex element

\rightarrow^R represents relationship

l represents level of node where $(0 \leq l \leq n-1)$, *n* is a finite positive number

Rule 1: Restructure n-ary many-to-many or one-to-many/many-to-one path link with partial functional dependency

To restructure n-ary many-to-many or one-to-many/many-to-one relationship we must avoid multi-nested path link. For each, n-ary relationship R ($n > 2$), many-to-many or one-to-many/many-to-one relationship type

Let $(\langle CE_a, l \rangle \rightarrow^R \langle CE_b, l_{+1} \rangle)$ be a path link and let $\theta_1: \langle att, l_{+2} \rangle \rightarrow \langle se, l_{+2} \rangle$ is a PFD, where $\langle ATT, l_{+2} \rangle$ and $\langle SE, l_{+2} \rangle$ are children for $\langle CE_b, l_{+1} \rangle$

- 1.1 Create a new complex element name $\langle CE_b_new, l \rangle$.
- 1.2 Insert node $\langle CE_b_new, l \rangle$ at the same level of level $\langle CE_a, l \rangle$ at the rightmost position of G-DTD.
- 1.3 Create a new relationship type of binary one-to-many binary *path link* between parent of $\langle CE_a, l \rangle$ and new complex element name $\langle CE_b_new, l \rangle$.
- 1.4 Replicate all children of $\langle CE_b, l_{+1} \rangle$ to be children of the complex element $\langle CE_b_new, l \rangle$.
- 1.5 Delete all children of $\langle CE_b, l_{+1} \rangle$ except the key attribute node.
- 1.6 PFD θ_1 is transformed to θ_1' : $\langle ATT, new_l \rangle \rightarrow \langle SE, new_l \rangle$ where new_l is a level equivalence to CE_b level.

Rule 2: Restructure binary many-to-many/many-to-one/one-to-many path link with transitive functional dependency

For each binary many-to-one relationship type

$$(\langle CE_a, l \rangle \rightarrow^R \langle CE_b, l_{+1} \rangle)$$

If exist, $\langle CE_a, l \rangle$ with attribute ATT_a and $\langle CE_b, l_{+1} \rangle$ with attribute ATT_b , and simple element SE_b

$$\text{Where } \theta_1: \langle ATT_a, l_{+1} \rangle \rightarrow \langle ATT_b, l_{+2} \rangle$$

$$\theta_2: \langle ATT_b, l_{+2} \rangle \rightarrow \langle SE_b, l_{+2} \rangle$$

$\theta_3: \langle ATT_a, l_{+1} \rangle \rightarrow \langle SE_b, l_{+2} \rangle$ is a TFD

- 2.1 Move up the set complex element $\langle CE_b, l_{+1} \rangle$ along with its children to the same level $\langle CE_a, l \rangle$
- 2.2 If parent of $\langle CE_a, l \rangle$ is a root node
 - (a) Then create a new relationship type of many-to-one *path link* between parent of $\langle CE_a, l \rangle$ with set complex element node $\langle CE_b, l_{+1} \rangle$
 - else
 - (b) Create new relationship type of many-to-many *path link* between parent of $\langle CE_a, l \rangle$
- 2.3 TFD $\theta_3: \langle ATT_a, l_{+1} \rangle \rightarrow \langle SE_b, l_{+2} \rangle$ is transformed to $\theta_3': \langle ATT_a, l_{+1} \rangle \rightarrow \langle SE_b, new_l \rangle$ where *new_l* is equivalence to level of $\langle ATT_a, l_{+1} \rangle$

Rule 3: Restructure binary many-to-many/many-to-one/one-to-many path link with global functional dependencies

Let $\theta_1: \langle ATT, l_{k-1} \rangle \rightarrow \{ \langle SE, l_{k-1} \rangle \}$ is GFD corresponding for complex element $\langle CE, l_{k-2} \rangle$

- 3.1 Create a new complex element name $\langle CE_new, l \rangle$
- 3.2 Insert node $\langle CE_new, l \rangle$ at level one ($l=1$) at the rightmost position of G-DTD
- 3.3 Create a new relationship type of binary many-to-one binary *path link* between root and new set complex element name $\langle CE_new, l \rangle$
- 3.4 Replicate attribute node $\langle ATT, l_{k-1} \rangle$ and simple node $\langle SE, l_{k-1} \rangle$
 - 3.4.1 Rename them as $\langle ATT, l_{new} \rangle$ and simple node $\langle SE, l_{new} \rangle$ respectively where $l_{new} = \text{level of } (\langle CE_new, l \rangle) + 1$
 - 3.4.1 Make them as children to node $\langle CE_new, l \rangle$
 - 3.4.2 Let new attribute node $\langle ATT, l_{new} \rangle$ of $\langle CE_new, l \rangle$ be a key node
- 3.5 Create a new relationship type of *part-of link* between attribute node $\langle ATT, l_{new} \rangle$ and simple element node $\langle SE, l_{new} \rangle$ with $\langle CE_new, l \rangle$

3.6 Delete simple element node $\langle SE, l_{n-1} \rangle$ from its original location and relationship

3.7. GFD $\theta_1: \langle ATT, l_{k-1} \rangle \rightarrow \{ \langle SE, l_{k-1} \rangle \}$ is transformed to $\theta_1': \langle ATT, l_{new} \rangle \rightarrow \{ \langle SE, l_{new} \rangle \}$

4.4.2 Normalization Algorithms

Normalization is a process that analyses and restructures the schema of an XML document to minimize redundancies with the help of data dependencies in the data. In our approach, the normalization algorithm takes G-DTD and set of data dependencies specified by the user as an input, and returns the normal form G-DTD as output. These algorithms apply the normalization rules i.e. Rule 1, Rule 2 and Rule 3 which have been presented in section 4.4.1. We propose three algorithms to transform a G-DTD into second normal form, third normal form and fourth normal form, respectively.

Algorithm Restructure 1XNF to 2XNF G-DTD

In Figure 4.3 we present an algorithm for 2XNF converting a G-DTD into a second normal form. This algorithm applies rule 1 (presented in section 4.4.1) which is used to eliminate redundancy through restructure n-ary many-to-many or one-to-many/many-to-one path links with PFD. To do this, firstly all path links between complex element nodes in G-DTD are identified from the root until last node. Then, the data dependencies given by the user will be identified and grouped into key dependency, GFD, TFD and PFD accordingly based on the definitions given in section 4.4.1. If there exist both PFD and multiple path links with many-to-many or one-or-many in the same path, it indicates that the G-DTD is not in a second normal form. To transform the G-DTD into a second normal, the PFD between attribute and simple element will be removed by placing them under a new complex element node. The basic idea of rule 1 requires creating a new complex element node then locating this new node at a different path and level so that the multi nested *path link* is restructured. As a consequence, a new relationship type of *path link* and *part-of* link is created associated to the new complex element node.

Input: The G-DTD schema diagram D and given set θ of specified data dependencies.

Do:

- (1) 1.1 Let CE_a is a root element, where CE is complex element node in D
- 1.2 Let $Cnodes = \{CE_a, CE_b, \dots, CE_n\}$ is a set complex element node in D
- 1.3 Let $Snodes = \{SE_a, SE_b, \dots, SE_n\}$ is a set simple element node in D
- 1.4 Let $Attnodes = \{ATT_a, ATT_b, \dots, ATT_n\}$ is a set of attribute nodes in D
- 1.5 Let $Hierarchical_Link = \{Hlink_1, Hlink_2, \dots, Hlink_n\}$ is set of hierarchical link in D
- 1.6 Let (D, θ) is a 1XNF

(2) If (D, θ) is a 2XNF, then return (D, θ)

(3) For each $Hlink_i \in \{Path_Link\}$ in D {

(3.1) For level = 0 to $k-1$ where $k = \text{Maximum level in } D$

(3.2) Let $Hlink_i, (CE_a, \text{level}) \rightarrow (CE_b, \text{level}+1) \in Path_Link$ such that CE_a is a parent for CE_b

//Where $Hlink_i$ is describes by a relation name denoted as $(name, n, pc, cc)$ where $name$ is a name of relation, n is a type of relation, pc is a parent constraint and cc is a child constraint//

(3.3) If $(pc = [N_1..N_2])$ and $(cc = [N_1..N_2])$ // N_1 represent zero or one
 N_2 represent one or many//

Then $Nested_Hierarchical_link = True;$

(3.4) level = level + 1 ;}

(4) If $Nested_Hierarchical_Link$

4.1 let $\theta_1: \langle ATT_a, \text{level} \rangle, \langle ATT_b, \text{level}+1 \rangle \rightarrow \langle SE, \text{level}+1 \rangle$

$\theta_2: \langle ATT_b, \text{level}+1 \rangle \rightarrow \langle SE, \text{level}+1 \rangle$

where $\langle ATT_b, \text{level}+1 \rangle$ and $\langle SE, \text{level}+1 \rangle$ are children for $\langle CE_b, \text{level} \rangle$

4.2 $\{\theta_2 \subset \theta_1\}$ then θ_2 is PFD where $\{\theta_2, \theta_1\} \in \theta$

4.3 Restructure (D, θ_2) by applying rule 1

Output: $(D, \theta)'$ in 2XNF

Figure 4.3: Algorithm 2XNF

Finally the data dependencies are updated on the basis of the new structure G-DTD. This process is repeated until all PFD(s) are eliminated. This process is illustrated in Figure 4.4.

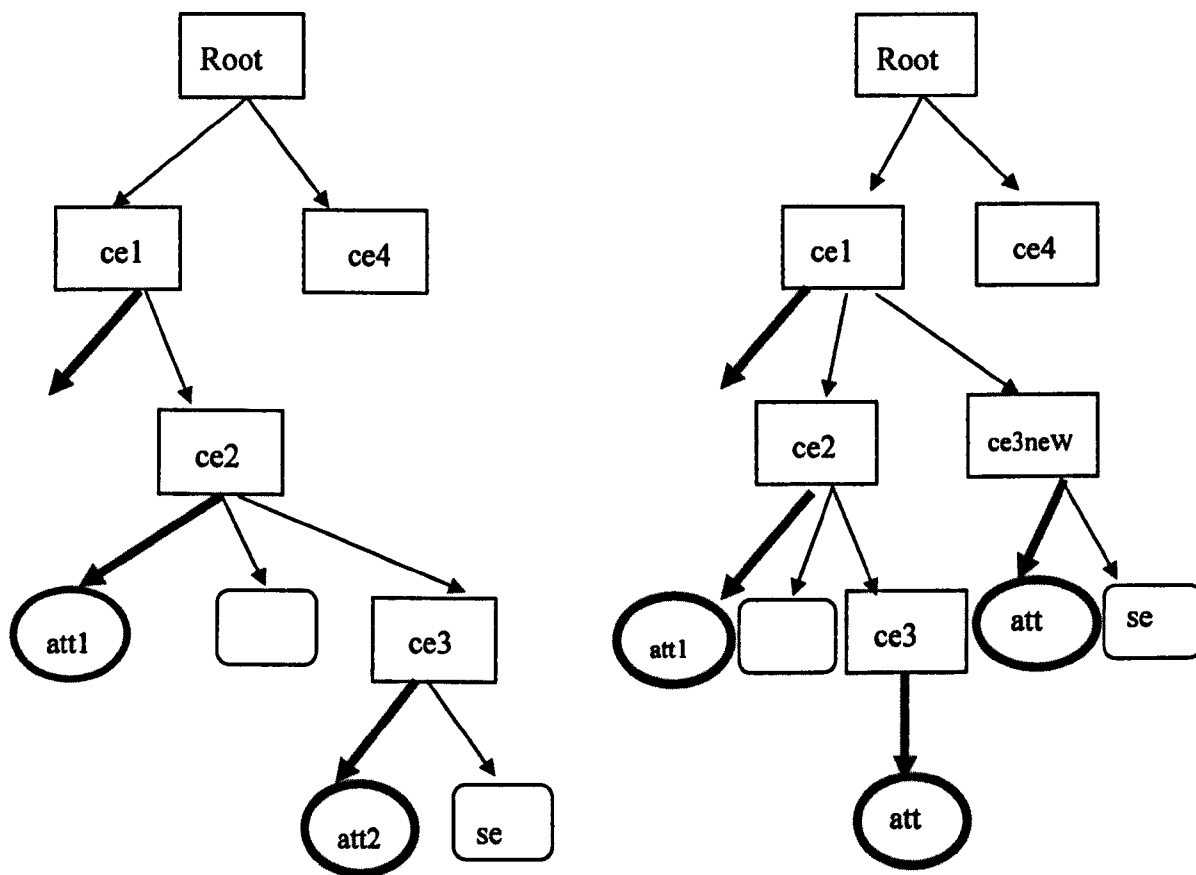


Figure 4.4: Create a New Complex Element Node

Algorithm restructure 2XNF to 3XNF G-DTD

Figure 4.5 shows an algorithm 3XNF to transform a G-DTD into a third normal form by applying rule 2 which have been presented in section 4.4.1. Like the 2XNF algorithm, all path links between complex elements nodes in G-DTD are identified from the root until to the last node. If there exist multiple nested one-to-many/many-to-many/many-to-one path links with TFD then the G-DTD will be restructured. To transform the G-DTD into a third normal form, the TFD between attribute and simple element will be removed. Generally, in rule 2 requires moving up a complex element node along with

its attribute and simple element node to be under a root node and creating a new relationship type of *path link* and *part-of link*. Finally data dependencies are updated on the basis of the new structure G-DTD. This process is repeated until all TFD(s) are eliminated. This process is illustrated in Figure 4.6.

Input: The G-DTD schema diagram D and given set θ of specified data dependencies.

Do:

- (1)
 - 1.1 Let CE_a is a root element, where ce is complex element node in D
 - 1.2 Let $Cnodes = \{CE_1, CE_2, \dots, CE_n\}$ is a set complex element node in D
 - 1.3 Let $Snodes = \{SE_1, SE_2, \dots, SE_n\}$ is a set simple element node in D
 - 1.4 Let $Attnodes = \{ATT_1, ATT_2, \dots, ATT_n\}$ is a set of attribute nodes in D
 - 1.5 Let $Path_Link = \{Hlink_1, Hlink_2, \dots, Hlink_n\}$ is set of path link in D
 - 1.6 Let (D, θ) is a 2XNF
- (2) If (D, θ) is a 3XNF, then return (D, θ)
- (3) For each $Hlink_i \in \{Path_Link\}$ in D {
 - (3.1) For level = 0 to $k-1$ where $k = \text{Maximum level in } D$
 - (3.2) Let $Hlink_i, (CE_a, \text{level} \rightarrow CE_b, \text{level}+1) \in Path_Link$ such that ce_a is a parent for ce_b

//Where $Hlink_i$ describe by relation name denoted as $(name, n, pc, cc)$ where $name$ is name of relation, n is type of relation, pc is a parent constraint and cc is a child constraint//
 - (3.3) If $(pc = [N_1..N_2])$ and $cc = [N_1..N_2])$ // N_1 represent zero or one
 N_2 represent one or many//

Then $Nested_Path_link = True;$
 - (3.4) level = level + 1;}
- (4) If $Nested_Path_Link$
 - 4.1 Let $\theta_1 : \langle ATT_a, \text{level}_{+1} \rangle \rightarrow \langle ATT_b, \text{level}_{+2} \rangle$

$\theta_2 : \langle ATT_b, \text{level}_{+2} \rangle \rightarrow \langle SE_b, \text{level}_{+2} \rangle$ where $\{\theta_2, \theta_1\} \in \theta$
 - 4.2 Then $\theta_3 : \langle ATT_a, \text{level}_{+1} \rangle \rightarrow \langle SE_b, \text{level}_{+2} \rangle$ is a TFD
 - 4.3 Restructure (D, θ_3) by applying rule 2

Output: (D, θ) in 3XNF

Figure 4.5: Algorithm 3XNF

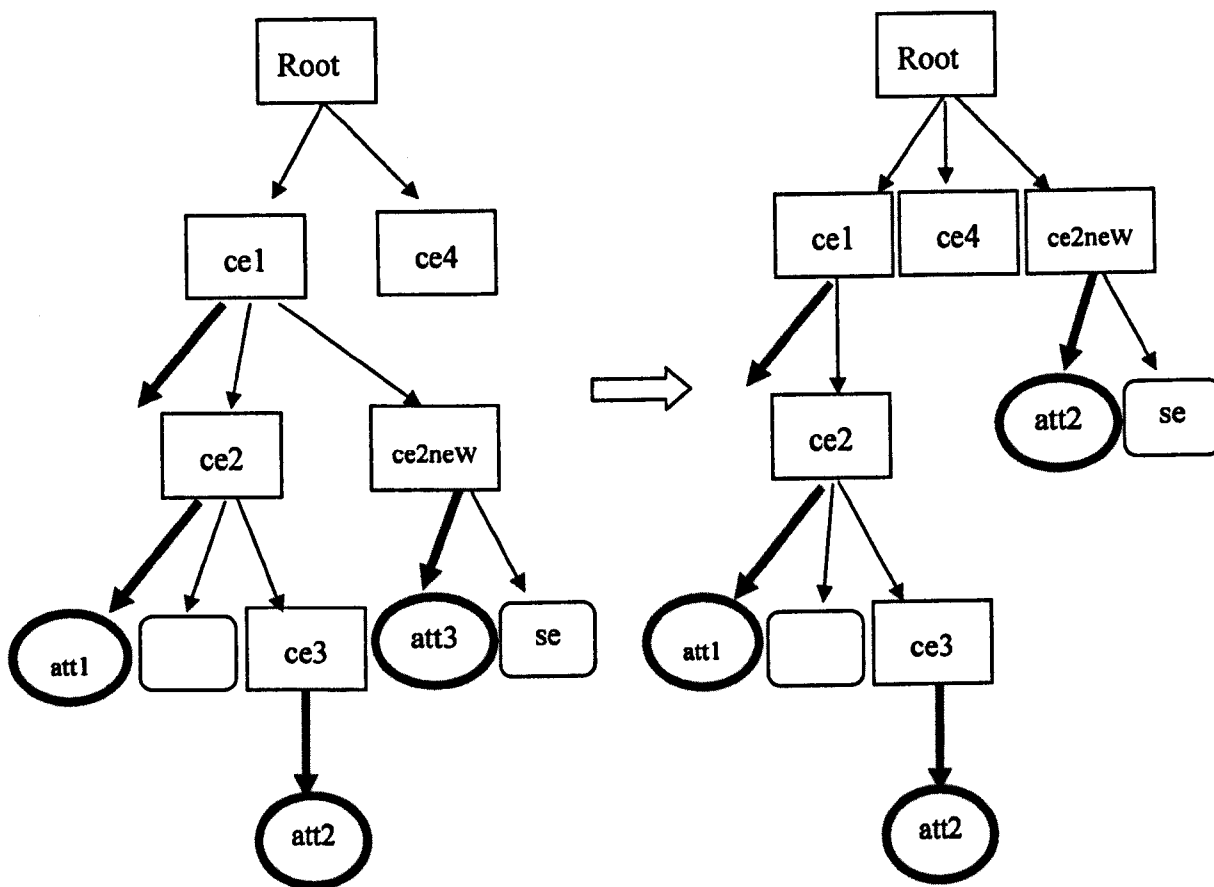


Figure 4.6: Moving Up a Complex Element Node

Algorithm restructure 3XNF to 4XNF G-DTD

Figure 4.7 presents an algorithm 4XNF to convert a G-DTD into a fourth normal form. The difference between 3XNF and 4XNF is the identification of GFD within the G-DTD's attributes/simple element node. This algorithm uses rule 3 to eliminate redundancy through global functional dependencies given in section 4.2.2. Three basic ideas used in the rule 3 are creating a new complex element node, creating a new relationship type of *path link* and *part_of link/has_A link* and replicating an attribute or simple element node. Unlike rule 1, rule 3 requires a new complex element node to be linked directly under the root node. This process is illustrated in Figure 4.8.

Input: The G-DTD schema diagram D and given set θ of specified data dependencies.

Do:

- (1)
 - 1.1 Let CE_a is a root element, where CE is complex element node in D
 - 1.2 Let $Cnodes = \{CE_a, CE_b, \dots, CE_n\}$ is a set complex element node in D
 - 1.3 Let $Snodes = \{SE_a, SE_b, \dots, SE_n\}$ is a set simple element node in D
 - 1.4 Let $Attrnodes = \{Att_a, Att_b, \dots, Att_n\}$ is a set of attribute nodes in D
 - 1.5 Let $Path_Link = \{Hlink_1, Hlink_2, \dots, Hlink_n\}$ is set of path link in D
 - 1.6 Let (D, θ) is a 3XNF
- (2) If (D, θ) is a 4XNF, then return (D, θ)
- (3) For each $Hlink_i \in \{Path_Link\}$ in D {
 - (3.1) For level = 0 to k-1 where k = Maximum level in D
 - (3.2) Let $Hlink_i, (CE_a, level) \rightarrow (CE_b, level+1) \in Path_Link$ such that CE_a is a parent for CE_b
//Where Hlink_i describe by relation name denoted as (name, n, pc, cc) where name is name of relation, n is type of relation, pc is a parent constraint and cc is a child constraint//
 - (3.3) If (pc= $[N_1..N_2]$ and cc = $[N_1..N_2]$) // N_1 represent zero or one
 N_2 represent one or many//
Then $Nested_Path_link = True;$
 - (3.4) level = level + 1; }
- (4) If $Nested_Path_Link$
 - 4.1 let $\theta_1: \langle ATT, level_k \rangle \rightarrow \langle SE, level_k \rangle$ is GFD where $\theta_1 \in \theta$
 - 4.3 Restructure (D, θ_1) by applying rule 3

Output: $(D, \theta)'$ in 4XNF

Figure 4.7: Algorithm 4XNF

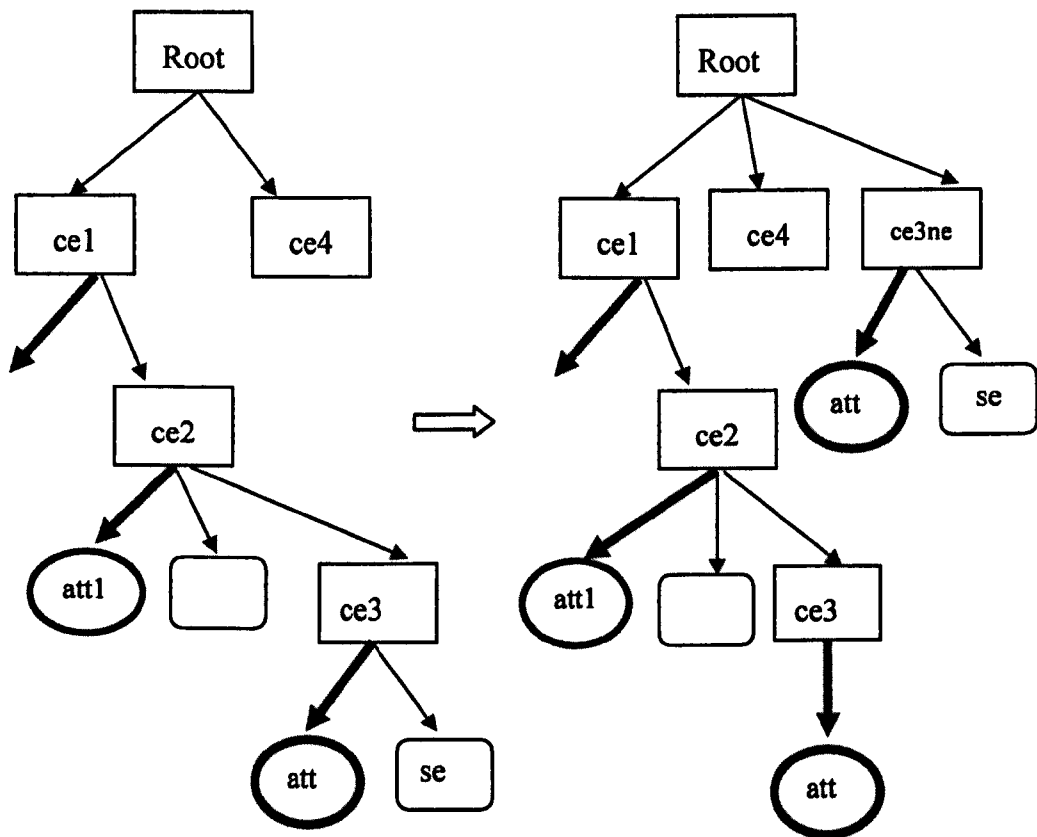


Figure 4.8: Moving a Complex Element Node under a Root Node

The algorithm presented in Figure 4.7 is an extension from the normalization algorithm proposed Arenas and Libkin (2004). For instance, rule 3 (see section 4.4.1) that included the algorithm has the similarities with the procedure *create element* and *moving attribute* in the XNF decomposition algorithm (Arenas and Libkin, 2004). This rule is used to eliminate redundancies caused by *semantic constraint* GFD only. However, in our algorithm, we also check for the *structural constraint* in the G-DTD model. Both of the algorithms presented in Figure 4.3 and Figure 4.5 are used another rules i.e. rule 1 and rule 2 (see section 4.4.1) which are used to eliminate redundancy caused by semantic constraints such as TFD and PFD and also the structural constraint relationship dependency. In our algorithm, we make a distinction between type of node to be created and where the node to be located and check the structure of G-DTD by using the

relationship dependency i.e. path link types. We simplify the process of this normalization algorithm as follows:

1. Semantic constraints are presented as FD and grouped into GFD, TFD and PFD according to our definition given in section 4.4
2. FD are processed according to number of level/ type of path links in G-DTD
3. FD are processed according to the hierarchy depth by using top-down approach
4. Structural constraints are presented as RD and identified as one-to-many/many-to-one or many-to-many path relationship.

4.5 Case study

In this section, again, we use a G-DTD (in Figure 4.2) and its instance of an XML university database (in Figure 4.1) as a case study. As shown in Figure 4.1, this XML document is prone to update anomaly. For instance, if the staff number (*tno*) of lecturer named “Bottaci” is changed to ‘125’, then two distinct places need to be updated. If any of them is not updated, then the information in the document becomes inconsistent. This anomaly was called an update anomaly by Codd(1972) and it arises because the instance is storing redundant information. To avoid this problem, the G-DTD needs to be in a normal form. To demonstrate the process of normalizing G-DTD from 1XNF to 4XNF, we present here step by step transformation of G-DTD according to 1XNF, 2XNF, 3XNF and 4XNF design with respect to the following constraints or data dependencies.

The following constraints are given based on the database designer's requirements:

Constraint 1: *Each set course, student and lecturer has a unique identifier.*

Constraint 2: *In the department, a student number (sno) determines fname and lname*

Constraint 3: *In the department, course number (cno) and student number (sno) determine lecturer number (tno)*

Constraint 4: *In the department, course number (cno) determines lecturer number(tno)*

Constraint 5: *In the department, course number (cno) and lecturer number (tno) determine lecturer name (tname).*

Constraint 6: *In the department, lecturer number (tno) determines lecturer name (tname)*

Using the definition of data dependency G-DTD given in section 4.2, the above constraints are categorized and represented as follows respectively.

- *Key dependency*

Constraint 1 is considered as key dependency since key attributes can determine their complex elements:

$\langle cno, 2 \rangle \rightarrow \langle course, 1 \rangle,$

$\langle sno, 3 \rangle \rightarrow \langle student, 2 \rangle,$

$\langle tno, 4 \rangle \rightarrow \langle lecturer, 3 \rangle$

- *Global functional dependency*

Constraint 2 and constraint 6 are classified as GFD because they involve attributes and simple elements at leaves.

$\langle sno, 3 \rangle \rightarrow \{ \langle fname, 3 \rangle, \langle lname, 3 \rangle \}$

$\langle tno, 4 \rangle \rightarrow \langle tname, 4 \rangle$

- *Transitive functional dependency*

Constraint 4: $\langle cno, 2 \rangle \rightarrow \langle tno, 4 \rangle$

Constraint 6: $\langle tno, 4 \rangle \rightarrow \langle tname, 4 \rangle$

Simple element $\langle tname, 4 \rangle$ is transitively dependent on attribute node key $\langle cno, 2 \rangle$ which derives the following TFD.

Constraint 7: $\langle cno, 2 \rangle \rightarrow \langle tname, 4 \rangle$

- *Partial functional dependency*

Constraint 5: $\langle cno, 2 \rangle, \langle tno, 4 \rangle \rightarrow \langle tname, 4 \rangle$

Constraint 6: $\langle tno, 4 \rangle \rightarrow \langle tname, 4 \rangle$

Constraint 6 is called PFD because the attribute node key, course number $\langle tno, 4 \rangle$ alone can be used to determine lecturer name $\langle tname, 4 \rangle$.

- *Relationship dependency*

Based on G-DTD given, there exist a binary many-to-many path link between *course* and *student* which represented as $\langle course, 1 \rangle \rightarrow^R \langle student, 2 \rangle$ and a ternary one-to-many path link between *course*, *student* and *lecturer* node which is represented as $(\langle course, 1 \rangle, \langle student, 2 \rangle \rightarrow^R \langle lecturer, 3 \rangle)$ where $R = (CST, 3, 1, *)$

4.5.1 1XNF G-DTD

The G-DTD illustrated in Figure 4.2 shows that it is in first normal form (1XNF) because each set of complex element nodes *course*, *student* and *lecturer* has *cno*, *sno* and *tno* as a unique key identifier respectively, while all simple element nodes and attribute nodes have one unique label. The *department* node is a root element since it is located at level 0. Moreover, the XML documents shown in Figure 4.1 satisfy and conform to 1XNF G-DTD in Figure 4.2.

4.5.2 2XNF G-DTD

G-DTD of Figure 4.2 is not in 2XNF because

- There exists a nested ternary path link type under many-to-one/many-to-many path link type, $\langle course, 1 \rangle, \langle student, 2 \rangle \rightarrow^{(cst,3,1,*)} \langle lecturer, 3 \rangle$. This dependency relationship involves complex element *course*, *student* and *lecturer* nodes. As a consequence information about *lecturer* is stored redundantly in the XML document and can cause update anomaly. If the information about the *lecturer* is changed, then it must be updated in all subtree *students* who are taking the same *course*.

- There exist constraints 5 and 6 which may give rise to PFD $\langle tno, 4 \rangle \rightarrow \langle tname, 4 \rangle$. The attribute node $\langle tno, 4 \rangle$ and simple element node $\langle tname, 4 \rangle$ are children of complex element *lecturer*.

To be in 2XNF, the structure of G-DTD needs to be restructured using algorithm 2XNF (as presented in section 4.4.2.). Based on this algorithm, the new complex element *lecturernew* along with its children is created and located at the same level as complex element *student* node. A one-to-many path link between *course* and *lecturernew* node is created. All children from the original complex element *lecturer* are deleted except key attribute node *tno*. In this way, the original semantic relationship is preserved in G-DTD. As a consequence, PFD under a ternary one-to-many path link is eliminated in complex element *lecturer*, but still preserved in complex element *lecturernew*, which is denoted as $\langle tno, 3 \rangle \rightarrow \langle tname, 3 \rangle$. At the same time, all constraints in the set of data dependencies that involve the constraint $\langle tno, 3 \rangle \rightarrow \langle tname, 3 \rangle$ are updated accordingly. Figure 4.9 and Figure 4.10 illustrate the structure of the G-DTD in 2XNF and the XML document conforms to 2XNF. Finally, the new version of the set of data dependencies will look as follows:

- *Key dependency*

Constraint 1:

$\langle cno, 2 \rangle \rightarrow \langle course, 1 \rangle$, $\langle sno, 3 \rangle \rightarrow \langle student, 2 \rangle$,
 $\langle tno, 4 \rangle \rightarrow \langle lecturer, 3 \rangle$, $\langle tno, 3 \rangle \rightarrow \langle lecturernew, 2 \rangle$

- *Global functional dependency*

Constraint 2: $\langle sno, 3 \rangle \rightarrow \{ \langle fname, 3 \rangle, \langle lname, 3 \rangle \}$

Constraint 6: $\langle tno, 3 \rangle \rightarrow \langle tname, 3 \rangle$

- *Transitive functional dependency*

Constraint 4: $\langle cno, 2 \rangle \rightarrow \langle tno, 3 \rangle$

Constraint 6: $\langle tno, 3 \rangle \rightarrow \langle tname, 3 \rangle$

Constraint 7: $\langle cno, 2 \rangle \rightarrow \langle tname, 3 \rangle$ is TFD

- *Partial functional dependency*

Constraint 5: $\langle cno, 2 \rangle, \langle tno, 3 \rangle \rightarrow \langle tname, 3 \rangle$

Constraint 6: $\langle no, 3 \rangle \rightarrow \langle tname, 3 \rangle$ is a PFD

- *Relationship dependency*

As shown in Figure 4.9, there exists a binary one-to-many relationship between complex element node *course* and complex element node *lecturernew*, represented as $\langle course, 1 \rangle \rightarrow^R \langle lecturernew, 2 \rangle$, binary many-to-many path link between complex element *course* and complex element *student*, $\langle course, 1 \rangle \rightarrow^R \langle student, 2 \rangle$ and a ternary one-to-many path link between *course*, *student* and *lecturer* node which is represented as $(\langle course, 1 \rangle, \langle student, 2 \rangle \rightarrow^R \langle lecturer, 3 \rangle)$.

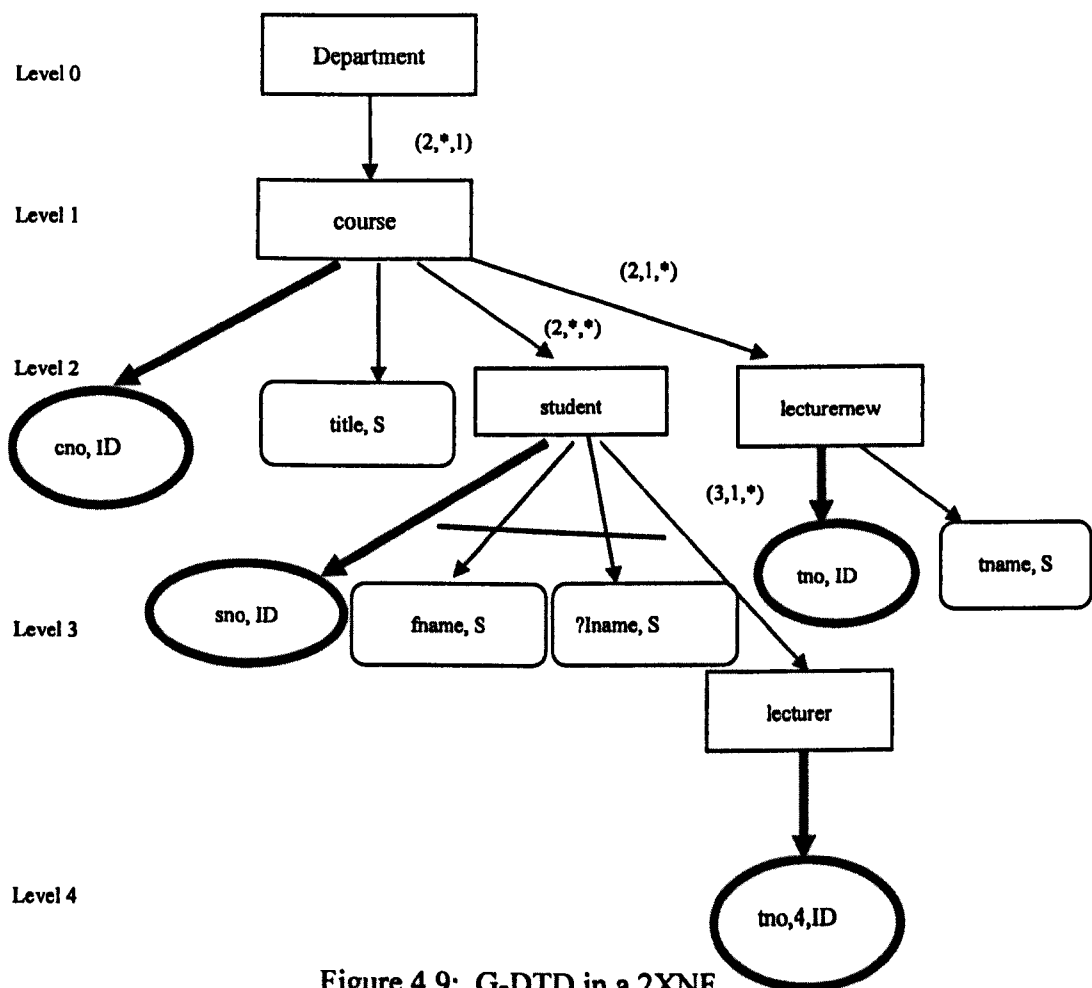


Figure 4.9: G-DTD in a 2XNF

```

<!DOCTYPE Department [
  <course>
    <course cno = "csc101">
      <title > XML database </title>
      <student >
        <student sno = "112344">
          <fname> David</fname>
          <lname> Grey </lname>
        </student >
        <lecturer>
          <lecturer tno = "123">
          </lecturer>
        < student >
          <student sno = "112345">
            <fname>Helen</fname>
          </student >
          <lecturer>
            <lecturer tno = "123">
            </lecturer>
          <lecturernew>
            <lecturernew tno = "123">
              <tname> Bing </tname>
            </lecturernew>
          </course>
        <course>
          <course cno = "csc201">
            <title > Database technique </title>
            < student >
              <student sno = "112344">
                <fname> David</fname>
                <lname> Grey </lname>
              </student >
              <lecturer>
                <lecturer tno = "123">
                </lecturer>
            < student >
              <student sno = "112346">
                <fname> Sally</fname>
                <lname> Teoh </lname>
              </student >
              <lecturer>
                <lecturer tno = "123">
                </lecturer>
            <lecturernew>
              <lecturernew tno = "123">
                <tname> Bing </tname>
              </lecturernew>
            </course>
          <course>
            <course cno = "csc102">
              <title > Z formal methods </title>
              <student >
                <student sno = "112344">
                  <fname> David</fname>
                  <lname> Grey </lname>
                </student >
                <lecturer>
                  <lecturer tno = "124">
                  </lecturer>
              < student >
                <student sno = "112345">
                  <fname>Helen</fname>
                </student >
                <lecturer>
                  <lecturer tno = "124">
                  </lecturer>
                <lecturernew>
                  <lecturernew tno = "124">
                    <tname> Bottaci</tname>
                  </lecturernew>
                </course>
              </Department>]

```

Figure 4.10: An XML Document That Conforms to 2XNF

4.5.3 3XNF G-DTD

Given the new set of data dependencies, the G-DTD of Figure 4.9 is not in 3XNF because

- There exist one-to-many path links between element *lecturernew* and *course* node under many-to-one path link.
- There exists TFP represented by constraint 7: $\langle cno, 2 \rangle \rightarrow \langle tname, 3 \rangle$ caused by *constraint 4* and *constraint 6* which involve complex element node *course* and complex element node *lecturernew*

To be in 3XNF G-DTD, the path link between complex element node *course* and complex element node *lecturernew* needs to be restructured and TFD is eliminated within *course* node and complex element *lecturernew*. The complex element node *lecturernew* along with its children is moved up and linked with *department* node. Because *department* is a root, a binary many-to-one path link is created. Figure 4.11 and Figure 4.12 present a new structure of a 3XNF G-DTD after eliminating the above constraints and the instance XML document conforms to 3XNF.

Having this structure, TFD under a ternary one-to-many path link is eliminated between complex element *course* and complex element *lecturernew*, but the semantic constraint is still preserved between them, which is indicated as $\langle cno, 2 \rangle \rightarrow \langle tname, 2 \rangle$. Finally, all constraints in the set of data dependencies that involve the constraint $\langle cno, 2 \rangle \rightarrow \langle tname, 2 \rangle$ need to be updated. As a consequence, the new version of set data dependencies in 3XNF will be as follows:

- *Key dependency*

Constraint1:

$\langle cno, 2 \rangle \rightarrow \langle course, 1 \rangle$, $\langle sno, 3 \rangle \rightarrow \langle student, 2 \rangle$, and $\langle tno, 4 \rangle \rightarrow \langle lecturer, 3 \rangle$,
 $\langle tno, 2 \rangle \rightarrow \langle lecturernew, 1 \rangle$

- *Global functional dependency(GFD)*

Constraint 2: $\langle sno, 3 \rangle \rightarrow \{ \langle fname, 3 \rangle, \langle lname, 3 \rangle \}$

Constraint 6: $\langle tno, 2 \rangle \rightarrow \langle tname, 2 \rangle$

- *Transitive functional dependency(TFP)*

Constraint 4: $\langle cno, 2 \rangle \rightarrow \langle tno, 2 \rangle$

Constraint 6: $\langle tno, 2 \rangle \rightarrow \langle tname, 2 \rangle$

Constraint 7: $\langle cno, 2 \rangle \rightarrow \langle tname, 2 \rangle$ is TFD

- *Partial functional dependency(PFD)*

Constraint 5: $\langle cno, 2 \rangle, \langle tno, 2 \rangle \rightarrow \langle tname, 2 \rangle$

Constraint 6: $\langle tno, 2 \rangle \rightarrow \langle tname, 2 \rangle$ is a PFD

- *Relationship dependency*

In 3XNF G-DTD, there exists a binary many-to-one path link between complex element node *department* and complex element node *lecturernew student*, represented as $\langle department, 0 \rangle \rightarrow^R \langle lecturernew, 1 \rangle$, a binary many-to-many path link between complex element *course* and complex element *student*, $\langle course, 1 \rangle \rightarrow^R \langle student, 2 \rangle$ and a ternary one-to-many path link between course, student and lecturer node, which is represented as $(\langle course, 1 \rangle, \langle student, 2 \rangle \rightarrow^R \langle lecturer, 3 \rangle)$.

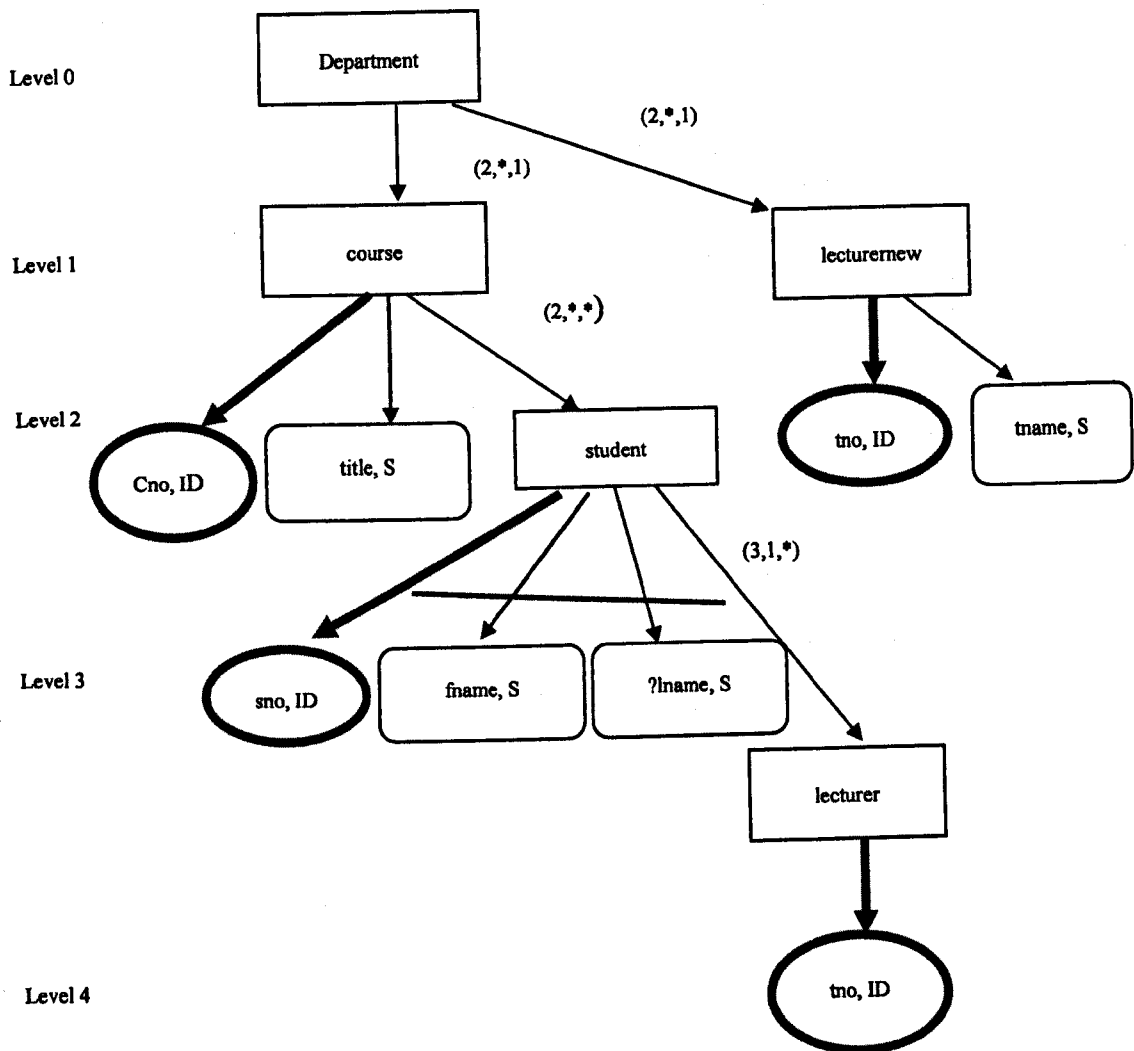


Figure 4.11: G-DTD in a 3XNF

```

<!DOCTYPE Department [
  <course>
    <course cno = "csc101">
      < title > XML database </title>
      <student >
        <student sno = "112344">
          <fname> David</fname>
          <lname> Grey </lname>
        </ student >
        <lecturer>
          <lecturer tno = "123">
            </lecturer>
        <student >
          <student sno = "112345">
            <fname>Helen</fname>
          </ student >
          <lecturer>
            <lecturer tno = "123">
              </lecturer>
            </course>
          <course>
            <course cno = "csc201">
              < title > Database technique </title>
              < student >
                <student sno = "112344">
                  <fname> David</fname>
                  <lname> Grey </lname>
                </ student >
                <lecturer>
                  <lecturer tno = "123">
                    </lecturer>
                < student >
                  <student sno = "112346">
                    <fname> Sally</fname>
                  </ student >
                </course>
              <lecturernew>
                <lecturernew tno = "123">
                  <tname> Bing </tname>
                </lecturernew>
                <lecturernew>
                  <lecturernew tno = "124">
                    <tname> Bottaci</tname>
                  </lecturernew>
                </lecturernew>
              </Department>]
    </name> Teoh </lname>
  </ student >
  <lecturer>
    <lecturer tno = "123">
      </lecturer>
    </course>
  <course>
    <course cno = "csc102">
      < title > Z formal methods </title>
      < student >
        <student sno = "112344">
          <fname> David</fname>
          <lname> Grey </lname>
        </ student >
        <lecturer>
          <lecturer tno = "124">
            </lecturer>
          < student >
            <student sno = "112345">
              <fname>Helen</fname>
            </ student >
            <lecturer>
              <lecturer tno = "124">
                </lecturer>
            </course>
          <lecturernew>
            <lecturernew tno = "123">
              <tname> Bing </tname>
            </lecturernew>
            <lecturernew>
              <lecturernew tno = "124">
                <tname> Bottaci</tname>
              </lecturernew>
            </lecturernew>
          </Department>]

```

Figure 4.12: An XML Documents Conform to 3XNF

4.5.4 4XNF G-DTD

Given updated data dependencies, 3XNF G-DTD of Figure 4.11 is not in 4XNF because

- (i) There exists GFD indicated in constraint 2: $sno \rightarrow \{fname, lname\}$ for set of complex elements *student* node and *course* node under a many-to-many path link. This GFD may cause update anomalies if the information about the *fname* and *lname* is changed.

To this GFD, a new of complex element node *studentnew* is created. The attribute node *sno* and simple element nodes *fname* and *lname* are replicated and they become children of the complex element *studentnew*. A *part_of* link is created between them accordingly. Both simple element node *fname* and *lname* are deleted from *student* node but key attribute node *sno* remains as a child for complex element *student* node.

Figure 4.13 illustrates the new structure of the 4XNF G-DTD. More importantly, as shown, all data dependencies are still preserved in the 4XNF G-DTD. Figure 4.14 is an XML document conforming to 4XNF G-DTD with no redundancy. Eventually, the G-DTD has the following dependencies.

- *Key dependency*

Constraint 1:

$\langle cno, 2 \rangle \rightarrow \langle course, 1 \rangle$, $\langle sno, 3 \rangle \rightarrow \langle student, 2 \rangle$, $\langle sno, 2 \rangle \rightarrow \langle studentnew, 1 \rangle$,
 $\langle tno, 4 \rangle \rightarrow \langle lecturer, 3 \rangle$,
 $\langle tno, 2 \rangle \rightarrow \langle lecturernew1 \rangle$

- *Global functional dependency*

Constraint 2: $\langle sno, 2 \rangle \rightarrow \{ \langle fname, 2 \rangle, \langle lname, 2 \rangle \}$

Constraint 6: $\langle tno, 2 \rangle \rightarrow \langle tname, 2 \rangle$

- *Transitive functional dependency*

Constraint 4: $\langle cno, 2 \rangle \rightarrow \langle tno, 2 \rangle$

Constraint 6: $\langle tno, 2 \rangle \rightarrow \langle tname, 2 \rangle$

Constraint 7: $\langle cno, 2 \rangle \rightarrow \langle tname, 2 \rangle$ is TFD

- *Partial functional dependency*

Constraint 5: $\langle cno, 2 \rangle, \langle tno, 2 \rangle \rightarrow \langle tname, 2 \rangle$

Constraint 6: $\langle tno, 2 \rangle \rightarrow \langle tname, 2 \rangle$ is a PFD

- *Relationship dependency*

In Figure 4.13, finally 4XNF G-DTD consists of a binary many-to-one path link between *department* and *course*, many-to-one path link between *department* and *lecturernew*, and binary many-to-one path link between *department* and *studentnew* which represented as $\langle department, 0 \rangle \rightarrow^R \langle course, 1 \rangle$, $\langle department, 0 \rangle \rightarrow^R \langle lecturernew, 1 \rangle$, and $\langle department, 0 \rangle \rightarrow^R \langle studentnew, 1 \rangle$. Meanwhile the semantic relationships between complex element *course* and complex element *student*, and *lecturer* remain the same.

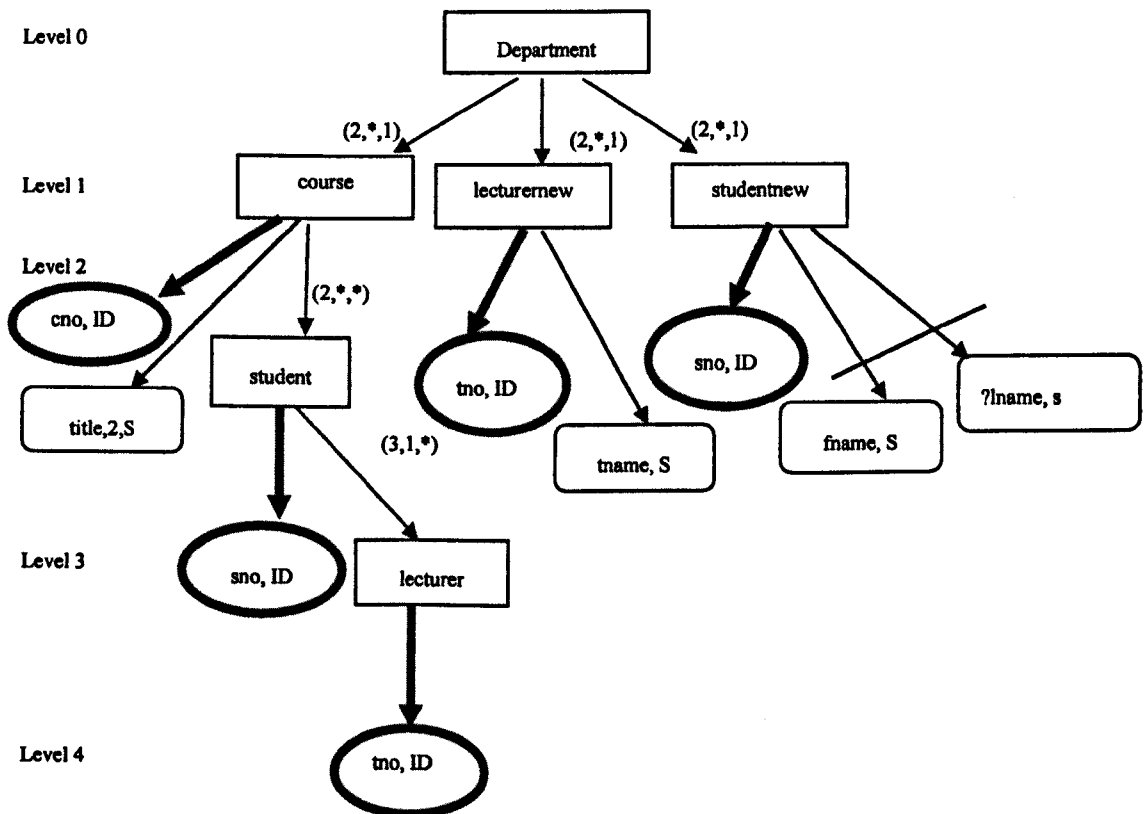


Figure 4.13: G-DTD in a 4XNF

Finally, based on the transformation rules given in Chapter 3 (Section 3.7), we map the 4XNF G-DTD to the new DTD. The final DTD can be shown as follows.

```
<!DOCTYPE Department [  
  <!ELEMENT Department(course*, lecturernew*, studentnew*)>  
  <!ELEMENT course(title, student*)>  
    <!ATTLIST course cno ID #REQUIRED>  
  <!ELEMENT title #PCDATA>  
  <!ELEMENT student (lecturer*)>  
    <!ATTLIST student sno ID #REQUIRED>  
  <!ELEMENT lecturer (EMPTY)>  
  <!ATTLIST lecturer tno ID #REQUIRED>  
  <!ELEMENT lecturernew* (tname)>  
    <!ATTLIST lecturernew tno ID #REQUIRED>  
  <!ELEMENT studentnew* (fname|?lname)>  
    <!ATTLIST studentnew sno ID #REQUIRED>  
  <!ELEMENT fname (# PCDATA)>  
  <!ELEMENT lname # PCDATA>  
>
```

Form the above DTD the following XML document with no redundancy is generated:

```

<!DOCTYPE Department [
  <course>
    <course cno = "csc101">
      < title > XML database </title>
      < student >
        <student sno = "112344">
          <lecturer>
            <lecturer tno = "123">
          </lecturer>
        </ student >
      < student >
        <student sno = "112345">
      </student >
    </course>
  <course>
    <course cno = "csc201">
      < title > Database technique </title>
      < student >
        <student sno = "112344">
          <lecturer>
            <lecturer tno = "124">
          </lecturer>
        </ student >
      < student >
        <student sno = "112346">
      </student >
    </course>
  <course>
    <course cno = "csc102">
      < title > Z formal methods </title>
      < student >
        <student sno = "112344">
          <lecturer>
            <lecturer tno = "124">
          </lecturer>
        </ student >
      < student >
        <student sno = "112345">
          <lecturernew>
            <lecturernew tno = "123">
              <name> Bing </name>
            </lecturernew>
          </lecturernew>
        <studentnew>
          <studentnew sno = "112344">
            <fname> David</fname>
            <lname> Grey </lname>
          </studentnew>
        <studentnew>
          <studentnew sno = "112345">
            <fname>Helen</fname>
          </studentnew>
        <studentnew>
          <studentnew sno = "112346">
            <fname> Sally</fname>
            <lname> Teoh </lname>
          </studentnew>
        </studentnew>
      </Department>]

```

Figure 4.14: An XML Document That Conforms to 4XNF

4.6 Comparisons of Proposed Approach with Existing Approaches

To evaluate this work, a comparison of our approach with existing approaches is discussed. The comparison is based on a number of criteria, specifically:

(1) Expression of DTD structure

In Arenas and Libkin's work, they used mathematical notations to represent the DTD structure, which is hard to understand for a normal designer. For instance, we present again here their formal definition of DTD as follows:

A DTD is defined to be $D = (E, A, P, R, r)$, where (Arenas and Libkin, 2004):

1. $E \subseteq El$ is a finite set of element types.
2. $A \subseteq Att$ is a finite set of attributes.
3. P is a mapping from E to element type definition defined in a regular expression $\alpha = \varepsilon | \tau' | \alpha | \alpha \cup \alpha | \alpha^*$ where ε is the empty sequence, $\tau' \in E$, and "U", "(", and "*" denote union, concatenation, and Kleene star, respectively.
4. R is a mapping from E to the power set of value of R : $\mathbb{P}(A)$
5. $r \in E$ and is called the root element type

The symbols ε and S represent element type declaration EMPTY and #PCDATA, respectively.

For example, reconsider the DTD in Figure 2.10 which is presented by Arenas and Libkin (2004) as follows:

$E = \{\text{Department, Course, Student, Lecturer}\}$,

$A = \{\text{cno, sno, tno}\}$

$r = \text{Department}$

Furthermore, P and R are defined as follows:

$P(\text{Department}) = \{\text{course}^*\}$, $P(\text{course}) = \{\text{title}, \text{Student}^*\}$, $P(\text{student}) = \{\text{fname}, \text{lname}, \text{lecturer}^*\}$, $P(\text{title}) = S$,

$P(\text{fname}) = S$, $P(\text{lname}) = S$,

$R(\text{Department}) = \emptyset$ $R(\text{course}) = cno$, $R(\text{student}) = sno$, $R(\text{lecturer}) = tno$.

The above definition clearly shows that textual grammar representation makes it difficult to be analysed and understood. In practise, it often causes difficulties when designing even a simple DTD. More importantly, the semantic constraint and relationship between the elements in the XML document cannot be represented precisely and clearly. For instance the relation between *course* and *student* is not defined explicitly. The semantic relation between the elements presents only one-to-many relationships by notation *, while other relationships such as many-to-many or many-to-one relationships are not defined. The semantic relation presents only the relationship between the parent and child while the relation between the child and parent is not defined. The level of each element is not defined explicitly to show the hierarchy of the elements

However, in our work, a graphical interpretation of G-DTD is used to visually represent a DTD structure which is used to describe an XML document. In this way, we believe the user can have better understanding of the DTD structure. Indeed, the Mok and Embley (2006) make the argument that: “*The graphical conceptual modelling languages offer one of the best human-oriented ways of describing an application*”

Representation of the G-DTD is slightly different from the Arenas and Libkin’s DTD. Firstly, we distinguish explicitly the difference between complex elements, simple elements and attributes. We emphasise that a simple element is an element with no child elements while an attribute is a key or candidate key of a complex element. The reason for this is to avoid using *S* as a text representation, which will make it easier to visualise during normalization process. Secondly, we present the G-DTD structure as a tree structure of elements using level notation which is similar to XML document structure, to provide an accurate picture of the XML document. The advantage of G-

DTD over DTD are: it allows users to define explicitly the structure of attribute nodes, simple element nodes and complex element nodes in a hierarchical way and also allows the user to determine the relationship dependency between the nodes. For G-DTD's notation itself, we adopt some notations from Chen (1976) and Dobbie et al.(2000) but we add a few semantics such as sequence, disjunction, path link, part_of link, has_A link between the nodes to reveal more semantics between the element nodes.

(2) XML Normal Forms

We have presented definitions of XML normal forms proposed by Arenas and Libkin, (2004), Vincent et al. (2004), Wang and Topor (2005), Kolahi (2007) and Yu and Jagadish (2008). Most of them generalised XNF from BCNF except Kolahi proposed an XML third normal form. The proposed work from Arenas and Libkin (2004) is most fundamental and achieves the best possible design (Kolahi, 2007) while the others are extensions and improvements of Arenas and Libkin's work. However, as discussed in Chapter 2 (Section 2.3.9.), Arenas and Libkin's XML normal forms are limited in their ability to capture certain semantic constraints. Most proposals represent semantic constraints using XML functional dependencies and key dependencies to define XML normal form. As shown in Chapter 2 (Section 2.3.7), the definition of XFD and normal form is very difficult to understand by users without a theoretical background. Thus, these limitations will directly affecting the application of XML normal form in real practice by the end user. We believe that defining the current definitions of XML normal form with simple presentation will help end users to apply and design XML documents in a simple way.

Arenas and Libkin (2004) defined a normal form for XML documents (XNF) is based on XML functional dependency (XFD) which is used to avoid update anomalies and redundancies. Instead of formal definition of DTD, they define also an XML document as a tree called 'XML tree' which consist of nodes, edges to represent a relation between parent and child node, and a root node. In an XML tree, they use paths to represent element nodes, starting form the root node until the last child node. The same path

definition also applies to the path in DTD denoted as $path(D)$. The child node is normally an attribute node or a string (denoted as symbol @ and S respectively). Every path in XML tree will contain at least one element node type. For example,

$Department.course.student.lecturer@tno$ and

$Department.course.student.lecturer.tname.S$ are all paths in XML document.

In Arenas and Libkin (2004), XFD is used to represent the semantic constraint which could cause data redundancy. They defined XFD as a path based on the idea from relational schema (Codd, 1974) and nested relational schema (Ling, 1985). For example, the constraint that two lecturer elements with same lecturer number (tno) must have the same $name$ is expressed as follows:

$XFD: Department.course.student.lecturer.@tno,$

$\rightarrow Department.course.student.lecturer.tname.S$ -----(4.1)

Where $Department.course.student.lecturer.@tno$ is a Left Hand Side (LHS) path and $Department.course.student.lecturer.tname.S$ is a Right Hand Side (RHS) path. Using this XFD definition, the following XML normal form is presented.

Definition XNF: Given a DTD and a set of Σ of XFDs over D , (D, Σ) is in XML normal form (XNF) if and only if for every nontrivial XFD $p \rightarrow q.@I$ or $p \rightarrow q.S$ or , the XFD $p \rightarrow q$ is also in $(D, \Sigma)^+$. Where p and q is a set of $path(D)$.(Arenas and Libkin, 2004)

Generally, the above definition means that a DTD is in a normal form (XNF) if every functional dependency defined over DTD is in XNF. The XFD is XNF if every LHS path can determine a unique value of a RHS path. In other words, XNF does not allow any redundancy in data values occurring in the leaves of the XML tree. Intuitively, this XNF ensures the value of $q.@I$ or $q.S$ will not be repeated in two different locations of the XML tree. For instance, XFD in equation 4.1 is not in XNF since the value for the element $lecturer$ name is not unique for the attribute given, as $lecturer$ name Bing

appears twice in the XML document as shown in Figure 4.1. Because of this XFD in equation 4.1, DTD is not in XNF.

Based on Arenas and Libkin's definition, recently, Kolahi (2007) has proposed a third normal form for XML by extending 3NF to XML. Kolahi adopted the notion of XML tree and DTD from Arenas and Libkin (2004) but extended the notion of a prime attribute from relational database to case of paths to XML tree. We present here the definition of prime attribute path in order to present X3NF:

Definition X3NF: XML specification (D, Σ) is in X3NF if and only if for every nontrivial XFD $p \rightarrow q.@l \in (D, \Sigma)^+$, we have that $p \rightarrow q \in (D, \Sigma)^+$ or $q.@l$ is a prime path. (Kolahi, 2007)

As a relational counterpart, a prime path is a path that uniquely determine path element of a 'tree tuple' from the root. Like the 3NF, X3NF tries to achieve a schema that can preserve the functional dependency and at the same time reduce data redundancy in XML documents. However, to date there no normalization algorithm has been developed based on X3NF definition.

Other than that, based on Arenas and Libkin's work, Lv et al. (2004) also proposed first, second and third normal forms which are considered partial functional dependency and transitive functional dependency respectively. They defined second and third normal forms as follows:

Definition 2XNF: XML specification (D, Σ) is in X2NF if and only if for every XFD $p_1.p_2.@l_1 \rightarrow p_1.p_2.p_3.@l \in (D, \Sigma)$, if there is another XFD' $p_1.@l_2 \rightarrow p_1.p_2.p_3.@l \in (D, \Sigma)$, there is no partial dependency $p_1.@l \rightarrow p_1.p_2.p_3.@l$ and $p_1 \neq r$. Where p_i is a set of $path(D)$. (Lv et al., 2004)

Definition 3XNF: XML specification (D, Σ) is in X3NF if and only if for every XFD $p_1.@l \rightarrow p_1.p_2.@l \in (D, \Sigma)$, if there is another XFD' $p_1.p_2.@l \rightarrow p_1.p_2.p_3.@l \in (D, \Sigma)$, there is no transitive dependency $p_1.@l \rightarrow p_1.p_2.p_3.@l$. Where p_i is a set of $path(D)$. (Lv et al., 2004)

Our work is similar to Arenas and Libkin (2004) but we simplify these definitions so that they are more practical, easy to understand and implement and give the same result. As shown in Section 4.3, we propose normal form for G-DTD (XNF G-DTD) at the schema level based from Arenas and Libkin's definition. Furthermore, we present also other normal forms such as 1XNF, 2XNF and 3XNF which are generalised from definitions given by Kolahi, (2007) and Lv et al. (2004). Generally, the nature of definition of the normal form G-DTD depends on a number of conditions. First, except root node, each complex element node must have an attribute node key. Second, there should not exist any nested path link between complex element nodes starting from root element node until last complex element node with global, transitive and partial functional dependency. The significant differences between our normal form and the one presented in Arenas and Libkin (2004), Kolahi (2007) and Lv et al. (2004) are:

- a) Firstly, the G-DTD model is proposed to assist in XML document design at the conceptual model. Our normal forms are based on a combination of different types of functional dependencies such as relationship dependency, transitive functional dependency, partial functional dependency, and global functional dependency definitions. We define these functional dependencies based on G-DTD structure without considering the path, which is simpler than in Arenas and Libkin's approach. However Arenas and Libkin (2004), Kolahi (2007) and Lv et al.(2004) they totally define their normal forms based on the XFD definitions. XFD is formally defined on the concept 'tree-tuple' and path (Arenas and Libkin, 2004). As we know, functional dependency is already the area where designers have the most problem specifying in relational models (Date, 2000), so making them more complicated and unfamiliar to designer make XML document design more difficult.
- b) Second, we apply our defined functional dependencies based on the structure of G-DTD at the schema level to present explicitly the position and level of complex elements, simple elements, attribute nodes and the semantic relationships. Based upon the analysis of these structural properties, the G-DTD schema is iteratively transformed into refined normal form, preserving the data dependencies. To

preserve data dependencies, all the data dependencies are updated iteratively to match with the current G-DTD structure. We differ fundamentally from the previous effort because we take the functional dependencies from a G-DTD schema rather than in various ways over path in XML document structure (instance). We believe that it is easier for designers to specify and understand functional dependency constraints in terms of a conceptual model than in terms of XML document structures. We therefore also offer our approach as a way to avoid having to specify these more complex and low-level constraints.

(3) Normalization Algorithm

Arenas and Libkin (2004) have proposed a normalization algorithm to transform XML design into XNF design. This algorithm performs iteratively two rules to produce XNF design.

(i) *Moving attributes.*

The rule is used to eliminate global dependency. Consider global XFD: $p \rightarrow q.@l$. To eliminate this XFD, the attribute $@l$ is moved from its original location to a new location of the last element of p

(ii) *Creating new element types*

The rule is used to eliminate local dependency. Consider XFD: $p, q_1.@l_1, \dots, q_n.@l_n \rightarrow q.@l$. To eliminate this type of XFD, a new element type e is created as a child of last p and attribute $@l_1, \dots, @l_n$ are copied for τ and $@l$ is moved from its original location to become an attribute of τ .

The algorithm proposed by Arenas and Libkin (2004) only considers the case of local and global functional dependency. To make this algorithm more flexible, we enhance the rules from Arenas and Libkin (2004) and Lv et al. (2004) with some modification to suit our normal forms definition. As presented in Chapter 4 Section 4.2, we next explain it in a more general way.

- (a) Given the initial DTD, it will be analysed and carefully investigated. Using our various notations, the recognition of complex elements, simple elements, attributes and semantic relationships which include n-ary path links types of many-to-many/many-to-one/one-to-many are carefully analysed. This will establish a better G-DTD.
- (b) Based on user requirements, the G-DTD is refined, by supplementing the additional properties and data dependencies such as GFD, TFD, PFD and relationship dependency.
- (c) Transform the initial G-DTD into normalised G-DTD i.e. 1XNF, 2XNF, 3XNF or XNF design. This procedure is iteratively implemented by taking the previous refined G-DTD until it becomes normal form. In order to establish a systematic approach to obtain a better G-DTD schema, we propose normalization rules. As shown in section 4.4.1, we add another rule (rule 1 and rule 2) which is moving up a complex element together with its children to another location and level. These rules are used to eliminate TFD and PFD in the G-DTD by considering one-to-many, many-to-many or many-to-one path links between complex element nodes in G-DTD.
- (d) Based upon the analysis of these structural semantic relationships, the G-DTD schema is iteratively transformed into refined normal form, preserving the data dependencies as well. It is basically required that the data dependency should be preserved throughout the transformation of the G-DTD model. To preserve the data dependencies, each data dependency is updated to match the new location of associated complex elements, simple elements and attributes in G-DTD. By the analysis of the data dependencies, the initial G-DTD schema may be transformed into a better schema which is normalised and simplified.

4.7 Discussion

Arenas and Libkin's approach has a remarkable impact on the principles of XML document normalization. However, we have found that XML database designers expect a somewhat different and more practical technique in such a way that:

- a) The definition has to be simple, precise, and understandable and should work with a minimum of abstract concepts, similarly to "the classical relational model normalization". We believe that introduction of difficult notations distinctively exceeding classical normal forms by having many types of concepts, is not desirable.
- b) The focus should be concretely on the graphical interpretation of XML schema model rather than textual representation. This is because graphical modelling language is widely accepted as a more visually effective means of specifying and communicating data requirements and suitable for users who have no technical background (Gustas, 2010; Halpin, 2010).
- c) We need to model structures and semantics of elements and attributes of the XML document schema (i.e. DTD) used for data storage and data manipulation. As stated by Feng et al. (2001), "*To enable efficient business application development in large-scale electronic commerce environment, it is necessary to describe and model real world semantics and their complex interrelationship*" (pp 391).
- d) It has to be flexible to enable designers to choose which normal form design is better for their application requirements and at the same time preserve the data constraints.

We believe that our method complements to Arenas and Libkin (2004) approach but we also consider all the four requirements that have been pointed up above when designing an XML document. Besides that, we found that the main features of G-DTD's notation are that it is particularly easy to improve XML structural design and more importantly, makes the XML normalization procedure simpler and practical. Since the semantic relationships represented in this approach are not complicated, it is thus easy to use G-DTD to maintain XML documents as well.

4.8 Summary

A set of normal forms called first normal form (1XNF), second normal form (2XNF), third normal form (3XNF) and fourth normal form (4XNF) for G-DTD have been defined for XML documents. Normalization algorithms have been proposed to transform from one normal form to another level of normal form. In this proposed algorithm, an original G-DTD is restructured by considering the tree structure, the level of nodes and the semantic relationship between nodes associated to different types of data dependencies such as partial functional dependency, transitive functional dependency and global functional dependency.

To demonstrate the concept, we provide a simple case study to illustrate the application of our proposed G-DTD normal forms and normalization rules to achieve a redundancy-free XML document. We show through a case study that having G-DTD as a conceptual model simplifies the complex procedure of XML document design and normalization. Even though the length of XML document is longer than the original document, the structure is free from data redundancy and update anomalies.

The normal form G-DTD presented in this chapter has shown three advantages. First, the designer can identify complex elements, simple elements and attributes graphically and can add the relationship types between the nodes from user specification. Hence, this will give more control to the designer to evaluate each successive normal form G-DTD. Second, normalizing the G-DTD can effectively remove redundancies and anomalies at a schema level. More importantly, it is able to preserve both DTD hierarchical structure and XML document structure and satisfy user requirement. Next, we will define formally the prototype system of XML document design using a formal specification method.

Chapter 5

Formal Specification of XML Document Design Model

5.1 Introduction

This chapter presents a formal specification of the XML document design system called XML design model (XML_DM) which comprises a conceptual model G-DTD and normalization procedure which we discussed in the previous chapter. This formal specification is used to describe a fundamental framework of what the XML_DM can do and also as an abstraction of a full complete system which can serve as a reliable blueprint for those who want implement the prototype later. This formal specification is important before the implementation of the real system is developed, as it allows a designer to understand the big picture of the system; and helps to discover error early in the lifecycle and reduce the overall cost of the project. As stated by Sommerville (2010),

“... it is easier to build a system from a formal specification than by using other methods. Coding from formal specification to be straight forward. The application of formal methods can also make the development process of each stage clearer. More importantly, monitoring is more reliable and thus development is less risky.”

The rest of the chapter is organized as follows. Section 5.2 explains the importance of formal specification and some related work. Section 5.3 presents the benefits of Z notation. In section 5.4 we present a conceptual design framework of an XML design model (XML-DM). All the specification will be formalised using Z specification language. In our work, we use a Z Word tools (Hall, 2010) where fuzz is used for typing and checking Z specification.

5.2 The Formal Specification and Its Importance

According to Spivey (1992), “Formal specification use mathematical notation to describe in a precise way the properties which an information system must have, without unduly constraining the way in which these properties are achieved”.

This statement also supported by Sommerville (2010), which stated that “..creating a formal specification force you to make a detail system analysis that usually reveals error and inconsistencies in the formal requirement specification”.

Amongst the important roles of formal specification are that it can be used to clarify the understanding of the problem, to provide a prototype to demonstrate the idea and it can be used as a basis for system design (Wang, 1999; Hall, 2000). Formal specification has been widely recognized as a precise way to define the structure of a complex software system such that its usages have increased in last two decades and now it is frequently used in industry (Mian and Zafar, 2010). More importantly, the literature has shown and proved that formal specification is particularly cost-effective with computerization of small-to-medium sized information systems.

For instance, the use of formal specification has been applied in scheduling of railway system design(Hexthausen and Peleska, 2000), air traffic control system(Paternof et al.,1998), a radiation therapy machine(Jacky, 1997), hypertext system (Wang, 1993; Halasz and Schwartz, 1994 ; D’Inverno and Hu, 1997), network security (Singh and Patterh, 2010), healthcare application (Coronato and De Pietro, 2010), web service system (Liu, F. et al., 2011) and many more.

An attempt of using formal specification as a means of developing databases is also found in the work of Boros (1994), which is focused on the provision of a framework of development, to enable software engineers to develop database specification and to solve database design problems.

On the other hand, formal specification also can be used for checking and verification. Related to semi-structured data design, there are many attempts to formally verify a

semi-structured data model. The most relevant work using formal specification to verify a data model called Object Relational Attribute for Semistructured(ORA-SS) is done by Lee et al. (2006), Wang et al. (2006), Lee et al. (2009) and Lee et al. (2010). They used different types of formal method language to present the syntax and semantics of the model. For instance, Lee et al. (2006) used Z formal language to validate the syntac and semantic of the ORA-SS model. They also validate the model to check the correctness of ORA-SS at both schema and instance level. Similar to this work, the formalization of ORA-SS using OWL was presented with improved verification performance. Recently, Lee et al. (2009) used a different approach to define a formal specification for ORA-SS using Prototype Verification System (PVS) language. In this work they also presented type checking and theorem proving abilities. In other research, Wang et al. (2006) presented a formalization of the ORA-SS notations using Alloy. However, to the best of our knowledge no formal specification has been developed to define an automatic system of XML document design. Thus, we can claim that our formal specification presented here is a novel approach for an XML document design prototype.

5.3 Z notations

Growing awareness of the need of formal specification has led to the development of various specification languages. The most popular specification languages are Z (Spivey, 1988) and VDM (Jones, 1986), which have been recommended as official standard software system specifications. Z is a formal specification language originally developed at the Programming Research Group at Oxford University. In our work, the Z specification language, in particular, is used as a means of formalization for a number of reasons. First, the language is based upon primitive mathematical notation such as set theory and first order predicate logic, making it accessible to researchers from a variety of different backgrounds. Second, it is expressive enough to allow a consistent, formal and unified representational of a system and its associated operations. Third, it is model oriented (Bottaci and Jones, 1995). Model-oriented specification language seems to be more appropriate to specify an XML design model and its operations. Moreover, it has been claimed that, in general, human being tend to find model-oriented methods

easier to understand. Finally, in particular, we have found Z is an established language, widely accepted and appropriate for building formal frameworks, necessary to enable a rigorous approach for any discipline including interactive conferencing system (D’Inverno et al., 1991), distributed artificial intelligence (D’Inverno and Hu, 1996), multi-agents systems (D’Inverno and Luck, 1996) and design hypertext models (Lange, 1990; Halasz and Schwartz, 1994; Wang, 1993; D’Inverno et al., 1997).

However, we found that the main disadvantage of using Z is that mapping both the *XML design model (XML_DM)* semantics and operations to Z can result in very verbose specifications. Intuitively, expressing the *XML design model (XML_DM)* semantic using a meta-model is much easier for non-specialists to understand and comprehend than a Z specification. This is a critical shortcoming which will need to be addressed where the user must have working knowledge of the formal notations. The main difficulty is making the right connections between the real world and the mathematical formalism. However, we believe that, even though many practitioners involved with the development of computer systems are not mathematicians and do not make regular use of mathematics, mathematical technique are increasingly important role in the development of software and it is appropriate that they should be known and applied by as many practitioners as possible. This is because the mathematics used in formal specification is very simple; it only uses *discrete mathematics* which is concerned more with sets and logic than with numbers.

5.4 The overview of XML Document Design Model (XML_DM)

The XML_DM model consists of three layers: the conceptual G-DTD layer, the normalizer layer and G-DTD translator layer, as illustrated in Figure 5.1. The G-DTD conceptual model layer helps the user to create, query, insert, and delete a node/link of a G-DTD model.

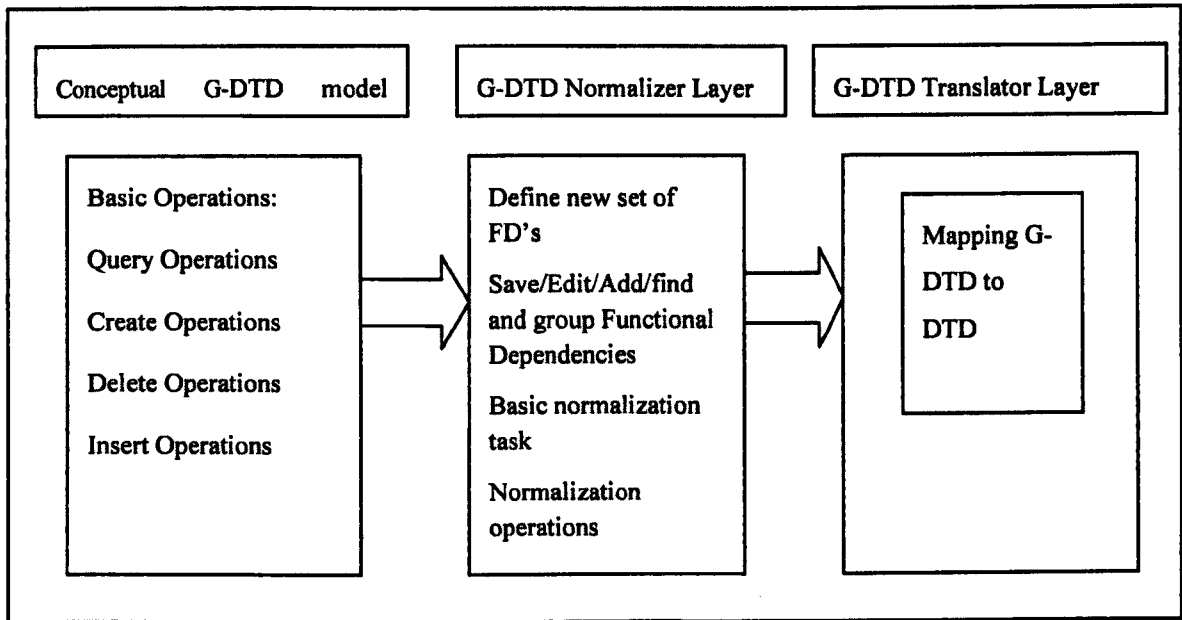


Figure 5.1: Layers of XML Document Design Model

It allows database designers/users to use a G-DTD model to describe XML documents by using simple notations provided at the interface layer. As presented in Chapter 3, G-DTD describes an XML document using basic sets such as complex element, simple element, attribute nodes and invariant relationships between these nodes. The details of its conceptual operations have been presented in Chapter 3, Section (3.6).

The G-DTD normalizer layer allows the user to normalize a G-DTD using the normalization rules and algorithms which have been presented in Chapter 4. This layer is used to assist the user to design redundancy-free XML documents by normalizing the G-DTD on the basis of a set of normal forms. This specification describes what XML_DM provides to achieve 1XNF, 2XNF, 3XNF and 4XNF of G-DTD. It will serve as a visual aid for the normalization process which is always easy to understand and interpret rather than a theoretical approach.

The G-DTD Translator layer provides the user the functionality to map a G-DTD back to DTD using transformation rules provided in Chapter 3 (Section 3.7), so that the standard template of a redundancy-free XML document can be generated which conforms and satisfies a new DTD.

We present the XML document design model (XML_DM) specification framework into three sections: formal specification of a conceptual G-DTD model, formal specification of G-DTD normalizer and formal specification of G-DTD translator. We present them next.

5.5 Formal Specification of a Conceptual G-DTD Model

In XML-DM environment consists of a conceptual G-DTD model which represents an abstract view of an XML document. It defines all the simple element, complex element and attributes nodes and the links between these nodes. To define those types of node, we define first the basic set used in a G-DTD.

5.5.1 Basic Type Definitions

[ID, Element_Name, Attribute_Name, Relation_Name]

where ID is a set of all unique ID for each element. Element_Name is the set of all possible XML element node names and Attribute_Name is the set of all possible XML attribute node names. Relation_Name is a set of relationship names. Terms like complex element or simple element are used but we will reserve these for specific concepts, and will be seen shortly.

The structure of the G-DTD is represented as collection of all types of nodes and links where links can only exist between a pair of nodes in the space. Nodes cannot be linked to themselves. As we described in Chapter 3 section 3.6, the characteristic of each type of node such as simple element, complex element and attribute node can be described using the following representation.

5.5.2 The Data Structure of G-DTD

Simple Element Node

SimpleElement_Node schema consists of identity, name, level and type of simple element. Simple element type can be classified either as single value, multivalued,

optional single value or optional multivalued. The type definition for a simple element is defined as follows:

SimpleElement_Type ::= singlevalue | multivalued | op_singlevalue | op_multivalued

SimpleElement_Node

identity : ID
name : Element_Name
level : N
se_type : SimpleElement_Type

Attribute Node

The Attribute_Node schema denotes the unique identifier of each complex element node. The Attribute_Node schema consists of identity, name, level and attribute type. The type of attribute can be either composite, required or reference and is defined by using the following attribute type definition.

Attribute_Type ::= composite | required | reference

Attribute_Node

identity : ID
name : Attribute_Name
level : N
att_type : Attribute_Type

Complex Element Node

The ComplexElement_Node schema is presented as follows which consists of its identity, complex element name and level.

ComplexElement_Node

identity : ID
name : Element_Name
level : N

Parent for Complex Element Node, Simple Element Node and Attribute Node

A parent for each type of related node, i.e. complex element node, simple element node and attribute node is described precisely in relationships. The functions *Parent_Ce*, *Parent_Se* and *Parent_Att* are defined using the axiomatic function as a total function because every complex element node, simple element node and attribute node must have its own parent node and no node can have more than one parent. In the state invariant, it is stated that complex element $ce1 \mapsto ce2 \in Parent_Ce$ means that $ce2$ is the parent of $ce1$ if and only if $ce1$ is not the same as $ce2$ and the level position of $ce2$ must always be less than the level position of $ce1$ by one level difference only. The same meaning is applied for the second and third predicates associated to both parent of a simple element node and parent of an attribute node.

Parent_Ce : *ComplexElement_Node* → *ComplexElement_Node*

Parent_Se : *SimpleElement_Node* → *ComplexElement_Node*

Parent_Att : *Attribute_Node* → *ComplexElement_Node*

∀ *ce1, ce2*: *ComplexElement_Node* •

$ce1 \mapsto ce2 \in Parent_Ce \Leftrightarrow$

$(ce1 \neq ce2 \wedge ce2.level - ce1.level = 1)$

∨

(∀ *se* : *SimpleElement_Node*; *ce*: *ComplexElement_Node* |

∃ *parent*: *ComplexElement_Node* •

$se \mapsto ce \in Parent_Se \Leftrightarrow se \in Parent_Se \langle parent \rangle \wedge$
 $se.level - ce.level = 1)$

∨

(∀ *att* : *Attribute_Node*; *ce*: *ComplexElement_Node* | •

∃ *parent*: *ComplexElement_Node* •

$att \mapsto ce \in Parent_Att \Leftrightarrow att \in Parent_Att \langle parent \rangle$
 $att.level - ce.level = 1)$

Relationships

On the G-DTD schema diagram, we distinguish three types of relationship; *Path_Link*, *Part_of_Link*, *Has_A_Link*. We define the schema of each type of relationship as follows:

(1) Path_Link

As described in Chapter 3 Section (3.4.3.5), path link has very important features. To be more precise we capture those features in following Path_Link schema. This schema consists of the relation between two complex element nodes. The state invariant states an ordered pair of complex element nodes $ce1 \mapsto ce2$ is an element of *path_link* if and only if $ce2$ is an immediate parent of $ce1$. The complex element $ce2$ is a parent of $ce1$ if and only if $ce1 \mapsto ce2 \in path_link^+$; that is to say, a transitive closure relation (Diller, 1994), and the relation is a cycle free if and only if no complex element node is mapped to itself. We are using transitive closure because the transitive closure allows complex element node to be directly reached in the same link (Diller, 1994).

<i>Path_Link</i>
$path_link : ComplexElement_Node \leftrightarrow ComplexElement_Node$ $name : Relation_Name$ $degree : \mathbb{N}_1$ $parent_constraint, child_constraint : (\mathbb{N} \times \mathbb{N}_1)$
$(\forall ce1, ce2 : ComplexElement_Node$ <ul style="list-style-type: none"> • $ce1 \mapsto ce2 \in path_link$ $\Leftrightarrow Parent_Ce(ce1) = ce2 \wedge ce1 \mapsto ce2 \in path_link^+$ $\wedge (\exists ce : ComplexElement_Node \bullet ce \mapsto ce \notin path_link^+)$ $\wedge (\forall name1, name2 : Relation_name \bullet name1 \neq name2)$ $\wedge (d : degree \bullet \# degree \geq 2)$ $\wedge (\forall pc : parent_constraint, cc : child_constraint, card : (\mathbb{N} \times \mathbb{N}_1) \bullet$ $pc = second\ card \geq first\ card \wedge$ $cc = second\ card \geq first\ card)$

The additional properties of path links such as name, degree of relationship, parent and child cardinality constraint are defined to a path link. For a relationship's name, the name of the relationship must be unique. The degree of relationship is represented as a natural number and must be equal to or greater than two. Every path link type in a G-DTD model has its associated parent and child constraints to set the lower limit and upper limit cardinality for them respectively.

(2) Part_of_Link

The schema Part_of_Link presents the relation between a complex element node and attribute node. It consists of two types of relations: Attributekey function and Compositekey relation. Attributekey function is total and injective because each complex element node has a unique attribute node. CompositeKey relation is a relation between a complex element and set of attributes. In the the first predicate, $ce \mapsto att \in AttributeKey$ means that att is an attributekey for ce if and only if attribute type is required. In the second predicate, $(ce \mapsto attcom) \in CompositeKey$ means that $attcom$ is a composite key for ce if and only if its attribute type is composite. The last predicate indicates that domain for function AttributeKey and relation CompositeKey is a member of a complex element node.

Part_of_Link

AttributeKey : ComplexElement_Node \mapsto Attribute_Node

CompositeKey : ComplexElement_Node \leftrightarrow Attribute_Node

$\forall ce : ComplexElement_Node; att : Attribute_Node \bullet$

$(ce \mapsto att) \in AttributeKey \Leftrightarrow att.att_type = required \wedge Parent_Att(att) = ce$

$\forall ce : ComplexElement_Node; attcom : Attribute_Node \bullet$

$(ce \mapsto attcom) \in CompositeKey \Leftrightarrow attcom.att_Type = composite$

$\wedge Parent_Att(attcom) = ce$

$dom AttributeKey \cup dom CompositeKey \in ComplexElement_Node$

(3) Has_A_Link

The schema Has_A_Link presents the relation between a complex element node and simple element node. The complex element node is declared as the parent node while the simple element node is a child node

$\begin{array}{l} \textit{Has_A_Link} \\ \textit{hasa} : \textit{ComplexElement_Node} \leftrightarrow \textit{SimpleElement_Node} \\ \forall ce : \textit{ComplexElement_Node}; se : \textit{SimpleElement_Node} \bullet \\ (ce \mapsto se) \in \textit{hasa} \Leftrightarrow se.se_type = \textit{singlevalue} \vee se.se_type = \textit{multivalued} \\ \vee se.se_type = \textit{op_singlevalue} \vee se.se_type = \textit{op_multivalued} \\ \wedge \textit{Parent_Se} \ se = ce \\ \textit{ran hasa} \in \mathbb{P}\textit{SimpleElement_Node} \end{array}$
--

5.5.3 Abstract state of Environment XML_DM

We next define the following structure of G-DTD which is used later in presenting a readable specification. The G-DTD consists of seven variables whose values are restricted by the state invariant. These are a root node type, set of ComplexElement_Node, and set of SimpleElement_Node type, set of Attribute_Node and set of relation Part_of, Has_A and Path_Link type. The following schema captures the abstract state of the G-DTD.

SchemaGDTD

$root : ComplexElement_Node$
 $Cnodes : \mathbb{F}ComplexElement_Node$
 $Snodes : \mathbb{F}SimpleElement_Node$
 $Attnodes : \mathbb{F}Attribute_Node$
 $PathLink : seq(\mathbb{P}Path_Link)$
 $HasA : Has_A_Link$
 $Partof : Part_of_Link$

$\exists_1 root : ComplexElement_Node \bullet root.level = 0$
 $\forall ce1, ce2 : Cnodes \mid ce1 \neq ce2 \bullet ce1.name \neq ce2.name$
 $\forall se1, se2 : Snodes \mid se1 \neq se2 \bullet se1.name \neq se2.name$
 $\forall att1, att2 : Attnodes \mid att1 \neq att2 \bullet att1.name \neq att2.name$
 $\forall partlink : Partof \bullet partlink.AttributeKey \neq \emptyset$
 $\forall hl : PathLink ; haslink : HasA ; partlink : Partof \bullet$
 $Cnodes = U\{dom\ hl.path_link, ran\ hl.path_link\}$
 $Snodes = ran\ haslink.hasa$
 $Attnodes = ran\ partlink.AttributeKey$

The first predicate of the *SchemaGDTD* states that there must exist one root node. The second, third and fourth predicates indicate that at any point in time, each complex element node, simple element node and attribute node must have a unique name. The last three predicates ensure that all types of nodes and relationships defined exist in *SchemaGDTD*.

Finally the abstract state of XML_DM environment consists of schema *SchemaGDTD*

Environment_XML_DM

SchemaGDTD

$\exists_1 S : SchemaGDTD \bullet S \neq \emptyset$

5.5.4 Initial state of Environment XML_DM

Before any operation can be performed on the model, we must define the initial state of environment XML_DM. In our case, the initial state of environment XML_DM refers to the situation in which there are no elements of *SchemaGDTD* existing in it. This is characterized by the following schema definition:

Initial_state
$\Delta Environment_XML_DM$
$Schema.Snodes = \emptyset$
$Schema.Cnodes = \emptyset$
$Schema.Attnodes = \emptyset$
$Schema.Partof = \emptyset$
$Schema.HasA = \emptyset$
$Schema.PathLink = \emptyset$

This schema describe the Initial_state in which set of simple element node, complex element node and attribute node are empty: in consequence, the PathLink, HasA and Partof relation is empty too

5.5.8 Manipulating the G-DTD in an Environment

The manipulation of G-DTD in an XML_DM environment includes four groups of operations known as querying, inserting, deleting and moving. We will formally define those operations on the basis of their rules description defined in Chapter 3 Section (3.6). However, before we present these operations we must first define the following functions and schemas, since they will be used in future operations schema definition.

(1) Get ID, Level and Node Numbers

Before querying, inserting or deleting any node, we should be aware of its properties. In order to achieve this aim we define the three functions *Get_ID*, *Get_Name*, *Get_Level* and *Get_Node_Type*, which identify the ID, name, level and node type of a node.

$Get_ID : ComplexElement_Node \leftrightarrow ID$ $Get_Name : ComplexElement_Node \leftrightarrow Element_Name$ $Get_Level : ComplexElement_Node \leftrightarrow \mathbb{N}$
$\forall ce : ComplexElement_Node \bullet$ $Get_ID\ ce = ce.ID$ $Get_Name\ ce = ce.Name$ $Get_Level\ ce = ce.Level$

(2) Create Complex Element Node

The *Create_NewComplexElement_Node* function is used to create a new complex element node. Each new node must have an instance of *ID*, *Element_Name* and *level*. The new complex element is added to a set of complex element nodes if it satisfies the *pre-condition* that the new node is not already be one of the members of complex element node in *schemaGDTD*. This is because only one unique complex element is allowed in G-DTD schema.

$Create_NewComplexElement_Node : (ID \times Element_Name \times \mathbb{N}_1)$ $\rightarrow ComplexElement_Node$
$\forall newid : ID; newname : Element_Name; l : \mathbb{N}_1; schema, schema' : SchemaGDTD \bullet$ $(\exists ce, newnode : ComplexElement_Node; $ $newnode = ce \bullet$ $(ce.identity = newid \wedge ce.name = newname \wedge ce.level=l) \wedge$ $newnode \notin schema.Cnodes \wedge$ $schema'.Cnodes = schema.Cnodes \cup \{newnode\}$ $\Rightarrow createNewComplexElement(newid, newname, l) = newnode)$

(3) Create Simple Element node

The *Create_SimpleElement_Node* function is used to create simple element node. Given the instance of *ID*, *Element_Name*, *level*, *simple element type* and a simple element node is created. The success of the operation relies on a given *pre-condition*. The new

simple element node is added if and only if it is not already one of the members of simple element node in schema GDTD. This is because only one unique simple element node is allowed in schema GDTD.

$\text{Create_SimpleElement_Node} : (ID \times \text{Element_Name} \times \mathbb{N}_1 \times \text{Se_Type}) \rightarrow \text{SimpleElement_Node}$ <hr style="width: 50%; margin-left: 0;"/> $\forall \text{newid} : ID; \text{newname} : \text{Element_Name}; l : \mathbb{N}_1; \text{type} : \text{Se_Type};$ $\text{schema} : \text{SchemaGDTD} \bullet$ $(\exists \text{se}, \text{newnode} : \text{SimpleElement_Node}; \text{schema}' : \text{SchemaGDTD} $ $\text{newnode} = \text{se} \bullet$ $(\text{se.identity} = \text{newid} \wedge$ $\text{se.name} = \text{newname} \wedge$ $\text{se.level} = l \wedge$ $\text{se.seType} = \text{type}) \wedge$ $\text{newnode} \notin \text{schema.Snodes} \wedge$ $\text{env'.Snodes} = \text{env.Snodes} \cup \{\text{newnode}\}$ $\Rightarrow \text{Create_SimpleElement_Node}(\text{newid}, \text{newname}, l, \text{type}) = \text{newnode})$
--

(4) Create Attribute node

The description of the *Create_Attribute_Node* function is similar to the *Create_SimpleElement_Node* function.

$$\text{Create_Attribute_Node} : (ID \times \text{Attribute_Name} \times \mathbb{N}_1 \times \text{Att_Type}) \rightarrow \text{Attribute_Node}$$

$$\forall \text{newid} : ID; \text{newname} : \text{Attribute_Name}; l : \mathbb{N}_1; \text{type} : \text{Att_Type};$$

$$\text{Env} : \text{Environment_XML_DM} \bullet$$

$$(\exists \text{att}, \text{newnode} : \text{Attribute_Node}; \text{schema}' : \text{SchemaGDTD} |$$

$$\text{newnode} = \text{att} \bullet$$

$$(\text{att.identity} = \text{newid} \wedge$$

$$\text{att.name} = \text{newname} \wedge$$

$$\text{att.level} = l \wedge$$

$$\text{att.attType} = \text{type}) \wedge$$

$$\text{newnode} \notin \text{env.Attnodes} \wedge$$

$$\text{schema'.Attnodes} = \text{schema.Attnodes} \cup \{\text{newnode}\}$$

$$\Rightarrow \text{createAttribute_Node}(\text{newid}, \text{newname}, l, \text{type}) = \text{newnode})$$

(5) Create Path_Link

Create_Path_Link is a function to create a new Path_Link between two complex element nodes. The function maps both given complex element node and complex element node and assigns between them a new Path_Link if and only if it satisfies the pre-conditions that the relation of these complex element nodes are not a cyclic relation and they have a parent relation. If the condition is satisfied, the new instances of relation name, level, parent and child constraint are assigned to a new Path_Link and added to set of Path_Link.

$$\text{Create_Path_Link} : (\text{ComplexElement_Node} \times \text{ComplexElement_Node}) \rightarrow \text{Path_Link}$$

$$\forall ce1 : \text{ComplexElement_Node}; ce2 : \text{ComplexElement_Node}; \text{schema} : \text{SchemaGDTD} \bullet$$

$$\begin{aligned} & \exists \text{new_Path_Link}, \text{newlink} : \text{Path_Link}; \\ & \text{deg} : \mathbb{N}_1; \text{pc}, \text{cc} : \mathbb{N} \times \mathbb{N}_1; \text{newname} : \text{Relation_Name}; \\ & \text{schema}' : \text{SchemaGDTD} \{ \\ & \text{new_Path_Link} = \text{newlink} \bullet \\ & \quad ce1 \mapsto ce2 \in \text{newlink.path_link} \Leftrightarrow \\ & \quad (ce1 \mapsto ce2 \notin \text{newlink.path_link}^+ \\ & \quad \wedge ce2 = \text{Parent_Ce}(ce1) \\ & \quad \wedge \text{newlink.name} = \text{newname} \\ & \quad \wedge \text{newlink.degree} = \text{deg} \\ & \quad \wedge \text{newlink.parent_constraint} = \text{pc} \\ & \quad \wedge \text{newlink.child_constraint} = \text{cc}) \\ & \quad \wedge \text{schema}'.\text{Path_Link} = \\ & \quad \quad \text{schema.Path_Link} \cup \{ \text{new_Path_Link} \} \\ & \quad \Rightarrow \text{Create_Path_Link}(ce1, ce2) = \text{new_Path_Link} \end{aligned}$$

(6) Create Has_A_Link

Create_Has_A_Link is a function to create a new *Has_A* link between the complex element node and simple element node. To build a *Has_A* link, a simple element node must have a parent which is a complex element node. When a new *Has_A* link is created, it is added to the set of *Has_A* links.

$$\text{Create_Has_A_Link} : (\text{ComplexElement_Node} \times \text{SimpleElement_Node}) \rightarrow \text{Has_A_Link}$$

$$\forall ce : \text{ComplexElement_Node}; se : \text{SimpleElement_Node}; \text{schema} : \text{SchemaGDTD} \bullet$$

$$\begin{aligned} & \exists \text{new_Haslink}; \text{newlink} : \text{Has_A_Link}; \text{schema}' : \text{SchemaGDTD} \{ \\ & \text{new_Haslink} = \text{newlink} \bullet \\ & \quad ce \mapsto se \in \text{newlink.hasa} \\ & \quad \wedge \text{schema}'.\text{HasA} = \text{schema.HasA} \cup \{ \text{new_Haslink} \} \\ & \quad \Rightarrow \text{Create_Has_A_Link}(ce, se) = \text{new_Haslink} \end{aligned}$$

(8) Deleting a Path_Link

When deleting a complex element node, we should consider the task of deleting its corresponding links. We further define a function, *Delete_Hierarchical_Link*, which describe the general task of deleting the link. By deleting the link the instance of link will automatically be deleted as well.

$$\begin{array}{l} \textit{Delete_Path_Link} : \\ \textit{ComplexElement_Node} \times \textit{ComplexElement_Node} \times \textit{Path_Link} \\ \times \textit{SchemaGDTD} \rightarrow \textit{SchemaGDTD} \\ \hline \forall ce1, ce2 : \textit{ComplexElement_Node}; link : \textit{Path_Link}; \\ schema, schema' : \textit{SchemaGDTD} \bullet \\ \quad link \in schema.\textit{PathLink} \\ \quad \textit{Delete_Path_Link}(ce1, ce2, link, schema) = schema' \Leftrightarrow \\ \quad ce1 \mapsto ce2 \in link \wedge \textit{Parent_ce}(ce2) = ce1 \\ \quad \Rightarrow schema'.\textit{PathLink}.hlink = \\ \quad \textit{PathLink} \triangleleft schema.\textit{PathLink}.link \end{array}$$

(9) Deleting a Has_A_Link

When deleting a simple element node, we should consider the task of deleting its corresponding links. We further define a function, *Delete_Has_A_Link*, which describes the general task of deleting the link between a complex element node and simple element node.

$$\begin{array}{l} \textit{Delete_Has_A_Link} : \\ (\textit{ComplexElement_Node} \times \textit{SimpleElement_Node} \times \textit{Has_A_Link} \\ \times \textit{SchemaGDTD} \rightarrow \textit{SchemaGDTD} \\ \hline \forall ce : \textit{ComplexElement_Node}; se : \textit{SimpleElement_Node}; \\ link : \textit{Has_A_Link}; schema, schema' : \textit{SchemaGDTD} \bullet \\ \quad link \in schema.\textit{HasA} \\ \quad \textit{Delete_Has_A_Link}(ce, se, link, schema) = schema' \\ \quad \Leftrightarrow ce \mapsto se \in has_a_link \wedge \textit{parent_se}(se) = ce \\ \quad \Rightarrow schema'.\textit{HasA}.link = \textit{HasA}.link \triangleleft schema.\textit{HasA}.link \end{array}$$

(10) Deleting a Part_of_Link

Similar to the above schema, we further define a function, *Delete_Part_of_Link*, which describes the general task of deleting the link between a complex element node and attribute node.

$$\begin{array}{l} \textit{Delete_Part_of_Link} : \\ (\textit{ComplexElement_Node} \times \textit{Attribute_Node} \times \textit{Part_of_Link} \times \textit{SchemaGDTD} \\ \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \rightarrow \textit{SchemaGDTD} \\ \hline \forall ce : \textit{ComplexElement_Node}; se: \textit{SimpleElement_Node} ; link: \textit{Part_of_Link}; \\ \textit{schema}, \textit{schema}': \textit{SchemaGDTD} \bullet \\ \quad \textit{link} \in \textit{schema.Partof} \\ \quad \textit{Delete_Part_of_Link} (ce, se, link, \textit{schema}) = \textit{schema}' \\ \quad \Leftrightarrow ce \mapsto se \in link \wedge \textit{parent_se} (se) = ce \\ \quad \Rightarrow \textit{schema}'.\textit{Partof}.link = \textit{Partof}.link \triangleleft \textit{schema.Partof}.link \end{array}$$

(11) Checking type of Path_Link

The following four functions are used to check the *Path_Link* Type.

$$\begin{aligned}
& \text{Is_Rel_One_to_One} : \text{Path_Link} \times \text{SchemaGDTD} \rightarrow \text{Boolean} \\
& \text{Is_Rel_One_to_Many} : \text{Path_Link} \times \text{SchemaGDTD} \rightarrow \text{Boolean} \\
& \text{Is_Rel_Many_to_One} : \text{Path_Link} \times \text{SchemaGDTD} \rightarrow \text{Boolean} \\
& \text{Is_Rel_Many_to_Many} : \text{Path_Link} \times \text{SchemaGDTD} \rightarrow \text{Boolean}
\end{aligned}$$

$$\begin{aligned}
& \forall \text{rel} : \text{Path_Link}; \text{schema} : \text{SchemaGDTD} \mid \\
& \text{rel} \in \text{schema.PathLink} \wedge \\
& \exists \text{pc} : \text{rel.parent_constraint}; \text{cc} : \text{rel.child_constraint} \bullet \\
& \quad \text{Is_Rel_One_to_One}(\text{rel}, \text{schema}) = \text{True} \Leftrightarrow \\
& \quad (\text{first rel.pc} = 1 \wedge \text{second rel.pc} = 1 \wedge \text{first rel.cc} = 1 \wedge \text{second rel.cc} = 1) \vee \\
& \quad \text{Is_Rel_One_to_Many}(\text{rel}, \text{schema}) = \text{True} \Leftrightarrow \\
& \quad (\text{first rel.pc} = 1 \wedge \text{second rel.pc} = 1 \wedge \text{first rel.cc} \geq 1 \wedge \text{second rel.cc} \geq 2) \vee \\
& \quad \text{Is_Rel_Many_to_One}(\text{rel}, \text{schema}) = \text{True} \Leftrightarrow \\
& \quad (\text{first rel.pc} \geq 1 \wedge \text{second rel.pc} \geq 2 \wedge \text{first rel.cc} = 1 \wedge \text{second rel.cc} = 2) \vee \\
& \quad \text{Is_Rel_Many_to_Many}(\text{rel}, \text{schema}) = \text{True} \Leftrightarrow \\
& \quad (\text{first rel.pc} \geq 1 \wedge \text{rel.second pc} \geq 2 \wedge \text{first rel.cc} \geq 1 \wedge \text{second rel.cc} \geq 2)
\end{aligned}$$

(12) Checking the root and last node

To check the root node and last node in the G-DTD, the following functions are used. The function *Is_root* has a complex element node and *schemaGDTD* as arguments and returns the value true if the given complex element is a root node.

The status of a result node is defined using the set of Boolean messages.

Boolean ::= True | False

$$\begin{aligned}
& \text{Is_root} : \text{ComplexElement_Node} \times \text{SchemaGDTD} \rightarrow \text{Boolean} \\
& \exists \text{ce} : \text{ComplexElement_Node}; \text{schema} : \text{SchemaGDTD} \bullet \\
& \quad \text{Is_root}(\text{ce}, \text{schema}) = \text{True} \Leftrightarrow \\
& \quad \text{ce} \in \text{schema.Cnodes} \wedge \text{ce.level} = 0
\end{aligned}$$

Similar to function *Is_root*, *Is_last* function has a simple element node and schema GDTD as arguments returns the value true if the node is a last node.

$$\begin{array}{|l}
\hline
Is_last : SimpleElement_Node \times SchemaGDTD \rightarrow Boolean \\
\hline
\forall se : SimpleElement_Node; schema : SchemaGDTD \bullet \\
Is_last(se, schema) = True \Leftrightarrow \\
se \in schema.Snodes \wedge se.level = maxlevel - 1
\end{array}$$

5.5.9 Operations of G-DTD in an Environment

(1) Query Operations

As presented in Chapter 3 Section (3.6), before manipulating the structure of any complex element node of G-DTD, we should be aware of its related nodes such as its attribute nodes and simple element nodes. Since the structure of G-DTD is like a tree structure, a child or descendant and parent or ancestor of given complex element node also need to be queried in some cases.

The status of a queried node is defined using the set of messages. It is defined by enumeration like this:

$$Report ::= Existence | Nonexistence | Inserted | Created$$

Based on this set, we define the following schema *Success* to output a confirmatory message that the operation being performed has been successfully completed.

$$\begin{array}{|l}
\hline
Success \\
report! : Report \\
\hline
report! = Existence \\
\hline
\end{array}$$

- **Query a Node**

Find_AttributeKey_Node shows how to get an attribute key associated to the given Complex Element Node

Find_AttributeKey_Node

\exists *Environment_XML_DM*

ce? : *ComplexElement_Node*

found_attkey! : *Attribute_Node*

report! : *Report*

\forall *schema: SchemaGDTD; part_of: Partof* •

ce? \notin dom *schema.part_of* \Rightarrow *report!* = *Nonexistence* \vee

dom *schema.part_of* \in *schema.Cnodes*

ce? \subseteq *schema.Cnodes*

ce? $\neq \emptyset$

found_attkey! = *schema.part_of.AttributeKey* (*ce?*)

Find_SimpleElement_Node schema describes that the Simple Element Node (*se!*) has been found and belongs to the complex element node of G-DTD defined in environment XML_DM.

Find_SimpleElement_Node

\exists *Environment_XML_DM*

ce? : *ComplexElement_Node*

found_se! : *SimpleElement_Node*

report! : *Report*

\forall *has_link: HasA; schema: SchemaGDTD* •

ce? \notin dom *schema.has_link.has_a* \Rightarrow *report!* = *Nonexistence* \vee

found_se! = *schema.has_link.has_a* (*ce?*)

Find_ComplexElement_Node schema describes that the ComplexElement_Node (*found_ce!*) has been found since its ID, name and level are equal to the input ID, name and level.

<p><i>Find_ComplexElement_Node</i></p> <p>\exists <i>Environment_XML_DM</i></p> <p><i>Ce</i> : <i>ComplexElement_Node</i></p> <p><i>ce_name?</i> : <i>Element_Name</i></p> <p><i>ce_level?</i> : \mathbb{N}</p> <p><i>ce_id?</i> : <i>ID</i></p> <p><i>found_ce!</i> : <i>ComplexElement_Node</i></p> <p><i>report!</i> : <i>Report</i></p> <hr/> <p>\forall <i>schema</i>: <i>SchemaGDTD</i> •</p> <p><i>ce</i> \in <i>schema.Cnodes</i></p> <p><i>ce?</i> \notin $\text{dom } \textit{schema.PathLink.path_link} \Rightarrow$</p> <p><i>report!</i> = <i>Nonexistence</i> \vee</p> <p>(<i>Get_Name ce</i> = <i>ce_name?</i> \wedge <i>Get_Level ce</i> = <i>ce_level?</i> \wedge</p> <p><i>Get_ID ce</i> = <i>ce_id?</i> \Rightarrow <i>found_ce!</i>)</p>

Finally, based on the schema definitions above, we can finally define the following schemas, which describes the state in which a complex element node, simple element node or attribute node has been successfully queried.

Query_SimpleElement_Node \triangleq *Find_SimpleElement_Node* \wedge *Success*

Query_AttributeKey_Node \triangleq *Find_AttributeKey_Node* \wedge *Success*

Query_ComplexElement_Node \triangleq *Find_ComplexElement_Node* \wedge *Success*

- **Querying a Related Nodes**

Query about the ancestor of complex element node can be done using a Path Link as described in *Find_Ancursor_schema*. We achieve this by forming the relational inverse transitive closure of a Path_Link.

Find_Ancesor

$$\begin{aligned} &\exists \textit{Environment_XML_DM} \\ &\textit{ce?} : \textit{ComplexElement_Node} \\ &\textit{anc_ce!} : \mathbb{P} \textit{ComplexElement_Node} \\ &\textit{report!} = \textit{Report} \end{aligned}$$

$$\begin{aligned} &\forall \textit{schema} : \textit{SchemaGDTD}, \textit{hl} : \textit{Path_Link} \bullet \\ &\quad \textit{ce?} \notin \text{dom } \textit{schema.hl.path_link} \Rightarrow \\ &\quad \textit{report!} = \textit{Nonexistence} \\ &\quad \vee \\ &\quad \textit{anc_ce!} = (\textit{hl.hierarchical_link}^+)^{\sim} \{\{\textit{ce?}\}\} \end{aligned}$$

We define the *Find_Descendants* schema, which shows how to browse for descendants of the given complex element node using a transitive closure of *path_link*.

Find_Descendants

$$\begin{aligned} &\exists \textit{Environment_XML_DM} \\ &\textit{ce?} : \textit{ComplexElement_Node} \\ &\textit{des_ce!} : \mathbb{P} \textit{ComplexElement_Node} \\ &\textit{report!} : \textit{Report} \end{aligned}$$

$$\begin{aligned} &\forall \textit{schema} : \textit{SchemaGDTD}, \textit{hl} : \textit{Path_Link} \bullet \\ &\quad \textit{ce?} \notin \text{dom } \textit{schema.hl.path_link} \Rightarrow \\ &\quad \textit{report!} = \textit{Nonexistence} \\ &\quad \vee \\ &\quad \textit{des_ce!} = (\textit{hl.path_link}^+)^{\downarrow} \{\{\textit{ce?}\}\} \end{aligned}$$

$\textit{Query_AncestorNode} \triangleq \textit{Find_Ancestors} \wedge \textit{Success}$

$\textit{Query_Descendants} \triangleq \textit{Find_Descendants} \wedge \textit{Success}$

$\textit{Query_Related_Nodes} \triangleq \textit{Query_AncestorNode} \wedge \textit{Query_Descendants}$

As we said in Chapter 3 Section (3.6), path links are only allowed to be defined between complex element nodes. Therefore, in the following schema definition, relating to the finding of path links, we assume that all complex element nodes are connected to a path link with a unique name. The user is allowed to find a *Path_Link* corresponding to the input *Path_Link* name. We specify this schema as follows:

Find_Path_Link

\exists *Environment_XML_DM*

hl_name? : *Relation_Name*

hl_degree? : \mathbb{N}_1

hl_pc?; *hl_cc?* : $\mathbb{N} \times \mathbb{N}$

hl! : *Path_Link*

found_hl : *Path_Link* \leftrightarrow *Relation_Name*

\forall *schema*: *SchemaGDTD* | \exists *hl*: *Path_Link* •

$\text{dom } \textit{found_hl} \subseteq \textit{schema.PathLink}$

$\textit{hl.name} = \textit{hl_name?} \wedge \textit{hl.degree} = \textit{hl_degree?} \wedge \textit{hl.parent_constarint} = \textit{hl_pc?}$

$\wedge \textit{hl.child_constraint} = \textit{hl_child_constraint} \Rightarrow$

$\textit{found_hl } \textit{hl_name?} = \textit{hl!}$

This schema means that the existing *Path_Link* whose name is equal to the input name is found.

Query_Path_Link \triangleq *Find_Path_Link* \wedge *Success*

(2) Insert Operations

• Insert Complex Element Node

Insert operation allows the user to add new complex element nodes to the G-DTD. This operation is captured by the schema *Insert_New_ComplexElement_Node*

<i>Insert_New_ComplexElement_Node</i>
Δ <i>Environment_XML_DM</i> <i>level?</i> : \mathbb{N} <i>newname?</i> : <i>Element_Name</i> <i>newid?</i> : <i>ID</i> <i>new_ce!</i> : <i>ComplexElement_Node</i> <i>newlink!</i> : <i>Path_Link</i>
<i>new_ce!</i> = <i>Create_NewComplexElement_Node</i> (<i>newid?</i> , <i>newname?</i> , <i>level?</i>) \exists_1 <i>root</i> : <i>ComplexElement_Node</i> , <i>schema</i> : <i>schemaGDTD</i> • <i>Is_root</i> (<i>parent_ce</i> (<i>new_ce!</i> , <i>schema</i>)) = <i>True</i> \Rightarrow <i>new_link!</i> = <i>Create_Path_Link</i> (<i>newnode</i> , <i>root</i>) <i>Is_root</i> (<i>parent_ce</i> (<i>new_ce!</i> , <i>schema</i>)) = <i>False</i> \Rightarrow <i>new_link!</i> = <i>Create_Path_Link</i> (<i>newnode</i> , <i>parent_ce</i> (<i>new_ce!</i>))

In this schema, the declaration Δ *Environment_XML_DM* alerts the user to the fact that the schema is describing a state change of schema *Environment_XML_DM* by inserting a new node into a *schemaGDTD*. Before the insertion is made, the precondition of this operation is that the new complex element to be inserted must not already be one in the *schemaGDTD*. This is because only one unique complex element is allowed in *schemaGDTD*. To capture the condition where the complex element node is already a member of *schemaGDTD*, the following schema is used:

<i>ComplexElement_Node_AlreadyExisted</i>
\exists <i>Environment_XML_DM</i> <i>ce_name?</i> : <i>Element_Name</i> <i>ce!</i> : <i>ComplexElement_Node</i> <i>found_ce</i> : <i>Element_Name</i> \leftrightarrow <i>ComplexElement_Node</i> <i>report!</i> = <i>Report</i>
\exists <i>schema</i> : <i>schemaGDTD</i> ; <i>ce</i> : <i>ComplexElement_Node</i> • <i>ce.name</i> = <i>ce_name?</i> \Rightarrow <i>found_ce ce_name?</i> = <i>ce!</i> \wedge <i>ce!</i> \in <i>schema.Cnodes</i> <i>report!</i> = <i>Existed</i>

If this condition is satisfied, the new node is created by using a *Create_NewComplexElement* function and relationship type is created between a parent or root node using a *Create_Path_Link* function. When the operation is successful, the

schema *success* outputs a confirmatory message *inserted* meaning that the operation being performed has been successfully completed.

<i>success</i>
<i>rep!</i> : <i>Report</i>
<i>rep!</i> = <i>Inserted</i>

To perform *Do_InsertNewComplexElementNode* operation the schema *Insert_New_ComplexElement_Node* will be conjunctive with the schema *success* disjunctive with schema *ComplexElement_Node_AlreadyExisted*

$$Do_InsertNewComplexElementNode \triangleq Insert_New_ComplexElement_Node \wedge success \vee ComplexElement_Node_AlreadyExisted$$

- **Insert Simple Element Node**

Insert operation allows the user to add new simple element nodes to the G-DTD. This operation is captured by the schema *Insert_NewSimpleElement_Node*.

<i>Insert_New_SimpleElement_Node</i>
Δ <i>Environment_XML_DM</i>
<i>level?</i> : <i>N</i>
<i>newname?</i> : <i>Element_Name</i>
<i>newid?</i> : <i>ID</i>
<i>type?</i> : <i>SimpleElement_Type</i>
<i>new_se!</i> : <i>SimpleElement_Node</i>
<i>new_has_a_link!</i> : <i>Has_A_Link</i>
<i>new_se!</i> = <i>Create_NewSimpleElement_Node</i> (<i>newid?</i> , <i>newname?</i> , <i>level?</i> , <i>Type?</i>)
<i>new_has_a_link!</i> = <i>Create_Has_A_Link</i> (<i>newnode</i> , <i>Parent_se</i> (<i>newnode</i>))

Similar to *Insert_New_ComplexElement_Node*, before the insertion is made, the precondition must be satisfied that the new simple element to be inserted must not already be one in the *schemaGDTD*. To capture the condition where the simple element node is already a member of *schemaGDTD*, the following schema is used:

SimpleElement_Node_AlreadyExisted

\exists <i>Environment_XML_DM</i> <i>se_name?</i> : <i>Element_Name</i> <i>se!</i> : <i>SimpleElement_Node</i> <i>found_se</i> : <i>Element_Name</i> \rightarrow <i>SimpleElement_Node</i> <i>report!</i> = <i>Report</i>
\exists <i>schema: schemaGDTD</i> ; <i>se</i> : <i>SimpleElement_Node</i> • <i>se.name</i> = <i>se_name?</i> \Rightarrow <i>found_se se_name?</i> = <i>se!</i> \wedge <i>se!</i> \in <i>schema.Snodes</i> <i>report!</i> = <i>Existed</i>

Finally the following schema is defined to describe the state in which a node has been successfully inserted:

$$Do_InsertNewSimpleElementNode \triangleq Insert_New_SimpleElement_Node \wedge success$$

$$\vee$$

$$SimpleElement_Node_AlreadyExisted$$

- **Insert Attribute Node**

The operation to insert attribute node is captured by the schema *Insert_Attribute_Node*

Insert_Attribute_Node

Δ <i>Environment_XML_DM</i> <i>level?</i> : \mathbb{N} <i>newname?</i> : <i>Attribute_Name</i> <i>newid?</i> : <i>ID</i> <i>Type?</i> : <i>Attribute_Type</i> <i>new_att!</i> : <i>Attribute_Node</i> <i>new_part_of_link!</i> : <i>Part_of_Link</i>
<i>new_att!</i> = <i>Create_Attribute_Node</i> (<i>newid?</i> , <i>newname?</i> , <i>level?</i> , <i>Type?</i>) <i>new_part_of_link!</i> = <i>Create_Part_of_Link</i> (<i>new_att!</i> , <i>Parent_att</i> (<i>new_att!</i>))

To capture the condition where the attribute node is already a member of *schemaGDTD*, the following schema is used:

$\underline{\textit{Attribute_Node_AlreadyExisted}}$ $\exists \textit{Environment_XML_DM}$ $\textit{att_name?} : \textit{Attribute_Name}$ $\textit{att!} : \textit{Attribute_Node}$ $\textit{found_att} : \textit{Attribute_Name} \mapsto \textit{Attribute_Node}$ $\textit{report!} = \textit{Report}$ <hr/> $\exists \textit{schema}:\textit{schemaGDTD}, \textit{att}:\textit{Attribute_Node} \bullet$ $\textit{att.name} = \textit{att_name?} \Rightarrow$ $\textit{found_att} \textit{att_name?} = \textit{att!} \wedge \textit{att!} \in \textit{schema.Attnodes}$ $\textit{report!} = \textit{Existed}$
--

Finally the following schema is defined to describe the state in which an attribute node has been successfully inserted:

$$\textit{Do_Insert_Attribute_Node} \triangleq \textit{Insert_Attribute_Node} \wedge \textit{success}$$

$$\vee$$

$$\textit{Attribute_Node_AlreadyExisted}$$

(3) Delete Operations

In the following, we give schema definitions showing how to delete complex element node, simple element node, attribute node and their related links in *schemaGDTD*, on the basis of the definitions of deletion operation in Chapter 3 (section 3.6).

- **Delete Complex Element Node**

The user is allowed to delete a complex element node from *schemaGDTD*. In order to capture what happens when a complex element node is deleted, we provide the following definition.

Delete_ComplexElement_Node

Δ *Environment_XML_DM*

Query_ComplexElement_Node

Query_SimpleElement_Node

Query_Attribute_Node

Query_Related_Nodes

\forall *schema* : *SchemaGDTD*; *child_ce* : \mathbb{P} *ComplexElement_Node* •

\exists *ce*; *parent_ce* : *ComplexElement_Node*;

se : *SimpleElement_Node*;

att : *Attribute_Node*;

parent_link : *Path_Link*; *has_a* : *Has_A_Link*;

part_of : *Part_of_Link*; *schema'* : *SchemaGDTD* |

ce = *found_ce*! \wedge *se* = *found_se*! \wedge *att* = *found_attkey*! \wedge

parent_ce = *anc_ce*! \wedge *child_ce* = *des_ce*! •

parent_ce \mapsto *child_ce* \in *ran parent_link*

Create_Path_Link(*parent_ce*, *child_ce*, *parent_link*, *schema*)
= *schema'*

Delete_Path_Link(*parent_ce*, *ce*, *parent_link*, *schema*)
= *schema'*

Delete_Part_of_Link(*ce*, *se*, *part_of*, *schema*) = *schema'*

Delete_Has_A_Link(*ce*, *att*, *has_a*, *schema*) = *schema'*

schema.Cnodes' = *schema.Cnodes* \ *ce*

schema.Snodes' = *schema.Snodes* \ *se*

schema.Attnodes' = *schema.Attnodes* \ *att*

\vee

ce \notin *schema.Cnodes*

report! = *Nonexistence*

Before the node can be deleted, the particular node and related nodes must be queried and identified with a given node. The inclusion of schema *Query_ComplexElement_Node* is used to find a corresponding complex element while schema *Query_SimpleElement_Node*, and *Query_Attribute_Node* is used to find its simple element node and attribute nodes respectively. The rationale of this process is to make sure its attribute node and simple element node are automatically deleted as well. While deleting a node, the link between parent node and child nodes of the deleted node

is created using a function *Create_Path_Link*. After the complex element node is deleted, the original *Path_Link*, *Part_of_Link* and *Has_A_Link* can be removed from the *SchemaGDTD* state space without affecting other links and nodes. However, no nodes can be removed unless all links to the nodes have been removed first. The complete specification of the delete operation for complex element node is captured as follows:

Do_Delete_ComplexElement_Node \triangleq *Delete_ComplexElement_Node* \wedge *success*

(4) Replicate Operations

- Replicate Attribute Node

<p><i>Replicate_Attribute_Node</i> Δ <i>Environment_XML_DM</i> <i>Query_AttributeKey_Node</i> <i>newnode?</i> : <i>Complex_Element_Node</i> <i>replicate_att!</i> : <i>Attribute_Node</i></p> <hr/> <p>\exists <i>new_level</i> : \mathbb{N}; <i>replicate_name</i> : <i>Attribute_Name</i>; <i>new_ID</i> : <i>ID</i> <i>new_id</i> = <i>ID</i> \rightarrow \mathbb{N} <i>replicate_name</i> = <i>Get_Name</i> (<i>found_attkey!</i>) <i>new_level</i> = <i>Get_Level</i> (<i>found_attkey!</i>) + 1 • <i>replicate_att!</i> = <i>Create_Attribute_Node</i> (<i>new_id</i>, <i>replicate_name</i>, <i>new_level</i>) <i>new_part_of_link!</i> = <i>Create_Part_of_Link</i> (<i>replicate_att!</i>, <i>newnode?</i>)</p>

Do_Replicate_Attribute_Node \triangleq *Replicate_Attribute_Node* \wedge *success*

- **Replicate SimpleElement Node**

$\text{Replicate_SimpleElement_Node}$ $\Delta \text{Environment_XML_DM}$ $\text{Query_SimpleElement_Node}$ $\text{newnode?} : \text{Complex_Element_Node}$ $\text{replicate_se!} : \text{SimpleElement_Node}$ <hr/> $\exists \text{new_level} : \mathbb{N}; \text{found_se!} : \text{SimpleElement_Node};$ $\text{replicate_name} : \text{Element_Name}; \text{new_id} : \text{ID} \mid$ $\text{new_id} = \text{ID} \rightarrow \mathbb{N}$ $\text{replicate_name} = \text{Get_Name}(\text{found_se!})$ $\text{new_level} = \text{Get_Level}(\text{newnode?}) + 1 \bullet$ $\text{replicate_se!} = \text{Create_Attribute_Node}(\text{new_id}, \text{replicate_name}, \text{new_level})$ $\text{new_has_a_link!} = \text{Create_Part_of_Link}(\text{replicate_se!}, \text{newnode?})$
--

$\text{Do_Replicate_SimpleElement_Node} \triangleq \text{Replicate_SimpleElement_Node} \wedge \text{success}$

5.6 Formal Specification of G-DTD Normalizer

The second layer of XML_DM is the G-DTD Normalizer. Normalization is a step by step process to transform the *schemaGDTD* into a new structure of *schemaGDTD*. As presented in Chapter 4 (Section 4.3), our defined normal forms G-DTD (1XNF, 2XNF, 3XNF and 4XNF) are achieved by restructuring each *schemaGDTD* using normalization rules.

We briefly defined again these normal forms. If each simple element node and attribute node in the *schemaGDTD* has an atomic label node and has no repeating labels and every complex element node has its own attribute key, it is said to be in a first normal form (1XNF). In this work, it is assumed that the *schemaGDTD* satisfies this conditions. A *schemaGDTD* is in a 2XNF if there exist no n-ary one-to-many/many-to-many path_link with PFD between attribute node and simple element nodes. The 3XNF *schemaGDTD* prohibits n-ary one-to-many/many-to-many path_links with TFD among its attribute nodes and simple element nodes. And finally, *schemaGDTD* is in 4XNF, if there exists no n-ary one-to-many/many-to-many path_link with GFD among its attribute nodes and simple element nodes.

All the above processes are embedded into the G-DTD normalizer to help the user to derive well-defined XML documents. In this layer, the user can input *schemaGDTD* with the set of FDs and then it will be normalised to another level of normal form based on her/his requirements. Generally, normalizing *schemaGDTD* involves a process of creating a new complex element node, splitting a sequence of path_link and moving a simple element node. In this section, we present the conceptual operations of the G-DTD normalizer model, which consist of two main operations: first to determine types of functional dependency which to classify into different PFD, TFD and GFD types and secondly to normalize operations.

5.6.1 Determine Type Functional Dependency Operation

In this operation, a user is allowed to input a set of FDs. As a start the user is required to input a set of FDs and then it will be classified and grouped according to the FD's definition presented in Chapter 4 (Section 4.2.2). The model will classify the set of FDs as a GFD, PFD and TFD on the basis of attribute nodes and simple element nodes in LHS and RHS of FDs. Every attribute node and simple element node is associated with its level and this indicates the depth of the nodes in the *schemaGDTD*. In this operation, for the purpose of simplicity, we make an assumption that every complex element node in *schemaGDTD* has an attribute key and every node, whether attribute node, simple

element node or complex element node, has a unique name. As defined in Chapter 4 (Section 4.2.2), FDs such as GFD, TFD and PFD are in ' $\langle X, \text{level} \rangle \rightarrow \langle Y, \text{level} \rangle$ ' form where X and Y represent the LHS and RHS of the FDs respectively. A set of FD of a *schemaGDTD* is presented as two element sets, one for LHS and the other for RHS set. Obviously, the order of attributes and element nodes in such a set is important and should be maintained with care throughout the normalization process.

5.6.2 Normalization Operations

After completing the classification of FDs, the user can choose the normalization process or directly obtain the required normal forms by selecting the normalization operation. In this operation the user is allowed to transform the *schemaGDTD* into a normal form one. Normally the procedure of normalization is very tedious since it involves theory. By having this model the user just clicks the button provided then he/she can choose which normal form he/she requires 1XNF, 2XNF, 3XNF or 4XNF. This of course will save a lot of user time and effort. This process is iteratively repeated until the user is satisfied with the answer given.

(1) Normalize 1XNF to 2XNF Operation

The transformation from 1XNF to 2XNF is supported by our normalization algorithm and rules which have been presented in Chapter 4 (Section 4.4). This normalization algorithm is embedded as a backend process of the operations to help the user to achieve the best results. In order to transform the *schemaGDTD* to 2XNF, all sets of PFD need to be determined. For every PFD, the corresponding complex element node, simple element node and attribute node are classified and eliminated from the *schemaGDTD*. The detailed process of elimination of these PFD is based on normalization rules (rule 1) provided in Chapter 4 (Section 4.4.1). Generally the process of normalization involved of restructuring the structure of *schemaGDTD* into a new structure by (1) creating a new complex element node, (2) creating a new *path_link*, *part_of_link* and *has_a_link* and (3) reallocating this new complex element node and new links to a new location.

(2) Normalize 2XNF to 3XNF Operation

In this operation, the *schemaGDTD* is transformed to 3XNF by eliminating the TFD. Before the process to eliminate the TFD starts, all the given set of FDs is browsed and sorted by grouping them together according the LHS and RHS of the given FD. Each of the elements from LHS and RHS of the FD is identified to determine its parent node, related node, location and level in the *schemaGDTD*. The detailed process of elimination of these TFD is based on normalization rules (rule 2) provided in Chapter 4 (Section 4.4.1). Generally, in this operation, the *schemaGDTD* is restructured by moving up the corresponding complex element node and its associated attribute nodes and simple element nodes to another level. The attribute key node will become an attribute key for a new complex element node.

(3) Normalize 3XNF to 4XNF Operation

Similar to the above operations, the *schemaGDTD* is transformed to 4XNF by eliminating the GFD. The detailed process of elimination of these GFD is based on normalization rules (rule 3) can be found in Chapter 4 (Section 4.4.1). This process of removing GFD involves restructuring *schemaGDTD* into a new structure by (1) creating a new complex element node (2) creating a path_link, part_of link and has_a link and (3) inserting this new complex element node to be directly under a root node of *schemaGDTD*.

5.6.3 Basic Functions of G-DTD Normaliser

In this section we present formally the formal model of G-DTD normalizer on the basis of conceptual operation defined in the Section 5.4.3 and Chapter 4. In this formal approach, *Functional_Dependencies* is used to represent a set of FDs with a binary relation over sets of elements.

$Functional_Dependencies \Rightarrow F_1 Element \leftrightarrow F_1 Element$

where we construct new type *Element* from *Attribute_Node* and *SimpleElement_Node*

$$\text{Element} ::= f_at \langle \langle \text{Attribute_Node} \rangle \rangle | f_se \langle \langle \text{SimpleElement_Node} \rangle \rangle$$

using two constructor functions f_at and f_se

We define a set of $[\text{Report_status}]$ for reporting the status of SchemaGTD.

$$\text{Report_status} ::= \text{First_NormalForm} | \text{Second_NormalForm} |$$

$$\text{Third_NormalForm} | \text{Fourth_NormalForm} | \text{Not_NormalForm}$$

Before we define the normalization operations, we need to define the following functions as a prerequisite to the main operations of normalize operation. We present them as follows:

(1) Find Global Functional Dependencies (GFD)

The function $\text{Find_Global_Dependencies}$ is defined to capture the process to classify and group all GFD. This classification of GFD is on the basis of definition presented in Chapter 4 (Section 4.2.2). As defined in section 4.2.2, the FD is classified as GFD if the set of elements of the LHS and RHS of FD is located at the same level and they share the same parent node.

$$\begin{array}{l} \text{Find_Global_Dependencies} : \mathbb{F} \text{Functional_Dependencies} \\ \rightarrow \mathbb{F} \text{Functional_Dependencies} \end{array}$$

$$\begin{array}{l} \forall \text{setofFD}, \text{global_fd} : \text{Functional_Dependencies} \bullet \\ \text{Find_Global_Dependencies}(\text{setofFD}) = \text{global_fd} \Leftrightarrow \\ \exists \text{schema} : \text{SchemaGDTD}; \\ \text{simple_element} : \text{SimpleElement_Node}; \\ \text{ce1}, \text{ce2} : \text{ComplexElement_Node} \\ \text{attribute} : \text{Attribute_Node}; \\ \text{elements} : \mathbb{F} \text{Element} | \\ \text{att} = f_at \text{ attribute} \wedge \text{att} \in \text{schema.Attnodes} \\ \text{ele} = f_se \text{ simple_element} \wedge \text{ele} \in \text{schema.Snodes} \\ \text{ce2} = \text{parent_se}(\text{ele}) \wedge \text{ce2} \in \text{schema.Cnodes} \\ \text{ce1} = \text{parent_att}(\text{att}) \wedge \text{ce1} \in \text{schema.Cnodes} \\ \text{ce1} = \text{ce2} \wedge \\ \text{att.level} = \text{ele.level} \wedge \\ \text{att} \mapsto \text{ele} \in \text{global_fd} \bullet (\text{att} \mapsto \text{ele}) \end{array}$$

(2) Find Transitive Functional Dependency (TFD)

The function *Find_Transitive_Dependencies* presented below is used to classify and group all TFD. Before the classification, all the simple elements and attributes in FD are queried on the basis of the definition stated in Chapter 4 (Section 4.2.2). To be more precise, we present it formally as follows.

$$\begin{array}{l} \text{Find_Transitive_Dependencies} : \mathbb{F}\text{Functional_Dependencies} \\ \quad \rightarrow \mathbb{F}\text{Functional_Dependencies} \\ \hline \forall \text{setofFD}, \text{transitive_fd} : \mathbb{F}\text{Functional_Dependencies} \bullet \\ \quad \text{Find_Transitive_Dependencies}(\text{setofFD}) = \text{transitive_fd} \Leftrightarrow \\ \quad \exists \text{schema} : \text{SchemaGDTD}; \\ \quad \text{simple_element} : \text{SimpleElement_Node}; \\ \quad \text{attribute} : \text{Attribute_Node}; \\ \quad \text{att1}, \text{att2}, \text{ele} : \mathbb{P}\text{Element}; \\ \quad \text{ce1}, \text{ce2} : \text{ComplexElement_Node} \mid \\ \quad \quad \text{att1} = f_at \text{ attribute} \wedge \text{att1} \in \text{schema.Attnodes} \\ \quad \quad \text{ele} = f_se \text{ simple_element} \wedge \text{ele} \in \text{schema.Snodes} \\ \quad \quad \text{att2} = f_at \text{ attribute} \wedge \text{att2} \in \text{schema.Attnodes} \\ \quad \quad \text{att1.level} < \text{att2.level} \wedge \\ \quad \quad \text{att2.level} < \text{ele.level} \wedge \\ \quad \quad \text{parent_att att1} \neq \text{parent_at att2} \neq \text{parent_se ele} \wedge \\ \quad \quad \{\text{att1} \mapsto \text{att2}, \text{att2} \mapsto \text{ele}\} \in \text{setofFD} \wedge \\ \quad \quad \text{att1} \mapsto \text{ele} \in \text{transitive_fd} \bullet (\text{att1} \mapsto \text{ele}) \end{array}$$

(3) Find Partial Functional Dependency (PFD)

The function *Find_Partial_Dependencies* defined below is used to find all possible PFD that exist in a set of FDs. This process is formally defined as follows:

$$\begin{array}{l}
 \text{Find_Partial_Dependencies} : \mathbb{F}\text{Functional_Dependencies} \\
 \qquad \qquad \qquad \qquad \qquad \qquad \rightarrow \mathbb{F}\text{Functional_Dependencies} \\
 \hline
 \forall \text{setofFD}, \text{partial_fd} : \text{Functional_Dependencies} \bullet \\
 \text{Find_Partial_Dependencies}(\text{setofFD}) = \text{partial_fd} \Leftrightarrow \\
 \quad \exists \text{schema} : \text{SchemaGDTD}; \\
 \quad \text{simple_element} : \text{SimpleElement_Node}; \\
 \quad \text{attribute} : \text{Attribute_Node}; \\
 \quad \text{elements}, \text{att1}, \text{att2}, \text{ele} : \mathbb{F}\text{Element}; \\
 \quad \text{ce1}, \text{ce2} : \text{ComplexElement_Node} \mid \\
 \quad \text{att1} = f_at \text{ attribute} \wedge \text{att1} \in \text{schema.Attnodes} \\
 \quad \text{att2} = f_at \text{ attribute} \wedge \text{att2} \in \text{schema.Attnodes} \\
 \quad \text{ele} = f_se \text{ simple_element} \wedge \text{ele} \in \text{schema.Snodes} \\
 \quad \text{att1.level} < \text{att2.level} \wedge \text{att2.level} = \text{se.level} \\
 \quad \text{parent_att att1} \neq \text{parent_att att2} \wedge \\
 \quad (\{\text{att1}, \text{att2}\} \mapsto \text{ele}, \text{att2} \mapsto \text{ele}) \in \text{setofFD} \wedge \\
 \quad \text{att2} \mapsto \text{ele} \in \text{partial_fd} \bullet (\text{att2} \mapsto \text{ele})
 \end{array}$$

(4) Get elements of Functional Dependencies

The function *getFD_elements* presented below is used to retrieve all elements in LHS and RHS of FDs

$$\begin{array}{l}
 \text{getFD_elements} : \text{Functional_Dependencies} \rightarrow \mathbb{F} \text{Element} \\
 \hline
 \forall \text{fds} : \text{Functional_Dependencies} \bullet \\
 \quad \text{getFD_elements}(\text{fds}) = \cup \{fd : \text{fds} \bullet \text{dom } fd \cup \text{ran } fd\}
 \end{array}$$

(5) Checking the Status of Schema GDTD

The current status of schemaGDTD can be identified using four different functions: Is_1XNF , Is_2XNF , Is_3XNF , Is_4XNF . These rules have been defined in Chapter 4 (Section 4.3). We formally capture each of rules as follows respectively:

- Check 1XNF

$Is_1XNF : (SchemaGDTD) \rightarrow Boolean$

$\forall schema : SchemaGDTD \bullet$

$Is_1XNF(schema) = true \Leftrightarrow$

$\exists_1 root : ComplexElement_Node \bullet root.level = 0$

$\forall ce1, ce2 : Cnodes \mid ce1 \neq ce2 \bullet ce1.name \neq ce2.name$

$\forall se1, se2 : Snodes \mid se1 \neq se2 \bullet se1.name \neq se2.name$

$\forall att1, att2 : Attnodes \mid att1 \neq att2 \bullet att1.name \neq att2.name$

$\forall hlink : Path_Link \bullet$

$dom\ hlink.path_link \in Cnodes \wedge$

$ran\ hlink.path_link \in Cnodes \wedge$

$root \notin ran\ hlink.path_link$

$\forall part_of_link : Part_of_Link \bullet$

$dom\ part_of_link \in Cnodes \wedge$

$ran\ part_of_link.AttributeKey \in Attnodes \wedge$

$root \notin dom\ part_of_link$

$\forall has_link : Has_A_Link \bullet$

$dom\ has_link.hasa \in Cnodes \wedge$

$ran\ has_link.hasa \in Snodes \wedge$

$root \notin dom\ has_link.hasa$

- Check 2XNF

$$\begin{aligned}
 & \underline{Is_2XNF : (SchemaGDTD \times \mathbb{F}Functional_Dependencies) \rightarrow Boolean} \\
 & \forall schema : SchemaGDTD; givenFD, partial_fd : Functional_Dependencies \bullet \\
 & \quad Is_2XNF (schema, givenFD) = true \Leftrightarrow \\
 & \quad \quad Is_1XNF (schema) = true \wedge \\
 & \quad \quad \neg (\forall hlink_1, hlink_2 : seq Path_Link \bullet \\
 & \quad \quad \quad Is_Rel_Many_to_One (hlink_1.1, schema) = True \wedge \\
 & \quad \quad \quad Is_Rel_Many_to_Many (hlink_2.2, schema) = True \wedge \\
 & \quad \quad \quad \exists_1 (hlink_n : seq Path_Link \mid lastlink = last hlink_n \bullet \\
 & \quad \quad \quad \quad Is_Rel_One_to_Many (lastlink, schema) = True)) \wedge \\
 & \quad \quad (\forall givenFD : Functional_Dependencies \bullet \\
 & \quad \quad \quad Find_Partial_Dependencies (givenFD) = partial_fd)
 \end{aligned}$$

- Check 3XNF

$$\begin{aligned}
 & \underline{Is_3XNF : (SchemaGDTD \times \mathbb{F}Functional_Dependencies) \rightarrow Boolean} \\
 & \forall schema : SchemaGDTD; \\
 & \quad givenFD, transitive_fd : Functional_Dependencies \bullet \\
 & \quad Is_2XNF (schema, givenFD) = true \Leftrightarrow \\
 & \quad \quad Is_2XNF (schema) = true \wedge \\
 & \quad \quad \neg (\forall hlink_1, hlink_2 : seq Path_Link \bullet \\
 & \quad \quad \quad Is_Rel_Many_to_One (hlink_1.1, schema) = True \wedge \\
 & \quad \quad \quad Is_Rel_Many_to_Many (hlink_2.2, schema) = True \wedge \\
 & \quad \quad \quad \exists_1 (hlink_n : seq Path_Link \mid lastlink = last hlink_n \bullet \\
 & \quad \quad \quad \quad Is_Rel_One_to_Many (lastlink, schema) = True)) \wedge \\
 & \quad \quad (\forall givenFD : Functional_Dependencies \bullet \\
 & \quad \quad \quad Find_Transitive_Dependencies (givenFD) = transitive_fd)
 \end{aligned}$$

- Check 4XNF

$$Is_4XNF : (SchemaGDTD \times \mathbb{F}Functional_Dependencies) \rightarrow Boolean$$

$$\forall schema : SchemaGDTD;$$

$$givenFD, partial_fd : Functional_Dependencies \bullet$$

$$Is_4XNF (schema, givenFD) = true \Leftrightarrow$$

$$Is_3XNF (schema) = true \wedge$$

$$\neg (\forall hlink_1, hlink_2 : seq Path_Link \bullet$$

$$Is_Rel_Many_to_One(hlink_1.1, schema) = True \wedge$$

$$Is_Rel_Many_to_Many(hlink_2.2, schema) = True \wedge$$

$$\exists_1 (hlink_n : seq Path_Link \mid lastlink = last hlink_n \bullet$$

$$Is_Rel_One_to_Many(lastlink, schema) = True)) \wedge$$

$$(\forall givenFD : Functional_Dependencies \bullet$$

$$Find_Global_Dependencies(givenFD) = global_fd$$

5.6.4 Restructure IXNF Schema Operations

To normalize 1XNF to 2XNF G-DTD, the *Restructure_IXNF_schemaGDTD* schema is defined to represent the process. The input of the operation is the *schemaGDTD* and set of FDs and the output of this operation is a new *schemaGDTD* in a 2XNF form. This schema uses the *Find_PathLink_with_Dependency* operation as a precondition to test whether path links of one-to-many/many-to-many or many-to-one relationship types exist in *SchemaGDTD*.

$$Find_PathLink_with_Dependency$$

$$\exists Environment_XML_DM$$

$$found_dependency! = Report$$

$$\forall hlink : Path_Link \mid$$

$$hlink \in current_schema?.PathLink \wedge$$

$$\neq hlink.degree \geq 2 \wedge$$

$$Is_Rel_Many_to_Many(hlink, Current_schema?) = true \vee$$

$$Is_Rel_One_to_Many(hlink, Current_schema?) = true \vee$$

$$Is_Rel_Many_to_One(hlink, Current_schema?) = true \bullet$$

$$found_dependency! = Existence$$

If the precondition is satisfied, all the PFDs will be extracted from the given set of FDs using a *Find_Partial_Dependency* function. In the *Restructure_LXNF_schemaGDTD* schema, for each of PFD, the associated parent for both LHS and RHS elements of the PFD is determined using *parent_att* and *parent_se* functions. When the parent node, which is a complex element type, is determined, a new complex element node is created and inserted using the operation *Insert_New_ComplexElement_Node*. Both *Insert_SimpleElement_Node* and *Insert_Attribute_Node* schemas are used to create a new simple element node and attribute node for the new complex element node. The function *Delete_Has_A_Link* is applied to delete all Has_A links between the former simple element node and its parent node. Finally the new *schemaGDTD* without the PFD is generated. The new schema in 2XNF form contains only the updated nodes and links.

Restructure_IXNF_schemaGDTD

Δ *Environment_XML_DM*

Find_PathLink_with_Dependency

Do_Insert_New_ComplexElement_Node

Do_Replicate_Attribute_Node

Do_Replicate_SimpleElement_Node

current_schema? : *schemaGDTD*

found_partial_fd!, *new_fd!*, *current_fd?* : \mathbb{F} *Functional_Dependencies*

result! : *Report_status*

check_dependency! : *Report*

$Is_1XNF(current_schema?) = true$

$Is_2XNF(current_schema?, current_fd?) \neq true$

$check_dependency! = found_dependency!$

$found_partial_fd! = Find_Partial_Dependencies(current_fd?) \wedge$

$\forall ce:ComplexElement_Node; se:SimpleElement_Node |$

$\exists elements:Elements; created_att:Attribute_Node;$

$created_se:SimpleElement_Node \bullet$

$elements = getFD_element(found_partial_fd!) \wedge$

$att \in ran\ elements \wedge se \in ran\ elements \wedge$

$parent_att\ att = ce \wedge parent_se\ se = ce \bullet$

$new_fd! = current_fd? \setminus found_partial_fd! \wedge$

$\forall schema': schemaGDTD \bullet$

$\exists created_ce:ComplexElement_Node; created_att:Attribute_Node;$

$created_se:SimpleElement_Node; new_link:Path_Link;$

$new_partlink:Part_of_Link; new_haslink:Has_A_Link \bullet$

$schema' = Delete_Has_A_Link(ce, se, part_of, current_schema?)$

$created_ce = new_ce! \wedge created_att = new_att! \wedge$

$created_se = new_se! \wedge new_link = newlink!$

$new_partlink = new_part_link!$

$new_haslink = new_has_link! \wedge$

$schema'.Attnodes = schema.Attnodes \cup created_att \wedge$

$schema'.Snodes = schema.Snode \cup created_se \wedge$

$schema'.Cnodes = schema.Cnodes \cup created_ce \wedge$

$schema'.PathLink = schema.PathLink \cup new_link \wedge$

$schema'.Partof = schema.Partof \cup new_partlink \wedge$

$schema'.HasA = schema.HasA \cup new_haslink \wedge$

$result! = Second_NormalForm$

Second_NormalForm_GDTD \triangleq *Restructure_1XNF_schemaGDTD* \vee

Not_NormalForm

5.6.5 Restructure 2XNF Schema Operations

To normalize 2XNF to 3XNF G-DTD, the schema *Restructure_2XNF_SchemaGDTD* is used to capture the process. The input for the schema is set of FDs and *schemaGDTD* while the result reports the status of the new schema. The precondition for this operation is to test whether the path links types existing in *SchemaGDTD* have one-to-many/many-to-many or many-to-one relationship. If the precondition is satisfied, all the PFD will be extracted from the given set of FDs and then eliminated. This process is captured by the following schema.

<p><i>Find_Path_Link_with_TFD</i></p> <hr/> <p>Δ <i>Environment_XML_DM</i> <i>new_fd?</i>, <i>current_fd?</i> : <i>FFunctional_Dependencies</i> <i>current_schema?</i> : <i>SchemaGDTD</i></p> <hr/> <p>\forall <i>hlink</i>: <i>Path_Link</i> <i>hlink</i> \in <i>PathLink</i> \wedge \neq <i>hlink.degree</i> \geq 2 \wedge <i>Is_Rel_Many_to_Many</i> (<i>hlink</i>, <i>current_schema?</i>) = true \vee <i>Is_Rel_One_to_Many</i> (<i>hlink</i>, <i>current_schema?</i>) = true \vee <i>Is_Rel_Many_to_One</i> (<i>hlink</i>, <i>current_schema?</i>) = true \wedge <i>transitive_fd</i> = <i>Find_Transitive_Dependencies</i> (<i>current_fd?</i>) • <i>new_fd</i> \neq <i>current_fd?</i> \setminus <i>transitive_fd</i></p>
--

Schema *Restructure_2XNF_SchemaGDTD* presents the operation to eliminate PFD from *SchemaGTD*. In this schema, three operations are included, *Query_Complex_Element_Node*, *Find_Simple_Element_Node* and *Find_AttributeNode* to determine the location of complex element, simple element and attribute node in schema G-DTD. Each path link type is identified to confirm that all the related complex element nodes must be a member of a set of complex element node in *schemaGDTD*. The new created complex element node can be linked to either a root node or another complex element

node. Once TFD(s) have been removed from the *SchemaGDTD*, the operation will generate a 2XNF result. Finally the total specification to normalize GDTD to 2XNF is given as follows.

<p><i>Restructure_2XNF_SchemaGDTD</i></p> <p>Δ <i>Environment_XML_DM</i></p> <p><i>Find_Path_Link_with_TFD</i></p> <p><i>Query_ComplexElement_Node</i></p> <p><i>Query_SimpleElement_Node</i></p> <p><i>Query_Attribute_Node</i></p> <p><i>current_schema?</i>, <i>new_schema!</i> : <i>schemaGDTD</i></p> <p><i>result!</i> : <i>Report_Status</i></p> <hr/> <p>$(\forall ce1, ce2, created_ce : ComplexElement_Node; hlink : Path_Link$ $ce1, ce2 \in current_schema?.CNodes \wedge$ $hlink \in current_schema?.PathLink$ $ce1 \mapsto ce2 \in hlink.path_link$ $ce2.level - ce1.level = 1 \Rightarrow ce2.level = ce1.level$ $\exists existed_att : Attribute_Node ;$ $existed_se : SimpleElement_Node ; schema' : schemaGTD$ $existed_att = found_att! \wedge existed_se = found_se! \bullet$ $existed_att.level \wedge existed_se.level = ce2.level + 1$ $(\exists root : ComplexElement_Node; new_link : Path_Link$ $root.level = 0 \wedge$ $Is_Root(parent_ce(ce1), current_Schema?) = true$ $\Rightarrow new_link = Create_Path_Link(ce2, root)$ $Is_Root(parent_ce(ce1), current_Schema?) = false$ $\Rightarrow new_link = Create_Path_Link(ce2, parent_ce(ce1))$ $current_schema?'.Attnodes = current_schema?.Attnodes \cup existed_att$ $current_schema?'.Snodes = current_schema?.Snodes \cup existed_se$ $current_schema?'.PathLink =$ $current_schema?.PathLink \cup new_link \bullet$ $result! = Third_NormalForm$</p>
--

Third_NormalForm_GDTD \triangleq *Restructure_2XNF_SchemaGDTD* \vee *Not_NormalForm*

5.6.6 Restructure 3XNF Schema Operations

To transform from 3XNF to 4XNF G-DTD, the schema *Restucture_3XNF_SchemaGDTD* is defined to represent the process. This schema include schema *Find_Path_Link_with_GFD* as defined in the following section.

Find_Path_Link_with_GFD
 Δ *Environment_XML_DM*
current_schema? : *SchemaGTD*
new_fd!, *current_fd?* : \mathbb{P} *Functional_Dependencies*

\forall *hlink* : *Path_Link* |
hlink \in *current_schema?*.*PathLink* \wedge
 \neq *hlink.degree* \geq 2 \wedge
Is_Rel_Many_to_Many (*hlink*, *current_schema?*) = true \vee
Is_Rel_One_to_Many (*hlink*, *current_schema?*) = true \vee
Is_Rel_Many_to_One (*hlink*, *current_schema?*) = true \wedge
global_fd = *Find_Global_Dependencies* (*current_fd!*) •
newfd! = *current_fd!* \ *global_fd*

The schema *Find_Path_Link_with_GFD* is used to capture an operation to eliminate path links of n-ary many-to-many or one-to-many type with GFD. This operation is similar to *Find_Path_Link_with_PFD* except the new created complex element node is inserted directly under a root node.

The schema *Restucture_3XNF_SchemaGDTD* has a set of GFD and *SchemaGDTD* while the result will be a new schema and a report in fourth normal form G-DTD. Like the previous schema, the same preconditions are tested and GFD will be removed from the set of FDs.

Restructure_3XNF_SchemaGDTD

Δ *Environment_XML_DM*

Find_Path_Link_with_GFD

Query_ComplexElement_Node

Insert_New_ComplexElement_Node

Insert_SimpleElement_Node

Insert_Attribute_Node

current_schema?, *new_schema!* : *schemaGTD*

global_FD? : *Functional_Dependencies*

result! : *Report_Status*

($\forall ce1, ce2, created_ce$: *ComplexElement_Node*; *hlink* : *Path_Link* ;
schema : *schemaGDTD* |

$hlink \in current_schema?.PathLink$

$ce1, ce2 \in current_schema?.CNodes \wedge$

$ce1 \mapsto ce2 \in hlink.path_link \bullet$

($\exists new_link$: *Path_Link* ; *created_att* : *Attribute_Node* ;

created_se : *SimpleElement_Node* ; *schema'* : *schemaGDTD* |

$created_ce = new_ce! \wedge created_att = new_att! \wedge$

$created_se = new_se!$

$new_link = newlink!$

$new_part_of = new_part_link!$

$new_has_link = new_has_link! \bullet$

($\exists elements$: *Elements* | $elements = getFD_element(global_FD?)$

$att \in \text{ran } element \wedge se \in \text{ran } elements$

$parent_att\ att = ce2 \wedge$

$parent_se\ se = ce2 \wedge$

$Delete_Has_A_Link(ce2, se, part_of, current_schema?) = schema'$

$schema'.Attnodes = schema.Attnodes \cup created_att \wedge$

$schema'.Snodes = schema.Snodes \cup created_se \wedge$

$schema'.Cnodes = schema.Cnodes \cup created_ce \wedge$

$schema'.PathLink = schema.PathLink \cup new_link \wedge$

$schema'.Partof = schema.Partof \cup new_part_of \wedge$

$schema'.HasA = schema.HasA \cup new_has_link))) \bullet$

$new_schema! = schema'$

$result! = Fouth_NormalForm$

We define the following schema, which reports the status of G-DTD as being in not normal form

<i>Not_Normal_Form</i>
<i>status!: Report_Status</i>
<i>status! = not_normalForm</i>

The total specification of the 4XNF is given as follows

Fourth_NormalForm_GDTD \triangleq *Restucture_3XNF_SchemaGDTD* \vee *Not_NormalForm*

5.7 Formal Specification of G-DTD Translator

Once the final *schemaGDTD* is derived it will then map back to a DTD format. The user can design a redundancy-free XML document on the basis of this new DTD format. The algorithm for this mapping purpose has been presented in Chapter 3 (Section 3.7). In this algorithm we provide a depth first traversal method for each element in *schemaGDTD* map to DTD format. We use flat mapping in this approach since it is simpler. The mapping process uses one-to-one mapping between nodes in *schemaGDTD* with building blocks in DTD. The main building blocks of DTD are element and attribute, defined by the tags `<!ELEMENT>` and `<!ATTLIST>` respectively. Keywords PCDATA and CDATA are used as string types for element and attribute respectively. We capture this process using the following definition.

The given set [*DTD*] and following free type definitions:

Operators::= + |? |*

Tag ::= DOCTYPE | ELEMENT | ATTLIST

keyword::= ID | REQUIRED | PCDATA | CDATA | EMPTY

We define the following function to map *directly* each complex element node, simple element node and attribute node of schemaGDTD to DTD structure. Function *ce_tag* is used to map complex element node to !ELEMENT tag while function *att_tag* is used to map attribute_node with !ATTLIST ID REQUIRED and function *se_tag* is used to map simpleelement_node with !ATTLIST PCDATA or EMPTY. Function *op* is to map all *types of operators* used by depending on the type of *semantic* relationship between the complex element nodes.

ce_tag : ComplexElement_Node → Tag
se_tag : SimpleElement_Node → Tag
att_tag : Attribute_Node → Tag
op : ComplexElement_Node → Operators

($\forall ce: \text{ComplexElement_Node}, att: \text{Attribute_Node}, se: \text{SimpleElement_Node} \mid$
 $ce \in \text{schema.Cnodes}$
 $att \in \text{schema.Attnodes}$
 $se \in \text{schema.Snodes} \bullet$

($Is_root\ ce$) $\Rightarrow ce_tag\ ce = DOCTYPE$

\vee

$\neg(Is_root\ ce) \Rightarrow ce_tag\ ce = ELEMENT$

$att_tag\ att = ATTLIST$

($\forall hl: \text{path_link}, ce1, ce2: \text{ComplexElement_Node} \mid$

$ce1 \mapsto ce2 \in hl \bullet$

$\forall operators: \{$

$op\ (ce2) = * \Leftrightarrow Is_one_to_many(hl) \vee hl.degree \geq 2$

$\vee op\ (ce2) = ? \Leftrightarrow Is_zero_to_one(hl) \wedge hl.degree \geq 2$

$\vee op\ (ce2) = + \Leftrightarrow Is_zero_to_many(hl) \wedge hl.degree \geq 2)$

$\text{Map} : \text{SchemaGDTD} \rightarrow \text{DTD}$ <hr/> $\exists \text{schema} : \text{SchemaGDTD}, D : \text{DTD} \bullet$ $\text{map schema} = D \Leftrightarrow$ $(\forall \text{ce} : \text{ComplexElement_Node}, \text{att} : \text{Attribute_Node}, \text{se} : \text{SimpleElement_Node}$ $\text{hl} : \text{path_link}, \text{has_a} : \text{Has_a_link}, \text{part_of} : \text{part_of} $ $\text{ce} \in \text{Cnodes}$ $\text{att} \in \text{AttNodes}$ $\text{se} \in \text{Snodes}$ $\text{hl} \in \text{PathLink} \bullet$ $\text{ce_tag}(\text{ce}) \vee$ $\text{ce} \mapsto \text{parent_ce} \sim (\text{ce}) \in \text{hl} \wedge \text{op}(\text{parent_ce} \sim (\text{ce}))$ $\text{ce} \mapsto \text{se} \in \text{has_a} \wedge \text{se_tag}(\text{se})$ $\text{ce} \mapsto \text{att} \in \text{part_of} \wedge \text{att_tag}(\text{att}))$

Finally the process to map the shcemaGDTD to a new DTD is captured by the schema *GDTD_Translator*. The input fot schema is *schemaDGTD* and the output is *DTD*.

In this schema the function *map* is used to map each node in *SchemaGDTD* to DTD structure.

GDTD_Translator <hr/> $\Delta \text{Environment_XML_DM}$ $\text{schema?} : \text{SchemaGDTD}$ $\text{output!} : \text{DTD}$ <hr/> $\text{output!} = \text{map}(\text{schema?})$

$$\text{DoGDTD_Translator} \triangleq \text{GDTD_Translator} \wedge \text{success}$$

5.8 Summary

In this chapter, we have constructed a formal specification of the XML document design prototype. The purpose of this specification is to help the user to understand the whole picture of the proposed prototype system. More importantly, by using Z, we can define precisely the data structure, semantic constraints and operations of the XML document

design system. In particular, the important components of the system, which are the conceptual model G-DTD and G-DTD normalizer, have been precisely formalized to capture both the structure of the XML document and the normalization process. It is observed that application of formal specification in XML normalization has increased the correctness of the system at the abstract level and gives more confidence to automate the process of XML document normalization.

Chapter 6

Specification Testing – A Case Study

6.1 Introduction

In this chapter we will test the specification that was constructed in Chapter 5 with the specific case study to check

“Does the specification describe for the properties of XML document design represent what the users want/are expecting to get?”

Because Z specifications are formal descriptions based on predicate logic, we can test them in ways that enable us to prove that they satisfy certain fundamental criteria in respect to the question above. This will increase the confidence in the implementation later.

In this chapter we are checking the properties of XML document design, in particular the G-DTD normaliser layer, to seek for the consistency of the operations defined in Chapter 4 with the specification defined in Chapter 5. With respect to the above question we must ensure that the specification presented in Chapter 5, specifically the G-DTD normaliser specification can be used to derive a required normal form design using the set of normal forms and normalization rules presented in Chapter 4. Even though we have illustrated this concept informally through the case study presented in Chapter 4 Section 4.5, in this chapter we formally demonstrate again the properties of XML document design through Z specification to show the validity of the specification. In this case study, the G-DTD normaliser layer is chosen as an example because it contains the important properties of XML document design such as 1XNF, 2XNF, 3XNF and 4XNF designs. To show the consistency and correctness of the specification of the G-DTD normaliser constructed in Chapter 5 (Section 5.6), we reapply the same case study as in Chapter 4 (Section 4.5) throughout this chapter.

The rest of the chapter is organized as follows. Section 6.2 presents an example of a G-DTD diagram in a Z specification. Section 6.3 presents the consistency of the operations in G-DTD normaliser by formally demonstrating the normalization steps to transform from INXF until to 4XNF design using Z specification. Finally, a summary of the work is presented in section 6.4.

6.2 Representing G-DTD Diagram in a Z Specification

We illustrate how a G-DTD schema diagram of the university database in Figure 4.2 can be represented using the Z specification as defined in Chapter 5 (Section 5.5). In a Z specification, a G-DTD schema is represented as a *SchemaGDTD*. The *SchemaGDTD* defines a G-DTD schema in sets and relations expressions with predicates representing associated constraints. The *SchemaGDTD* contains a *root*, *Cnodes*, *Snodes* and *Attnodes* which represent a root, a set of complex element nodes, a set of simple element nodes and a set of attribute nodes respectively. The sets of relations named *Partof*, *HasA* and *PathLink* are presented as relations to represent a set of ordered pairs between a set of complex element nodes and a set of attribute nodes, a set of ordered pairs between a set of complex element nodes and a set of simple element nodes and a set of ordered pairs between a set of complex element nodes and a set of complex element nodes respectively. In this specification, *PathLink* is represented as a sequence of a set of relation between complex element nodes and complex element nodes. Figure 6.1 presents the *SchemaGDTD* for the G-DTD diagram represented in a Z specification. As shown in Figure 6.1, a root which is $(1, department, 0)$ is a special case of a complex elements node which is always located at level 0. The set *Cnodes* contains four complex element nodes: $\{(1, department, 0), (2, course, 1), (5, student, 2), (9, lecturer, 3)\}$. The set of *Snodes* contains four simple element nodes: $\{(4, titles, 2), (7, fname, 3), (8, lname, 3), (11, name, 4)\}$ and the set of *Attnodes* consists of three attribute nodes: $\{(3, cno, 2), (6, sno, 3), (10, tno, 4)\}$.

The *Partof* relation is a type of total and injective function because each complex element node in *Cnodes* except the root node $(1, Department, 0)$ has a unique attribute node. The range of the *Partof* relation is a set of attribute nodes: *Attnodes*, for instance,

$\{(2, \text{course}, 1) \mapsto (3, \text{cno}, 2), (5, \text{student}, 2) \mapsto (6, \text{sno}, 3), (9, \text{lecturer}, 3) \mapsto (10, \text{tno}, 4)\}$.
 The *HasA* relation has its domain as a set of complex element nodes and its range as a set of simple element nodes; *Snodes*, for instance: $\{(2, \text{course}, 1) \mapsto (4, \text{titles}, 2), (5, \text{student}, 2) \mapsto \{(7, \text{fname}, 3), (8, \text{lname}, 3)\}, (9, \text{lecturer}, 3) \mapsto (11, \text{name}, 4)\}$. For the *HierarchicalLink*, both its domain and range belong to a set of complex element nodes: *Cnodes*, for instance $\langle \{(1, \text{department}, 0) \mapsto (2, \text{course}, 1), 2, 1..n, 1..1), ((2, \text{course}, 1) \mapsto (5, \text{student}, 2), 2, 1..n, 1..n), ((5, \text{Student}, 2) \mapsto (9, \text{lecturer}, 3)) \rangle$

Root = $\{(1, \text{department}, 0)\}$
Cnodes = $\{(1, \text{department}, 0), (2, \text{course}, 1), (5, \text{student}, 2), (9, \text{lecturer}, 3)\}$
Snodes = $\{(4, \text{titles}, 2), \{(7, \text{fname}, 3), (8, \text{lname}, 3)\}, (11, \text{name}, 4)\}$
Attnodes = $\{(3, \text{cno}, 2), (6, \text{sno}, 3), (10, \text{tno}, 4)\}$
Partof = $\{(2, \text{course}, 1) \mapsto (3, \text{cno}, 2), (5, \text{student}, 2) \mapsto (6, \text{sno}, 3),$
 $(9, \text{lecturer}, 3) \mapsto (10, \text{tno}, 4)\}$
HasA = $\{(2, \text{course}, 1) \mapsto (4, \text{titles}, 2),$
 $(5, \text{student}, 2) \mapsto \{(7, \text{fname}, 3), (8, \text{lname}, 3)\},$
 $(9, \text{lecturer}, 3) \mapsto (11, \text{name}, 4)\}$
PathLink = $\langle \{(1, \text{department}, 0) \mapsto (2, \text{course}, 1), 2, 1..n, 1..1),$
 $((2, \text{course}, 1) \mapsto (5, \text{student}, 2), 2, 1..n, 1..n),$
 $((5, \text{student}, 2) \mapsto (9, \text{lecturer}, 3), 2, 1..1, 1..n) \rangle$

Figure 6.1: *SchemaGDTD*

$$\begin{aligned}
KD: & \{(3, cno, 2) \rightarrow (2, Course, 1), \\
& (6, sno, 3) \rightarrow (5, Student, 2), \\
& (10, tno, 4) \rightarrow (9, lecturer, 3)\} \\
GFD = & \{(6, sno, 3) \rightarrow \{(7, fname, 3), (8, lname, 3)\}, \\
& (10, tno, 4) \rightarrow (11, tname, 4)\} \\
TFD = & (3, cno, 2) \rightarrow (10, tno, 4) \\
& (10, tno, 4) \rightarrow (11, tname, 4) \\
& (3, cno, 2) \rightarrow (11, tname, 4) \\
PFD = & \{(3, cno, 2), (10, tno, 4)\} \rightarrow (11, tname, 4), \\
& (10, tno, 4)\} \rightarrow (11, tname, 4)\}
\end{aligned}$$

Figure 6.2: Set of KD and FDs

As shown in Figure 6.2, FD is defined as a set of homogenous relations between a set of attributes/simple elements and a set of attributes/simple elements

6.3 Consistency of the Operations in G-DTD Normalizer

In this section, we present formally the G-DTD normaliser operations by demonstrating how *SchemaGDTD* is transformed from IXNF, 2XNF, 3XNF and 4XNF design on the basis of a set of normal forms and normalization rules presented in Chapter 5 (Section 5.6).

6.3.2 Normalize 1XNF to 2XNF

Given *SchemaGDTD* with a set of KD and FD as shown in Figures 6.1 and 6.2 respectively, G-DTD normaliser will check the status of *SchemaGDTD* using the function *Is_2XNF*. The *SchemaGDTD* is not in the 2XNF form since it does not satisfy some of the predicates listed in the function *Is_2XNF*. This can be proved as follows:

Function *Is_2XNF* contains three predicates:

$$(1) \text{Is_1XNF}(\text{schema}) = \text{true}$$

The first predicate is true since the *SchemaGDTD* has satisfied all the predicates as shown in section 6.3.1

$$(2) \neg(\forall \text{hlink}_1, \text{hlink}_2: \text{seq Path_Link} |$$

$$\text{Is_Rel_Many_to_One}(\text{hlink}_1.1, \text{schema}) = \text{True} \wedge$$

$$\text{Is_Rel_Many_to_Many}(\text{hlink}_2.2, \text{schema}) = \text{True} \wedge$$

$$\exists_1 (\text{hlink}_n: \text{seq Path_Link} | \text{lastlink} = \text{last hlink}_n \bullet$$

$$\text{Is_Rel_One_to_Many}(\text{lastlink}, \text{schema}) = \text{True}))$$

$$\neg (\text{True} \wedge \text{True} \wedge \text{True}) \text{ Negation of these predicates returns a false value}$$

In the second predicate, each *Path_Link* is identified in a sequence starting from the first position until the last position of the *Hiearchical_Link*. Function *Is_Rel_Many_to_One* is used to check if $(1, \text{department}, 0) \mapsto (2, \text{course}, 1), 2, 1..n, 1..1)$ is a many_to_one relationship. This returns a true value since it satisfies the following condition:

$$(\text{first rel. pc} \geq 1 \wedge \text{second rel. pc} \geq 2 \wedge \text{first rel. cc} = 1 \wedge \text{second rel. cc} = 1)$$

Function *Is_Rel_Many_to_Many*, returns a true value for the *Path_Link* $((2, \text{Course}, 1) \mapsto (5, \text{Student}, 2), 2, 1..n, 1..n)$ since it satisfies the following condition:

$(first\ rel.pc \geq 1 \wedge rel.second\ pc \geq 2 \wedge first\ rel.cc \geq 1 \wedge second\ rel.cc \geq 2)$

and Function *Is_Rel_One_to_Many* returns a true value for the Path_Link ((5,Student, 2) \mapsto (9, lecturer, 3), 2, 1..1, 1..n) since it satisfies the following condition:

$first\ rel.pc \geq 1 \wedge second\ rel.pc \geq 1 \wedge first\ rel.cc = 1 \wedge second\ rel.cc = 2)$

Note : *pc* and *cc* refer to parent constraint and child constraint respectively.

(3) $\neg (\forall givenFD : Funcional_Dependencies \bullet$

$\{ \exists pfd : partial_FD \mid pfd = Find_Partial_Dependencies (givenFD) \wedge pfd \in partial_dependency \} \bullet pfd)$

The third predicate indicates that the function *Find_Partial_Dependencies* returns a true value since PFD (10, tno, 4) \rightarrow (11, name, 4) is found in the set of *givenFD* as it satisfies the following predicate:

$(\forall fds : given\ set\ FD :$

$\exists fd1:fd2 : fds \mid fd1.2 \cap fd2.1 \neq \emptyset \wedge fd1.2 = fd2.2 \bullet fd2 \in partial_FD)$

For instance: $fd1: \{(3, cno, 2), (10, tno, 4)\} \rightarrow \{(11, tname, 4)\}$

$fd2: \{(10, tno, 4)\} \rightarrow \{(11, tname, 4)\}$

thus, the following statement returns a true value as

$= fd1.2 \cap fd2.1 \neq \emptyset \wedge fd1.2 = fd2.2$

$= \{(3, cno, 2), (10, tno, 4)\} \cap \{(10, tno, 4)\} \neq \emptyset \wedge (11, tname, 4) = (11, tname, 4)$

$\Rightarrow fd2: (10, tno, 4) \rightarrow (11, tname, 4) \text{ is a PFD}$

Hence, $\neg (\text{True}) = \text{False}$

As a result of the conjunction of all the three predicates with the values (*True* \wedge *False* \wedge *False*), the function *Is_2XNF* will return the **False** value. This means the *SchemaGDTD* is not in the 2XNF. To transform the *SchemaGDTD* to the 2XNF design, the following schema is applied

Second_NormalForm_GDTD \triangle *Restructure_1XNF_schemaGDTD* \vee *Not_NormalForm*

As presented in Chapter 5 (Section 5.6.4), Schema *Restructure_1XNF_schemaGDTD* consists of the following schemas inclusion

- (1) *Find_PathLink_with_Dependency*
 - (1.1) *Find_Partial_Dependencies*
- (2) *Do_Insert_New_ComplexElement_Node*
- (3) *Do_Replicate_Attribute_Node*
- (4) *Do_Replicate_SimpleElement_Node*

We present here how each of the above functions is used to normalize the structure of 1XNF *SchemaGDTD*.

Step 1: *Find_PathLink_with_Dependency*

The *Find_PathLink_with_Dependency* schema returns the value **true** since there exist many-to-one and many-to-many relationships in the set of relations *PathLink* in the *SchemaGDTD*. This process is executed using the functions *Is_Rel_Many_to_One* and *Is_Rel_Many_to_Many* (the process similar to predicate in function *Is_2XNF*). For instance, with the existence of the following relations in the *PathLink* set, the schema will generate an *Existence* report.

$\langle \{(1, Department, 0) \mapsto (2, Course, 1), 2, 1..n, 1..1),$
 $((2, Course, 1) \mapsto (5, Student, 2), 2, 1..n, 1..n)\} \rangle$

Step 1.1: Function *Find_Partial_Dependencies* is used to get the PFD in the *givenFD*.

In this function, one argument is given, which is a set of FD and returns the PFD

The function *Find_Partial_Dependencies* returns $(10, tno, 4) \rightarrow (11, name, 4)$ as it satisfies the following predicate:

$(\forall fds: givenFD :$

$\exists fd1:fd2: fds \mid fd1.2 \cap fd2.1 \neq \emptyset \wedge fd1.2 = fd2.2 \bullet fd2 \in partial_FD)$

Step 1.2: Then, the parent of LHS and RHS of PFD is determined using the relation

Has_A_Link and *Part_of_Link*. Hence,

$Has_A_Link \sim (10, tno, 4) \models (9, lecturer, 3) \wedge$

$Part_of_Link \sim (11, name, 4) \models (9, lecturer, 3)$

Step 2: Do_Insert_New_ComplexElement_Node

Once the complex element node $(9, lecturer, 3)$ is determined, the following schema is applied.

$Do_Insert_New_ComplexElement_Node \triangleq Insert_NewComplexElement_Node \wedge success$
 \vee
 $ComplexElement_Node_AlreadyExisted$

Step 2.1 *Insert_New_ComplexElement_Node* will insert a newly created complex element node into a new position in the *SchemaGDTD*. Before the insertion is made, a new complex element node must be created using the function *Create_NewComplexElement_Node*. In this function, the properties of the new complex element node such as ID, name and level are created. The ID is generated automatically based on the *preorder traversal method*. For instance in this case, the ID is 12, the name for the new complex element is given as *lecturer_new* based on the complex element node $(9, lecturer, 3)$ which has been determined from step 1.2. The level is equivalent

to a parent of complex element node (9, lecturer, 3). This process is demonstrated using the following predicates:

(1) *Using parent_ce function*

$= (9, \text{lecturer}, 3) \mapsto (5, \text{Student}, 2) \in \text{parent_ce}$

$\Leftrightarrow (9, \text{lecturer}, 3) \neq (5, \text{Student}, 2)$

$\Leftrightarrow \text{Get_level}(5, \text{Student}, 2) < \text{Get_Level}((9, \text{lecturer}, 3))$

$\Leftrightarrow 3 - 2 = 1$

$\Rightarrow (5, \text{Student}, 2)$

(2) $\text{Get_level}(\text{lecturer_new}) = 2$ and $\text{Get_ID}(\text{lecturernew}) = 12$

(3) $\text{newnode} = (12, \text{lecturer_new}, 2)$

(4) $(12, \text{lecturer_new}, 2) \notin \text{schema.Cnodes}$

(5) $\text{schema'.Cnodes} = \text{schema.Cnodes} \cup \{(12, \text{lecturer_new}, 2)\}$

Step 2.2 The (12, lecturer_new, 2) is linked with a parent node using the following statement

$\text{new_link!} = \text{Create_Path_Link}(\text{newnode}, \text{parent_ce}(\text{parent_ce ce!}))$

A *Path_Link* is created between new node (12, lecturer_new, 2) and the parent of (5, student, 2) which is (2, Course, 1).

The instance of a new *Path_Link* which is (2, 1..1, 1..n) is replicated from the previous *Path_Link* between (2, Course, 1) and (12, lecturer_new, 2). This process is shown using the following predicates:

$(2, \text{Course}, 1) \mapsto (12, \text{lecturer_new}, 2) \in \text{newlink.PathLink} \Leftrightarrow$

(1) $(2, \text{Course}, 1) \mapsto (12, \text{lecturer_new}, 2) \notin \text{newlink.PathLink}^+ \quad (\text{True})$

(2) $(2, \text{Course}, 1) = \text{Parent_ce}(12, \text{lecturer_new}, 2) \quad (\text{True})$

(3) $newlink = (2, Course, 1) \mapsto (12, lecturer_new, 2) \wedge newlink.degree = 2 \wedge$

(4) $newlink.pc = 1..1 \wedge newlink.cc = 1..n$

(5) $schema'.PathLink =$

$schema.PathLink \cup \{(2, Course, 1) \mapsto (12, lecturer_new, 2), 2, 1..1, 1..n\}$

Step 3: Do_Replicate_Attribute_Node

$Do_Replicate_Attribute_Node \triangleq Replicate_Attribute_Node \wedge success$
 \vee
 $Attribute_Node_AlreadyExisted$

$newnode? = (12, lecturer_new, 2)$ and $oldnode? = (9, lecturer, 3)$

(1) $found_attkey! = partof \langle (oldnode) \rangle$

$= Partof \langle (9, lecturer, 3) \rangle$

$= (10, tno, 4)$

(2) $new_ID = ID \mapsto n$

$= 13$

(3) $replicate_name = Get_Name(found_attkey!)$

$= Get_Name \langle (10, tno, 4) \rangle$

$= tno$

(4) $new_level = Get_Level(newnode)+1$

$= Get_Level(12, lecturer_new, 2)+1$

$= 2+1 = 3$

(5) $replicate_att! = Create_Attribute_Node(new_ID, replicate_name, new_level)$

$= Create_Attribute_Node(13, tno, 3)$

(6) $new_Partof_Link = Create_Partof_Link(replicate_attribute, newnode)$

$$\begin{aligned}
&= \text{Create_Partof_Link} ((13, \text{tno}, 3), (12, \text{lecturer_new}, 2)) \\
&= (12, \text{lecturer_new}, 2) \mapsto (13, \text{tno}, 3) \in \text{Partof} \\
&= \text{schemaGDTD'.Partof} = \text{schemaGDTD.Partof} \cup \\
&\quad (12, \text{lecturer_new}, 2) \mapsto (13, \text{tno}, 3)
\end{aligned}$$

Step 4: Do_Replicate_SimpleElement_Node

Do_Replicate_SimpleElementNode \triangleq *Replicate_SimpleElement_Node* \wedge *success*

Schema *Replicate_SimpleElement_Node* consists of following predicates

newnode? = (12, *lecturer_new*, 2) and *oldnode?* = (9, *lecturer*, 3)

- (1) $\text{found_se!} = \text{HasA } \emptyset (\text{oldnode?}) \emptyset$
 $= \text{HasA } \emptyset (9, \text{lecturer}, 3) \emptyset$
 $= (11, \text{tname}, 4)$
- (2) $\text{new_ID} = \text{ID} \mapsto n$
 $= \text{ID} \mapsto 14$
- (3) $\text{replicate_name} = \text{Get_Name} (\text{found_se!})$
 $= \text{Get_Name} (11, \text{tname}, 4)$
 $= \text{tname}$
- (4) $\text{new_level} = \text{Get_Level}(\text{newnode})+1$
 $= \text{Get_Level} (12, \text{lecturer_new}, 2)+1$
 $= 2+1 = 3$

- (5) $replicate_se \neq Create_SimpleElement_Node(new_ID, replicate_name, new_level)$
 $= Create_SimpleElement_Node(14, tname, 3)$
- (6) $new_has_a_link! = Create_Has_A_Link(replicate_se, newnode)$
 $= Create_Has_A_Link((14, tname, 3), (12, lecturer_new, 2))$
 $= (12, lecturer_new, 2) \mapsto (14, tname, 3) \in HasA$
 $= schemaGDTD'.HasA = schemaGDTD.HasA \cup$
 $(12, lecturer_new, 2) \mapsto (14, tname, 3)$
- (7) $Snodes' = Snodes \setminus oldnode$
 $= Snodes \setminus (11, tname, 4)$
- (8) $Schema'.HasA = schema.HasA \setminus (9, lecturer, 3) \mapsto (11, tname, 4)$

Step 5: Update a set of complex element nodes, simple element nodes and attribute nodes

The process of updating is done using the following predicates:

- (1) $schema'.Attnodes = schema.Attnodes \cup (13, tno, 3)$
- (2) $schema'.Snodes = schema.Snodes \oplus (14, tname, 3)$
- (3) $schema'.Cnodes = schema.Cnodes \cup (12, lecturer_new, 2)$
- (4) $schema'.HierarchicalLink = schema.HierarchicalLink \cup$
 $\{(2, Course, 1) \mapsto (12, lecturer_new, 2), 2, 1..1, 1..N\}$
- (5) $schema'.Partof = schema.PartofLink \cup \{(12, lecturer_new, 2) \mapsto (13, tno, 3)\}$
- (6) $schema'.HasA = schema.HasA \cup (12, lecturer_new, 2 \mapsto 14, tname, 3)$

Step 6: Update the data dependencies

Data dependencies are updated using the following predicates

- (1) $KD' = KD \cup \{(13, tno, 3) \mapsto 12, lecturer_new, 2)\}$
- (2) $FD' = FD \setminus \{(10, tno, 4) \rightarrow (11, tname, 4)\}$
- (3) $FD' = FD \cup \{(13, tno, 3) \rightarrow (14, tname, 3)\}$

The following Figure 6.3 and Figure 6.4 are the *SchemaGDTD* in a 2XNF design and updated set of FDs respectively.

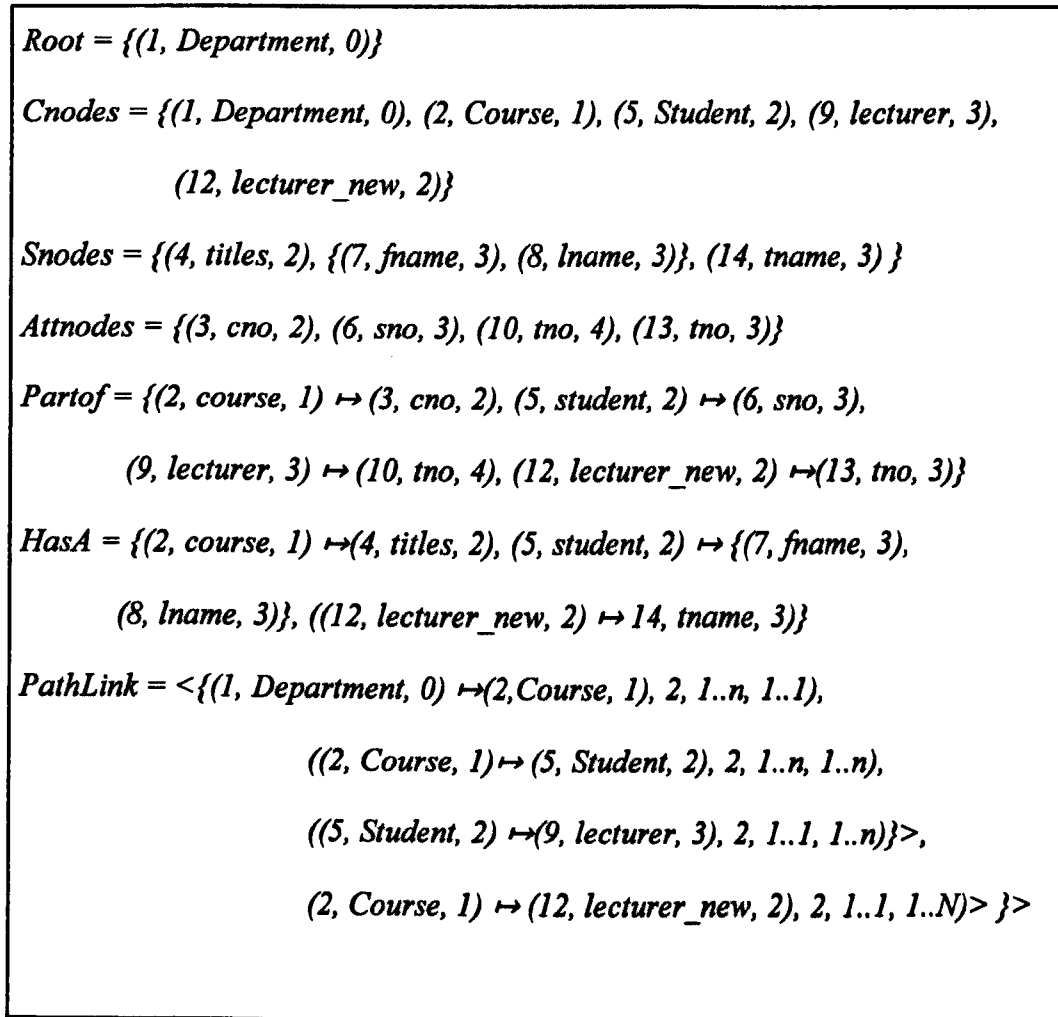


Figure 6.3: *Schema GDTD* in 2XNF

$$\begin{aligned}
KD = & \{(3, cno, 2) \rightarrow (2, Course, 1), \\
& (6, sno, 3) \rightarrow (5, Student, 2), \\
& (10, tno, 4) \rightarrow (9, lecturer, 3), \\
& (13, tno, 3) \rightarrow (12, lecturer_new, 2)\} \\
GFD = & \{(6, sno, 3) \rightarrow \{(7, fname, 3), (8, lname, 3), \\
& (13, tno, 3) \rightarrow (14, tname, 3)\} \\
TFD = & (3, cno, 2) \rightarrow (13, tno, 3) \\
& (13, tno, 3) \rightarrow (14, tname, 3) \\
& (3, cno, 2) \rightarrow (14, tname, 3) \\
PFD = & \{(3, cno, 2), (6, sno, 3)\} \rightarrow (13, tno, 3), \\
& (13, tno, 3) \rightarrow (14, tname, 3)\}
\end{aligned}$$

Figure 6.4: Set of KDs and FDs

6.3.3 Normalize 2XNF to 3XNF

Given *SchemaGDTD* with a set of FDs as shown in Figure 6.3 and Figure 6.4, first, GTD normaliser will check the status of *SchemaGDTD* using function *Is_3XNF*. If the *SchemaGDTD* is in 3XNF, the function will return the value **true**.

Function *Is_3XNF* contains three predicates:

$$(1) \text{ Is_2XNF (schema) = true}$$

The first predicate is true since the *SchemaGDTD* has satisfied all the predicates as shown in Figure 6.3

$$(2) \neg(\forall hlink_1, hlink_2: seq \text{ Path_Link} |$$

$$\text{Is_Rel_Many_to_One (hlink}_1.l, \text{ schema) = True} \wedge$$

$Is_Rel_Many_to_Many(hlink_2.2, schema) = True \wedge$

$\exists_1 (hlink_n: seq Path_Link \mid lastlink = last hlink_n \bullet$

$Is_Rel_One_to_Many(lastlink, schema) = True))$

$\neg (True \wedge True \wedge True)$ Negation of these predicates returns a **false** value

In the second predicate, each Path_Link is identified in a sequence starting from the first position until the last position of the Path_Link. Function *Is_Rel_Many_to_One* is used to check if $(1, department, 0) \mapsto (2, course, 1), 2, 1..n, 1..1)$ is a many_to_one relationship. This returns a true value since it satisfies the following condition:

$(first\ rel.pc \geq 1 \wedge second\ rel.pc \geq 2 \wedge first\ rel.cc = 1 \wedge second\ rel.cc = 1)$

Function *Is_Rel_Many_to_Many*, returns a true value for the Path link $((2, Course, 1) \mapsto (5, Student, 2), 2, 1..n, 1..n)$ since it satisfies the following condition:

$(first\ rel.pc \geq 1 \wedge rel.second\ pc \geq 2 \wedge first\ rel.cc \geq 1 \wedge second\ rel.cc \geq 2)$

and Function *Is_Rel_One_to_Many* returns a true value for the path link $((5, Student, 2) \mapsto (9, lecturer, 3), 2, 1..1, 1..n)$ since it satisfies the following condition:

$first\ rel.pc \geq 1 \wedge second\ rel.pc \geq 1 \wedge first\ rel.cc = 1 \wedge second\ rel.cc = 2)$

Note : *pc* and *cc* refer to parent constraint and child constraint respectively.

(1) $\neg (\forall givenFD : Funcional_Dependencies \bullet$

$\{ \exists tfd: transitive_FD \mid pfd = Find_Transitive_Dependencies(givenFD)$

$\wedge pfd \in partial_dependency\} \bullet tfd)$

The third predicate indicates the function *Find_Transitive_Dependencies* returns a true value since TFD $(3, cno, 2) \rightarrow (14, tname, 3)$ is found in the set of *givenFD* as it satisfies the following predicate:

($\forall fds$: given set *XFD* :

$$\exists fd1, fd2, fd3: fds \mid (fd1.1 \cap fd2.1) = \emptyset \wedge (fd1.2 \cap fd2.2) = \emptyset \wedge fd1.2 = fd2.1 \wedge fd3 = fd1.1 \rightarrow fd2.2 \bullet fd3 \in transitive_FD) \bullet fd3$$

For instance: $fd1 : (3, cno, 2) \rightarrow (13, tno, 3)$

$fd2 : (13, tno, 3) \rightarrow (14, tname, 3)$

$fd3 : (3, cno, 2) \rightarrow (14, tname, 3)$

Thus, the following statement returns a true value since

$$(3, cno, 2) \cap (13, tno, 3) = \emptyset \wedge (13, tno, 3) \cap (14, tname, 3) = \emptyset$$

and $fd3 : (3, cno, 2) \rightarrow (14, tname, 3)$ is a TFD

Hence, $\neg(\text{True}) = \text{False}$

The disjunction of all the predicates in the *Is_3XNF* function returns a false value. This means the *SchemaGDTD* is not in the 3XNF. To transform the *SchemaGDTD* to the 3XNF form, the following schema is applied.

Third_NormalForm_GDTD \triangleq *Restructure_2XNF_schemaGDTD* \vee *Not_NormalForm*

As presented in Chapter 5 (Section 5.6.5), Schema *Restructure_2XNF_schemaGDTD* consists of the following schemas inclusion:

- (1) Find_PathLink_with_Dependency
- (2) Query_ComplexElement_Node
- (3) Query_SimpleElement_Node
- (4) Query_AttributeKey_Node

We present here how each of the above functions is used to normalize the structure of 2XNF *SchemaGDTD* to 3XNF design.

Step 1: Find_PathLink_with_Dependency

The schema will return the value **true** since there exist the following many-to-one and many-to-many relationship types in the PathLink set:

$(1, \text{Department}, 0) \mapsto (2, \text{Course}, 1), 2, 1..n, 1..1)$,

$(2, \text{Course}, 1) \mapsto (5, \text{Student}, 2), 2, 1..n, 1..n)$

Step 1.2: Function *Find_Transitive_Dependencies* is used to get the TFD in the set of given XFD. In this function, one argument is given which is a set of XFD and return the TFD

The function *Find_Transitive_Dependencies* returns $(3, \text{cno}, 2) \rightarrow (14, \text{tname}, 3)$ since it satisfies the following predicate:

$(\forall fds : \text{givenFD} \bullet$

$\exists fd1, fd2, fd3: fds \mid (fd1.1 \cap fd2.1) = \emptyset \wedge (fd1.2 \cap fd2.2) = \emptyset) \wedge fd1.2 = fd2.1 \wedge$
 $fd3 = fd1.1 \rightarrow fd2.2 \bullet fd3 \in \text{transitive_FD}) \bullet fd3$

Step 2 : Query_ComplexElement_Node

After the TFD is found, the parent of LHS and RHS of TFD: $(3, cno, 2) \rightarrow (14, tname, 3)$ is determined using the relation *Has_A_Link* and *Part_of_Link*. Hence,

$$\Leftrightarrow \text{Part_of_Link} \sim ((3, cno, 2)) \neq (2, course, 1)$$

$$\Leftrightarrow \text{Has_A_Link} \sim ((14, tname, 3)) \neq (12, lecturer_new, 2)$$

$$\Leftrightarrow (2, course, 1) \mapsto (12, lecturer_new, 2) \in \text{PathLink} \quad (\text{True})$$

Step 2.1

Once the complex elements $(2, course, 1)$ and $(12, lecturer_new, 2)$ nodes are determined, the current position of the $(12, lecturer_new, 2)$ node is changed to be equivalent to $(2, course, 1)$ level.

$$\Leftrightarrow (2, course, 1) \mapsto (12, lecturer_new, 2) \in \text{PathLink}$$

$$\Leftrightarrow \text{lecturer_new.level} = \text{Get_Level}(2, course, 1)$$

$$\Rightarrow (12, lecturer_new, 1)$$

Step 3: Query_SimpleElement_Node

The simple element node of $(12, lecturer_new, 2)$ is determined using the relation *Has_A* link and the level of node is changed to one level only. Hence,

$$\Leftrightarrow \text{found_se!} = \text{schema.Has_A} \text{ (ce?)}$$

$$= \text{schema.Has_A} \text{ ((12, lecturer_new, 2))}$$

$$= (14, tname, 3)$$

$$\Rightarrow \text{Simple_Element.level} = \text{Get_Level}(\text{found_se!}) - 1$$

$$= 3 - 1 = 2$$

$$\Rightarrow (14, tname, 2)$$

Step 4: Query_AttributeKey_Node

The attribute node of $(12, \text{lecturer_new}, 2)$ is determined using the relation Part_of link.

$$\begin{aligned} \Leftrightarrow \text{found_attkey!} &= \text{schema.Part_of } \langle \text{ce?} \rangle \\ &= \text{schema.part_of} \langle (12, \text{lecturer_new}, 2) \rangle \\ &= (13, \text{tno}, 3) \\ \Rightarrow \text{Get_Level_Attribute} &= \text{Get_Level}(\text{found_attkey!})-1 \\ &= (13, \text{tno}, 2) \end{aligned}$$

Step 5: Create_Path_Link

The $(12, \text{lecturer_new}, 1)$ is linked with a parent node using the following statement:

$$\text{new_link!} = \text{Create_Path_Link}(\text{newnode}, \text{parent_ce}(\text{parent_ce ce!}))$$

The path_link is created between newnode $(12, \text{lecturer_new}, 1)$ with parent of $(5, \text{Student}, 2)$ which is $(1, \text{department}, 0)$.

The instance of path_link, $(2, 1..1, 1..n)$ is replicated from the previous path_link between $(2, \text{Course}, 1)$ and $(12, \text{lecturer_new}, 2)$. This process is shown using the following predicates

$$\begin{aligned} (1, \text{department}, 0) \mapsto (12, \text{lecturer_new}, 1) \in \text{newlink.PathLink} &\Leftrightarrow \\ (1) \quad (1, \text{department}, 0) \mapsto (12, \text{lecturer_new}, 1) \notin \text{newlink.PathLink}^+ & \text{ (True)} \\ (2) \quad (1, \text{department}, 0) = \text{Parent_ce}(12, \text{lecturer_new}, 1) & \text{ (True)} \\ (3) \quad \text{newlink} = (1, \text{department}, 0) \mapsto (12, \text{lecturer_new}, 1) \wedge \text{newlink.degree} = 2 \wedge & \\ (4) \quad \text{newlink.pc} = 1..n \wedge \text{newlink.cc} = 1..1 & \\ (5) \quad \Rightarrow \text{schema'.PathLink} = \text{schema.PathLink} \cup & \\ & \{(1, \text{department}, 0) \mapsto (12, \text{lecturer_new}, 1), 2, 1..n, 1..1\} \end{aligned}$$

Step 6: Create_Has_A_Link

$$\begin{aligned} \text{newlink} &= (12, \text{lecturer_new}, 1) \mapsto (14, \text{tname}, 2) \in \text{newlink.hasa} \\ &= \text{schema'.HasA} = \text{schema.HasA} \oplus (12, \text{lecturer_new}, 1) \mapsto (14, \text{tname}, 2) \end{aligned}$$

Step 7: Create_Part_of_Link

$$\begin{aligned} \text{newlink} &= (12, \text{lecturer_new}, 1) \mapsto (13, \text{tno}, 2) \in \text{newlink.part_of} \\ &= \text{schema'.Partof} = \text{schema.Partof} \oplus \\ &\quad \{12, \text{lecturer_new}, 1\} \mapsto (13, \text{tno}, 2)\} \end{aligned}$$

Step 8: Update set of complex element nodes, simple element nodes and attributes nodes

The set of nodes are updated using the following predicates

- 1) $\text{schema'.Attnodes} = \text{schema.Attnodes} \oplus (13, \text{tno}, 2)$
- 2) $\text{schema'.Snodes} = \text{schema.Snodes} \oplus (14, \text{tname}, 2)$
- 3) $\text{schema'.Cnodes} = \text{schema.Cnodes} \oplus (12, \text{lecturer_new}, 1)$

Step 9: Update the data dependencies

The KD and FD are updated using the following predicates

- (1) $KD' = KD \oplus (13, \text{tno}, 2) \rightarrow (12, \text{lecturer_new}, 1)$
- (2) $TFD' = TFD \oplus (3, \text{cno}, 2) \rightarrow (14, \text{tname}, 2)$

The following Figure 6.5 is the *SchemaGDTD* in the 3XNF form consisting of the following sets and relations.

$Root = \{(1, Department, 0)\}$
 $Cnodes = \{(1, Department, 0), (2, Course, 1), (5, Student, 2), (9, lecturer, 3),$
 $(12, lecturer_new, 1)\}$
 $Snodes = \{(4, titles, 2), \{(7, fname, 3), (8, lname, 3)\}, (14, tname, 2)\}$
 $Attnodes = \{(3, cno, 2), (6, sno, 3), (10, tno, 4), (13, tno, 2)\}$
 $Partof = \{(2, course, 1) \mapsto (3, cno, 2), (5, student, 2) \mapsto (6, sno, 3),$
 $(9, lecturer, 3) \mapsto (10, tno, 4), (12, lecturer_new, 1) \mapsto (13, tno, 2)\}$
 $HasA = \{(2, course, 1) \mapsto (4, titles, 2), (5, student, 2) \mapsto \{(7, fname, 3),$
 $(8, lname, 3)\}, (12, lecturer_new, 1) \mapsto (14, tname, 2)\}$
 $PathLink = \{<\{(1, Department, 0) \mapsto (2, Course, 1), 2, 1..n, 1..1),$
 $((2, Course, 1) \mapsto (5, Student, 2), 2, 1..n, 1..n),$
 $((5, Student, 2) \mapsto (9, lecturer, 3), 2, 1..1, 1..n)\}>,$
 $<(1, Department, 0) \mapsto (12, lecturer_new, 1) 2, 1..n, 1..1>$

Figure 6.5: *SchemaGDTD* in 3XNF

$$\begin{aligned}
 KD = & \{(3, cno, 2) \rightarrow (2, Course, 1), (6, sno, 3) \rightarrow (5, Student, 2) \\
 & (10, tno, 4) \rightarrow (9, lecturer, 3), \\
 & (13, tno, 2) \rightarrow (12, lecturer_new, 1)\} \\
 GFD = & \{(6, sno, 3) \rightarrow \langle (7, fname, 3), (8, lname, 3) \rangle, \\
 & (13, tno, 2) \rightarrow (14, tname, 2)\} \\
 TFD = & \{(3, cno, 2) \rightarrow (13, tno, 2) \\
 & (13, tno, 2) \rightarrow (14, tname, 2) \\
 & (3, cno, 2) \rightarrow (14, tname, 2)\} \\
 PFD = & \{\langle (3, cno, 2), (6, sno, 3) \rangle \rightarrow (13, tno, 2), \\
 & (13, tno, 2) \rightarrow (14, tname, 2)\}
 \end{aligned}$$

Figure 6.6: Set of FDs

6.3.4 Normalize 3XNF to 4XNF

Given *SchemaGDTD* with a set of FD as shown in Figure 6.5 and Figure 6.6, the G-DTD normalizer will check the status of *SchemaGDTD* using function *Is_4XNF*. If the *SchemaGDTD* is in the 4XNF form, the function will return the **true** value. However the *SchemaGDTD* is not in the 4XNF design as it does not satisfy some of the predicates listed in function *Is_4XNF*. This can be proved as follows:

Function *Is_4XNF* contains three predicates

$$(1) \text{ Is_1XNF (schema) = true}$$

The first predicate is true since the *SchemaGDTD* has satisfied all the predicates as shown in section 6.3.2

(2) $\neg(\forall hlink_1, hlink_2: seq Path_Link |$
 $Is_Rel_Many_to_One(hlink_1.1, schema) = True \wedge$
 $Is_Rel_Many_to_Many(hlink_2.2, schema) = True \wedge$
 $\exists_1 (hlink_n: seq Path_Link | lastlink = last hlink_n \bullet$
 $Is_Rel_One_to_Many(lastlink, schema) = True))$
 $\neg (True \wedge True \wedge True)$ Negation of these predicates return false value

In the second predicate, each Path_Link is identified in a sequence starting from the first position until the last position of the Hieararchical_Link. Function *Is_Rel_Many_to_One* is used to check if (1, Department, 0) \mapsto (2, Course, 1), 2, 1..n, 1..1) is a *Many_to_One* relationship type. This function returns a true value since its predicate satisfies the following condition:

$(first\ rel.pc \geq 1 \wedge second\ rel.pc \geq 2 \wedge first\ rel.cc = 1 \wedge second\ rel.cc = 1)$

Function *Is_Rel_Many_to_Many*, returns a true value for the Path_Link ((2, Course, 1) \mapsto (5, Student, 2), 2, 1..n, 1..n) since it satisfies the following condition:

$(first\ rel.pc \geq 1 \wedge rel.second\ pc \geq 2 \wedge first\ rel.cc \geq 1 \wedge second\ rel.cc \geq 2)$

and Function *Is_Rel_One_to_Many* returns a true value for the Path_Link ((5, Student, 2) \mapsto (9, lecturer, 3), 2, 1..1, 1..n) since it satisfies the following condition:

$first\ rel.pc \geq 1 \wedge second\ rel.pc \geq 1 \wedge first\ rel.cc = 1 \wedge second\ rel.cc = 2)$

Note : *pc* and *cc* are refer to parent constraint and child constraint respectively.

(3) $\neg (\forall givenFD : Funcional_Dependencies \bullet$
 $\{ \exists gfd: global_FD | gfd = Find_Global_Dependencies(givenFD) \wedge$
 $gfd \in global_dependency\} \bullet gfd)$

The third predicate indicates that the function Find_Global_Dependencies returns a true value since GFD $\{(6, sno, 3) \rightarrow \langle (7, fname, 3), (8, lname, 3) \rangle\}$ is found in the set of givenFD since it satisfies the following predicate:

$$\begin{aligned}
 & (\forall fds: givenFD \bullet \\
 & (\exists fd: fds, att: Attribute_Node; se: SimpleElement_node; ce: ComplexElement_node | \\
 & current_element = getFD_element (fd) \wedge current_element = att \cup se \\
 & \wedge get_level (att) = get_level (se) \wedge Is_last (se, schema) = true \\
 & \wedge has_link (att) = part_of_link (se) \Rightarrow att \mapsto se \in global_FD) \bullet fd)
 \end{aligned}$$

For instance: $fd: \{(6, sno, 3) \rightarrow \langle (7, fname, 3), (8, lname, 3) \rangle\}$

Since all attribute nodes and simple element nodes are located at the same level, which where is the last level in the *schemaGDTD*.

Hence, $\neg (True) = False$

The conjunction of all the three predicates with $(True \wedge False \wedge False)$ will return a **False** value. This means that the *SchemaGDTD* is not in the 4XNF. To transform the *SchemaGDTD* to the 4XNF form, the following schema is applied:

Fourth_NormalForm_GDTD \triangleq *Restructure_3XNF_schemaGDTD* \vee *Not_NormalForm*

As presented in Chapter 5 (Section 5.6.6), Schema *Restructure_3XNF_schemaGDTD* consists of the following schemas inclusion

(1) Find_Path_Link_with_Dependency

(1.1) Find_Global_Dependencies

- (2) Do_Insert_New_ComplexElement_Node
- (3) Do_Replicate_Attribute_Node
- (4) Do_Replicate_SimpleElement_Node

We present here how each of the above functions is used to normalise the structure of *SchemaGDTD*.

Step 1: *Find_Path_Link_with_Dependency_schema* returns the value **true** since there exist many-to-one and many-to-many relationship in a set of relation PathLink in the *SchemaGDTD*. For instance, with the existence of the following path links,

$\langle \{(1, Department, 0) \mapsto (2, Course, 1), 2, 1..n, 1..1),$
 $((2, Course, 1) \mapsto (5, Student, 2), 2, 1..n, 1..n) \rangle,$
 $\langle \{(1, Department, 0) \mapsto (12, lecturer_new, 1) 2, 1..n, 1..1) \} \rangle$

the schema then generates an *Existence* report.

Step 1.1: Function *Find_Global_Dependencies* is used to get the GFD in the given FD. In this function, one argument is given which is a set of FD and returns the GFD

The function *Find_Global_Dependencies* returns

$\{(6, sno, 3) \rightarrow \{(7, fname, 3), (8, lname, 3)\} \}$

Step 1.2: Then, the parent of LHS and RHS of GFD is determined using the relation *Has_A_Link* and *Part_of_Link*. Thus,

$Has_A_Link \sim \{(7, fname, 3), (8, lname, 3)\} \neq (5, student, 2) \wedge$

$Part_of_Link \sim ((6, sno, 3)) \neq (5, student, 2)$

Step 2: Once the complex element $(5, student, 2)$ is determined, schema

$$Do_InsertNewComplexElement_Node \triangleq Insert_NewComplexElement_Node \wedge$$

$$success \vee$$

$$ComplexElement_Node_AlreadyExisted$$

is applied.

Step 2.1 *Insert_New_ComplexElement_Node* will insert a newly created complex element node into a new position in the *schemaGDTD*. Before the insertion is made, the new complex element node must be created using the function *Create_NewComplexElement_Node*. In this function, the properties of the new complex element node such as ID, name and level, are created. The ID is generated automatically based on the ordered *preorder traversal method*. For instance, in this case the ID is 15, the name for a new complex element is given as *student_new* based on the complex element node (5, *student*, 2) which has been determined from step 1.2. The level is equivalent to level 1. This process is demonstrated using the following predicates

- (1) $student_new.level = 1$ and $student_new.ID = 15$
- (2) $newnode = (15, student_new, 1)$
- (3) $(15, student_new, 2) \notin schema.Cnodes$
- (4) $schema'.Cnodes = schema.Cnodes \cup \{(15, student_new, 1)\}$

Step 2.2 The (15, *student_new*, 1) is linked with a root node using the following statement

$new_link! = Create_Path_Link(newnode, root_node)$

A *Path_Link* is created between the newnode (15, *student_new*, 1) and root node which is (1, *department*, 0).

The instance of a new Path_Link which is (2, 1..n, 1..1) is replicated from the previous Path_Link between (1, department, 0) and (15, student_new, 1). This process is shown using the following predicates:

- $$(1, department, 0) \mapsto (15, student_new, 1) \in newlink.PathLink \Leftrightarrow$$
- (1) $(1, department, 0) \mapsto (15, student_new, 1) \notin newlink.PathLink$ (True)
 - (2) $Is_root(1, department, 0)$ (True)
 - (3) $newlink = (1, department, 0) \mapsto (15, student_new, 1)$
 - (4) $newlink.degree = 2 \wedge$
 - (5) $newlink.pc = 1..1 \wedge newlink.cc = 1..n$
 - (6) $schema'.PathLink = schema.PathLink \cup$
 $\{(1, department, 0) \mapsto (15, student_new, 1), 2, 1..n, 1..1\}$

Step 3: Do_Replicate_Attribute_Node

$$Do_Replicate_Attribute_Node \triangleq Replicate_Attribute_Node \wedge success$$

$$\vee$$

$$Attribute_Node_AlreadyExisted$$

$newnode? = (15, student_new, 1)$ and $oldnode? = (5, lecturer, 2)$

- (1) $found_att! = partof \downarrow (oldnode?) \downarrow$
 $= Part\ of \downarrow (5, lecturer, 2) \downarrow$
 $= (6, sno, 3)$
- (2) $newID = ID \mapsto n$
 $= 16$
- (3) $replicate_name = Get_Name(found_att!)$
 $= Get_Name((6, sno, 3))$
 $= sno$

- (4) $new_level = Get_Level(newnode) + 1$
 $= Get_Level(15, student_new, 1) + 1$
 $= 1 + 1 = 2$
- (5) $replicate_attribute = create_Attribute_Node(newID, replicate_name, new_level)$
 $= create_Attribute_Node(16, sno, 2)$
- (6) $new_Partof_link! = Create_Partof_Link(replicate_attribute, newnode)$
 $= Create_Partof_Link((16, sno, 2), (15, student_new, 1))$
 $= (15, student_new, 1) \mapsto (16, sno, 2) \in Partof$
- (7) $schemaGDTD'.Partof = schemaGDTD \cup (15, student_new, 1) \mapsto (16, sno, 2)$

Step 4: Do_Replicate_SimpleElement_Node

$Do_Replicate_SimpleElementNode \sqcap Replicate_SimpleElement_Node \wedge success$

Schema Replicate SimpleElement_Node consists of the following predicates:

$newnode? = (15, student_new, 1)$ and $oldnode? = (5, student, 2)$

- (1) $found_se! = HasA \downarrow (5, student, 2) \downarrow$
 $= \{(7, fname, 3), (8, lname, 3)\}$
- (2) $new_ID = ID \mapsto n$
 $= ID \mapsto \{17, 18\}$
- (3) $replicate_name = Get_Name(oldnode?)$
 $= Get_Name(5, student, 2)$
 $= \{fname, lname\}$
- (4) $new_level = Get_Level(newnode?) + 1$

$$= \text{Get_Level}(15, \text{student_new}, 1) + 1$$

$$= 1 + 1 = 2$$

$$(5) \quad \text{replicate_se} = \text{Create_SimpleElement_Node}(\text{new_ID}, \text{replicate_name}, \text{new_level})$$

$$= \text{Create_SimpleElement_Node}(17, \text{fname}, 2),$$

$$= \text{Create_SimpleElement_Node}(18, \text{lname}, 2))$$

$$(6) \quad \text{new_has_a_link!} = \text{Create_Has_A_Link}(\text{replicate_se}, \text{newnode})$$

$$= \text{Create_Has_A_Link}(\{(17, \text{fname}, 2), (15, \text{student_new}, 1)\})$$

$$= \text{Create_Has_A_Link}(\{(18, \text{lname}, 2), (15, \text{student_new}, 1)\})$$

$$= (15, \text{student_new}, 1) \mapsto (\{(17, \text{fname}, 2)\} \in \text{HasA})$$

$$= (15, \text{student_new}, 1) \mapsto (\{(18, \text{lname}, 2)\} \in \text{HasA})$$

$$(7) \quad \text{schemaGDTD'.HasA} = \text{schemaGDTD} \cup$$

$$\{(15, \text{student_new}, 1) \mapsto (\{(17, \text{fname}, 2)\})$$

$$(15, \text{student_new}, 1) \mapsto (\{(18, \text{lname}, 2)\})\}$$

$$(8) \quad \text{Snodes}' = \text{Snodes} \setminus \text{oldnode}$$

$$= \text{Snodes} \setminus \{(7, \text{fname}, 3), (8, \text{lname}, 3)\}$$

Step 5: Update the set of complex element nodes, simple element nodes and attribute nodes using the following predicates

$$(1) \quad \text{schema'.Attnodes} = \text{schema.Attnodes} \cup (16, \text{sno}, 2)$$

$$(2) \quad \text{schema'.Snodes} = \text{schema.Snodes} \oplus (\{(17, \text{fname}, 2), (18, \text{lname}, 2)\})$$

$$(3) \quad \text{schema'.Cnodes} = \text{schema.Cnodes} \cup (15, \text{student_new}, 1)$$

$$(4) \quad \text{schema'.HierarchicalLink} = \text{schema.HierarchicalLink} \cup$$

$$\{(1, \text{department}, 0) \mapsto (15, \text{student_new}, 1), 2, 1..n, 1..1\}$$

$$(5) \quad \text{schema'.Partof} = \text{schema.Partof} \cup (15, \text{student_new}, 1) \mapsto (16, \text{sno}, 2)$$

$$(6) \quad \text{schema'.HasA} = \text{schema.HasA} \cup$$

$$(15, \text{student_new}, 1) \mapsto \{(17, \text{fname}, 2), (18, \text{lname}, 2)\}$$

Step 6: Update the data dependencies using the following predicates

$$(1) \quad \text{KD}' = \text{KD} \cup \{(16, \text{sno}, 2) \rightarrow (15, \text{student_new}, 1)\}$$

$$(2) \quad \text{FD}' = \text{FD} \cup \{(16, \text{sno}, 2) \rightarrow \{(17, \text{fname}, 2), (18, \text{lname}, 2)\}\}$$

The following Figures 6.7 and 6.8 are the *SchemaGDTD* in the 4XNF and set of FD.

$$\begin{aligned} \text{Root} &= \{(1, \text{Department}, 0)\} \\ \text{Cnodes} &= \{(1, \text{Department}, 0), (2, \text{Course}, 1), (5, \text{Student}, 2), (9, \text{lecturer}, 3), \\ &\quad (12, \text{lecturer_new}, 1), (15, \text{student_new}, 1)\} \\ \text{Snodes} &= \{(4, \text{titles}, 2), \{(17, \text{fname}, 2), (18, \text{lname}, 2)\}, (14, \text{tname}, 3)\} \\ \text{Attnodes} &= \{(3, \text{cno}, 2), (6, \text{sno}, 3), (10, \text{tno}, 4), (13, \text{tno}, 3), (16, \text{sno}, 2)\} \\ \text{Partof} &= \{(2, \text{course}, 1) \mapsto (3, \text{cno}, 2), (5, \text{student}, 2) \mapsto (6, \text{sno}, 3), \\ &\quad (9, \text{lecturer}, 3) \mapsto (10, \text{tno}, 4), (12, \text{lecturer_new}, 2) \mapsto (13, \text{tno}, 3), \\ &\quad (15, \text{student_new}, 1) \mapsto (16, \text{sno}, 2)\} \\ \text{HasA} &= \{(2, \text{course}, 1) \mapsto (4, \text{titles}, 2), \\ &\quad (15, \text{student}, 2) \mapsto \{(17, \text{fname}, 3), (18, \text{lname}, 3)\}, \\ &\quad (12, \text{lecturer_new}, 2) \mapsto (14, \text{tname}, 3)\} \\ \text{PathLink} &= \langle \{(1, \text{Department}, 0) \mapsto (2, \text{Course}, 1), 2, 1..n, 1..1), \\ &\quad (2, \text{Course}, 1) \mapsto (5, \text{Student}, 2), 2, 1..n, 1..n), \\ &\quad (5, \text{Student}, 2) \mapsto (9, \text{lecturer}, 3), 2, 1..1, 1..n)\} \rangle, \\ &\quad \langle \{(1, \text{Department}, 0) \mapsto (2, \text{Course}, 1), 2, 1..n, 1..1) \\ &\quad (2, \text{Course}, 1) \mapsto (12, \text{lecturer_new}, 2), 2, 1..n, 1..1)\} \rangle \end{aligned}$$

Figure 6.7: *SchemaGDTD* in 4XNF

$$\begin{aligned}
KD = & \{(3, cno, 2) \rightarrow (2, Course, 1), \\
& (6, sno, 3) \rightarrow (5, Student, 2), \\
& (10, tno, 4) \rightarrow (9, lecturer, 3), \\
& (13, tno, 2) \rightarrow (12, lecturer_new, 1) \\
& ((16, sno, 2) (15, student_new, 1))\} \\
GFD = & \{(16, sno, 2) \rightarrow \{< ((17, fname, 2), (18, lname, 2))>, \\
& (13, tno, 2) \rightarrow (14, tname, 2)\} \\
TFD = & (3, cno, 2) \rightarrow (13, tno, 2) \\
& (13, tno, 2) \rightarrow (14, tname, 2) \\
& (3, cno, 2) \rightarrow (14, tname, 2) \\
PFD = & \{(3, cno, 2), (13, tno, 2) \rightarrow (14, tname, 2), \\
& (13, tno, 2) \rightarrow (14, tname, 2)\}
\end{aligned}$$

Figure 6.8: Set of KDs and FDs

6.4 Summary

In this chapter, we have tested the formal specification of the XML document design prototype, specifically the GDTD normaliser operations. To show the consistency and the correctness of the specification, we reused the same case study as provided in Chapter 4 and demonstrated step by step how the schema GDTD can be transformed from 1XNF design to 4XNF design in a formal way. We show that our specification for the GDTD normalizer operations defined in Chapter 5 is consistent with the normal forms and normalization rules defined in Chapter 4. This proves that our approach can be used as a prototype to design a non-redundant XML document and gives confidence that our prototype can be implemented successfully to generate an automatic XML document schema design.

Chapter 7

Conclusion and Future Work

7.1 Contributions of the Research

This thesis has examined the requirements and problems of XML documents schema design. This thesis argues that to produce a non-redundant schema of an XML document for application A , we should first produce a conceptual model, $G\text{-DTD}$ at schema level and then apply a normalization rule to transform $G\text{-DTD}$ into a normal form $G\text{-DTD}'$ and finally convert the conceptual model $G\text{-DTD}'$ back to the XML schema DTD (see also statement of hypothesis in the Introduction).

To assess and support this research hypothesis, a main research aim and several objectives were defined. In the following discussion, we revisit these objectives and summarize how, and to what extent, they have been achieved.

Objective 1. To investigate how design guidelines for relational schema are applied to XML database schema design using normalization theory. This involves examining XML functional dependency (XFD) concepts, discussing various definitions of XML normal forms based on these XFDs and highlighting their strengths and limitations.

This thesis investigated an approach to database design and discussed theory that has been developed to design non-redundant schema related to relational databases in general. The review of database design theory presented in Chapter 2 provides the basic concepts like data dependency, such as functional dependency, key dependency and multi valued dependency. These data dependencies are formal constraints among attributes, which are used as the main tool for formally measuring the semantic relations among attributes. The functional dependencies, key dependencies and multi-valued

dependencies can be used to group attributes into a normal form relation schema. To address the normalization process, algorithms for 3NF and BCNF design that are based on functional dependency are presented.

The thesis also described how design guidelines for relational schemas are applied to XML document schema design using a normalization theory. As a relational database, XML documents are also associated to a schema. Previous studies used DTD as a schema for XML documents. In this thesis, we have presented and compared thoroughly the notion of data dependency, namely, XML functional dependencies (XFDs) by looking into the approaches taken to define the XFDs. The advantages and disadvantages of the XFDs are highlighted with particular emphasis on semantic expressiveness, which is a desirable property for defining XML normal forms.

In this thesis, various definitions of XML normal forms proposed by Arenas and Libkin (2004), Vincent et al. (2004), Wang and Topor (2005), Kolahi (2007) and Yu and Jagadish(2008) are presented and discussed. Most of them proposed the XNF, except Kolahi proposed XML third normal form (X3NF). From the study of the characteristics, we identified that XML normal form XNF, proposed by Arenas and Libkin (2004) achieves the best possible design from the point of view of eliminating redundancies in XML documents (Kolahi, 2007). Arenas and Libkin (2004) have defined XFDs and XML normal forms (XNF) entirely on the concept of 'tree tuple' within the context of the XML document and DTD.

However, it was found that the problem with Arenas and Libkin's approach is that the way they express the semantic constraint i.e. XFD is complicated due to the textual presentation of a DTD. This led to difficulty in term of XML normal form notion and posed an obstacle to database designer in designing redundancy-free XML document schema. We have therefore suggested that this approach needs simplification for the

benefit of the users. The new approach and method are explained further in the next objectives.

Objective 2. To propose a systematic approach to simplify XML document schema design by proposing a graphical XML schema based on DTD called Graph Data Type Definition (G-DTD) at the schema level. We believe having the G-DTD model as a tool could describe the structure of XML documents at the schema level clearly and precisely

This thesis has proposed G-DTD, as a conceptual model to describe XML document at the schema level in a precise and simple way. Following the review of prominent current XML models, we have decided to adopt some of ER diagram (Chen, 1976) and ORA-SS diagram (Dobbie et al., 2000) notation in the G-DTD's notation, based on the argument made in Chapter 3. We accommodated semantic constraints in the G-DTD explicitly to help to achieve automation of XML document schema normalization.

In Chapter 3, the objective and the rationale of having G-DTD were discussed. The structure and semantics of G-DTD were introduced and developed to describe the XML document at the schema level. G-DTD consists of five main parts: complex element node, simple element node, attribute node, root node and relationship type. G-DTD represents an XML document in a simpler and clearer manner compared with the original textual representation, DTD. The second important property of the G-DTD is that it has path links, Part_of link, and Has_A link type. These properties distinguished G-DTD from other data models. More importantly, the semantic relationship between complex element types can be modelled at the schema level using n-ary one-to-many/many-to-one/many-to-many path relationships.

The thesis also presented the conceptual operations of the G-DTD model, which describes the dynamic properties of the model. These operations are classified into five main parts: Query operations, Insert operations, Delete Operations, Replicate Operations and Update Operations. These operations are important as they are basic operations during the normalization process, which we discussed further in Chapter 5.

The result of G-DTD offers a clear and precise meaning for designers by providing them with information on the structure of attributes, simple element and complex element types and the semantic relationship between them. This ultimately helps contribute to a better understanding of DTD and DTD design. In Chapter 4, the thesis has shown that having the G-DTD as a tool can assist in the design process, i.e. the normalization process which as it is always easier to understand and interpret than a theoretical approach.

Once the G-DTD is in a normal form design, it is important to transform it back to DTD. To enable G-DTD to be used practically, the thesis proposed transformation rules to transform from G-DTD back to DTD.

Objective 3. To redefine a set of normal forms for G-DTD on the basis of Arenas and Libkin's rule (2004) and Lv et al.'s rules (2004), which is easy to understand and implement programmatically. To achieve this, a basic property of XML normal form, which is functional dependency, is proposed, such as relationship dependency, partial functional dependency, transitive functional dependency and global functional dependency. In the context of XML document schema normalization, it is important to develop normalization rules to transform an XML document schema into a normalised one.

The thesis has refined a set of normal forms for G-DTD called as First Normal Form (1XNF), Second Normal Form (2XNF), Third Normal Form (3XNF) and Fourth Normal Form (4XNF). The set of normal forms for G-DTD have been generalised from Arenas and Libkin (2004) and Lv et al.'s (2004) normal form. In defining these normal forms we have adopted and applied traditional data dependencies based on functional dependency, transitive dependency and partial dependency.

The thesis enhanced the normalization algorithm proposed by Arenas and Libkin (2004) by adding two rules. One rule is to check whether a semantic constraint caused by GFD, TFD or PFD existed. If one of these constraints exists, a complex element node together with its children are moved up to another location and level or a new complex element node is created and put under a root node. These rules are used to eliminate TFD and

PFD in the G-DTD. The other rule is to check the structure of G-DTD by considering one-to-many, many-to-many or many-to-one path links between complex element nodes in G-DTD. Particularly, these algorithms have restructured the original G-DTD by considering the tree structure, the level of nodes and semantic relationship types between nodes associated with a given set of functional dependencies.

In Chapter 4, a case study illustrated and demonstrated that the application of G-DTD normal forms and normalization rules achieved a redundancy-free XML document schema design. In this case study, we showed how a G-DTD can simplify the complex procedure of XML document design and normalization. Even though the length of the XML document is longer than the original document, the structure is free from data redundancy and update anomalies. Although our approach uses a simple case study, it supports the claim that users (designers) can benefit in development of real XML document schema normalization.

The thesis has compared the approach with the previous work proposed by Arenas and Libkin (2004), Lv et al. (2004) and Kolahi (2007) based on three criteria: Expression of DTD structure, XML normal forms and normalization algorithm. From the discussion, we found that our approach complements to Arenas and Libkin's approach, since it produces the same result as Arenas and Libkin. However, our approach provides an alternative method which is easy to implement and more practical, because it is a simple, precise and understandable and more importantly it works with a minimum of abstract concepts. It is relatively easier for a schema designer. Another benefit is the new set of normal form presented in this thesis, which has shown three advantages. First, the designer can indentify complex elements, simple elements and attributes graphically and can add the relationship types between the nodes from the user specification. This will give more control to the designer to evaluate each successive normal form G-DTD. Second, normalizing the G-DTD can effectively removes redundancies and anomalies at a schema level. More importantly, it is able to preserve both DTD tree structure and XML document structure and satisfy user requirements.

Objective 4. To develop a prototype of an XML document schema design using a formal approach. More specifically, to propose a formal framework of XML document normalization using Z formal specification language in order to give a precise and a clearer understanding of the whole system requirement.

The thesis proposed a novel prototype of an XML Document Schema Design. This is to support the claim that users (designers) can profitably bring formal specification to bear in development of a real XML Document Schema Design tool. The complete framework and formal specification of the XML Document Design model was presented in Chapter 5. The full specification of the model using Z notation was constructed, which gives the precise and clear meaning of the model.

This formal description is divided into three main layers: the G-DTD layer which aims to define the data types and operations precisely to reveal the semantic constraint of the G-DTD model; the G-DTD normalizer layer which presents the operations and functions required to transform from one normal form to another normal form. A specification to define normalization rules for normalization of G-DTD schema is constructed and applied to the rules for normalization presented in Chapter 4. Finally the G-DTD translator layer which presents the operation to map from G-DTD back to DTD is presented. The specification has shown that it is both possible and promising to use our approach to design non-redundant XML document schema in a practical way.

Objective 5. The final research objective is to test the specification constructed to show the consistency of the specification using a simple case study

The thesis developed a justification of the XML document design specification. The justification can be found in Chapter 6. The rationale of this specification testing is to enable us to demonstrate that the specification constructed in Chapter 5 is satisfied and consistent with certain fundamental criteria of XML document design. This will increase the confidence in the implementation later. We tested the properties of XML document design, in particular in G-DTD normaliser layer, to seek the validity of the specification and assurance that the specification can be used to derive a required

normal form one using our normalization algorithm. It can therefore increase the confidence in the design model before the decision is taken to progress towards the implementation of the model.

7.2 Limitations and Future Research

Having discussed the contributions that this research has made to the current state-of-the-art in XML design, we would like to look at the limitations of the work, and promising avenues for future research.

- (1) The scope of the research is applied and limited to a simple DTD schema with simple application only. However this approach can be extended in future to be applied to a more complex DTD. In this way, the G-DTD model can be enriched to allow users to declare more general applications with specific integrity constraints. This model can also be extended to another schema language such as XSD. Such extension is trivial because the syntax and semantics used in the model can be generalized to XSD with some changes, as there is a similarity between XSD and DTD in structure (Bex et al., 2004). Recently, XSD has become one of the primary schemas for XML documents and is widely supported by many applications (Lv. and Yan, 2006). This is because XSD addresses most of shortcomings of DTDs and in particular, is more expressive than DTDs and most importantly it uses XML as the syntax for describing schemas (Marten et al., 2007). XSD itself provides a rich set of data description mechanisms, including mixed/fixd value, empty content and cardinality constraints. It would be interesting to incorporate there features into the G-DTD model.
- (2) Another limitation is the time to implement the system. We provide only a precise framework of XML document design using Z specification. Of course, developing a formal specification is not easy and quite challenging; hence we are bound to make mistakes. However, writing a specification led to the discovery of some difficulties in the earlier design and enabled the final prototype to meet the definition of the data model. Thus, the programming effort is much reduced when there is a formal specification of the data model. For future work, an automatic

XML document design system could be developed based on our formal specification in order to derive an automatic generation of non-redundant XML document design. The XML_DM system will consist of three major components: (1) a graphical input layer to assist users in drawing their application-specific G-DTD. This input must be conducted in an easy, natural and user-friendly manner for the purpose of editing and drawing a G-DTD. (2) A normalization layer that performs the automatic normalization process, which is always easy to understand and interpret rather than a theoretical approach. This can offer a facility to normalize the schema in one click. This can enable the XML document designer to have this automation performed at a very early stage of XML document design, i.e. at the conceptual level. (3) An output layer that shows the transformation results in a DTD schema file.

- (3) In theory, another avenue for potential future research is using formal methods to help validate the G-DTD model. Proofs of important properties of the G-DTD model could, in principle help the designer to build a confidence in the claim that the model captures the intuitively desired structure and meanings. Formal verification of G-DTD properties against such a specification could similarly build confidence that the specification will be satisfied by the implementation. Our work provides a concrete starting point for exploring this idea.

Bibliography

- Abiteboul, S. and Segoufin, L. (2006). Representing and querying XML with incomplete information. *ACM Transaction on Database System*, Vol 31(1), pp. 208-254.
- Abiteboul, S., Hull, R. and Viannu, V. (1995). *Foundation of Database*. Reading: Addison-Wesley.
- Abiteboul, S., Buneman, P., and Suciu, D. (2000). *Data on the Web: From Relation to Semistructured Data and XML*. California: Morgan Kaufmann.
- Amalio, N., Stepney, S., Polack, F. (2004). Formal proof from UML Models. In *Proceeding of the 6th International Conference on Formal Engineering Methods, ICFEM*, Lecture Notes in Computer Science, Vol 3308, Springer, pp. 418-433.
- Armstrong, W. W. (1974). Dependency structures of database relationships. In *IFIP Congress*, pp. 580-583.
- Anutariya, C., Wuwongse, V., Nantajeewarawat, E., and Akama, K., (2000). Towards a Foundation for XML Document Database. *Electronic Commerce and Web Technologies, Lecture Notes in Computer Science*, Vol. 1875, pp. 324 -333
- Apparao, V and Byrne, S. (1998). Document Object Model (DOM) level 1 specification. W3C Recommendation, Available at <http://www.w3.org/TR/1998/PR-DOM-Level-1-19980818>.
- Arenas, M. (2006). Normalization theory for XML. *SIGMOD record*, Vol 35(4), pp. 57-64.
- Arenas, M., Fan, W., and Libkin, L. (2002). What's hard about XML schema constraints? *Lecture Notes in Computer Science* 2543 , pp. 269-278.
- Arenas, M. and Libkin, L. (2004). A normal form for XML documents. *ACM Transaction on Database System*, Vol 29(1) , pp. 195-232.

- Arenas, M. and Libkin, L. (2005). An information-theoretic approach to normal form for relational and XML data. *Journal of the ACM*, Vol 52.(2) , pp. 246-283.
- Atzeni, P. and De Antonellis, V. (2003). *Relational Database Theory: A comprehensive Introduction*. London: Addison-Wesley.
- Beeri C. and Bernstein, P. A. (1979). Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, Vol 4(1). pp. 30-59
- Beeri, C., Bernstein, P. and Goodman, N. (1978). A sophisticate's introduction to database normalization theory. *In Fourth International Conference on Very Large DataBases*, Seattle, pp. 113-124.
- Beeri, C., Fagin, R.,and Howard, J. (1977). A Complete axiomatization for functional and multivalued dependencies in database relations. *ACM SIGMOD International Conference on Management of Data*, New York, USA, pp. 47-61.
- Bernstein, P. A. (1976). Synthesizing third normal form relations from functional dependencies. *ACM Transactions on Database Systems*, Vol 1(4). pp. 277- 298.
- Bertino, E.; Guerrini, G.; and Mesiti, M. (2008). Measuring the structural simillarity among XML documants and DTDs. *Journal Intelligence Information System*, Vol 30 , pp. 55-92.
- Bex,G.J., Neven, F., Bussche, J.V. (2004). DTD versus XML Schema. A Practical Study. *In Proceeding of the Seventh International Workshop on the Web and Databases*, Paris, France, pp. 79-84.
- Bidiot, N., Cerrito, S., and Thion, V. (2004). A first step towards modelling semistructured data in hybrid multimodal logic. *Journal of Applied Non-Classical Logic*, Vol 14. pp.
- Biggs, N. L. (1985). *Discrete Mathematics*. London: Oxford University Press.

- Bird, L., Goodchild, A., and Halpin, T. (2000). Object role modelling and XML-schema. *In Proceedings of the Ninth International Conference of Conceptual Modeling(ER2000)*, Salt Lake City, Utah, pp. 309-322.
- Biskup, J. (1995). Achievement of relational database schema design theory revisited semantic in database. *Lecture Notes Computer Science*, Vol 1066, pp.14-44.
- Biskup, J., Dayal, U., Bernstein, P. A.(1979). Synthesizing independent database schemas. *In Proceeding of the 1979 ACM SIGMOD International Conference on Management of Data*. New York, USA, pp. 143-151.
- Bisova, V., and Richta, K. (September 2000). Transformation of UML models into XML. *In Proceeding of the 2000 ADBIS-DASFAA symposium on Advance in Database and Information Systems*, Prague Czech Republic, pp. 33-45.
- Borros, D. (1994). On the formal specification and derivation of relational database application. PhD Thesis, Department of Computer Science. University of Glasgow.
- Bottaci, L., and Jones, J. (1995). *Formal Specification using Z*. London: International Thomson Publishing Inc.
- Bourret, R.(2007). XML database product.
<http://www.rpbourret.com/xml/XMLDatabaseProds.htm>.
- Bowen, J. (2003). *Formal Specification and documentation using Z: A case study approach*, London: International Thompson Computer Press.
- Buneman, P., Davidson, B.S., Fan, W., Hara, S.C., and Tan, W.C. (2003). Reasoning about keys for XML. *Information System* 28(8), pp. 1037-1063.
- Buneman, P., Fan, W., Simeon, J., and Wienstein, S. (2001). Constraints for semistructured data and XML. *SIGMOD Record*, Vol 30, pp. 47-54.
- Calvanese, D., Giacomo, G.D., and Lenzerini, M. (1999). Representing and reasoning on XML documents: A Descriptive Logic Approach. *Journal of Logic and Computation*, Vol 9, pp. 295-318.

- Chaudhri, B.A., Rashid, A. and Zicari, R. (2003). *XML Data Management; Native XML and XML-Enabled Database Systems*. USA: Pearson Education.
- Chen, P. P., (1976). The entity-relational model: Towards a unified view of data. *ACM Transaction on Database System*, Vol 1(1), pp. 9-36.
- Choi, B. (2002). What are Real DTDs like. *Technical Reports, Department of Computer and Information Science, University of Pennsylvania* .
- Clack ,J. and Murata, M. (2001). RELAX NG Specification. December 2001, <http://www.oasis-open.org/committee/relax-ng/spec-20011203.html>.
- Clack, J. (2001). TREX - Tree Regular Expression for XML: language Specification. February 2001. . <http://www.thaiopensource.com/trex/spec.html>.
- Codd, E. (1970). A relational model of data for large shared data banks. *Communication of the ACM*, Vol 13 (6) , pp. 377-387.
- Codd, E. (1972). Further normalization of the database relational model. In *Database system, Computer Science Symposia Series 6*. Englewood Cliffs, New York: Prentice Hall.
- Codd, E. (1974). Recent investigation into relational database system. *Proc IFIP congress*, Stockholm, Sweden, pp 1017-1021.
- Connolly, T.M and. Begg, C.E (2002). *Database Systems: A practical approach to Design, Implementation and Management*. USA: Addison Wesley.
- Conforti,G., and Ghelli, G. (2003). Spatial tree logics to reason about semistructured Data. In *Proceeding of 11th Italian Symposium and Advance Database System*, Italy, pp. 37-48.
- Conrad,R., Scheffner,D., and Freytag, J. (2000). XML conceptual modelling using UML. In *Proceeding of the International Conference on Conceptual Modeling* , New York, USA, pp. 558-571.

- Coronato, A. and De Petro, G. (2010). Formal specification of wireless and pervasive healthcare applications. *ACM Transaction on Embedded Computing Systems*, Vol.10(1), Article 12. pp. 1-18.
- Cover, T., and Thomas, J. (1991). *Element of Information Theory*. New York: Wiley.
- Date, C. (2000). *An Introduction to Database Systems*. London: Addison Wesley.
- D'Inverno, M and Hu, M.J. (1997). A Z specification of the soft-link hypertext model. *ZUM'97: 10th International Conference of Z Users, Lecture Notes in Computer Science*, Reading, UK, pp. 297-316.
- D'Inverno, M, Justo, G.R. and Howell, P. (1991). A formal framework for specifying design methodologies. *Software Process: Improvement and Practice*, Vol 2(3), pp. 181-195.
- D'Inverno, M. and Luck, M. (1996). A formal view of social dependence networks. In *Distributed Artificial Intelligence Architecture and Modelling: Proceeding of the First Australian Workshops on Distributed Artificial Intelligence, Lecture Notes in Artificial Intelligence*, Springer, 1087, pp. 115-129.
- Diller, A. (2001). *Z: An Introduction To Formal Methods*. London: John Wiley and Sons.
- Dobbie, G., Xiaoying, W., Ling, T.W., and Lee, M.L. (2000). ORA-SS: An object-relationship-attribute model for semi-structured data. Technical Report, Department of Computer Science, National University of Singapore .
- Embley, D. and Mok, W.Y. (2001). Developing XML documents with guaranteed "good" properties. In *Proceedings of the 20th International Conference on Conceptual Modeling*, London, UK, pp. 426-441.
- Fagin, R. (1977). Multivalued dependencies and a new normal form for relational databases. *ACM Transaction on Database System*, Vol 2(3), pp. 262-278.

- Fan, W. and Libkin, L. (2002). On XML Integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3), pp. 386-406.
- Fan, W. and Simeon, J. (2000). Integrity constraints for XML. In *Proceedings of the Nineteenth ACM SIGMOD -SIGACT Symposium on Principle of Database Systems*, New York, USA, pp. 23-34.
- Fan, W. and Simeon, J. (2003). Integrity constraint for XML. *Journal of Computer and System Sciences*, Vol 66(1), pp. 254-291.
- Feng, L., Chang, E., and Dillon, T. (2002). A Semantic network-based design methodology for XML documents. *ACM Transactions on Information Systems*. Vol 20(4), pp. 390-421.
- Florescu, D. and Kossmann, D. (1999). Storing and querying XML data using RDMS. *IEEE Computer Society on Data Engineering*, pp. 27-34.
- Goldman, R., and Widom, J. (1997). Dataguides: enabling query formulation and optimization in semistructured database. In *Proceeding of the 23rd International Conference on Very Large Databases*, Athens, pp. 436-445.
- Gustas, R. (2010). A look behind conceptual modelling constructs in information system, analysis and design. *International Journal of Information System Modeling and Design*, Vol 1(1), pp 79-108.
- Hall, A. (1990). Seven myths of formal methods. *EEE Software*, pp 11-19.
- Hall, A. (1998). What does industry need from formal specification techniques?. In *Proceeding of the second workshop of Industrial Strength Formal Specification Techniques*, IEEE Computer Society, Washington DC, New York. pp. 2-7.
- Hall, A. (2000). Realising the benefit of formal methods. *Formal Methods and Software Engineering*, Lecture Notes Computer Science, Vol (3785). Springer. pp. 1-4.

- Hall, A. (2010), Z word tools. Retrieved on September 5, 2010 Accessed <http://http://sourceforge.net/projects/zwordtools/files/>.
- Halpin, T. (2010). Object Role Modelling: principle and benefit. *International Journal of Information System Modeling and Design*, Vol 1(1), pp 33-55.
- Halasz, F. and Schwartz, M.(1994). The dexter hypertext. *Communication of the ACM*, Vol. 37(2), pp.30-39.
- Hartmann, S. and Link, S. (2006). Deciding implications for functional dependencies in complex-value databases. *Theoretical Computer Science* , pp. 212-240.
- Hartmann, S. and Link, S. (2004). Multi-valued dependencies in the presence of list. *ACM Transaction on Database System*, pp. 330-341.
- Hartmann, S., Link,S., Schewe K.D. (2005). Functional dependency over XML documents with DTDs. *ACTA Cybern*, pp. 153-171.
- Hartmann, S. and Link, S. (2003). More functional dependencies for XML. *Lecture Notes in Computer Science*, pp. 355-369.
- Hegaret, P.(2002). *The W3C Document Object Model(DOM)*. Available at <Http://www.w3.org/2002/07/26/dom-article>.
- Hexthausen, A and Peleska, J.(2000). Formal development and verification of a distributed railway control system. *IEEE Transaction on Software Engineering*, Vol 26(8). pp. 687-701.
- Hunter, D. (2000). *Beginning XML*. USA:Wrox Press.
- Jacob, K. (2004). Integration of life science databases. *Biosilico* Vol 2(2) , pp. 61-68.
- Jacky, J. (1997). Specifying a safe critical control system in Z. *Journal of IEEE Transaction on Software Engineering*, Vol 21(2). pp 95-106.

- Jones, C.B. (1986). *Systematic software development using VDM*. Englewood Cliffs, New Jersey : Prentice-Hall
- Kanellakis, P. (1990). Elements of relational database theory. In *Handbook of Theoretical computer Science*, Vol B , pp. 1074-1144.
- Kim, S.W., Shin, P.S., Kim, Y.H., and Lim, H.C. (2002). A data model and algebra for document -centric XML document. *Lecture Notes in Computer Science*, Vol 2344 , pp. 714 -723.
- Kolahi, S. (2007). Dependency-preserving normalization of relational and XML data. *Journal of Computer and system Sciences*, Vol 73, pp. 636-647.
- Kolahi, S. and Libkin, L. (2006). On redundancy vs dependency preservation in normalization: an informatics-theoretic study of 3NF. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, USA, pp. 114-123
- Kolahi, S. and Libkin, L. (2007). XML design for relational storage. *Journal of Computer and System Sciences*, Vol 73(4), pp.636-647.
- Lange, D.B. (1990). A formal approach to hypertext using post-prototype formal specification. *Lecture Notes in Computer Science*, Vol 428, pp.99-121
- Lee, L.M., Ling, W.T., and Low, L.W. (2002). Designing functional dependencies for XML. In *Proceeding of the 8th International Conference on Extending Database Technology: Advance in Database Technology*, London, UK, pp. 27-36.
- Lee, S.J., Dobbie, G., Sun, J. and Groves, L. (2010). Theorem prover approach to semi-structured data design. *Formal Methods System Design*, Vol 37, pp. 1-60
- Lee, S.J., Sun, J., Dobbie, G., and Groves, L. (2009). Formal Verification of Semi-structured Data in PVS. *Journal of Universal Computer Science*, Vol 15(1), pp. 241-272
- Lee, S.J, Sun, J., Dobbie, G. and Li, Y.F. (2006) . A Z Approach in Validating ORA-SS Data Models. *Electronic Notes in Theoretical Computer Science*, Vol. 157, pp. 95-109.

- Lee, S.J, Sun, J., Dobbie, G., Groves, L., and Li, Y.F. (2008). Correctness Criteria for Normalization of Semi-structured Data. *In Proceeding 19th Australian Conference on Software Engineering*. Australia, pp. 248-257.
- Lee, S.Y., Lee, M.L., Ling, T.W., and Kalinichenko, L.A. (1999). Designing good semi-structured databases. *Lecture Notes in Computer Science*, Vol. 890, pp. 131-145.
- Leftonen, M. (2006). Preparing heterogeneous XML for full-text search. *ACM Transaction on Information System*, Vol 24(4) , pp. 455-474.
- Ley, M. (2002). *DBLP*. <http://www.informatik.uni-trier.de/~ley/db/index.html>.
- Libkin, L. (2007). Normalization theory for XML. *In Proceeding of the 5th International XML Database Symposium*, Austria, pp.1-13.
- Lightfoot, D. (1991). *Formal Specification Using Z*. London: The Macmillan Press Ltd.
- Ling, T.W, (1985). A normal form for entity-relationship diagram. *Proceeding 4th International Conference on Entity Relationship Approach*, pp. 24-35.
- Ling, T.W., Lee, M.L., and Dobbie, G. (2005). *Semistructured Database Design*, New York, USA: Springer.
- Ling T.W. and Yan, L.L. (1994). NF-NR: A practical normal form for nested relations. *Journal of Systems Integration*, Vol 4(4), pp. 309-340.
- Liu, F. , Li, C., Yu, J, (2011). Description of web service composition model based on Z notation. *In International Conference on Computer Science and Automatic Engineering (CSAE) ,IEEE* , Shanghai, pp. 587-591.
- Lu, J.J. and Renjen, S. (2005). Normalizing XML Schemas through relational. *43rd ACM Southeast Conference USA*, pp. 220-221.
- Luck, M. and D'Inverno, M.(1995). Structuring a Z specification to provide a formal framework for autonomous agent systems. *ZUM'95: 9th International Conference of Z Users, Lecture Notes in Computer Science*, Springer-Verlag. Berlin, pp.48-62.

- Lv, T., Gu, N., and Yan, P. (2004). Normal forms for XML documents. *Information and Software Technology*, pp. 839-846.
- Lv, T. and Yan, P. (2006). Mapping DTDs to relational schemas with semantic constraint. *Elsevier-Information and Software Technology* 48, pp. 245-252.
- Lv, T. and Yan, P. (2007). XML normal forms based on constraint_tree-based functional dependencies. *Lecture Notes in Computer Science*, Vol 4537, pp. 348-357.
- Ma, Z.M. and Yan, L. (2007). Fuzzy XML data modelling with UML and relational data models. *Elsevier: Data and Knowledge Engineering* (63), pp. 972-996.
- Mani, M., Lee, D., and Muntz, R. R. (2001). Semantic data modeling using XML Schemas. In *Proceeding of 20th International Conference on Conceptual Modelling*, London, UK, pp. 149-163.
- Mannila, H. and Raiha, K.J. (1989). Practical algorithms for finding prime attributes and testing normal forms. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGAT Symposium on Principles of Database Systems*, pp. 128-133.
- Martens, W., Neven, F., and Schwentick, T. (2007). Simple off the shelf abstraction for XML schema. *SIGMOD Record*, Vol 36(3), pp. 15-22.
- McHugh J, Abiteboul S, Quass D, and Wisdom J.(1997). Lore: a database management system for semistructured data. *SIGMOD Record*, Vol 26(3), pp. 54-66
- Melton, J. and Simon, A.R. (1993). *Understanding the new SQL: A Complete Guide*. USA: Morgan Kaufmann.
- Mian, N.A and Zafar, N.A.(2010). Key analysis of normalization process using formal techniques in DBRE. In *Proceeding of Second International Conference on Computer Engineering and Application*, IEEE Computer Society, pp. 370-374.
- Mok, W. Y.(2002). A comparative study of various normal forms. *IEEE Transaction on Knowledge and Data Engineering*, Vol 14, pp.369-385.

- Mok, W. Y. and Embley, D. (2006). On utilizing variables for specifying FDs in data-centric XML documents. *Data and Knowledge Engineering*, Vol 60, pp 494-510.
- Mok, W. Y., and Embley, D. (2006). Generating compact redundancy-free XML documents from conceptual-model hypergraphs. *IEEE Transaction on Knowledge. Data Data Engineering*. Vol 18(8), pp.1082-1096
- Mok, W.Y., Ng Y., and Embley, D. (1996). A normal form for precisely characterizing redundancy in nested relations. *ACM Transaction on Database System*, Vol 21(1), pp. 77-106.
- Moller, A. and Schwartzback, M. (2006). *An Introduction to XML and Web Technology*. England:Addison Wesley.
- Murata, M. (1999). Hedge automata: A formal model for XML schemata, Fuji Xerox Information System, Available at www.xml.gr.jp/relax/hedge_nice.html.
- Murata, M., Lee, D., Mani, M, and Kawaguchi, K. (2003). Taxonomy of XML Schema language using formal language. *ACM Transaction and Database*, Vol 45(7), pp 27-67.
- Nevan, F. (2002). Automata theory for XML researchers. *SIGMOD Record*, Vol 31(3) , pp. 39-46.
- Ozsoyoglu, M. and Yuan, L. (1989). On the normalization in nested relationa databases. In *Nested Relations and Complex Object* , Springer, pp. 243-271.
- Ozsoyoglu, M. and Yuan, L. (1987). A new normal form for nested relations. *ACM Transaction on Database Sytsem*, Vol 12(1), pp. 111-136.
- Pankowski, T. (2009). Transformation of XML data into XML normal form. *Informatica*, Vol 33, pp. 417-430.
- Papakonstantinou, Y., Molina,G. and Wisdom, J. (1995). Object exchange across heterogenous information sources. In *Proceeding of the Eleventh International Conference on Data Engineering*, Taipei, Taiwan, pp. 251-260.

- Partsch, H. A. (1990). *Specification and Transformation of Program*. New York: Springer-Verlag
- Paterno, F., Santoro, C. and Tahmassebi, L.(1998). Formal models for cooperative task, concept and application for route air traffic control. In *Proceeding of 5th International Workshop on Design and Specification*, New York, USA, pp. 1-10.
- Philippi, S. and Kohler, J.(2004). Using XML technology for the ontology based semantic integration of the life science databases, *IEEE Transaction Information Technology Biomed*, Vol 8(2), pp. 154-160.
- PIR International Protein Sequence Database.*
(<http://pir.georgetown.edu/pirwww/search/textpsd.html>).
- Powell, G. (2007). *Beginning XML Databases*. USA: Wiley .
- Ratcliff, B. (1994). *Introduction Specification using Z: A Practical Case Study Approach*. England: McGraw-Hill Student Company Europe.
- Rosen, K.H. (1995). *Discrete Mathematics and Its Applications*. Third Edition, New York: McGraw-Hill.
- Runapongsa, K., and Patel, J.M. (2002). Storing and querying XML data in object-relational DBMSs. *Lecture Notes in Computer Science* 2490, pp. 266-285.
- Saaltink, M. (1997). ZUM'97: Z formal specification notation. *Lecture Notes in Computer Science*, Vol 1212, pp. 72-75.
- Sahuguet, A. (2000). Everything you ever wanted to know about DTDs, but were afraid to ask. In *Proceesings of the International Conference on the Web and Databases*, Texas, pp. 69-74.
- Schewe, K. (2005). Redundancy, dependencies and normal forms for XML databases. *Sixteenth Australasian Database Conference(ADC2005)*, Australia, pp. 7-16.

- Schwentick, T. (2007). Automata for XML-Survey. *Journal of Computer System Sciences*, Vol 73(3), pp. 289-315.
- Simpson, A. (2002). *Discrete Mathematics by Example*, England:McGraw-Hill.
- Singh, M. and Patterh, M.S. (2010). Formal specification of common criteria based access control policy model. *International Journal of Network Security*, Vol. 11(3), pp. 139-148.
- Sommerville, I. (2010). *Software Engineering*, 9th Edition, USA:Addison Wesley.
- Spivey, J.M. (1988). *Understanding Z*. Cambridge: University Press,Cambridge.
- Spivey, J.M. (1992). *The Z notation: A reference manual*. International Series in Computer Science, Prentice Hall.
- Syukur, Z., Alias, N., Mohamed Halip, M.H. and Indrus, B. (2007) Formal validation on the safety of sack protocol using theorem proving technique. *Journal of Computer Science*, Vol 3(6). pp. 449-453.
- Tatarinov, I., Ives, Z., Halevy, A. and Weld, D. (2001). Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. Santa Barbara, Colifornia, pp. 412-424.
- Varlamis, I. and Vazirgiannis, M. (2001). Bridging XML-Schema and relational database. A system for generating and manipulating relational database using valid XML documents. *DogEng*, pp. 105-114.
- Vincent, M.W., and Levene, M. (2000). Restructuring partitioned normal relations without information loss. *SIAM Journal on Computing*, Vol 39(5), pp. 1550-1567.
- Vincent, M.W. and Liu, J.(2003). Multivalued dependencies in XML, *Lecture Notes in Computer Science 2712*, pp.4-18.

- Vincent, M.W., Liu, J., and Liu, C. (2004). Strong functional dependencies and their application to normal form in XML. *ACM Transactions on Database Systems*, Vol 29(3), pp. 445-462.
- Vincet, M., Liu, J., and Mohania, M. (2007). On the equivalence between FDs in XML and FDs in relations. *Acta Informatica*, pp. 2007-247.
- Wang, B. (1993). Integrating database and hypertext to support documentation environments. In PhD. Thesis. University of York.
- Wang, B. (1999). A hybrid system approach for supporting digital libraries. *International Journal on Digital Libraries*, Vol(2), Springer, pp. 91-110.
- W3C. (2010). XQuery: An XML Query Language, Available at [HTTP://www.w3.org/TR/2010/REC-xquery-20101214](http://www.w3.org/TR/2010/REC-xquery-20101214)
- W3C. (2001). XML Schema . W3C recommendation, Available at [HTTP://www.w3.org/TR/XMLschema](http://www.w3.org/TR/XMLschema).
- W3C. (1998). XML Specification DTD, available at <http://www.w3.org/XML/1998/06/xmlspec-report-1991010.html>.
- Wang, J. (2005). A comparative study of functional dependencies for XML. *Lecture Notes in Computer Science*, Vol 3399, pp. 308-319.
- Wang, J. and Topor, R. (2005). Removing XML data redundancies using functional and equality-generating dependencies. In *Proceeding of the 16th Australasian Database Conference*, Australia, pp. 65-74.
- Wang, L., Dobbie, G., Sun, J., and Groves, L.(2006). Validating ORA-SS data models using Alloy. In *Proceeding of the 2006 Australian Software Engineering Conference*, IEEE Computer Society, Australia, pp 231-242.
- Woodcock, J. and Davies, J. (1996). *Using Z:Specification, Refinement and Proof*. International Series in Computer Science, England: Prentice Hall.

- Wordsworth, J. (1992). *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*. England: Addison-Wesley.
- Wu, X., Ling, T.W., Lee, M.L., and Dobbie, G. (2001). Designing semistructured databases using ORA-SS model. In *Proceeding International Conference on Web Information System Engineering*, Kyoto, Japan, pp. 171-180.
- Wu, Y. (2004). Normalization design of XML database schema for eliminating redundant schemas and satisfying lossless join. In *International Conference on Web Intelligence IEEE*, Beijing, China, pp. 660-663.
- Wuwongse, V., Akama, K., Anutariya, C., and Nantajeewarawat, E. (2003). A data model for XML Databases. *Journal Intelligence Information System*, Vol 20(1), pp. 63-80
- Wyke, R. A., and Watt, A. (2002). *XML Schema Essentials*. New York: Wiley.
- Yu, C. and Jagadish, J.H. (2006). Efficient discovery of XML data redundancies. *VLDB'06 Korea*: ACM, pp. 103-114.
- Yu, C. and Jagadish, J.H. (2008). XML schema refinement through redundancy detection and normalization. *The VLDB Journal*, pp. 203-223.
- Yuliana, O.Y. and Chittayasothorn, S. (2005). XML schema re-engineering using a conceptual schema approach. *International Conference on Information Technology: Coding and Computing*. Las Vegas, Nevada, pp 25-31.

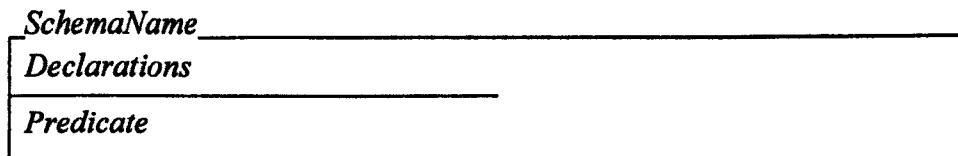
Appendix A

Glossary of Z Notation

This appendix presents a glossary of the Z notation used in this thesis. The glossary is based on the glossary of Z notation presented in Spivey (1982)

A.1 Schema Notation

Schema is the basic unit in Z. It contains a two-dimensional graphical notation for grouping together. It has the following basic form:



The *declaration* part is used to define variable names, each with a type. The *predicate* part presents a relationship between the variables declared in the *declaration* part

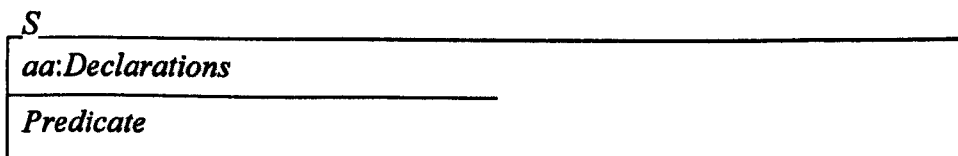
The schema also can be written in a linear form as follows:

$$\textit{SchemaName} \triangleq [\textit{Declaration} \mid \textit{Predicate}]$$

A.1.1 The Δ and Ξ Conventions in Schema

The Δ notation is used for a schema to represent the change of state. Suppose we have a schema S. In a linear form the schema S can be written as follows:

For example $\Delta S \triangleq S \wedge S'$



Then ΔS can be represented as follows:

ΔS
S
S'
<i>Predicate</i>

In a linear form the schema S can be written as follows:

$$\Delta S \triangleq S \wedge S'$$

The Ξ notation is used for a schema to represent no change of state

ΞS
ΔS
$aa = aa'$

A.2 Axiomatic Definitions

According to Spivey's definition (Spivey, 1982), an axiomatic schema introduces a global variable definition. It can be used throughout the whole specification. It has the following basic form:

<i>Declarations</i>
<i>Predicate</i>

A.3 Set in Z

Z	the set of integers (whole numbers)
N	the set of natural numbers (≥ 0)
N_1	the set of positive natural numbers (≥ 1)
$t \in S$	t is an element of S
$t \notin S$	t is not an element of S
$S \subseteq T$	S is contained in T
$S \subset T$	S is strictly contained in T

\emptyset or $\{\}$	the empty set
$\mathbb{P}S$	Power set: the set of all subsets of S
$\mathbb{F}S$	the set of finite subsets of S
$S \cup T$	Union
$S \cap T$	Intersection
$S \setminus T$	Difference
\neq	not equal

A.4 Relations in Z

X, Y are sets and R is the name of a relation

$X \times Y$	the set of ordered pairs of X and Y
$X \leftrightarrow Y$	the set of relations from X to Y
$x \mapsto y$	(x, y)
$\text{dom } R$	the domain of a relation $= \{ x: X \mid (\exists y: Y . x R y) \bullet x \}$
$\text{ran } R$	the range of a relation $= \{ y: Y \mid (\exists x: X . x R y) \bullet y \}$
$R(S)$	the relational image of S in R
$S \triangleleft R$	the relation R domain restricted to S
$S \triangleright R$	the relation R range restricted to S
$S \triangleleft R$	the relation R domain anti-restricted to S
$S \triangleright R$	the relation R range anti-restricted to S
R^+	the repeated self- composition of R
$R \sim$	the inverse of R

A.5 Function in Z

$X \rightarrow Y$	the set of partial functions from X to Y $= \{ f: X \leftrightarrow Y \mid \text{ran } f \in X \rightarrow Y \wedge \forall y: \text{ran } f \bullet (\exists_1 x: \text{dom } f \bullet (x, y) \in f) \}$
$X \rightarrow Y$	the set of total functions from X to Y $= \{ f: X \rightarrow Y \mid \text{dom } f = X \bullet f \}$
$X \rightarrow Y$	the set of total injection functions from X to Y

fx or $f(x)$ the function f applied to x
 $f \oplus g$ functional overriding $\equiv (\text{dom } g \triangleleft f) \cup g$

A.6 Sequences in Z

Seq X The set of sequence whose elements are drawn from X
 $\equiv \{ S:N \rightarrow X \mid \text{dom } S = 1..#S \}$

#S The length of the sequence S

$\langle x_1, \dots, x_n \rangle$ $\equiv \{ 1 \mapsto X_1, \dots, n \mapsto X_n \}$

Head S $\equiv S1$

Last S $\equiv S \#S$