Extending XML-RL With Updates

by

Li Lu

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Science
Information and System Science

School of Mathematics and Statistics

Carleton University

Ottawa, Ontario

April, 2003

© Copyright

2003, Li Lu

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-83503-0

Our file Notre référence

ISBN: 0-612-83503-0

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

Extending XML-RL With Updates

submitted by

Li Lu

Dr. Cyril W.L. Garner
(Director, School of Mathematics and Statistics)

Dr. Mengchi Liu
(Thesis Supervisor)

Carleton University

April, 2003

Abstract

XML has been used extensively for information representation and exchange over the Web. How to access XML data has been a hot topic with many XML query languages published. How to update XML data is becoming an important issue as the role of XML has expanded beyond traditional applications. So far, several languages have been proposed for updating XML data, but they have two major shortcomings. One is that these updating languages are based on low-level graph-based or tree-based data models. Update requests are thus expressed in a nonintuitive and unnatural way, and update statements are too complicated to comprehend. The other is that there is still no consensus about the logical foundation for XML update. This thesis presents a novel declarative XML update language based on the high-level data model, the XML-RL data model. We formally define the syntax and semantics of the XML-RL update language based on the concepts of logical programming, giving our language a logical foundation. We also design the architecture of XML-RL system and implement a prototype system. Compared with other existing XML update languages, ours has the following features. First, our language is based on a high-level data model rather than graph-based or tree-based data models. Therefore, update requests can be expressed in a more intuitive and natural way in our language than in the other languages. Specifically, our language can express complex update requests to multiple levels of a hierarchy in a simple and uniform manner. Second, our language directly supports

the functionality of updating complex objects while graph-based or tree-based update languages do not support these operations. Third, our update language supports both ordered or unordered XML data while some languages, which are based on relational or object relational databases only support unordered XML data. Lastly, our language modifies object name, object value and object in a unified way by introducing three kinds of logical binding variables: *name variable*, *value variable*, and *object variable*.

Acknowledgements

First of all, I would like to express my sincere thanks to Professor Mengchi Liu, my supervisor, for his academic and technical guidance, valuable and profound discussions on both my research and thesis, and also for his financial support during my study.

I also want to thank Professor Guoren Wang, a visiting Professor in the School of Computer Science, for his great technical help and valuable discussions and suggestions on my research work.

I would like to thank Lingyan Xu, Xiaofei Jia, Tao Fu and other colleagues for their assistance.

I would also like to recognize the efforts of the Examination Committee for evaluation of this work.

Dedication

To my beloved husband Shaoying Wei, for his understanding and support during the whole period of this study, and to my lovely daughter Yin Wei, for the happiness and relaxation she has brought to my life.

Contents

Abstract	iii
Acknowledgements	v
Dedication	vi
Table of Contents	vii
List of Tables	x
List of Figures	xi
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Objective and Contribution	4
1.3 Outline	5
Chapter 2 Background	6
2.1 The Emergence of XML	6
2.2 XML Markup	7
2.3 DTD	9
2.4 DOM Model	11

Chapter 3	Related Work	13
3.1	Database Point of View	13
3.1.1	Oracle9i XML SQL Utility	14
3.1.2	DB2 XML Extender	16
3.1.3	Comments on Oracle9i XSU and IBM DB2 Extender	20
3.2	Semi-structured Point of View	21
3.2.1	Data Models for XML as Semi-structured Data	21
3.2.2	XML-GL	22
3.2.3	Lorel	25
3.2.4	Comments on XML-GL and Lorel	27
Chapter 4	XML-RL Query Language Overview	28
4.1	Data Model	28
4.2	Mapping XML to XML-RL Model	30
4.3	Query Language	32
Chapter 5	XML-RL Update Language	34
5.1	Syntax	35
5.1.1	Schema Syntax	35
5.1.2	Database Syntax	38
5.1.3	Syntax of XML-RL Update Language	41
5.2	Variables	44
5.3	Example of Updates	47
5.3.1	Insertion	47
5.3.2	Deletion	51
5.3.3	Modification	52

5.3.4	Combination	54
5.4	Semantics of Update Language	56
5.4.1	Semantics of Schema	56
5.4.2	Semantics of Database	59
5.4.3	Semantics of Update Language	63
Chapter 6	Design and Implementation of XML-RL System	69
6.1	Requirements of XML-RL System	69
6.2	XML-RL System Architecture	70
6.2.1	User Interface	70
6.2.2	Web Server	72
6.2.3	Language Processor	74
6.3	Query/Update Processor	76
6.3.1	Command Handler	77
6.3.2	XML Loader	77
6.3.3	Operation Processor	78
Chapter 7	Conclusion and Future Work	82
7.1	Conclusion	82
7.2	Future Work	83
	Bibliography	84
Appendix A	Syntax of XML-RL Language	89

List of Tables

3.1	FXTRADE Table	16
3.2	AccountType	16

List of Figures

2.1	A DOM Representation of Part Data in Example 1	12
3.1	XML-GL Query Example	24
3.2	XML-GL Update Example	24
3.3	Lorel XML-based Graph Model	26
5.1	An Example of a Complex Element Object	51
6.1	XML-RL System Architecture	71
6.2	XML-RL Command Line Interface	72
6.3	XML-RL Language Processor	75
6.4	XML-RL Query/Update Processor	76
6.5	Variable Guide	78
6.6	Result Repository	80
6.7	Class Structure of Operation Processor	81

Chapter 1

Introduction

1.1 Motivation

Data stored in different forms can be classified into two classes: structured and unstructured. Structured data obeys a rigid schema and can be found in databases. It can be queried and manipulated using some various languages based on its structure. Unstructured data is normally found in a textual file and does not conform to a pre-defined internal structure, therefore it cannot be queried and manipulated using a machine understandable language. As the amount of data available on-line grows rapidly, more and more data is semi-structured. This kind of data is neither completely unstructured, nor strictly structured, but has a logical structure nevertheless. Since it has some kind of structure, it presents the possibility to be queried and manipulated using a language. However, because semi-structured data does not conform to a standard database framework, it is impossible to simply utilize an existing database language to query and update the data. XML [16], as one form of semi-structured data, is gaining more and more attention. XML is a general markup language for documents, thus it can be adopted not only for web information, but also

document information, i.e. any kind of semi-structured information. This feature is very beneficial to a variety of richly structured applications over the web. Thus, as an emerging standard for data representation and exchange on the web, XML is being adopted by more and more applications as their means of information description. Therefore, how to retrieve XML data becomes an important issue. So far, several languages have been proposed, such as XPath [6], XML-QL [35], XQL [14], Quilt [28], W3C XQuery [9], YATL [36] and XML-RL [11] etc. Although retrieval is important for XML, the manipulation of XML documents is also important for many XML applications. The modification operations are used to change document structures and contents. A useful XML document manipulation language should include both the ability to query and the ability to modify XML documents.

In the past several years, a few languages for both querying and updating XML data have been proposed, such as Oracle9i XML SQL Utility [20], DB2 XML Extender [22,23], Lorel [2,3], XML-GL [5], CXQuery [7], Updating XML [32], and XPathLog [12]. Some of them are inspired directly by XML and treat an XML document as a semi-structured database, while some share the commonalities with database update languages and treat an XML document as a relational database or an object relational database. For example, IBM DB2 Extender and Oracle9i XSU support the decomposition of XML data into relational storage or object relational storage respectively, and the user would then need to work with relational data as with any other relational data. In particular, any update on XML data has to be specified using SQL, and then be executed on the underlying relational database. This requires users to be aware of not only the underlying storage system, but also the particular mapping between the XML model and the storage model. In other words, there is a lack of abstraction for specifying native updates to XML data independent of the

underlying storage models. On the contrary, Lorel, XML-GL, CXQuery, Updating XML and XPathLog treat XML documents as semi-structured databases. They are based on graph-based or tree-based data models, which are primarily for machine representation. As a result, these querying and updating languages are too complicated for end users, as their query statements and update requests are expressed in a complicated and unnatural way from a database point of view. This would hinder the development of the design and standardization of document manipulation languages. Meanwhile, existing update languages relying on either a traditional database or a semi-structured database having quite a few preliminary update features and so many issues need deeper study for complex updating, such as meta-data updating, multiple levels updating, multiple documents updating and IDREFs updating etc.

XML manipulation languages are currently still in their infancy. Discussion is still going on with no standards available. An adequate update facility function is critical for the development and standardization of XML languages.

Recently, a novel data model for XML was proposed in [10], in which XML data is viewed in a way similar to complex object models [13]. Based on this high-level data model, a rule-based declarative XML query language, called XML-RL, has been presented in [11]. Benefiting from its underlying high level data model and declarative features, XML-RL possesses the potential to become a fully functional XML manipulation language. However, before it can really reach this objective, some enhancements are required, especially in its update features as updating the content of a database is one of the most fundamental operations that a database system must provide. Consequently, extending the XML-RL query language and providing a simple, powerful, and high-level update language to explore the full power of XML, are of great practical significance.

1.2 Objective and Contribution

This thesis focuses on the design of the XML-RL update language, and the development of techniques to process updates efficiently. Our objectives are to formally define the syntax and semantics of the XML-RL update language, and to design an architecture for the XML-RL system, and to implement a prototype system.

This thesis provides solutions for the above objectives and has the following contributions:

1. Redefining the objects, terms and expressions, and then defining the formal syntax and semantics of the schema of the database, so that the existing XML document querying can smoothly integrate with our newly added XML document updating.
2. Defining the formal syntax and semantics of the update language based on logical programming, presenting a minimal set of primitive update operations to fulfill the update requirements. All valid changes to XML data can be specified by one or a sequence of our primitive update operations.
3. Expressing complex update requests in a straightforward way.
4. Supporting updates to multiple documents. The update operation can be applied to more than one XML document at the same time. This provides a natural way to exchange data between XML documents, even if these documents reside across a network.
5. Supporting the functionality of directly updating complex objects.
6. Supporting updates in both ordered and unordered XML data.

7. Modifying both meta data and data in a unified way by introducing three kinds of logical binding variables: *object variable*, *value variable*, and *name variable*.
8. Implementing a prototype system for the XML-RL language to show its basic functionality.

Two papers from parts of this thesis have been accepted by international conferences, one in CAiSE'03 [40], another in IDEAS (2003) [41].

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 gives background information on XML. Chapter 3 reviews several query and update languages for XML, including Lorel [3], Oracle 9i XML SQL Utility [20], XML-GL [5] and DB2 XML Extender [22,23]. Chapter 4 introduces the XML-RL data model [10] and its query language [11]. Chapter 5 presents the formal syntax of the schema, the database, and the XML-RL update language. We illustrate the update language by a number of examples, including insertion, deletion, and replacement of element and attribute objects. Also, we formally describe the semantics of the XML-RL update language, including well-formed schema, well-formed databases and various update operations. Chapter 6 describes the architecture of the XML-RL system and the implementation issues for the XML-RL prototype system. The last chapter concludes the thesis and discusses future work.

Chapter 2

Background

2.1 The Emergence of XML

In 1986 the Standard Generalized Markup Language (SGML) [15] became an international standard for defining descriptions of the structure and content of different types of electronic documents. SGML, the “mother tongue” of HTML [17] and XML [16,27], is used for describing thousands of different document types in many fields of human activity. However, various barriers exist in delivering SGML over the Web. These barriers include the lack of widely supported Stylesheets, complex and unstable software because of SGML’s broad and powerful options, and obstacles to interchange of SGML data because of varying levels of SGML compliance among SGML software packages. These difficulties have condemned SGML to being a successful niche technique rather than a mainstream tool.

HTML, a subset of SGML, is the most frequently used document type on the Web. It defines a single, fixed type of document with markup that lets us describe a common class of simple office style reports, with headings, paragraphs, lists, illustrations, etc, and some provisions for hypertext and multimedia. It was defined to allow the

transfer, display and linking of documents over the Internet and is the key enabling technology for the World Wide Web. But HTML was designed as a markup language with simple structures and strong emphasis on formatting, and was weak for encoding content. It was not designed to encode the structure and semantics needed for complex applications.

Because of the lack of SGML support in mainstream Web browsers, most applications that deliver SGML information over the Web have to convert SGML to HTML. This down-translation removes much of the intelligence of the original SGML information. That lost intelligence virtually eliminates information flexibility and poses a significant barrier to reuse, interchange, and automation. For this reason, an eXtensible Markup Language (XML) was developed by the World Wide Web Consortium (W3C) in 1996. XML is a highly functional subset of SGML. The purpose of XML is to specify a SGML subset that works very well for delivering SGML information over the Web. When mainstream Web browsers support XML, it is believed that it will be very easy to publish SGML information on the Web.

2.2 XML Markup

XML is a markup language for documents containing structured information. XML can be used to structure text in such a way that it is readable by both humans and machines. It also presents a simple format for the exchange of information across the Internet.

A markup language is a mechanism to specify structures in a document. The XML specification defines a standard way to add markup to documents. An XML document is composed of markup and content. There are six kinds of markup that can occur in an XML document: elements and attributes, entity references, comments,

processing instructions, marked sections, and document type declarations.

XML markup focuses on the description of information structure and content as distinct from its presentation. The data structure and its syntax are defined in a Document Type Definition (DTD) specification [18], which is a derivative of SGML, and defines a series of tags and their constraints. In contrast to information structure, the presentation issues are addressed by XSL (XML Style Language) [25], which is also a W3C standard for expressing how XML based data should be rendered. With XSL, XML has a potential to be used as an exchange format for general structured data, and to increase the productivity of authors, and to enhance maintenance while retaining the features that HTML has provided.

Example 1. The following is a fragment of an XML document called *csdept.xml*, which consists of two *faculty* elements, one *staff* element and three *student* elements.

```
<csdept>
  <faculty id = "F100", supervisee = "S200 S400">
    <name>
      <firstname> Andrew </firstname>
      <lastname> Castro </lastname>
    </name>
    <position> Associate Professor</position>
    <salary> $50,000 </salary>
  </faculty>
  <faculty id = "F200">
    <name>
      <firstname> Bob </firstname>
      <lastname> Smith</lastname>
    </name>
    <position> Associate Professor</position>
    <salary> $40,000 </salary>
  </faculty>
  <staff id = "SF100">
    <name>
      <firstname> Linda </firstname>
      <lastname> White </lastname>
```

```

        </name>
        <email> linda@scs.carleton.ca</email>
    </staff>
    <student id = "S200", supervisor = "F100">
        <name>
            <firstname> Alisar </firstname>
            <lastname> Smith </lastname>
        </name>
        <program> Software Engineering </program>
    </student>
    <student id = "S400", supervisor = "F100">
        <name>
            <firstname> Jack </firstname>
            <lastname> Weiss </lastname>
        </name>
        <program> Software Engineering </program>
    </student>
    <student id = "S600">
        <name>
            <firstname> Mary</firstname>
            <lastname> Lee </lastname>
        </name>
        <program> Hardware Engineering </program>
    </student>
</scdept>

```

2.3 DTD

XML documents can be classified into two categories: wellformed and valid. An XML document is wellformed if it obeys the syntax of XML. For example, non-empty tags must be properly nested and each nonempty start tag must have the corresponding end tag. A wellformed document is valid if it conforms to a DTD [18]. A DTD is a file (external, included directly in the XML document, or both) which contains a formal definition of a particular type of XML documents. A DTD states

what names can be used for element types, where they may occur, how each element relates to the others, and what attributes an element may have.

An important use of DTD is document validation: many current tools allow the user to verify if an XML document conforms to a DTD. Unfortunately, DTD is quite limited. It does not specify any concrete (basic) data type and the contents of documents are handled as text strings. The need for data types is not only important for validation, but also used by all query languages in order to perform computations and comparisons between different values. XML Schema [19] is an ongoing work that enriches the description of the context semantics by allowing the definition of data types and constraints on the domain of such data types. The XML element content and attribute values content can be then validated with the type of data to which they are declared to belong.

Example 2. The following shows an example of DTD for *csdept.xml*. The DTD specifies a *csdept* element with at least one *faculty*, at least one *student*, and zero or more *staff*. A *faculty* element has an *ID* attribute and an *IDREFS* attribute, a *name*, a *position*, and a *salary* element. A *staff* element has an *ID* attribute, a *name*, and an *email* element. A *student* has an *ID* attribute and an *IDREF* attribute, a *name*, and a *program* element. A *name* element has a *firstname* and a *lastname* element, and zero or one *middlename* element. The data type of *firstname*, *lastname*, *middlename*, *position*, *salary*, *email* and *program* is *#PCDATA*, that is parsed characters.

```
<DOCTYPE csdept[
  <!ELEMENT csdept (faculty+ | staff* | student+)>
  <!ELEMENT faculty (name, position, salary)>
  <!ELEMENT name (firstname, middlename?, lastname)>
  <!ELEMENT firstname (#PCDATA)>
  <!ELEMENT middlename (#PCDATA)>
  <!ELEMENT lastname (#PCDATA)>
  <!ELEMENT position (#PCDATA)>
```

```

<!ELEMENT salary (#PCDATA)>
<!ATTLIST faculty id ID #REQUIRED>
<!ATTLIST faculty supervisee IDREFS #IMPLIED>
<!ELEMENT staff (name, email)>
<!ELEMENT email (#PCDATA)>
<!ATTLIST staff id ID #REQUIRED>
<!ELEMENT student (name, program)>
<!ELEMENT program (#PCDATA)>
<!ATTLIST student id ID #REQUIRED>
<!ATTLIST student supervisor IDREF #IMPLIED>
]>

```

2.4 DOM Model

The W3C DOM, Document Object Model [24], is a platform-independent and language-neutral interface for representing and manipulating an XML or HTML document. The W3C DOM includes a model for how a standard set of objects representing HTML and eXtensible Markup Language (XML) documents is combined, and an interface for accessing and manipulating them.

The DOM specifies a tree-based representation for an XML document. A top-level document instance is the root of the tree, and has a single child which is the top-level element instance; this element has children nodes representing the content and any sub-elements, which may have further children, and so forth. The DOM also provides the API for a user to traverse the resulting tree, access element and attribute values, insert and delete nodes, and convert the tree back into XML. Figure 2.1 is a DOM tree which represents parts of the data in Example 1.

The DOM is useful for modifying XML documents, because users can create a DOM tree, modify it by adding new nodes and moving subtrees around, and subsequently produce a new XML document as output. Users can also construct a DOM

Figure 2.1: A DOM Representation of Part Data in Example 1

tree themselves, and convert it to XML. This is often a more flexible way of producing XML output than simply writing `<tag1>...</tag1>` to a file.

While the DOM does not require the entire tree be resident in memory at one time, the DOM implementation currently does keep the whole tree in RAM. It is possible to write an implementation that stores most of the tree to disk or a database and reads in new sections as they are accessed, but this has not been done yet. This means there may not have been enough memory to process very large documents as a DOM tree. A SAX [26] handler, on the other hand, can potentially churn through amounts of data far larger than the available RAM. SAX is a standard interface for eventbased XML parsing: it reports parsing events (such as the start and end of elements) directly to the application through callbacks and does not build a tree for the whole document.

Chapter 3

Related Work

There have been several languages that support both query and update operations to XML documents. This chapter reviews four of them that are considered to be the most important and promising ones.

3.1 Database Point of View

XML document as a data container can be decomposed and then stored in relational or object relational tables, with the XML tags mapped to their respective columns in the database tables. This treatment provides the possibility of utilizing the matured and standardized SQL query language to manipulate XML documents based on the corresponding relational or object relational database representation. The decomposed XML document can then be regenerated into a composed XML document as its original form after the operations are done.

There are various ways to solve the problem of effective, automatic conversion of XML data into and out of relational databases or object relational databases, and DTD into the database schema. Some vendors such as Oracle and IBM have developed tools to assist in converting XML documents into relational tables.

3.1.1 Oracle9i XML SQL Utility

Oracle9i XML SQL Utility (XSU) [20] models provide SQL-to-XML to convert between database tables and XML documents by modeling XML document elements as a collection of nested tables. Enclosed elements are modelled by employing the Oracle Object Datatype. If the XML and object-relational model in the database are not completely matched, "XML-to-SQL" might require either data model amending (converting it from relational into object-relational) or restructuring the original XML document. XSU's XML-to-SQL mapping can extract data from an XML document and apply this data to updating or deleting values of the appropriate columns, or using a canonical mapping, by inserting the data into the appropriate columns of a table. We use the following examples to illustrate its functionality.

An XML to SQL mapping is modelled by an object-relational model construction rule as follows: XML attributes are ignored; an atomic element with its name and content is mapped onto table columns; a nested XML element is mapped onto an object reference of the appropriate type. Mapping rules are implicitly embedded in the database model.

Example 3. Consider the following DTD and XML document.

```
<!ELEMENT FXTRADE      (CURRENCY1,CURRENCY2,AMOUNT,SETTLEMENT,ACCOUNT)>
<!ELEMENT ACCOUNT      (BANKCODE,  BANKACCT)>
<!ELEMENT CURRENCY1    (#PCDATA)>
<!--ATTLIST CURRENCY1  e-dtype      NMTOKEN    #FIXED  "string"
                      e-dsize      NMTOKEN    #FIXED  "3">
<!ELEMENT CURRENCY2    (#PCDATA)>
<!--ATTLIST CURRENCY2  e-dtype      NMTOKEN    #FIXED  "string"
                      e-dsize      NMTOKEN    #FIXED  "3">
<!ELEMENT AMOUNT        (#PCDATA)>
<!--ATTLIST AMOUNT     e-dtype      NMTOKEN    #FIXED  "decimal">
<!ELEMENT SETTLEMENT    (#PCDATA)>
<!--ATTLIST SETTLEMENT e-dtype      NMTOKEN    #FIXED  "date">
```

```

<!ELEMENT BANKCODE      (#PCDATA)>
<!ATTLIST BANKCODE      e-dtype      NMTOKEN      #FIXED      "string">
<!ELEMENT BANKACCT      (#PCDATA)>
<!ATTLIST BANKACCT      e-dtype      NMTOKEN      #FIXED      "string">

```

FXTRADE.xml.

```

<FXTRADE>
  <CURRENCY1>GBP</CURRENCY1>
  <CURRENCY2>JPY</CURRENCY2>
  <AMOUNT>10000</AMOUNT>
  <SETTLEMENT>20010325</SETTLEMENT>
  <ACCOUNT>
    <BANKCODE>812</BANKCODE>
    <BANKACCT>00365888</BANKACCT>
  </ACCOUNT>
</FXTRADE>

```

The following are the corresponding table schema in the object relational database:

```

CREATE TABLE FXTRADE {
  CURRENCY1      CHAR (3),
  CURRENCY2      CHAR (3),
  AMOUNT         NUMERIC (18,2),
  SETTLEMENT     DATE,
  ACCOUNT        AccountType // object reference}

```

```

CREATE TYPE AccountType as OBJECT {
  BANKCODE      VARCHAR (100),
  BANKACCT      VARCHAR (100)}

```

```

CREATE TABLE AccountType {
  BANKCODE      VARCHAR (100),
  BANKACCT      VARCHAR (100)}

```

The corresponding databases generated from the above schema are in Table 3.1 and Table 3.2. The ACCOUNT in the above schema is modelled as an object reference of type AccountType.

CURRENCY1	CURRENCY2	AMOUNT	SETTLEMENT	ACCOUNT
GBP	JPY	10000	20010325	AccountType

Table 3.1: FXTRADE Table

BANKCODE	BANKACCT
812	00365888

Table 3.2: AccountType

After the database is created, we can query and update the database by using the SQL language. For example, to insert a new row element that contains values of CURRENCY1, CURRENCY2, AMOUNT, use the form:

```
insert into FXTRADE (CURRENCY1, CURRENCY2, AMOUNT)
values (ABF,FGH, 20000)
```

The following statement updates the AMOUNT in the FXTRADE table.

```
update FXTRADE
set AMOUNT=30000
where ACCOUNT=AccountType(812,00365888)
```

3.1.2 DB2 XML Extender

IBM DB2 XML Extender [22, 23] provides the support for the XML-based operations that stores or retrieves XML documents from a DB2 database. It has two access and storage methods for using DB2 as an XML repository:

- XML column: stores and retrieves entire XML documents as DB2 column data
- XML collection: decomposes XML documents into a collection of relational tables, or composes XML documents from a collection of relational tables

The first method is used to store intact XML documents in DB2. XML column works well for archiving documents. The documents are inserted into columns that

are enabled for XML and can be selected, updated and retrieved. The element and attribute data can be mapped to DB2 tables (side tables), which in turn can be indexed for fast structural search. This method is suitable for retrieving XML data.

The second method is used to map XML document structures to DB2 tables so that users can either compose XML documents from existing DB2 data, or decompose (store un-tagged element or attribute content) XML documents into DB2 data. This method is suitable for modifying XML data, therefore, we use the second method to illustrate its functionality.

To create a database for an XML document, first, DTDs are stored in the DTD repository, and a DB2 table called DTD_REF whose schema name is *db2xml*. Each DTD in the DTD_REF table has a unique ID. The mapping between the database tables and a document is defined by means of a Data Access Definition (DAD) file. The DAD refers to a processed document DTD, thus providing a bridge between an XML document, its DTD, and mapping rules onto database tables.

Example 4. The following is an example of DAD for FXTRADE.xml in Example 3.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "dad.dtd">
<DAD>
  <dtdid>FXTRADE.DTD</dtdid>
  <validation>YES</validation>
  <Xcollection>
    <prolog>?xml version="1.0"?</prolog>
    <doctype>!DOCTYPE FXTRADE FXTRADE.DTD </doctype>
    <root_node>
      <element_node name="FXTRADE">
        <RDB_node>
          <table name="FXTRADE"/>
          <table name="ACCOUNT" key="ID"/>
          <condition>
            FXTRADE.ACCOUNT=ACCOUNT.ID
```

```

        </condition>
    </RDB_node>
    <element_node name="CURRENCY1">
        <text_node>
            <RDB_node>
                <table name="FXTRADE"/>
                <column name="CURRENCY1" type="CHAR(3)"/>
            </RDB_node>
        </text_node>
    </element_node>
    <element_node name="CURRENCY2">
        <text_node>
            <RDB_node>
                <table name="FXTRADE"/>
                <column name="CURRENCY2" type="CHAR(3)"/>
            </RDB_node>
        </text_node>
    </element_node>
    <element_node name="AMOUNT">
        <text_node>
            <RDB_node>
                <table name="FXTRADE"/>
                <column name="AMOUNT" type="DECIMAL(18,2)"/>
            </RDB_node>
        </text_node>
    </element_node>
    <element_node name="SETTLEMENT">
        <text_node>
            <RDB_node>
                <table name="FXTRADE"/>
                <column name="SETTLEMENT" type="DATE"/>
            </RDB_node>
        </text_node>
    </element_node>
    <element_node name="ACCOUNT">
        <element_node name="BANKCODE">
            <text_node>
                <RDB_node>
                    <table name="ACCOUNT"/>
                    <column name="BANKCODE"

```

```

        <type="VARCHAR(100)"/>
    </RDB_node>
</text_node>
</element_node>
<element_node name="BANKACCT">
    <text_node>
        <RDB_node>
            <table name="ACCOUNT"/>
            <column name="BANKACCT"
                <type="VARCHAR(100)"/>
            </RDB_node>
        </text_node>
    </element_node>
</element_node> <!--end of Account element-->
</element_node> <!-- end of FxTrade element -->
</root_node>
</Xcollection>
</DAD>

```

DAD defines a mapping between XML elements and relational database columns by employing `element_node` to `RDB_node` associations. The top_level element-node `FXTRADE` is defined as a join between tables `FXTRADE` and `ACCOUNT`, with field `ID` in `ACCOUNT` table as a primary key. Child element `CURRENCY1` is mapped onto field `CURRENCY1` in the table `FXTRADE`, and so on. Child elements of `ACCOUNT`, `BANKCODE`, and `BANKACCT` respectively are defined in the corresponding columns of `ACCOUNT` table. Atomic XML elements are tagged in DAD as `text_node`. In the example above all elements, with the exception of `FXTRADE` and `ACCOUNT`, are atomic.

After the XML document is decomposed as a DB2 table, SQL can be used to query and update it.

Example 5. Finding all amounts having a value over 10000.

```

select AMOUNT
from FXTRADE F, ACCOUNT A

```

```
where F.AMOUNT > 10000
and F.ACCOUNT=A.ID
```

Example 6. The following is an update example that changes settlement value.

```
update FXTRADE
set SETTLEMENT = 20010327
where ACCOUNT.BANKCODE = 812
and FXTRADE.ACCOUNT=ACCOUNT.ID
```

After the operations are completed, the resulting XML document can be extracted from the DB2 table.

Both storing XML into a database and extracting XML from a database rely on mapping rules defined in the DAD.

3.1.3 Comments on Oracle9i XSU and IBM DB2 Extender

For both Oracle XSU and IBM DB2 Extender, we may use their database query and update facilities to manipulate XML documents. For users, there is no learning gap. And the language itself, SQL, has been proven highly efficient and reliable. However, in general, this is not a perfect approach for XML updating since it has the following drawbacks:

- Both approaches must rely on a DBMS which is mostly platform dependant. This is incompatible with the motivation of XML which is designed to be platform independent.
- Before a user can use SQL to query the document, the document must be converted to database tables. After the operation, it must be converted back. This adds a lot of overhead and reduces performance.

- Both approaches do not support updating ordered XML data because of the limitation of the SQL language .
- When converting an XML document to database representation, a DB2 Extender user has to define the convert rule in DAD. This is a learning gap for the user. For Oracle 9i XSU, the user must create a perfectly matching database schema for the XML document. And since XSU cannot store XML attribute values, the user must change the XML attribute to an XML element. This reduces the expressing power of XML attribute.

3.2 Semi-structured Point of View

XML is actually a semi-structured document. Thus, it is very natural to query and update an XML document from a semi-structured point of view.

3.2.1 Data Models for XML as Semi-structured Data

A clear and effective query language for semi-structured data relies on an appropriate data model, which is a collection of abstract notations for representing the organization of data. XML permits the representation of both structured and semi-structured data. In fact, an XML document is often associated with a DTD, which dictates its syntactic structure. However, in many cases, a syntactically correct XML document does not have a DTD. The data model underlying the query languages tries to represent appropriately the XML data structure with or without a DTD. All the approaches of query languages rely on a data model.

All the data models proposed to represent XML data can be categorized into two main kinds: labelled trees and directed labelled graphs. Labelled tree models are

used when the reference attributes (IDREF) are interpreted as strings, as any other XML attribute, while the directed labelled graph models are adopted when the reference attributes, which allow one to refer to another element, are interpreted as references between elements. The choice between the two kinds of models depends on the weight given to the semantics of an XML document in the query language. In fact, this choice affects the kind of queries that it is possible to express in the query language. With the latter, a stronger weight is given to the semantics of the references between elements.

The graph structure has been chosen by XML-QL [35] and XML-GL[5]. The tree structure has been chosen by XSL [25], XQL [14], CXQuery [7], Updating XML [32]. Actually, nearly all the official proposals of the W3C consider the tree structure as the data model for XML documents. The query language Lorel [2,3] allows one to pose queries relying on both data models.

Our thesis concentrates on the update part of the query languages. Unfortunately, most of the existing XML query languages only support query operations, not updates. In the next two sections, we review two of the XML query languages that support update operations: XML-GL and Lorel.

3.2.2 XML-GL

XML-GL stands for Graphical Language for Querying and Restructuring XML Document [5]. It was designed at University Di Milano. The originality of XML-GL with respect to other proposals for querying XML documents, like XML-QL, XQuery and XQL, is that queries are formulated visually, using a graph-based formalism close to the structure of XML documents (e.g, comparable to the visual representations of XML documents offered by XML authoring tools). However, XML-GL is not a

visual interface over a conventional, textual, query language, but a graph-based query language with both its syntax and semantics defined in terms of graph structures and operations [38]. The typical structure of an XML-GL query on a set of XML is a pair of graphs (or, as we will see, a pair of sets of graphs). The graphs on the left side extract information from the document collection and match each of them with user provided predicates. The graphs on the right side indicate which elements retrieved in the lefthand part should appear in the result, and construct or restructure the information to be produced as output.

Example 7. We use the following XML segment to demonstrate XML-GL query:

```
<manufacturer>
  <name>Mercury</name>
  <year>1999</year>
  <model>
    <mo_name>Sable LT</mo_name>
    <rank>9</rank>
  </model>
  ....
</manufacturer>
```

An example of XML-GL query is depicted in Figure 3.1, which finds all the manufacturer elements having a model with rank less than 10. As the result, the manufacturer elements, with all their content, are retained.

In Figure 3.1, we use LHS to indicate the left hand side and RHS to indicate the right hand side. In the LHS, we select manufacturer elements where some models have rank less than 10. The query extracts all the occurrences of the manufacturer elements satisfying the stated conditions. The elements used in the RHS are to construct the result which are exactly those manufacturer objects retrieved in the LHS with all the sub-elements appearing in the input XML documents. The result is a new XML document enclosed within the standard element result.

Figure 3.1: XML-GL Query Example

XML-GL supports simple update functions. Insertion, deletion, and modification of elements are graphically implemented by means of I, D, and U labelled arrows. Insertion and modification functions have corresponding LHS and RHS elements, while deletion just needs a LHS element. If we want to insert a new manufacturer after all manufacturers where some models have rank less 10, we may use the graph shown in Figure 3.2.

Figure 3.2: XML-GL Update Example

The result is a new XML document with a newly added manufacturer element and its subelement.

3.2.3 Lorel

Lorel [2], from Stanford's Lore Semi-Structured Database System, was originally designed for semi-structured data. It has now been extended to XML data [3].

Lorel is based on a labelled, ordered graphic data model. In this model, all entities are objects that can either be atomic or complex. The nodes in the graph represent the data elements and the edges represent the element-subelement relationship. Each node representing a complex data element contains a tag and an ordered list of attribute-name/atomic-value pairs; atomic data element nodes contain string values. There are two different types of edges in the graph: (i) normal subelement edges (shown as a solid line), labelled with the tag of the destination subelement; (ii) cross-link edges (shown as a dash line), labelled with the attribute name that introduced the cross-link.

Example 8. Consider the following XML document segment.

```
<DBGGroup>
  <Member Name="Smith" Advisor="m1" >
    <Age>28</Age>
  </Member>
  <Member ID="m1" Project="p1">
    <Name>Jones</Name>
    <Advisor>Ullman</Advisor>
  </Member>
  <Project ID="p1" Member="m1">
    <Title>Lore</Title>
  </Project>
</DBGGroup>
```

Example 8 can be mapped into a Lorel model as shown in the Figure 3.3.

Lorel's query language uses SQL-like style for querying XML data. For example, the following statement can be used to find the age of a member whose name is Smith.

```
select DBGGroup.Member.age
```

Figure 3.3: Lorel XML-based Graph Model

where `DBGroup.Member.Name = "Smith"`

Lorel also supports a simple and expressive update language with the functions of creating both elements and attributes and modifying the value of an existing element or attribute. Update requests are mainly expressed using the following general form:

```
Update <object-selector>.<label>(+/-/:)= <expression>
from <from-clause>
where <where-clause>
```

where the `<object-selector>` determines an element or attribute to be updated. It is usually a unique object result of a query. The `<expression>` identifies a set `O` of objects and `o ∈ O`. If the operator is `+=`, then new edges are created from `o` to each object in `O` and given the label `<label>`. If the operator is `-=`, then existing edges with the label `<label>` from `o` to objects in `O` are removed. If the operator is `:=`, all edges from `o` with label `<label>` are removed and new edges with label `<label>`

are introduced between *o* and object in *O*. The *from* and *where* clauses are the select statement in the Lorel.

Example 9. To add a new member whose name is Clark to DBGroup.

```
update P.Member.Name += "Clark"
from DBGroup.Project P
where P.Title = "Lorel"
```

3.2.4 Comments on XML-GL and Lorel

The model used in Lorel and XML-GL is not an abstract data model. Therefore it cannot provide a mechanism to perform any type of checking according to the XML DTD or XML Schema. XML-GL does not support tag variable binding so that there is no way to update an XML tag. Lorel is the first query language for XML documents. It supports simple update features without considering multiple level and multiple document updates, etc. Lorel relies on many built-in functions for updating operations, and the update operation cannot fully support the ordered XML document.

Chapter 4

XML-RL Query Language Overview

XML-RL [10, 11] is a rule-based declarative query language for extracting and constructing data from XML documents. In XML-RL, XML documents are treated as an extensional database, while queries and functions expressed in rules are treated as an intensional database. It incorporates many ideas from the deductive database and logic programming languages. It also includes some commonly used functions in its queries. In this chapter, we first briefly introduce the data model and mapping rules, then explain its query language.

4.1 Data Model

The XML-RL data model represents an XML document in a natural way as in complex object data models. In order to model basic concepts of XML specification, the following five kinds of objects are proposed in the XML-RL data model.

1. An *element object* represents an element with tag and value pair, for example,

```
name->[firstname->Andrew, lastname->Castro],  
program->Software Engineering.
```


Symbol '→' is used to separate the name of an object from the value of the object.

2. An *attribute object* represents an attribute with name and value pair, for example,

@id→F100, @supervisee→{S200, S400}.

Symbol '@' indicates the following is an attribute object.

3. A *tuple object* represents the relationship among elements and attributes in XML, for example,

[@id→F100, @supervisee→{S200, S400},
name→[firstname→Andrew, lastname→Castro],
position→Assistant Professor,
salary→\$50,000]

where the pair of square brackets is used to construct a tuple object.

4. A *list object* represents multiple values of an attribute or element, for example,

{S100, S600}

The pair of curly brackets is used to construct a list object.

5. A *lexical object* represents the constant value of an element or an attribute, for example,

Marry, Databases.

4.2 Mapping XML to XML-RL Model

In this section, we discuss in brief how to convert an XML document to the complex object model. From the XML document in Example 1, we can see that there are five kinds of basic concepts: *string data*, *simple attribute*, *complex attribute* (i.e. IDREFS attribute), *simple element* and *complex element*. Correspondingly, there are five kinds of objects in the complex data model: *lexical object*, *attribute object*, *element object*, *tuple object* and *list object*. The informal mapping rules from the XML document to the complex object model are as follows.

1. A *string data* is transformed into a lexical object. For example, string data *Computer Science* is mapped to lexical object *Computer Science*.
2. A *simple attribute* is transformed into an attribute object. For example, an attribute *id*="S200" of element *student* is mapped to an attribute object *@id*→*S200*.
3. A *complex attribute* is transformed into an attribute object with a value of lexical list object. For example, an IDREFS attribute *supervisee*="S200 S400" of element *faculty* is mapped to an attribute object *@supervisee*→{*S200*, *S400*}.
4. A *simple element* is transformed into an element object. For example, *<salary> \$40,000 </salary>* is mapped to *salary*→*\$40,000*.
5. A *complex element* is transformed into an element object with a value of tuple object that is composed of other simple objects or nested complex objects. For example,

```
<student id = "S600">
  <name>
    <firstname> Mary </firstname>
```

```

        <lastname> Lee </lastname>
    </name>
    <program> Hardware Engineering </program>
</student>

```

is transformed into

```

student->[@id->S600,
          name->[firstname->Mary, lastname->Lee],
          program->Hardware Engineering]

```

Model 1. Using the above mapping rules, the XML document in Example 1 can be naturally and straightforwardly converted into the complex object model as follows:

```

csdept->[faculty->[@id->F100, @supervisee->{S200, 400},
                  name->[firstname->Andrew, lastname->Castro],
                  position->Assistant Professor,
                  salary->$50,000],
        faculty->[@id->F200,
                  name->[firstname->Bob, lastname->Smith],
                  position->Associate Professor,
                  salary->$40,000],
        staff->[@id->SF100,
                name->[firstname->Linda, lastname->Stevens],
                email->linda@scs.carleton.ca],
        student->[@id->S200, @supervisor->F100,
                  name->[firstname->Alisar, lastname->Smith],
                  program->Software Engineering],
        student->[@id->S400, @supervisor->F100,
                  name->[firstname->Jack, lastname->Weiss],
                  program->Software Engineering],
        student->[@id->S600,
                  name->[firstname->Mary, lastname->Lee],
                  program->Hardware Engineering]]

```

4.3 Query Language

XML-RL is a query language based on the data model introduced above. It is composed of rules which have a head and a body. The body is for extracting the source information by binding variables while the head is for result constructing. The specification expression is:

query $qexp_1, \dots, qexp_n$

construct $cexp$

where $qexp_1, \dots, qexp_n$ are querying expressions and $cexp$ is the result constructing expression.

Example 10. Consider the sample XML document in Example 1, the following query is to find all faculty members whose salary is more than \$45,000 and whose position is Associate Professor.

```
query (http://www.scs.carleton.ca/csdept.xml)/
      csdept/$f(faculty->[salary>$45,000,
                        position->Associate Professor])
construct (http://www.scs.carleton.ca/result.xml)/csdept/$f
```

The variable \$f matches an element with tag name faculty. The result is as follows:

```
<csdept>
  <faculty id = "F100", @supervisee = "S200 S400">
    <name>
      <firstname> Andrew </firstname>
      <lastname> Castro </lastname>
    </name>
    <position> Assistant Professor </position>
    <salary> $50,000 </salary>
  </faculty>
</csdept>
```

Example 11. The following query is to find the values of all students who are in the Software Engineering program.

```

query (http://www.scs.carleton.ca/csdept.xml)
  /csdept/student->$s(program->Software Engineering)
construct (http://www.scs.carleton.ca/result.xml)/csdept/student->$s

```

The variable \$s matches the value of two student elements. For constructing the value result properly, we need to construct a tag name student.

The result is as follows:

```

<csdept>
  <student id = "S200", supervisor = "F100">
    <name>
      <firstname> Alisar </firstname>
      <lastname> Smith </lastname>
    </name>
    <program> Software Engineering </program>
  </student>
  <student id = "S400", supervisor = "F100">
    <name>
      <firstname> Jack </firstname>
      <lastname> Weiss </lastname>
    </name>
    <program> Software Engineering </program>
  </student>
</csdept>

```

Example 12. The following query is to find the IDs for all students whose supervisor is Andrew Castro.

```

query (http://www.scs.carleton.ca/csdept.xml)
  /csdept/faculty->[@supervisee->{$id},
                    name->[firstname->Andrew,
                          lastname->Castro]]
construct (http://www.scs.carleton.ca/result.xml)/result:{$id}

```

The variable {\$id} holds a list of values. The result is as follows:

```

<result>{S200, S400}</result>

```

Chapter 5

XML-RL Update Language

In this section, we propose an XML-RL update language to extend the XML-RL query language. Using the XML-RL update language, we can *insert*, *delete* and *modify* an atomic or complex XML object within one or multiple XML documents in a straightforward way.

We aim for a minimal set of primitive update operations to fulfill the update requirements. All valid changes to manipulate XML data should be specifiable by one or by a sequence of our primitive update operations.

In section 5.1, we define the formal syntax of the schema, the database and the XML-RL update language. In section 5.2, we discuss variables from different aspects. In section 5.3, we illustrate our update language by some examples, including insertion, deletion, replacement of element and attribute objects. Finally, we formally present the semantics of the XML-RL update language, including well-formed schemas, well-formed databases and the semantics of various update operations.

5.1 Syntax

In this section, we define the formal syntax of schema, database and update language.

5.1.1 Schema Syntax

We assume the existence of the following sets:

1. A set $\mathcal{C}$ of constants;
2. A set $\mathcal{V}$ of variables. A variable name begins with \$ followed by a constant, for example, \$salary;
3. a set of basic types $\mathcal{T} = \{\#PCDATA, CDATA, ID, IDREF, IDREFS\}$.

Definition 1. An *attribute rule* is of the form $@A_E \leftarrow t$, where $A \in \mathcal{C}$ is an attribute name, $E \in \mathcal{C}$ is the element name to which the attribute belongs, and $t \in (\mathcal{T} - \{\#PCDATA\})$ is the data type of the attribute.

Definition 2. An *element rule* is of the form $E \leftarrow \mathcal{B}$, where $E \in \mathcal{C}$ is an element name and $\mathcal{B}$ is one of the following forms:

1. $\#PCDATA$. Such a rule is called an *atomic* element rule;
2. B , where $B \in \mathcal{C}$ is an element name. Such a rule is called a *single* rule;
3. $B?$, where $B \in \mathcal{C}$ is an element name. Such a rule is called an *optional* rule;
4. $B+$, where $B \in \mathcal{C}$ is an element name. Such a rule is called a *duplicate* rule;
5. B^* , where $B \in \mathcal{C}$ is an element name. Such a rule is called an *arbitrary* rule;

6. $B_1|...|B_n$ with $n > 0$, where for each $B_i(1 \leq i \leq n)$, $B_i \in \mathcal{C}$ is an element name.

Such a rule is called a *choice* rule;

7. $@A_{1E}, ..., @A_{mE}, E_1, ..., E_n$ with $m \geq 0$ and $n \geq 0$, where $@A_{iE} \in \mathcal{C}(1 \leq i \leq m)$ is an attribute name of the element E to be described by the rule and $E_i \in \mathcal{C}(1 \leq i \leq n)$ is an element name. Such a rule is called a *tuple* rule;

8. Forms composed by rule 1, 2, 3, 4, 5, 6 and 7 in a usual way. Theses are called a *composite* rule.

The following are examples of various rules.

Attribute rules : $@id_{faculty} \leftarrow ID, @supervisor_{student} \leftarrow IDREF$

Atomic element rule : $program \leftarrow \#PDCATA$

Single rule : $faculty \leftarrow name$

Optional rule : $name \leftarrow firstname, middlename?, lastname$

Arbitrary rule : $staff \leftarrow staff^*$

Duplicate rule : $faculty \leftarrow faculty^+$

Choice rule : $publication \leftarrow book|journal|conference|\#PCDATA$

Tuple rules : $faculty \leftarrow @id_{faculty}, name, position, salary,$
 $student \leftarrow @id_{student}, @supervisor_{student} \leftarrow IDREF, name, program$

Composite rules : $department \leftarrow (faculty^+|staff^*|student^+),$
 $university \leftarrow (@id_{university}, department^*)$

Definition 3. Let $P=A \leftarrow B$ be either an *element rule* or an *attribute rule*. Then A is the head of the rule, denoted by $head(P)$, and B is the body of the rule, denoted by $body(P)$.

Definition 4. A schema S consists of two parts: a root element R and a set of rules P , denoted by the ordered pair (R, P) . $root(S)$ is used to refer to R and

$rules(S)$ is used to refer to P .

A DTD document can be transformed into a schema in a straightforward way. In Example 2, the root element is *csdept* and the set of rules is shown as follows.

Model 2. The following is the schema for Example 2.

```
csdept ← (faculty+ | staff* | student+)
faculty ← @idfaculty, @superviseefaculty, name, position, salary
staff ← @idstaff, name, email
student ← @idstudent, @supervisorstudent, name, program
name ← firstname, middlename?, lastname
firstname ← #PCDATA
middlename ← #PCDATA
lastname ← #PCDATA
position ← #PCDATA
salary ← #PCDATA
email ← #PCDATA
program ← #PCDATA
@idfaculty ← ID
@idstaff ← ID
@idstudent ← ID
@superviseefaculty ← IDREFS
@supervisorstudent ← IDREF
```

5.1.2 Database Syntax

In the following, we first give the formal syntax of objects, then we give the formal syntax of database.

Definition 5. The notion of *objects* is defined inductively as follows:

1. Let $c \in C$ be a constant. Then c is a *lexical object*.
2. Let $o_1, \dots, o_n$ be objects with $n \geq 0$. Then $\{o_1, \dots, o_n\}$ is a *list object*. In particular, if each $o_i (1 \leq i \leq n)$ is a *lexical object*, then $\{o_1, \dots, o_n\}$ is called a *lexical list object*.
3. Let $a \in C$ be a constant and o be either a constant or a *lexical list object*. Then $@a \rightarrow o$ is an *attribute object* and a is the *name* of the object and o is the *value* of the object.
4. Let $e \in C$ be a constant and o be either a *lexical object*, a *list object* or a *tuple object*. Then $e \rightarrow o$ is an *element object* and e is the *name* of the object and o is the *value* of the object.
5. Let $@a_1, \dots, @a_m$ be attribute objects and $e_1, \dots, e_n$ be element objects with $m \geq 0$ and $n \geq 0$. Then $[@a_1, \dots, @a_m, e_1, \dots, e_n]$ is a *tuple object*. In particular, it represents an *empty object* in the case of $m=0$ and $n=0$.

The following are examples of various objects:

Lexical object: Software Engineering

List object: $\{S200, S400\}$

Attribute objects: @id→F200, @supervisee→{S200, S400}

Tuple object: [firstname→Andrew,lastname→Castro]

[illegible]

program $\rightarrow$ Software Engineering]

Definition 6. Let o_1 and o_2 be two objects. Then $o_1 \in o_2$ if o_1 is a nested object of o_2 .

Definition 7. Attribute objects and element objects are called *named* objects while lexical objects, list objects and tuple objects are called *anonymous* objects. The following are some examples of named objects and anonymous objects.

Named objects: @id $\rightarrow$ S100, salary $\rightarrow$ \$40,000

Anonymous objects: Professor, {S200, S400},
[name $\rightarrow$ [firstname $\rightarrow$ Jack, lastname $\rightarrow$ Weiss]]

Definition 8. Let o be an object. Then $name(o)$ is defined as a function to get the name part of o . In particular, it returns *null* if o is an *anonymous* object. Similarly, $value(o)$ is defined as a function to get the value part of o .

Definition 9. Let o be an object. Then o is *regular* if o is not an element object with a value of list object and $\nexists o' \in o$ (o' is an element object and o' has a value of list object). If an object is *regular*, we say it has a *regular* form.

For example, in the complex object model of Model 1, the csdept element object is a regular form because two faculty element objects and three student element objects are not nested into an element object with a value of list object.

Definition 10. Let o be an object. Then o is *normalized* if $\exists o' \in o$ (o' is a

named object with a value of tuple $[o'_1, \dots, o'_n]$ and for any two adjoining objects o_i and o_{i+1} ($1 \leq i < n$) and $(name(o_i) = name(o_{i+1}))$. If an object is *normalized*, then we say it has a *normalized* form.

Model 3. The following is a normalized form for Model 1.

```
csdept->faculty->{@id->F100, @supervisee->{S200, S400},
                  name->[firstname->Andrew, lastname->Castro],
                  position->Assistant Professor,
                  salary->$50,000},
               [@id->F200,
                name->[firstname->Bob, lastname->Smith],
                position->Associate Professor,
                salary->$40,000]},
staff->[@id->SF100,
        name->[firstname->Linda, lastname->Stevens],
        email->linda@scs.carleton.ca],
student->{@id->S200, @supervisor->F100,
          name->[firstname->Alisar, lastname->Smith],
          program->Software Engineering],
         [@id->S400, @supervisor->F100,
          name->[firstname->Jack, lastname->Weiss],
          program->Software Engineering],
         [@id->S600,
          name->[firstname->Mary, lastname->Lee],
          program->Hardware Engineering]}
```

In this thesis, the *regular* form of XML documents is used for the data model because it is the most natural and straightforward way to represent XML objects as in the complex object model. Therefore, the extensional database representing XML documents is defined as follows.

Definition 11. A *database* DB of a schema S is defined as the ordered pair (S, o) , where o is a named element object with *regular* form.

5.1.3 Syntax of XML-RL Update Language

In this section, we present the formal syntax of the XML-RL update language.

We first define terms and expressions, then we define the update operations.

In XML-RL, a term is a direct reflection from the XML-RL data model.

Definition 12. The notion of *terms* is defined as follows:

1. Let $v \in \mathcal{V}$ be a variable. Then $\$v$ is a *variable term*.
2. Let $c \in \mathcal{C}$ be a constant. Then c is a *lexical term*.
3. Let $t_1, \dots, t_n$ with $n \geq 0$, t_i ($0 \leq i \leq n$) be a term. Then $\{t_1, \dots, t_n\}$ is a *list term*.
4. Let a be a constant or variable and t be either a lexical term, variable term or a list of lexical terms. Then $@a \rightarrow t$ is an *attribute term* and t is the *value* of attribute a .
5. Let e be a constant or variable and t be an term. Then $e \rightarrow t$ is an *element term* and t is the *value* of element e .
6. Let $@a_1, \dots, @a_m$ be attribute terms and $e_1, \dots, e_n$ be element terms with $m \geq 0$ and $n \geq 0$. Then $[@a_1, \dots, @a_m, e_1, \dots, e_n]$ is a *tuple term*. In particular, it represents an empty term in the case of $m=0$ and $n=0$.

The following are examples of various terms.

Lexical terms : 2314343, 440 Albert Street

Variable terms : \$name, \$id

List terms : {S100, S200, S300}, {F100}

Attribute terms: @id → \$id, @supervisee → {S200, S400}

Element terms : firstname → Andrew, salary → \$50,000

Tuple terms : [name→[firstname→Bob, lastname→Smith]],
 [faculty→[@id→F200,
 name→[firstname→Bob, lastname→Smith],
 position→Associate Professor,
 salary→\$40,000]]

Definition 13. A term is *ground* if it contains no variables.

In XML-RL, the path expression is an ordered list of elements with its contents.

Definition 14. The notion of expressions is defined as follows:

1. *Arithmetic*, *logical*, and *string* expressions are defined using terms in the usual way.
2. Let $\$v$ be a variable term. Then three kinds of variable selection expressions are defined as follows:
 - Let $e \rightarrow t$ be a named object term, then $\$v=e$ is a variable name selection expression.
 - Let $e \rightarrow t$ be a named object term, then $e \rightarrow \$v$ is a variable value selection expression.
 - Let e be a term, then $\$v(e)$ is a variable object selection expression.
3. Let L be a list term and A be an arithmetic expression. Then $L[A]$ is a *list selection* expression.
4. Let E_1 and E_2 each be either an element term, a tuple term, or a *variable selection* expression. Then E_1/E_2 and $E_1//E_2$ are *path selection* expressions, where E_1/E_2 indicates E_2 is the immediate child of E_1 and $E_1//E_2$ indicates E_1 is an ancestor of E_2 .

5. Let U be a url and P be an element term, a variable selection expression or a path selection expression. Then $(U)/P$ and $(U)//P$ are *query* expressions.

The following are examples of expressions.

List selection expression : @supervisee[3]

Variable selection expressions : \$faculty(faculty→[firstname→Bob]),
\$student(student→\$studentValue)

Path selection expressions : /csdept/faculty→[@id→F200],
//student→[name→[firstname→Marry]]

Query expressions :

(www.scs.carleton.ca)/csdept/faculty→[name→[firstname→Bob,lastname→Smith]]

Definition 15. An expression is *ground* if it contains no variables.

Definition 16. Let E be a term and $\$v$ be a variable term. Then *insert E into $\$v$* is an *insert-into* declaration.

Definition 17. Let E be either a *lexical* term, a *variable* term, a *list* term, an *element* term, or a *tuple* term, and $\$v$ be a variable term, Then *insert E before $\$v$* and *insert E after $\$v$* are *insert-before* and *insert-after* declarations.

Definition 18. Let E be a term and $\$v$ be a variable term. Then *replace $\$v$ with E* is a *replacement* declaration.

Definition 19. Let $\$v$ be a variable term. Then *delete $\$v$* is a *deletion* declaration.

Definition 20. Let $qexp_1, \dots, qexp_n$ be query expressions and $uexp_1, \dots, uexp_m$ be update declarations. Then

query $qexp_1, \dots, qexp_n$
 $uexp_1, \dots, uexp_m$

is an update query.

In our update query, the query expressions named query part and update expressions named update part may be one of the following five basic forms:

- **insert** *content* **into** \$v;
- **insert** *content* **before** \$v;
- **insert** *content* **after** \$v;
- **delete** \$v;
- **replace** \$v **with** *content*;

where the query part is used to query one or more XML documents. The query result could be bound to variable \$v to hold XML objects. These variables can be used in the update part as an updating object, updating context, and content of the update. A binding variable may be used in multiple update expressions. The update part is used to modify one or more XML objects from one or more XML documents. The order of update executions is the order in which they appear in the expressions.

5.2 Variables

XML-RL supports rich kinds of logical variables and restrictions on them within expressions. Variables can be categorized differently based on different criteria.

1. According to what kind of objects they represent, there could be five types of variables:
 - *Element variable* corresponds to element object.
 - *Attribute variable* corresponds to attribute object.

- *Tuple variable* corresponds to tuple object.
- *List variable* corresponds to list object.
- *Lexical variable* corresponds to lexical object.

2. According to the object contents they represent, variables could be grouped into three kinds: *object variable*, *name variable* and *value variable*.

- An *object variable* is used to hold an object including the name and the value of it. Consider the expression:

`(url)/csdept/$f`

where `$f` holds all component objects of the Computer Science Department object, including two faculty objects, one staff object, and three student objects. If we only want to hold one of the faculty objects, then we can apply a restriction on the logical variable within the expression, for example,

`(url)/csdept/$f(faculty->[name->[lastname->Smith]])`

where `$f` binds a faculty object. If we do not care about the value of the object variable, then we can use an anonymous variable `$` in the restriction to match any value, for example,

`(url)/csdept/$f(faculty->$)`

where `$f` holds all faculty objects of Computer Science Department. Also, the restriction can be simplified to `(url)/csdept/$f(faculty)`. Of course, we can apply a restriction only on the value of the object variable. Consider the expression,

`(url)//$f($->[name->[lastname->Smith]])`

where anonymous variable \$ in the expression matches any tag name. The results of this query are the second faculty object and the first student object.

- A *name variable* is used to hold the name of the element or attribute object. The value of this kind of variable can only be string. Consider the expression,

`(url)/department/$n->$, $n=faculty`

where the variable \$n holds the names of faculty objects.

- A *value variable* is used to hold the value of object. Consider the expression,

`(url)/csdept/faculty->$f`

where the variable \$f is used to hold the values of all faculty.

Of course, a restriction can be applied to both structure variables and value variables in a way similar to object variables.

3. According to the number of objects a variable represents at the same time, there could be two types of variables: *single variable* and *list variable*. A *single variable* is used in the regular way and a *list variable* uses a pair of curly brackets with a single variable inside. For example, `name->$name` is a *single variable* referring to the value of the name element one at a time; `name->{$name}` indicates the variable holds a list of values of all applicable name elements at the same time.

5.3 Example of Updates

In this section, we illustrate features of the XML-RL update language using many examples. In the XML-RL update language, the update operation is dependent on the query result. If the query returns no result, XML-RL update skips the update operation that depends on this result and continues to execute the next update operation. If any part of the update expression causes an error, either a syntax error or validation error, XML-RL aborts the update operation and no update occurs.

We will discuss insertion in Section 5.3.1, deletion and modification in Section 5.3.2 and 5.3.3. In Section 5.3.4, we discuss mixed update, multiple level update and copy semantics. We use Example 1 in Chapter 2 throughout this chapter. To save space, we use URL instead of the real url address.

5.3.1 Insertion

There are three primitive insertion operations, *insert-before*, *insert-after* and *insert-into*. The *insert-before* operation is used to insert an XML object into the selected XML object as its preceding sibling for ordered XML data. Similarly, the *insert-after* operation is used to insert an XML object into the selected XML object as its following sibling for ordered XML data. The *insert-into* operation is used to insert an XML object as a child at arbitrary position of the selected XML object for unordered data, or as a last child for ordered XML data. We can use these three operations to do an insertion for simple objects and complex objects, including attribute, IDREFS attributes, simple element, and complex element. Because attributes are always unordered, the *insert-before* and *insert-after* operations are not applicable to them.

We first discuss the insertion of simple attributes, then discuss the insertion of a simple element. Lastly, we discuss the insertion of a complex element.

Example 13. The following update statement inserts an attribute *@supervisor* that points to faculty *Bob Smith* into the *student* element whose name is *Mary Lee*.

```
query (URL)/csdept/$s(student->[name->[firstname->Mary,
                                     lastname->Lee]])
      (URL)/csdept/faculty->[@id->$id,name->[firstname->Bob,
                                             lastname->Smith]]
insert @supervisor->$id into $s
```

The variable *\$s* is used to hold the student object whose name is Mary Lee and variable *\$id* is used to hold the value of *@id* attribute object of the Bob Smith faculty object. After updating, an attribute named supervisor with value *\$id* will be added into the student object.

If an attribute object with name *@id* is already contained in the selected element object *\$s*, then the insertion operation is prohibited. This guarantees no duplicate attributes under one parent element according to attribute specification.

Note that although it is natural to use the *insert-into* operation for unordered data, it can also be used for ordered data. In this case, the object is simply inserted into the target object as its last child.

Example 14. The following update statement adds an element *workAddress* to each faculty element as the last child.

```
query (URL)/csdept/$f(faculty->$)
insert workAddress->311 Bell St into $f
```

The variable *\$* is used to hold any value of the faculty object. The simplest form is :

```
(URL)/csdept/$f(faculty)
```

If we want to add the object to each faculty as the following sibling object of the name object we can use the *insert-after* operation as follows.

```

query (URL)/csdept/faculty/$n(name->$)
insert workAddress->311 Bell St after $n

```

In example 14, we also show that the same updating operation applies to each applicable object in a document that matches a certain criteria. From the examples, we can see that the insert-into operation can ensure the workAddress object is inserted into each applicable object. But this would not be true if we use the insert-after operation.

The case for the insert-before operation is similar.

XML-GL also provides the insertion operation to insert simple objects. But it needs two primitive operations for this purpose. One is for the node object creation and the other is for the edge object linking, because they use graph-based models to represent XML data.

Now we discuss the insertion of complex objects, including IDREFS attribute insertion, IDREFS entry insertion and complex element insertion. An IDREF attribute insertion is similar to a simple attribute insertion except that the value expression of IDREFS attribute objects is different from that of simple attribute objects.

Since attributes are unordered, insert-before and insert-after are not applicable for attribute updating. But if the attribute type is IDREFS, its value is a named ordered list of IDs, in this case, insert-before and insert-after are applicable for the attribute update. Consider the following example.

Example 15. The following update statement adds a new attribute supervisee that contains the id of student Mary Lee to faculty Bob Smith.

```

query (URL)/csdept/$f(faculty->[name->[firstname->Bob,
                                     lastname->Smith]]),
      (URL)/csdept/student->[@id->$id,name->[firstname->Mary,
                                     lastname->Lee]]
insert @supervisee->{$id} into $f

```

The variable `{$id}` is used to construct a list object. Variable `$id` holds the value of the `@id` attribute object of the selected student object and variable `$f` holds the selected faculty object. Therefore, the above statement inserts the object `@supervisee->{S600}` into the faculty object.

If we use the insert-into operation to insert an entry into a list object, then the entry is inserted at the end of the list. We can also use the insert-before or insert-after operation to insert an entry at the proper position in the list of IDREFS.

Example 16. The following update statement adds the id of the student object whose name is Mary Lee to an IDREFS attribute of the faculty object whose name is Andrew Castro.

```
query (URL)//faculty->[@supervisee->{$s},name->[firstname->Andrew,
                                     lastname->Castro]],
      (URL)//student->[@id->$id,name->[firstname->Mary,
                                     lastname->Lee]]
insert $id after {$s}.position(0)
```

The list value variable `{$s}` holds a list of student ids and the single value variable `$id` holds another student id to be inserted. `{$s}.position(0)` is a built-in function which indicates the first member of the list objects held by `{$s}`.

The following example discusses the insertion of complex elements.

Example 17. Insert a new faculty Charis Adson after Bob Smith.

```
query (URL)/csdept/$f(faculty->[name->[firstname->Bob,
                                     lastname->Smith]])
insert faculty->[@id->F700,
               name->[firstname->Charis,lastname->Adson],
               position->Professor,
               salary->$70,000]
after $f
```

The variable `$f` holds a faculty element for Bob Smith. After the update operation, a new faculty element is added after Bob Smith.

In order to insert the above complex element object, the XML-GL language needs 12 node object creation primitive operations and 12 edge linking primitive operations as shown in Figure 5.1, where complex elements are represented as solid rectangles, simple elements as blank rectangles, attributes as circles, and string data as triangles. Our language can directly support the insertion of complex elements because it can directly express complex element objects due to the data model.

Figure 5.1: An Example of a Complex Element Object

5.3.2 Deletion

The delete operation allows users to remove objects from an XML document. The deleted objects are usually the ones returned by the query part, either objects or their values. We cannot remove the name of an existing object. When the value of an element is deleted, the value of object will be set to null.

Example 18. The following update statement deletes the value of salary for a faculty whose id is F200.

```

query (URL)/csdept/faculty->[@id->F200, salary->$s]
delete $s

```

The variable \$s holds the value of salary objects of the faculty object whose id is F200. Deleting \$s means the values of the selected salary objects are set to null. When a complex element object is deleted, its sub-elements and attributes are removed recursively. If the object is referred by other object, a cascading deletion is performed.

Example 19. The following update statement deletes a student whose name is Alisar Smith from the Computer Science Department.

```

query (URL)/csdept/$s(student->[name->[firstName->Alisar,
                                         lastName->Smith]])
delete $s

```

The variable \$s holds the student object whose name is Alisar Smith. The deleted student object contains two attribute objects, @id→S200 and @supervisor→F100, and two element objects, name→[firstname→Alisar, lastname→Smith] and program→Software Engineering. All these sub-objects are recursively removed. Thus the deleted student object will no longer be referenced by the faculty object whose id is F100.

5.3.3 Modification

The replace operation can be used to modify the existing objects. The modified object must be returned by the query part before the replace operation can be applied. The replace operation can replace name of an object, value of an object, or both, depending on what the binding variable references.

Example 20. The following update statement changes the name of @id for all student objects to @sid.


```

query (URL)/department/student->[@$id->$],$id=id
replace $id with sid

```

The variable \$id holds the name of attribute @id objects of student objects. The *replace* clause is used to change attribute name @id to @sid. In the Lorel language, object name updating can be indirectly supported in a two-step way. First, select value from existing object and assign it to a new object name; second, delete the existing object. This is actually a combination of creation and deletion operations. Our language supports the change of tag names in the same syntax with value and object updates.

Similar to the insertion of complex objects, our update language can express the replacement operations of complex objects and complex values in a simple but efficient way. Some XML update languages cannot support the replacement of complex objects and complex values because they cannot express complex objects and complex values due to the graph or tree data models used.

Example 21. The following update statement modifies the value of the faculty object whose name is Bob Smith.

```

query (URL)/department/faculty->$fv(name->[firstname->Bob,
                                         lastname->Smith])
replace $fv with [@id->F700,
                  name->[firstname->Charis,lastname->Adson],
                  position->Professor,
                  salary->$70,000]

```

In the above update statement, the value of the faculty is replaced with complex values. Of course, the replace operation can also be used to update complex objects. For example,

```

query (URL)/department/$fv(faculty->[name->[firstname->Bob,
                                         lastname->Smith]])
replace $fv with faculty->[@id->F700,

```

```

name->[firstname->Charis,lastname->Adson],
position->Professor,
salary->$70,000]

```

This update statement has the same function as the above one, but it updates the faculty object rather than the value of the faculty object because \$v is used to hold the faculty object rather than its value.

5.3.4 Combination

In sections 5.3.1 to 5.3.3, we have discussed the insertion, deletion and replacement operations, and shown some unique features of our update language, such as insertion and replacement of complex objects, replacement of tag names and object values. In this section, we discuss some other features our update language provides, including mixed update, copy semantics and multiple level update.

In our update language, one or more insertions, deletions, and replacements can be performed together within one update statement, but we only allow deletion of the current scope. In other words, a combination update cannot delete parents of the current scope. This prevents situations in which a user deletes parents and then tries to update their children.

Example 22. The following update statement changes the attribute name id to sid and the value of program from Software Engineering to Hardware Engineering for a student element, and inserts a new homeAddress element after name element.

```

query (URL)/csdept/student->[@$id->$, $id=@id,
                                $n(name->[firstname->Alisar,
                                                lastname->Smith]),
                                program->$a]

replace $id with sid,
replace $a with Hardware Engineering,
insert homeAddress->440 Albert St after $n

```

The variable `$id` holds the name of attribute `@id` object, `$n` holds the name element object and `$a` holds the value of the program element object of the student object whose name is Alisar Smith. Three different update requests are mixed together within one update statement. In this example, we cannot first delete the student object and then try to modify its content.

Besides reference semantics, the XML-RL data model supports copy semantics. Therefore, our update language also supports update operations based on copy semantics. If an existing object is used as part of the content of an update operation, then the copy of the existing object (rather than reference) is used for this update.

Example 23. Suppose a faculty Cantor Cliton works in the Math Department, and his data is at <http://www.carleton.ca/madept.xml>. *madept.xml* and *csdept.xml* have the same DTD. Now assume that he also works in the Computer Science Department. Therefore, we need to copy his data from the site of the Math Department to the site of the Computer Science Department. The new faculty element should be positioned after faculty element whose name is Bob Smith.

```
query (http://www.carleton.ca/madept.xml)/
      madept/$fm(faculty->[name->[firstname->Cantor,
                                   lastname->Cliton]]),
      (http://www.carleton.ca/csdept.xml)/
      csdept/$fc(faculty->[name->firstname->Bob,
                           lastname->Smith]])
insert $fm after $fc
```

After updating, the data about Cantor Cliton is copied into the XML document of the Computer Science Department as the sibling following Professor Bob Smith.

Sometimes we need to update multiple level elements in a hierarchy. Our update language supports multiple level update requests in a flat and natural way by using the logical variables.

Example 24. The following update statement adds a new attribute `dname` to the Computer Science Department and raises all associate professors salaries by 5% in this department.

```
query (URL)/$d(csdept->[faculty->[salary->$s,
                                position->Associate Professor]])
insert @dname->Computer Science into $d,
replace $s with $s+($s*5%)
```

The variable `$d` holds the `csdept` object under `URL` level while variable `$s` holds the value of salary of the applicable faculty under `faculty` level. The update operations apply to the binding variables that refer to different level objects.

5.4 Semantics of Update Language

In this section, we define the semantics of schema, database and the update language. We first discuss well-formed schema and well-formed objects, then define well-formed databases based on well-formed schema and well-formed objects. Finally we define the formal semantics of the update language.

5.4.1 Semantics of Schema

We first define *non-terminating* rules, and *reachable* and *unreachable* symbols, then define a class of *well-formed* schema.

Definition 21. Let P be a set of rules. Then a rule p is *non-terminating* with respect to P if and only if p is not an atomic element rule or an attribute rule and $\exists B \in \text{body}(p)$ such that

1. $\exists q \in (P - \{p\}), \text{head}(q) = B$, or
2. for $\forall q \in (P - \{p\}) (\text{head}(q) = B$ and q is non-terminating with respect to $P - \{p\})$.

Example 25. For the set of rules $P=\{A \leftarrow B, C, D, B \leftarrow E, F, C \leftarrow \#PCDATA, D \leftarrow H, G, E \leftarrow \#PCDATA, F \leftarrow \#PCDATA, G \leftarrow \#PCDATA\}$, the *non-terminating* and *terminating* rules are as follows.

Non-terminating: $A \leftarrow B, C, D, D \leftarrow H, G$

Terminating : $B \leftarrow E, F, C \leftarrow \#PCDATA, E \leftarrow \#PCDATA, F \leftarrow \#PCDATA, G \leftarrow \#PCDATA$

Definition 22. Let $c \in \mathcal{C}$ be a constant, p a rule, and P the set of rules. Then c is *unreachable* from p with respect to P if and only if $c \neq \text{head}(p)$ and there does not exist a sequence of rules $q_1, q_2, \dots, q_n$ with $n \geq 1$ such that $(\text{head}(p) \in \text{body}(q_n)) \wedge (\text{head}(q_n) \in \text{body}(q_{n-1})) \wedge \dots \wedge (\text{head}(q_i) \in \text{body}(q_{i-1})) \wedge \dots \wedge (\text{head}(q_2) \in \text{body}(q_1)) \wedge (\text{head}(q_1) = c)$. Similarly, we can define c is *reachable* from p with respect to P .

Example 26. For the set of rules $P=\{A \leftarrow B, C, B \leftarrow D, E, D \leftarrow F, G, H \leftarrow I, J, I \leftarrow K, L, K \leftarrow M, N\}$, A is reachable from rules $B \leftarrow D, E$ and $D \leftarrow F, G$, and H is reachable from rules $I \leftarrow K, L$ and $K \leftarrow M, N$, while A is unreachable from rules $I \leftarrow K, L$ and $K \leftarrow M, N$ and H is unreachable from rules $B \leftarrow D, E$ and $D \leftarrow F, G$.

Definition 23. A *well-formed* schema S is defined as follows:

1. for $\forall p_1 \in \text{rules}(S) \forall p_2 \in \text{rules}(S) (\text{head}(p_1) \neq \text{head}(p_2))$;
2. for $\forall p \in \text{rules}(S) (p \text{ is not non-terminating with respect to } \text{rules}(S))$;
3. for $\forall p \in \text{rules}(S) (\text{root}(S) \text{ is reachable from } p \text{ with respect to } \text{rules}(S))$.

Definition 24. Let S be a schema, $E \in \mathcal{C}$ be a constant and P be a set of rules of the schema S . Then the projection E to P is defined as the set of rules $\Pi_E P = \{p \mid p \in P \text{ and } E \text{ is reachable from } p \text{ with respect to } P\}$.

For Example 25, $\Pi_B P = \{B \leftarrow E, F, E \leftarrow \#PCDATA, F \leftarrow \#PCDATA\}$, $\Pi_D P = \{D \leftarrow H, G, G \leftarrow \#PCDATA\}$. For Example 26, $\Pi_A P = \{A \leftarrow B, C$,

$B \leftarrow D, E, D \leftarrow F, G\}$ and $\Pi_H P = \{H \leftarrow I, J, I \leftarrow K, L, K \leftarrow M, N\}$.

Definition 25. Let $E \in \mathcal{C}$ be a constant and S be a schema. Then the ordered pair $(E, \Pi_{Erules}(S))$ is defined as the subschema of schema S with respect to E .

It is obvious that $\Pi_{Erules}(S) \subseteq rules(S)$ according to Definition 24.

Theorem 1. All subschemas of a *well-formed* schema are also *well-formed*.

PROOF. Suppose that S is a *well-formed* schema and $(E, \Pi_{Erules}(S))$, a subschema of S , is not well-formed, then

1. $\exists p_1 \in \Pi_{Erules}(S) \exists p_2 \in \Pi_{Erules}(S) (head(p_1) = head(p_2))$ or
2. $\exists p \in \Pi_{Erules}(S) (p \text{ is non-terminating with respect to } \Pi_{Erules}(S))$ or
3. $\exists p \in \Pi_{Erules}(S) (E \text{ is unreachable from } p \text{ with respect to } \Pi_{Erules}(S))$

Case 1. Suppose $\exists p_1 \in \Pi_{Erules}(S) \exists p_2 \in \Pi_{Erules}(S) (head(p_1) = head(p_2))$.

Because $\Pi_{Erules}(S) \subseteq rules(S)$, $p_1 \in rules(S)$ and $p_2 \in rules(S)$. Thus the expression $\exists p_1 \in rules(s) \exists p_2 \in rules(S) (head(p_1) = head(p_2))$ is true. Therefore schema S is not *well-formed* according to Definition 23. This is a contradiction since S is *well-formed*.

Case 2. Suppose $\exists p \in \Pi_{Erules}(S) (p \text{ is non-terminating})$.

Suppose that the set of rules $Q = \{q_1, \dots, q_n\}$ contains all rules in $\Pi_{Erules}(S)$ from which $head(p)$ is reachable. According to Definition 21, there exists a symbol $B \in body(q_1) \cup \dots \cup body(q_n)$ such that $\nexists r \in \Pi_{Erules}(S) (head(r) = B)$.

Because schema S is *well-formed*, B is *terminating* with respect to $rules(S)$. Therefore there exists a rule $s \in rules(S) (s \notin \Pi_{Erules}(S) \wedge head(s) = B)$. According to Definition 21, E is *reachable* from rule s with respect to $\Pi_{Erules}(S)$. Therefore $s \in \Pi_{Erules}(S) (head(s) = B)$. This is a contradiction with $\nexists r \in \Pi_{Erules}(S) (head(r) = B)$.

Case 3. Suppose $\exists p \in \Pi_{Erules}(S) (E \text{ is unreachable from } p)$.

This is a contradiction with Definition 25.

Therefore a subschema of a *well-formed* schema is also *well-formed*.

5.4.2 Semantics of Database

In this section, we first define *well-formed* objects and satisfaction of a *well-formed* object with respect to a *well-formed* schema. Then we define a class of *well-formed* database.

Definition 26. A *well-formed* object is defined inductively as follows:

1. A lexical object is always *well-formed*.
2. A list object $\{o_1, \dots, o_n\}$ is *well-formed* if each $o_i (1 \leq i \leq n)$ is a lexical object.
3. Let $@a \rightarrow o$ be an attribute object. Then it is *well-formed* if the following hold:
 - (a) a is a lexical object.
 - (b) o is either a lexical object or a *well-formed* list object.
4. Let $e \rightarrow o$ be an element object. Then it is *well-formed* if the following hold:
 - (a) e is a lexical object.
 - (b) o is a lexical object or a *well-formed* tuple object.
5. Consider a tuple object $(@a_1, \dots, @a_m, e_1, \dots, e_n)$ with $m \geq 0$ and $n \geq 0$. Then it is *well-formed* if the following hold:
 - (a) Each $@a_i (1 \leq i \leq m)$ is a *well-formed* attribute object.
 - (b) Each $e_i (1 \leq i \leq n)$ is a *well-formed* element object.

Theorem 2. Let o be a *well-formed* object. Then $\forall o' \in o$ are also *well-formed*.

PROOF. Immediate from Definition 26.

Definition 27. Let S be a *well-formed* schema, o be a *well-formed* named object and C be a constant. Then the satisfaction of o with respect to S is defined recursively as follows:

1. if o is a simple attribute object $@A \rightarrow C$, and $\forall p \in \text{rules}(S)(\text{head}(p)=A \wedge \text{body}(p)=C)$ then o is *satisfiable* for S ;
2. if o is a lexical list attribute object $@A \rightarrow \{C_1, C_2, \dots, C_n\}$ with $n \geq 1$, and $\forall p \in \text{rules}(S)(\text{head}(p)=A \wedge \text{body}(p)=IDREFS)$, then o is *satisfiable* for S ;
3. if o is a simple element object $E \rightarrow C$ and $\text{root}(S)=E$, $\exists p \in \text{rules}(S)(\text{head}(p)=\text{root}(S))$, and one of the following holds for rule p , then o is *satisfiable* for S ;
 - (a) p is an *atomic* rule;
 - (b) p is a *single* rule $E \leftarrow B$, and o is *satisfiable* for subschema $(\text{name}(B), \Pi_{\text{name}(B)}S)$
 - (c) p is an *optional* rule $E \leftarrow B?$, and o is *satisfiable* for subschema $(\text{name}(B), \Pi_{\text{name}(B)}S)$
 - (d) p is a *duplicate* rule $E \leftarrow B+$, and o is *satisfiable* for subschema $(\text{name}(B), \Pi_{\text{name}(B)}S)$
 - (e) p is an *arbitrary* rule $E \leftarrow B*$, and o is *satisfiable* for subschema $(\text{name}(B), \Pi_{\text{name}(B)}S)$
 - (f) p is a *choice* rule $E \leftarrow (E_1 | \dots | E_n)$ with $n \geq 1$, $\exists E_i \in \{E_1, \dots, E_n\}$ E_i is #PC-DATA or o is *satisfiable* for subschema $(\text{name}(E_i), \Pi_{\text{name}(E_i)}S)$.

- (g) p is a tuple rule $E \leftarrow (E_1, \dots, E_n)$ with $n \geq 1$, $\exists E_i \in \{E_1, \dots, E_n\}$ (E_i is #PC-DATA or o is *satisfiable* for subschema $(name(E_i), \Pi_{name(E_i)}(S))$), and for each other $\exists E_j \in \{E_1, \dots, E_n\}$ E_j is either #PCDATA, $E_j?$, E_j^* .
4. if o is an element object with a value of tuple $[E \rightarrow [\textcircled{a}_1, \dots, \textcircled{a}_m, e_1, \dots, e_n]]$ with $m \geq 0$ and $n \geq 0$, $root(S) = E$, and $\exists p \in rules(S) (head(p) = root(S))$, and one of the following holds for rule p , then o is satisfiable for S ;
- (a) $m=0$ and $n=0$, that is o is an empty object with element type E ;
 - (b) p is an *atomic* rule, $m=0$ and $n=1$, and e_1 is *satisfiable* for subschema $(name(e_1), \Pi_{name(e_1)}(S))$;
 - (c) p is a *single* rule $E \leftarrow B$, $m=0$ and $n=1$, $name(e_1) = B$, and e_1 is *satisfiable* for subschema $(name(e_1), \Pi_{name(e_1)}(S))$;
 - (d) p is an *optional* rule $E \leftarrow B?$, $m=0$ and $n=1$, $name(e_1) = B$, and e_1 is *satisfiable* for subschema $(name(e_1), \Pi_{name(e_1)}(S))$;
 - (e) p is a *duplicate* rule $E \leftarrow B+$, $m=0$, $name(e_i) = B (1 \leq i \leq n)$, and e_i is *satisfiable* for subschema $(name(e_i), \Pi_B(S))$;
 - (f) p is an *arbitrary* rule $E \leftarrow B^*$, $m=0$, $name(e_i) = B (0 \leq i \leq n)$, and e_i is *satisfiable* for subschema $(name(e_i), \Pi_B(S))$;
 - (g) p is a *choice* rule $E \leftarrow (B_1 | \dots | B_s)$, $m=0$ and $n=1$, and e_1 is *satisfiable* for subschema $(name(e_1), \Pi_B(S))$;
 - (h) p is a *tuple* rule $E \leftarrow (\textcircled{A}_1, \dots, \textcircled{A}_t, B_1, \dots, B_s)$, $m=t$ and $n \leq s$, and
 - (i) $a_i (q \leq i \leq m)$ is *satisfiable* for $\Pi_{name(a_i)_E}(S)$;
 - (ii) there is a sequence $B_{i1}, \dots, B_{in}$, for each $e_j (1 \leq j \leq n)$ $name(e_j) = B_{ij}$, for each $B_k (B_k \notin \{B_{i1}, \dots, B_{in}\} \wedge B_k \in \{B_1, \dots, B_s\} \wedge B_k$ is of the form either

$B_k?$ or B_k^*) and e_i is *satisfiable* for $\Pi_{name(e_i)}(S)$;

Theorem 3. Let S be a well-formed schema and o is satisfiable for S . Then for $\forall o' \in o$, o' is satisfiable for $\Pi_{name(o')}S$.

PROOF. Immediate from Definition 27.

Definition 28. A database $\mathcal{DB}=(S, o)$ is *well-formed* if the following hold:

1. S is a *well-formed* schema;
2. o is a *well-formed* element object;
3. o is *satisfiable* for schema S .

Definition 29. Let $C \in \mathcal{C}$ be a constant and $\mathcal{DB} = (S, o)$ be a database. Then a sub-database of $\mathcal{DB}$ with respect to C , denoted by $\Pi_C \mathcal{DB}$, is defined as the ordered pair $(\Pi_C S, \{o_1, \dots, o_n\})$ such that $o_1 \in o, \dots, o_n \in o$, $name(o_1) = \dots = name(o_n) = C$.

For example, for the database in Example 1 and 2,

$$\begin{aligned} \Pi_{student} \mathcal{DB} = & ((student, \{student \leftarrow @id_{student}, @supervisor_{student}, name, program, \\ & @id_{student} \leftarrow ID, \\ & @supervisor_{student} \leftarrow IDREF, \\ & name \leftarrow firstname, middlename?, lastname, \\ & firstname \leftarrow \#PCDATA, \\ & middlename \leftarrow \#PCDATA, \\ & lastname \leftarrow \#PCDATA, \\ & program \leftarrow \#PCDATA \}), \\ & student \rightarrow [@id \rightarrow S200, @supervisor \rightarrow F100, \\ & name \rightarrow [firstname \rightarrow Alisar, lastname \rightarrow Smith], \\ & program \rightarrow Software Engineering], \end{aligned}$$

$\text{student} \rightarrow [\text{@id} \rightarrow \text{S400}, \text{@supervisor} \rightarrow \text{F100},$
 $\text{name} \rightarrow [\text{firstname} \rightarrow \text{Jack}, \text{lastname} \rightarrow \text{Weiss}],$
 $\text{program} \rightarrow \text{Software Engineering}],$
 $\text{student} \rightarrow [\text{@id} \rightarrow \text{S600},$
 $\text{name} \rightarrow [\text{firstname} \rightarrow \text{Mary}, \text{lastname} \rightarrow \text{Lee}],$
 $\text{program} \rightarrow \text{Hardware Engineering}]]).$

Theorem 4. Given a *well-formed* database $\mathcal{DB} = (S, o)$ and its sub-database $\Pi_C \mathcal{DB} = (\Pi_C S, \{o_1, \dots, o_n\})$, then $\forall o_i (1 \leq i \leq n)$ is *satisfiable* for $\Pi_C S$.

PROOF. Immediate from Theorems 1, 2 and 3.

5.4.3 Semantics of Update Language

In the above sections, we defined the formal semantics of schema and database. In this section we first define the formal semantics of expressions. Then we define the formal semantics of the update language described in Section 5.1.3.

Definition 30. Let $\mathcal{DB} = (S, o)$ be a database. Then the *level* of a nested object $o' \in o$, denoted by $\text{level}(o')$, is defined as the depth of o' nested in o . In particular, the level of the root object in the database is defined as 0, i.e. $\text{level}(o) = 0$.

For example, in the database in Example 1 and 2, $\text{level}(\text{department}) = 0$, $\text{level}(\text{faculty}) = \text{level}(\text{student}) = 1$.

Definition 31. Let $\mathcal{DB} = (S, o)$ be a *well-formed* database. Then the semantics of expressions on $\mathcal{DB}$ are defined inductively as follows.

1. The semantics of *arithmetic*, *logical* and *string* expressions are defined in the usual way.

2. Let $L[A]$ be a list selection expression and L be the target list object $\{o_1, \dots, o_n\}$. Then the result of the expression is o_A in the case of $1 \leq A \leq n$. Otherwise the result is *null*.
3. Let $\$v=n$ be a variable name selection expression. Then the result of the expression is a string.
4. Let $\$v(e)$ be a variable object selection expression. Then the result of the expression is defined as $(\Pi_e S, \{o' | o' \in o \text{ and } name(o')=e \text{ and } e(o') \text{ is true}\})$.
5. Let $e \rightarrow \$v$ be a variable value selection expression. Then the result of the expression is as follows.
 - If $\$v$ hold a lexical term, then the result is the same as in 3 above.
 - If $\$v$ hold a tuple term, then the result is the same as in 4 above.
 - If $\$v$ holds an attribute or an element term e , then the result is $(\Pi_e S, \{o' | o' \in o \text{ and } name(o')=e\})$.
6. Let E_1/E_2 be a path selection expression. Suppose that the result of E_1 is $(\Pi_A S, \{o_{11}, \dots, o_{1m}\})$ and the result of E_2 is $(\Pi_B S, \{o_{21}, \dots, o_{2n}\})$. Then the result of the expression is defined as $(\Pi_B S, \{o' | o' \in \{o_{21}, \dots, o_{2n}\} \text{ and } \exists o'' (o'' \in \{o_{11}, \dots, o_{1m}\} \wedge o' \in o'' \wedge level(o'') - level(o') = 1)\})$. We can define the semantics of path selection expression $E_1//E_2$ in a similar way.
7. Let $P, (U)/P$ and $(U)//P$ be query expressions. We can define the semantics for query expressions $(U)/P$ and $(U)//P$ similar to item 6 except that the result of expression (U) is (S, o) . Here, consider the query expression P . If P is a path expression, then its semantics are defined by item 6. If it is an

element term, then its result is defined as the sub-database $(\Pi_P S, \{o' | o' \in o \text{ and } name(o')=P\})$.

Definition 32. Let o and s be objects, and $o' \in o$ a nested object of o . Then the *substitution* of object s for o' in o , $o(o'/s)$, is defined as: object o' within object o is replaced by object s .

Theorem 5. Let $\mathcal{DB}=(S,o)$ be a well-formed database and t be an object satisfiable for $\Pi_{name(t)}S$. For $\forall s \in o$, if $name(s)=name(t)$, then $o'=o(s/t)$ is satisfiable for S .

PROOF. The theorem can be proofed inductively as follows.

1. Assume $level(s)=0$, that is $s=o$, then $o'=t$ and $\Pi_{name(s)}S=S$. Because t is satisfiable for $\Pi_{name(t)}S$, $o'=o(s/t)$ is satisfiable for S .
2. Assume $level(s)=1$, o is composed of $o_1, \dots, o_{i-1}, o_i, o_{i+1}, \dots, o_n$ and $o_i = s$. Because $\mathcal{DB}=(S,o)$ is a well-formed database, $o_i(1 \leq i \leq n)$ is satisfiable for $\Pi_{name(o_i)}S$ according to Theorem 4. Thus, $o'=o(s/t)$ is composed of $o_1, \dots, o_{i-1}, t, o_{i+1}, \dots, o_n$. Because $name(s)=name(t)$ and t is satisfiable for $\Pi_{name(t)}S$, o' is satisfiable for S according to Definition 27.
3. The case $level(s)=n$ can be proven by induction.

Theorem 6. Let $\mathcal{DB}=(S,o)$ be well-formed, $o'=o(s/t)$ a substitution of object t for s in o , $level(s)-level(o'')=1$ (i.e. o'' is the parent object of s), and $o'''=o''(s/t)$. If o''' is satisfiable for $\Pi_{name(o''')}(S)$, then o' is satisfiable for S .

PROOF. Immediate from Theorem 5.

Definition 33. Let $\mathcal{DB} = (S, o)$ be a *well-formed* database, and *query* $q_1, \dots, q_n$ u an updating query on the database. Then the semantics of the updating query, $\mathcal{DB}'=(S, o')$, are defined as follows:

1. *insert* E into $\$v$.

- (a) $\$v$ is an element object with a value of tuple $s = name(s) \rightarrow [@a_1, \dots, @a_m, e_1, \dots, e_m]$. Then $o' = o(s/s')$ is defined as: if E is an attribute object, then $s' = name(s) \rightarrow [@a_1, \dots, @a_m, E, e_1, \dots, e_n]$; if E is an element object, then $s' = name(s) \rightarrow [@a_1, \dots, @a_m, e_1, \dots, e_n, E]$. According to Theorem 6, if s' is not satisfiable for $\Pi_{name(s)}S$, then the insert-into operation is not allowed.
- (b) $\$v$ is an object with a value of list object $s \rightarrow \{ID_1, \dots, ID_n\}$. Then $o' = o(s/s')$ and $s' = \{ID_1, \dots, ID_n, E\}$.
- (c) Otherwise, the insert-into operation is not allowed.

2. *insert* E before $\$v$. Assume the parent object of $\$v$ is s .

- (a) s is an element object with a value of tuple $s = name(s) \rightarrow [@a_1, \dots, @a_m, e_1, \dots, e_m]$. Then $o' = o(s/s')$ is defined as: if $\$v$ is an element object and $e_i = \$v$, then $s' = name(s) \rightarrow [@a_1, \dots, @a_m, e_1, \dots, e_{i-1}, E, e_i, \dots, e_n]$. According to Theorem 6, if s' is not satisfiable for $\Pi_{name(s)}S$, then the insert-before operation is not allowed.
- (b) s is an object with a value of list object $s \rightarrow \{ID_1, \dots, ID_n\}$ and $\$v = ID_i$. Then $o' = o(s/s')$ and $s' \rightarrow \{ID_1, \dots, ID_{i-1}, E, ID_i, \dots, ID_n\}$.
- (c) Otherwise, the insert-before operation is not allowed.

3. *insert* E after $\$v$. We can define the semantics of the insert after operation in a similar way to the insert before operation.

4. *delete* $\$v$.

- (a) If $\$v$ is the value of object s , then $o'=o(s/s')$ and s' is $\text{name}(s) \rightarrow \text{null}$.

According to Theorem 6, if s' is not satisfiable for $\Pi_{\text{name}(s)}S$, then the deletion operation is not allowed.

- (b) If the parent object of $\$v$ is an element object with a value of tuple

$s = \text{name}(s) \rightarrow [@a_1, \dots, @a_m, e_1, \dots, e_n]$, then if $\$v = @a_i$ then

$s' \rightarrow \text{name}(s) \rightarrow [@a_1, \dots, @a_{i-1}, @a_{i+1}, \dots, @a_m, e_1, \dots, e_n]$; if $\$v = e_i$, then

$s' = \text{name}(s) \rightarrow [@a_1, \dots, @a_m, e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n]$; If e_i has an id attribute

@id is a reference by another element r

$r = \text{name}(r) \rightarrow [@a_{r1}, \dots, @a_{ri} \rightarrow \{id_1, \dots, id_{i-1}, id, id_{i+1}, \dots, id_k\}, @a_{rm}, e_{r1}, \dots, e_{rn}]$,

then, $r' = \text{name}(r) \rightarrow [@a_{r1}, \dots, @a_{ri} \rightarrow \{id_1, \dots, id_{ri-1}, id_{ri+1}, \dots, id_k\}, @a_{rm}, e_{r1}, \dots, e_{rn}]$.

According to Theorem 6, if s' is not satisfiable for $\Pi_{\text{name}(s)}S$ and r' is not satisfiable for $\Pi_{\text{name}(r)}S$, then the deletion operation is not allowed.

- (c) Otherwise, the deletion operation is not allowed.

5. replace $\$v$ with E .

- (a) Assume $\$v$ is a value of object s . Then a new object s' is constructed with

$\text{name}(s) \rightarrow E$ and $o' = o(s/s')$. According to Theorem 5, if s' is not satisfiable for $\Pi_{\text{name}(s)}S$, then the replacement operation is not allowed.

- (b) Assume $\$v$ is the name of object s . Then a new object s' is constructed

with $E \rightarrow \text{value}(s)$ and $o' = o(s/s')$. Assume that t is the parent of s . According to Theorem 6, if $t' = t(s/s')$ is not satisfiable for $\Pi_{\text{name}(t)}S$, then the replacement operation is not allowed.

- (c) Assume $\$v$ is an object and t is the parent object of $\$v$ with a value of

tuple $t = \text{name}(t) \rightarrow [@a_1, \dots, @a_m, e_1, \dots, e_n]$. If $\$v = @a_i$ then a new object t' is constructed with $\text{name}(t) \rightarrow [@a_1, \dots, @a_{i-1}, E, @a_{i+1}, \dots, @a_m, e_1, \dots, e_n]$ and

$o'=o(t/t')$. Similarly, if $v=e_i$, a new object t' is constructed with $\text{name}(t) \rightarrow [@a_1, \dots, @a_m, e_1, \dots, e_{i-1}, E, e_{i+1}, \dots, e_n]$ and $o'=o(t/t')$. According to Theorem 5, if t' is not satisfiable for $\Pi_{\text{name}(t)}S$, then the replacement operation is not allowed.

- (d) Otherwise, the replacement operation is not allowed.

Chapter 6

Design and Implementation of XML-RL System

In this chapter, we present the design and implementation of our XML-RL system. We start with defining a set of system requirements, then we present the architecture of the system. Our core system structure will be presented in section 6.3.

6.1 Requirements of XML-RL System

To support the design goals of the XML-RL language, our XML-RL system is designed to meet the following requirements:

1. Functionality requirements.
 - 1.1 System must support all the syntax and operations of the XML-RL language.
 - 1.2 System must support querying and updating XML files over the network.
 - 1.3 System must support the validation of both the original and updated XML document against its DTD if there is one.
 - 1.4 System should provide an interface with both human operator and other software

programs.

2. Reliability requirements:

2.1 System should provide a commit point. All changes can only take effect after this commit point. All changes before the commit point can be rolled back.

2.2 System should record all changes to log for future reference.

3. Security requirements:

3.1 System must have a user management subsystem and prevent unauthorized users from accessing XML documents for which they don't have privileges.

4. Performance requirements:

4.1 System should try to find balance between memory usage and data accessing time. Loading all related XML documents into memory would save data accessing time. However, as the size of XML documents increases, loading all related XML documents into memory could cause the system to run out of memory.

6.2 XML-RL System Architecture

The XML-RL system is composed of three sub-systems: *User Interface*, *Web Server* and *Language Processor*. Their relationship is shown in Figure 6.1.

6.2.1 User Interface

User interface is responsible for taking commands entered by users, sending them to the server for processing, receiving the results from the server and finally displaying the results to the user. The user input includes system administrative commands,

Figure 6.1: XML-RL System Architecture

query and update requests and other system supported commands. There are three user interfaces available in the system: *command line interface*, *graphical user interface* and *web interface*. They reside in the client side and are connected to the system server using a network connection. Users need to install our client package before they can run our *command line interface* and *graphical user interface*, while the *web interface* only needs a web browser. The *web interface* only supports query and update requests. *Command line interface* and *graphical user interface* have the same functionalities and support all commands. Our implemented prototype system currently supports *command line interface*. An user may start the XML-RL Command Line Interface by typing "XML-RL" along with the user name and password. On XML-RL prompt, the user can type in query and update command or other commands supported by the system. The system will display the processing status to the user while it is processing the command. After the processing is completed, it will

display the result to the user. Figure 6.2 shows the screen shot of XML-RL CLI when we tried to use the replace command to update the position of a faculty.

```
C:\WINDOWS\System32\cmd.exe
E:\>XML-RL -u root -p
password:*****
***** Welcome to XML-RL CLI *****

xml-rl>query
>http://localhost/csdept.xml/
>csdept/faculty->lname->if firstame->Bob,lstname->Smith,position->$p1
>replace $p with Professor
>g
processing ...
operation successful

xml-rl>query
>http://localhost/csdept.xml/
>csdept/$f<faculty->lname->if firstame->Bob,lstname->Smith1)
>constructing
>result.xml/csdept/$f
>g
processing ...
operation successful

result.xml
<csdept>
  <faculty id="F200">
    <name>
      <firstname>Bob</firstname>
      <lastname>Smith</lastname>
    </name>
    <position>Professor</position>
    <salary>$40,000</salary>
  </faculty>
</csdept>

xml-rl>
xml-rl>quit
E:\>
```

Figure 6.2: XML-RL Command Line Interface

6.2.2 Web Server

As a requirement of the XML-RL language, the XML-RL system client and updated XML documents could reside over the network. Therefore, our system must have a network server to support network communications. We chose a *web server*

to fulfill this task based on the following reasons:

First, a *web server* can do whatever other communication methods can do, including taking local and remote clients' requests, sending back the results to the clients and uploading and downloading documents.

Second, the *web server* is standardized. This will give our system the extendability and scalability.

Third, we need a *web server* anyway since the system is going to support a web interface.

Fourth, there are many free web servers. Among them, Apache Tomcat is the most famous. Using the *web server* will free our development from concerning about the communication details.

The *web server* in our system is responsible for the communication between the language processor and the client interface. The tasks of the web server are as follows:

- Taking user requests from the client interface, either from a command line interface, a GUI, or a web interface (browser);
- Calling the corresponding software module in the language processor to handle the request;
- Retrieving the result from the corresponding module;
- Finally, formatting the result and sending the result back to the client.

Apache Tomcat is chosen as our web server. By using Tomcat web server, we only need to develop Servlets to fulfill the above tasks with minimum awareness of communication details.

Servlets are modules of Java code that run in a server application to answer client requests. Servlets make use of the Java standard extension classes in the packages

javax.servlet (the basic Servlet framework) and javax.servlet.http (extensions of the Servlet framework for Servlets that answer HTTP requests). Since Servlets are written in the highly portable Java language and follow a standard framework, they provide a means to create sophisticated server extensions in a server and operating system independent way.

The contents of incoming requests from the client can be found in the message header and the message body. When the XML-RL Servlet handles the request, it always first extracts related information from the header. From the message header, the Servlet will know who makes the request and what the client wants to do. If it is a XML-RL query/update request, it will extract the content of the request from the message body and call the language processor to process the request.

6.2.3 Language Processor

The XML-RL Language Processor is responsible for handling users' requests and is where the query and update operations are done. It is comprised of *Authentication Server*, *Logging System*, *Locking Scheduler* Subsystems and *Query/Update Processor*. Figure 6.3 shows their relationship.

In our language processor, the *Query/Update Processor* is the core subsystem. Other subsystems play the service roll for the core system. We will present its detail in section 6.3.

The *Authentication Server* is responsible for user authentication and user session handling. Each time a request comes into the *Query/Update Processor* in the language processor, after the *Query/Update Processor* parses the request and knows the content of the request, it will call the *Authentication Server* to check if the user has the privilege to do the operation or not. If yes, the request will be processed. The user

Figure 6.3: XML-RL Language Processor

privilege includes his right to read from and write to a specific document.

The *Logging System* is responsible for recording all the changes made to a document. Whenever a system failure or operation abort happens, we can restore the original state of the document.

The *Locking Scheduler* is responsible for concurrent control of the system. Whenever a read or a write operation is executed, the *Query/Update Processor* will call the *Locking Scheduler* for permission before it can proceed. The *Locking Scheduler* will either place a read lock, a write lock or reject the operation according the locking status of the requested object.

Currently the *Logging System* and the *Locking Scheduler* have not been implemented in our prototype system. As there are many issues needed to be investigated to implement them, they could be the potential thesis topics. We will leave these as the future works.

6.3 Query/Update Processor

The *Query/Update Processor* is the core system of our XML-RL system. All the query/update requests are processed here.

The XML-RL Query/Update Processor system consists of three modules: the *XML Document Loader*, the *Command Handler* and the *Operation Processor*. The *XML Document Loader* loads involved XML documents into the XML-RL system, while the *Command Handler* parses the user command into the form which our *Operation Processor* can understand. The *Operation Processor* is in charge of analyzing language, processing query/update requests and managing the XML-RL data model. Their relationship is shown in Figure 6.4. We will discuss these parts and their relations in detail as follows.

Figure 6.4: XML-RL Query/Update Processor

6.3.1 Command Handler

The Command handler is composed of *Command Dispatcher* and *Command Parser*.

The *Command Parser* parses the incoming command into the form required by our *Query/Update Processor*, and then passes it to the *Command Dispatcher*. JLEX is used to construct a lexical analyzer to break up the characters of the command into meaningful tokens and then JCUP is used to generate a LALR parser based on the specification of our XML-RL language grammar.

The *Command Dispatcher* takes the parsed command and executes the command. For query/update requests, it will try to locate the related XML documents. If the document is located at remote sites, it will try to download the XML document to the local host using HTTP request. Then it will call the *XML Loader* to load the XML document into the memory. Next, it will call the *Operation Processor* to process the query or update request and wait for the result. Finally, after receiving the result, it will send the result back to the client.

6.3.2 XML Loader

The XML Loader is composed of *XML Parser* and *Model Transformer*.

The *XML Parser* uses JDOM API to parse the XML document into JDOM Object Model. The reason we use JDOM is that it is designed for Java application, therefore it is lightweight, optimized, and much easier for a Java developer to use.

The *Model Transformer* takes the JDOM and builds the XML-RL data model based on the JDOM. We implement XML-RL mapping rules defined in Section 4.2 to map XML-RL data model to JDOM. From the implementation point of view, XML-RL data model is recursively composed of a pair of name and content. The content can be 0 to n elements, 0 to n attributes, or a string, or their combinations. All these

objects can be retrieved from the corresponding element object of JDOM. Therefore, when we construct the XML-RL data model, the major member of the XML-RL node is the reference to the corresponding JDOM element. Doing this not only saves the system memory, but also allows us use the API provided by the JDOM package for a better performance.

6.3.3 Operation Processor

When the *Operation Processor* receives a valid query/update requests from the *Command Dispatcher*, it will process the requests as follows:

First, it will check the command semantics, and, if passed, the process will proceed.

Second, it constructs a variable guide tree to list all variables from the query statement and their possible representing values. As shown in Figure 6.5, the variable guide tree is implemented as a *general tree*. Each node contains two parts: the *variable path* and *variable name*. The *Variable Guide* tree is used to keeps track of the relationships between variables and the object types they represent. This information will be used to construct the *Result Repository* and guide the performing of query/update operations.

Figure 6.5: Variable Guide

Third, it will construct a *Result Repository* tree. The *Result Repository* is used to keep possible values of variables satisfying the query predicates. Also it keeps the hierarchical relationships amongst the values of the variables, which plays an important role for result constructing and update performing. As shown in Figure 6.6, which corresponds to the *Variable Guide* in Figure 6.5, the *Result Repository* is also implemented as a *general tree*. There are two kinds of nodes: *variable nodes* represented as bold rounded rectangles, and *value nodes* represented as shadowed rounded rectangles. The *variable nodes* are ordered in the *Result Repository* based on the order they appear in the query. Each variable node has a pointer pointing to its previous viable node and another pointer pointing to the possible value nodes it will represent. The *value node* contains two pointers: *parent pointer* and *sibling pointer*. Each variable value in a *value node* is a reference to corresponding XML-RL objects rather than a real object value. The query condition will apply to the corresponding value node in the *Result Repository* tree. If the node satisfies the condition, it will be marked as true. Otherwise, it will be marked as false. All sibling nodes marked as true are now referred by the corresponding variable node.

Fourth, if the command is an update command, it will construct an update result tree by copying the original XML-RL modelled data as its initial data. Then it will use the query result in *Result Repository* and the update command to apply the update operation to this update result tree. As we mentioned previously, the value node in the *Result Repository* tree is actually a reference to a corresponding XML-RL objects in the XML-RL modelled data. This gives us the convenience to apply the update command to the update result tree through the *Result Repository* tree.

Finally, it will transform the result tree back to an XML document, validate the document, and send this document to the client through the web server.

Figure 6.6: Result Repository

Figure 6.7 shows the class structure of the *Operation Processor* module.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we present the design and implementation of the XML-RL update language. We review some of the major XML update languages proposed in recent years and present our comments on their advantages as well as their limitations. We then present the formal syntax and semantics for an update extension to XML-RL query language. We propose a minimal set of primitive update operations and rules to fulfill the update requirements. Our update language incorporates some techniques and ideas from the existing updating languages with our uniform framework that is advantageous over other XML update languages because: our language is logically specified in a high-level and concise manner. It can represent update requests in a simple and natural, but powerful way; the update operation can be applied to more than one XML document in the same time; it supports updates in both ordered and unordered XML data; it modify both meta-data and data in a unified way by the introduction of three kinds of logical binding variables: *object variable*, *value variable*, and *name variable*; We designed the architecture for the XML-RL system. Our architecture considered the issues of performance, security, scalability and optimization.

It can fully utilize all the power of the XML-RL language. We also implement a prototype system to demonstrate the basic functions of our system.

7.2 Future Work

There are some points that could not be accommodated within the time frame of this thesis, that are worth investigating in a future work. In particular, we plan to investigate the following aspects:

- Query and update optimization: our language still lacks a cost-based query optimizer which is necessary for a fully functional XML query system. This will be left to future investigation.
- Update transaction processing: there are still no mechanisms to handle concurrent transactions and recoverability(for both transaction abort and system failure) for XML-RL. For a fully functioning system, we will investigate a serializable locking scheduler and a logging system in the future.
- Full implementation: the current prototype system only implemented the basic features of our system. It did not support update over the network and provide a programming interface for applications; it did not implement the GUI and WEB interface. A fully functional XML-RL system will be implemented in the future.

Bibliography

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundation of Databases*. Addison Wesley, 1995.
- [2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener. *The Lorel Query Language for Semistructured Data*. Int. J. on Digital Libraries 1997, 1(1): 68-88.
- [3] Goldman, R., Mchugh, J., Widom, J. *From semistructured data to XML: Migrating the Lorel data model and guery language*. In ACM SIGMOD WEBDBWorkshop99, Philadelphia, Pennsylyania. (1999) 25-30
- [4] A. Bonifati and S. Ceri. *Comparative Analysis of Five XML Query Languages*. SIGMOD Record 29(1): 68-79 (2000).
- [5] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, L. Tanca. *XML-GL: A Graphical Language for Querying and Restructuring XML Documents*. Proceedings of the Eighth International World Wide Web Conference, Toronto, Canada May 1999.
- [6] J. Cark and S. DeRose. *XML Path Language (XPath)*. ver. 1.0. Tech. Report RECxpath- 19991116, W3C, Nov. 1999.
- [7] Y. Chen and P. Revesz. *CXQuery A Nodel XML Query Language*. <http://citeseer.nj.nec.com/539624.html>.

- [8] A. Deutsch, M.F. Fernandez, D. Florescu, A.Y. Levy, D. Suci. *A Query Language for XML*. Proceedings of the Eighth International World Wide Web Conference, Toronto, Canada May 1999.
- [9] M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, N. Walsh. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft 15 November 2002.
- [10] M. Liu. *A logical foundation for XML*. In Proceedings of the 14th International Conference Advanced Information Systems Engineering(CAiSE02), Toronto, Canada, May 2002, 568-583.
- [11] M. Liu and T.W. Ling. *Towards declarative XML querying*. In Proceedings of The 3rd International Conference on Web Information Systems Engineering(WISE02), Singapore, December 2002.
- [12] W. May. *XPathLog: A Declarative, Native XML Data Manipulation Language*. In proceedings of International Database Engineering Applications Symposium, IDEAL01, July 2001, Grenoble, France. 123-128.
- [13] T. Ruhl and H. E. Bal. *The nested object model*. In Proceedings of 6th ACM SIGOPS European Workshop on Matching Operating Systems to Application Needs, September 1994, Dagstuhl Castle, Germany, 134-137.
- [14] H. Ishikawa, K. Kubota, Y. Kanemasa. *XQL: A Query Language for XML Data*. W3C Workshop on Query Language, Boston, Massachussets, USA, 1998.
- [15] Martin Bryan. *An Introduction to the Standard Generalized Markup Language(SGML)*. <http://www.personal.u-net.com/sgml/sgml.htm>
- [16] W3C. *Extensible Markup Language (XML)*. <http://www.w3.org/XML/>

- [17] Elizabeth Castro. *HMML For The World Wide Web*. Peachpit press, 1998
- [18] W3C. *XML DTD*.
- [19] W3C. *XML Schemas*. <http://www.oasis-open.org/cover/schemas.html>
- [20] Oracle. *XML SQL Utility (XSU)*
<http://www.sbg.ac.at/docu/man/oracle/9i/appdev.901/a88894/adx07xsu.htm>
- [21] Igor Dayen. *Storing XML in Relational Databases*
<http://www.xml.com/pub/a/2001/06/20/databases.html>
- [22] IBM. *DB2 XML Extender*.
<http://www-3.ibm.com/software/data/db2/extenders/xmlext/>
- [23] IBM. *Web services Object Runtime Framework: implementing Web services with XML Extender, Version 7.2 (Beta)*
<http://www-3.ibm.com/software/data/db2/extenders/xmlext/docs/v72wrk/dxxservl.htm>
- [24] W3C. *Document Object Model (DOM)*. <http://www.w3.org/DOM/>
- [25] Henry S. Thompson. *An Introduction to XSL*.
<http://www.ltg.ed.ac.uk/ht/swindon.html>
- [26] SAX. <http://www.saxproject.org/?selected=about>
- [27] Arciniegas A. *XML Developer's Guide*. McGraw-Hill, 1 edition, 2001.
- [28] D. Chamberlin, J. Robie, D. Florescu. *Quilt: An XML Query Language for Heterogeneous Data Sources*. In Proc. of Third Intl Workshop WebDB, Dallas, USA, May 2000, 1-25.

- [29] P. Atzeni, G. Mecca, and P. Merialdo. *Semistructured and structured data in the web: Going back and forth*. In Workshop on Management of Semistructured Data, 1997.
- [30] M. Fernandez, D. Florescu, and A. Levy. *A Query Language for a Web-Site Management System*. SIGMOD Record, 26(3):411, 1997.
- [31] D. Lee and W. Chu. *Comparative analysis of six XML schema languages*. SIGMOD Record, 29(3):7687, 2000.
- [32] I. Tatarinov, Z.G. Ives, A.Y. Halevy, D.S. Weld. *Updating XML*. In Proceedings of 2001 SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, May 2001.
- [33] W3C. *XML Query Requirements*. <http://www.w3.org/TR/xmlquery-req>
- [34] Berthod Daum, Udo Merten. *System Architecture With XML*. Morgan Kaufmann. 2003
- [35] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, Dan Suciu *XML-QL: A Query Language for XML*. <http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/>
- [36] Sophie Cluet, Jerome Simon *YATL: A Functional and Declarative language for XML* <http://citeseer.nj.nec.com/cluet00yatl.html>
- [37] S.Ceri, G.Gottlob, L.Tanca. *Logic Programming and Database*. Springer-Verlag Berlin Heidelberg 1990

- [38] Sara Comai, Ernesto Damiani, Roberto Posenato, Letizia Tanca. *A schema-based approach to modeling and querying www data*. In Proceedings of FQAS'98, Roskilde, May 1998.
- [39] Dallon Quass. *Ten Features Necessary for an XML Query Language*
<http://www.w3.org/TandS/QL/QL98/pp/quass.html>
- [40] Li Lu, Mengchi Liu, Guoren Wang *XML Update Language*. In Proceedings of the 15th Conference On Advanced Information Systems Engineering(CAISE'03), Klagenfurt/Velden, Austria, 16-20 June, 2003.
- [41] Guoren Wang, Mengchi Liu, Li Lu *Extending XML-RL With Update*. In Proceedings of 7th International Database Engineering and Application Symposium (IDEAS 2003), Hong Kong, July 16-18, 2003.

Appendix A

Syntax of XML-RL Language

<i>query</i>	::=	<i>qexp</i> ' , ' <i>qexp</i>
<i>qexp</i>	::=	FullElement
FullElement	::=	Url ElementPath
ElementPath	::=	ElementPathUnit ElementPath ElementPathUnit
ElementPathUnit	::=	DepthID Element DepthID Attribute DepthID Variable
DepthID	::=	'/' '//'
Variable	::=	SingleVar ListVar ObjectVar
SingleVar	::=	' \$ ' IDENTIFIER
ListVar	::=	' { ' SignleVar ' } '
ObjectVar	::=	SingleVar ' (' Element ') '

uexp	::=	'insert' Content 'into' Variable 'insert' Content 'after' Variable 'insert' Content 'before' Variable 'delete' Variable 'replace' Content 'with' Variable
Content	::=	IDENTIFIER {IDENTIFIER } Variable Element Attribute Tuple
FileName	::=	IDENTIFIER '.xml'
Url	::=	URL FileName
AttList	::=	Attribute ', ' Attribute
EleList	::=	Element ', ' Element
ElementName	::=	IDENTIFIER
AttributeName	::=	IDENTIFIER
Attribute	::=	AttributeName→IDENTIFIER AttributeName→{IDENTIFIER} AttributeName→Variable
Element	::=	ElementName→IDENTIFIER ElementName→Element ElementName→Tuple ElementName→ {Tuple}

		ElementName → Variable
Tuple	::=	' [' AttList ' , ' EleList '] '
Operator	::=	>
		<
		=
		<=
		>=
		!=
		+
		-
		*
		÷