THE UNIVERSITY OF CALGARY

COCALEREX: An Engine for Converting Catalog-based and Legacy Relational Databases into XML

by

Chunyan Wang

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
NOVEMBER, 2004

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "COCALEREX: An Engine for Converting Catalog-based and Legacy Relational Databases into XML" submitted by Chunyan Wang in partial fulfillment of the requirements for the degree of Master of Science.

Reda Alhajj, Ph.D., Supervisor
Computer Science

Ken Barker, Ph.D.
Computer Science

Svetlana N. Yanushkevich, Ph.D.
Electrical and Computer Engineering

Date

Abstract

Researchers mainly focus on finding XML schema for a relational database with a corresponding well-defined catalog. However, not all existing relational databases have all required catalog information; also legacy relational databases exist and should be re-engineered into XML. Motivated by this, we present COCALEREX, which is a data-analysis-based approach for converting into XML both relational databases providing the required catalog information fully or partially and legacy relational databases without well-defined metadata. We apply reverse engineering to extract from the analyzed database the Entity-Relationship (ER) model, which is converted into XML schema. Deriving the ER model empowers the proposed approach to smoothly consider binary and n-ary relationships during the mapping into XML. COCALEREX is capable of producing flat and nested XML schema. It also has a user-friendly interface that displays the result of each phase of the process. Experimental results are encouraging, demonstrating the applicability and effectiveness of the proposed approach.

Acknowledgments

I would like to express my sincere appreciation to my supervisor Dr. Reda Alhajj for his guidance and enlightenment during my study at The Department of Computer Science at The University of Calgary.

I would like to thank Mr. Anthony Lo who spent many hours working with me through discussion and implementation.

I would also like to thank the other members of my examining committee: Dr. Ken Barker and Dr. Svetlana N. Yanushkevich for reviewing this work and for their valuable comments.

My gratitude goes to my husband Xuming Chen, my daughter Cong Chen, and my son Xi Richard Chen for their supports.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

eXtensible Markup Language (XML) has been defined by the World Wide Web Consortium (W3C) and is considered the new universal format for structured documents and data on the Internet. As the Internet is becoming a major means of disseminating and sharing information, and as the amount of XML data is increasing substantially, there is an increasing demand to store, manage, and query XML in an efficient way. Thus, both legacy and catalog-based relational databases should be mapped into XML. In this thesis, we developed a user-friendly system that satisfies this purpose.

We are interested in the following four main processes: (1) extracting the ER model from an existing catalog-based or legacy relational database by applying reverse engineering techniques; (2) transforming the ER model to flat or nested XML schema; (3) transforming an XML schema into relational schema; and (4) publishing XML document with XML schema.

1.1 The Motivation and Overview of the Proposed Solution

XML has emerged and is being gradually accepted as the standard format for publishing and exchanging data over the Internet. Since most business data currently stored and maintained in RDBMS is still dominant, it is important to automate the process of generating XML documents containing information from existing databases. We

try to preserve as much information as possible during the transformation process. The Relational-to-XML transformation involves mapping names of the relational tables and attributes into XML names of elements and attributes, creating XML hierarchies, and processing values in an application specific manner. As described in the literatures, researchers primarily consider transforming relational databases that have rich corresponding catalogs. Although a large number of the existing relational databases are classified as legacy, the transformation of legacy relational databases to XML has received little attention. Legacy database systems are characterized by old-fashioned architecture, lack of the related documentation (missing or vague catalog), and non-uniformity resulted from numerous extensions.

Realizing the importance of transforming legacy databases into XML documents, we have developed a system, named **Co**nversion of **Ca**talog-based and **Le**gacy **Re**lational databases to **X**ML (COCALEREX) which handles the transforming process for both catalog-based and legacy databases. Our approach benefits from the previous finding on reverse engineering of legacy databases detailed by Alhajj [Alh03], which leads to the understanding of the design of an existing relational database. Unfortunately, some commercial RDBMS do not support the functionality necessary to retrieve primary and foreign keys information from metadata. For example, even the latest version of MySql does not fully support such functionality. COCALEREX can extract all the required metadata either from the catalog or by analyzing the database content.

Two basic steps are identified in the process of transforming relational databases into XML. First, the ER model from the given relational database is reconstructed. This process requires knowing the metadata. For legacy relational databases, reverse

engineering is employed to deduce information about functional dependencies, keys, and inclusion dependencies. For catalog-based databases, COCALEREX connects to the utilized RDBMS using JDBC/ODBC to obtain all the required available metadata information; reverse engineering is employed here to extract information missing from the catalog, if any. Second, the obtained ER model is transformed into XML schema in a process called *forward engineering*. Our approach smoothly handles all types of relationships allowed in the ER model, including many-to-many and nary relationships. COCALEREX is capable of doing two-way mapping, i.e., it can also transform an XML schema into a corresponding relational schema.

COCALEREX has been developed, based on the framework described in [WLAB04]. Users may use COCALEREX to specify viewing the underlying relational data as either flat or nested XML structure. Also, users can specify the nesting sequence. Users can directly view the result of each phase during the process. Finally, COCALEREX is composed of four main modules: 1) EELRR- **E**xtracting **E**R Model from **L**egacy **R**elational Database by **R**everse Engineering Module; 2) EECR- **E**xtracting **E**R Model from **C**atalog-based **R**elational Database Module; 3) ER2X- **ER** model to **XML** Module; 4) X2R- **X**ML schema to **R**elational schema Module. Experimental results are encouraging, demonstrating the applicability and effectiveness of the proposed approach.

1.2 Contributions

The outcome of this thesis is a user-friendly approach that equally converts legacy and catalog-based database into XML with a corresponding flat or nested schema.

The main contributions of this thesis may be enumerated as follows:

1. We have developed a system, named COCALEREX (**Co**nversion of **Ca**talog-based and **Le**gacy **Re**lational databases to **X**ML), which handles the reengineering of relational databases into XML. The implemented reverse engineering process constructs produces an ER model for a legacy database. The same process is also used for catalog-based databases when the catalog information is, at least, partially available. The forward engineering process produces the XML schema and the XML document(s).

2. COCALEREX can be used as a tool to redesign existing relational databases and can let users view XML of the underlying relational data.

3. Two-way mapping between relational databases and XML is provided.

4. COCALEREX can effectively map many-to-many and nary relationships.

5. COCALEREX can generate nested XML structures base on users' specification.

6. COCALEREX provides a user-friendly interface for users to the view the result of each phase of the process.

1.3 Outline of the Thesis

The thesis is organized as follows.

Chapter 2 gives an overview of the background material necessary to understand the work carried out in this thesis, including XML, XML schema, XML

query languages, other XML technologies, and XML database systems.

Chapter 3 discusses the transformation from legacy databases into XML. It covers the process of extracting the information necessary to produce the ER model with many-to-many and n-ary relationships identified. Identifying such relationships adds flexibility to the conversion process.

The transformation from XML into relational schema is discussed in Chapter 4.

In Chapter 5, more details are presented about COCALEREX including its architecture, and implementation. We also give some of the results.

Finally, we conclude this work in Chapter 6 with discussion about the remaining problems and future work.

Chapter 2

Background Material

2.1 Introduction

XML is considered the standard universal format for structured and exchanged data over the Internet. It has been designed to improve the functionality of the Web by providing more flexible and adaptable information identification. It is also extensible, unlike the HyperText Markup Language (HTML) which has a fixed format for displaying predefined data on the Web. The flexible nature of XML makes it an ideal basis for defining arbitrary languages. Although the XML syntax is flexible, it is constrained by a grammar that governs the permitted tag names, attachment of attributes to tags, and so on. All XML documents must conform to these basic grammar rules defined by an XML schema language (e.g., a Document Type Definition (DTD) or an XML schema). An XML document is valid if it conforms to a given DTD or an XML schema.

Relational databases get more and more employed to store the content of a website. At the same time, XML is fast emerging as the dominant standard at the hypertext level of website management describing pages and links between them. Thus, the integration of XML with relational databases to enable the storage, retrieval, and update of XML documents is of major importance. Since there are several disadvantages of this approach that are discussed later in this chapter, many researchers have developed some native XML repositories to manage XML documents.

To better understand this thesis, we introduce some basic information about XML in Section 2.2. XML databases are presented in Section 2.3.

2.2 Basic Knowledge about XML

As the World Wide Web (WWW) becomes a major means of disseminating and sharing information, there is an exponential increase in the amount of data in a web-compliant format such as HTML and XML. Specifically, the XML model is a novel textual representation of hierarchical (tree-like) data where a meaningful piece of data is bounded by matching starting and ending tags, such as <name> and </name>. Due to the simplicity of XML compared to Standard Generalized Markup Language (SGML) and its relatively powerful expressiveness compared to HTML, XML has become ubiquitous. XML data has to be managed in databases.

2.2.1 What is XML?

When asked for a concise definition of XML, one could say that XML is a *markup language for structured documents* [XML02]. It has been standardized by W3C and serves as a syntactical framework for data interchange on the Internet. XML's support from leading players in the software and communications industry has facilitated its quick assumption of the role of the ubiquitous and universal data interchange format.

XML is a rooted tree representation of data. One XML document or document fragment representing a book is shown below, and the tree it represents is shown in Figure 2.1.

<Book>

<Author>A. B. Chaudhri</Author>

<Author>A. Rashid</Author>

<Author>R. Zicari</Author>

<Publisher name='Addison-Wesley' />

</Book>

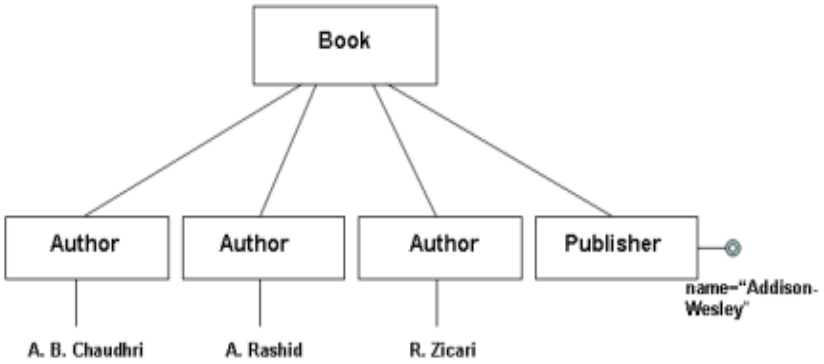


Figure 2.1: The tree representation of the book XML document

An XML document contains seven types of constructs. Of the seven, the most relevant constructs are **elements**, **attributes**, and **values**. In this example, the Book, the these Authors and the Publisher are elements; name is an Attribute; "A. B. Chaudhri", "A. Rashid", "R. Zicari" and "Addison-Wesley" are values. Values can occur as children of element or as attribute values. In this thesis, we prefer the element-oriented, rather then the attribute-oriented approach. XML also has

the notion of child elements. For example, the child elements of Book are the two Author elements and the Publisher element.

An XML document typically consists of two major components: schema and data. The schema describes the data and is specified in one of the proposed schema language notations. One of the most popular XML schema languages is DTD. XML is a textual representation of the hierarchical data model. The meaningful piece of the XML document is bounded by matching starting and ending tags such as <name> and </name>. The main building blocks of the XML model are element and attribute as follows:

<elem-name attr-name='attr-val'> elem-content </elem-name>

In DTD, elements and attributes are defined by the keywords <!ELEMENT> and <!ATTLIST>, respectively.

<!ELEMENT> <elem-name> <elem-content-model>

<!ATTLIST> <attr-name> <attr-type> <attr-option>

Element content model is the logical structure of the element contents based on the regular expressions such as "?" (0 or 1 instance), "*" (0 or many instances), or "+" (1 or many instances). In the following example, the element paper contains only one instance of sub-element title, one or many instances of sub-element author, and zero or many instances of sub-element citation:

<!ELEMENT paper (title, author+, citation*)>

2.2.2 XML Schema Languages

An XML document typically consists of two major components: schema and data. The schema describes the data and is specified in one of the proposed schema language notations. Since there are different requirements of the application, we need to choose a different XML schema language to describe our XML data. Among the dozens XML schema languages recently proposed, we briefly review three representative schema languages.

Document Type Definition

Since XML is a way to describe structured data, there should be a means to specify the structure of an XML document. A DTD [DTD02] is used to specify valid elements that can occur in a document, the order in which they can occur, and constrains imposed on certain aspects of these elements. An XML document that conforms to a given DTD is considered to be valid. The following is a list of the different means of constraining the contents of an XML document.

A sample XML fragment:

<Book ID=“88BB”>

<Title>XML Data Management</Title>

<Author>A. B. Chaudhri</Author>

<Author>A. Rashid</Author>

<Author>R. Zicari</Author>

<Publisher>Addison-Wesley</Publisher>

<Year>2003</Year>

</Book>

DTD was the original means of specifying the structure of an XML document and a holdover from XML's roots as a subset of SGML. The users can use DTD to define the structure of the XML. DTDs have different syntax from XML and are used to specify the order and occurrence of elements in an XML document. The following is a DTD for the above XML fragment.

DTD for sample XML fragment:

<!ELEMENT Book (Title, Author+, Publisher, Year)>

<!ATTLIST Book ID CDATA (#REQUIRED)>

<!ELEMENT Title (#PCDATA)>

<!ELEMENT Author (#PCDATA)>

<!ELEMENT Publisher (#PCDATA)>

<!ELEMENT Year (#PCDATA)>

The main building blocks of DTD are *elements and attributes*. Elements and attributes are defined by the keywords <!ELEMENT> and <!ATTLIST>. The above example DTD specifies that the **Book** element has four subelements, Title, Author, Publisher and Year. Keywords #PCDATA and CDATA are used as string types for elements and attributes. #REQUIRED means not-null. The element content model is the logical structure of the element contents based on the regular expressions such as "?" (0 or 1 instance), "*" (0 or many instances), or "+" (1 or many instances). Thus, in this example, the element **Book** contains only one

instance of subelement Title, Publisher and Year, but one or many instances of subelement Author.

XML Schema (XSD)

The W3C XML schema recommendation [Sch02] provides a sophisticated means of describing the structure and constraints on the content model of XML documents. W3C XML schema supports more data types than DTD, allows for the creation of custom data types, and supports object oriented programming concepts. XML schema has recently been gradually accepted by the software industry.

XML schema for sample XML fragment:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" >
    <element name="Book" maxOccurs="unbounded">
        <complexType>
            <sequence>
            <element name="ID" type="string"/>
            <element name="Title" type="string"/>
            <element name="Author" type="string"/>
            <element name="Publisher" type="string"/>
            <element name="Year" type="gYear"/>
            </sequence>
        </complexType>
        <key name="Pkey"/>
            <selector xpath="Book/"/>
```

```
            <field xpath="@ID"/>
        </key>
    </element>
</schema>
```

The above schema specifies a **Book** complex type (this means that it can have elements as subelement) that contains ID, Title, Author, Publisher, and Year elements in sequence. In this example, we define **ID** as an element not an attribute and also define ID as a "key" element.

DTDs have served well for almost twenty years as the primary mechanism of describing structured information in the SGML and HTML communities. They are considered too limited for many data-interchange applications. For example, DTDs can only specify that elements are text strings, text strings with other child elements mixed together or child elements without text. Furthermore, they are not formulated in XML syntax and provide only very limited support for data types or namespaces. XML schema tries to overcome some of the deficiencies, and like DTDs, are developed and standardized by the W3C.

The example shows an XML schema for the sample document. Clearly, XML schema is more complex description language than DTDs and offers more control over the document structure and text data by introducing types such as generic **string** while providing more specific ones like **gYear** for Gregorian year. Since XML schema is too complex to provide even an overview of all its features, only general principles are provided here.

Predefined Types. The standard provides a set of commonly used *simple*

types and allows the definition of *complex types* which are composed of simple types. Regular expressions, list, and set constructs all permit sophisticated structures in text types.

Type Inheritance. XML schema encourages the reuse of previously defined structures, no matter whether they are user-defined or pre-defined in the standard. Subtypes can add more elements to a supertype but also represent a subset of values. Groups of attributes and elements can be named for later reuse.

Controlled Extensibility. There are mechanisms to define whether a subtype can be derived from a given type, but also whether some type can be substituted by another one. There is also the notion of abstract types that must be substituted and of equivalence types.

Documentation. To allow for specifying both domain-specific and application-specific knowledge, XML schema provides dedicated elements like **appInfo**, **documentation**, and **annotation** elements for annotating schemas for both human readers and machines.

Uniqueness Constraints, Keys, and References. It is also possible to declare uniqueness constraints on certain attributes of child elements. This mechanism enables keys and references too.

Namespaces. Sometimes, it is desirable that documents conform to more than one schema. To achieve this, XML schema contains the necessary tools to enable fine-grained control over namespaces.

XML schema primer can be found in [Sch02]. XML schema is written in XML syntax. It supports more data types and namespaces than DTDs. It uses "key" and "keyref" elements to represent identities and references; this is useful for the mapping between relational data and XML. Because of these features of XML schema, many researchers focus on the conversion from relational data to XML schema and vice versus.

RELAX NG

RELAX (Regular Language description for XML) is novel XML schema language developed by Murata [Mur03]. It has been standardized by the Japanese Standard Association (JSA). RELAX is based on clean principles of hedge automata. It has the capability of expressing context-sensitive schema rules by means of non-terminal symbols. At the time of writing, RELAX and TREX are being merged into RELAX NG, under the OASIS Technical Committee.

Any regular tree grammar can be expressed in RELAX NG. RELAX NG has two significant extensions:

- Content models of RELAX NG can constrain both elements and attributes.
- RELAX NG provides the shuffle operator as well as usual operators for regular expressions.

RELAX NG represents production rules by **define** elements. The attribute **name** of a **define** element specifies a non-terminal in the left-hand side. The child elements of the **define** element captures the right-hand side. A terminal symbol and a content model in the right-hand side are represented by a child **element** element and the children of this **element** element.

To increase readability, RELAX NG allows production rules not to have a terminal in the right-hand side. Such production rules provide syntactic sugar and can be safely expanded without loss of information.

For example, consider the following RELAX NG schema:

```
<grammar>
    <start> <ref name="AddressBook"/> </start>
    <define name="AddressBook">
        <element name="addressBook">
            < zeroOrMore> <ref name="Card"/> </zeroOrMore>
        </element>
    </define>
    <define name="inline">
        <zeroOrMore>
            <choice>
                <element name="bold"> <ref name="inline"/> </element>
                <element name="italic"> <ref name="inline"/> </element>
            </choice>
        </zeroOrMore>
    </define>
</grammar>
```

Here **AddressBook** is a non-terminal that produces a tree, and **inline** is a non-terminal that produces a list of tree.

2.2.3 XML Query Languages

As the amount of XML data on the Web increases, the ability to access and query the data becomes increasingly important. Towards this goal, several XML query languages have been proposed [BL01]. Three query languages (Lorel, XML-QL, XQuery) are described briefly. Lorel [AQM+97] is considered a suitable representative of the family of languages for semistructured data. XML-QL [XML03] is the first query language in XML syntax. XQuery is the first proposal of a W3C standard query language for XML, which embeds the experience of previously defined query languages. In the following, a brief review of each query language is provided:

1. Lorel was originally designed for querying semistructured data and has been extended to XML data [AQM+97]; it was conceived and implemented at Stanford University. It is a user-friendly language in the SQL/OQL style, it includes a strong mechanism for type coercion and permits powerful path expressions, useful when the structure of a document is not known in advance [AQM+97].

2. XML-QL was designed at AT&T Labs; it has been developed as part of Strudel Project [BL01]. XML-QL language extends SQL with an explicit CONSTRUCT clause for building the document resulting from the query and uses the element patterns (patterns built on top of XML syntax) to match data in an XML document. XML-QL can express queries as well as transformations for integrating XML data from different sources [BL01].

3. XQuery [Lan03] is the first W3C proposal for a standard query language, published in February 2001 and revised in June 2001. The current proposal of

XQuery version 1.0 is drawn mostly from Quilt, a newly-conceived query language for XML, which inherits the experiences of several past query languages and attempts to unify them. XQuery assembles many features from previously defined languages. It is designed with the goal of being expressive, of exploiting the full versatility of XML and of combining information from diverse data sources [BL01].

2.2.4 Other XML Technologies

The following presents the XML technologies SAX, DOM, XPath and XPointer. SAX and DOM are arguably the most predominant technologies for working with XML documents, so the advantages and drawbacks associated with them will therefore be discussed. XPath is a specification to address specific parts of an XML document using path based regular expressions. XPointer is an extension of XPath. SOAP is used to invoke code over the Internet using XML and HTTP.

SAX

The Simple API for XML (SAX) is an example of an event based framework for parsing XML documents. SAX is not itself an XML parser, but defines a set of callback methods that are to be called by the parser when an event occurs, such as when the parser encounters the start tag and end tag of elements. To use SAX, an application programmer registers a SAX handler with the parser and begin parsing the document. The handler is notified of events when the callback methods are invoked by the parser. An XML document is thereby presented as a linear sequence of events. SAX provides no abstract representation of the document, only events,

which makes it difficult to work with and modify an XML document.

DOM

The Document Object Model (DOM) [DOM03] is an example of an API providing an abstraction of XML documents. In DOM, XML documents are represented as objects in a tree structure. DOM defines interfaces for each different entity in an XML document (elements, character data blocks, attributes, etc.), and specifies methods for traversing the structure and manipulating the document. Unlike SAX, that provides stream-based access to the XML data, DOM provides tree-structured access to nodes in the object representation. Furthermore, DOM provides ways to manipulate XML data, whereas SAX only provides access to data. DOM loads the entire XML document into an object structure, using a parser (actually, often a SAX parser) to build the object structure. It is thus, not possible to traverse and manipulate the object structure until the entire document that has been read. Since DOM has to read the entire XML document into the memory to build the tree structure, a large amount of RAM may be required when working with large XML documents.

XLink

XLink [Wil01] is the XML Linking Language which allows elements to be inserted into XML documents to create and describe links between resources. The language uses XML syntax to create hyperlinks. Unlike HTML, XLink links do not need to be stored within one of the documents they link. They can be stored in a separate document and link explicit regions of other documents. The regions are referred to using the XPointer language.

XPath

XML documents are structured as arbitrary nested hierarchical lists of information elements. XPath [XPa03] is a specification language designed by W3C. Being a subset of XQuery, it is an expression language for addressing parts of an XML document, used by various XML technologies such as XSL Transformations (XSLT) and XML Pointer Language (XPointer). As the name implies XPath uses path notation for navigating through the hierarchical structure of an XML document. Like DOM, XPath views an XML document as a tree structure consisting of nodes of different types, each representing the entities of a document.

XPointer

XPointer [Wil01] is a language defined as an extension of XPath. It allows XML resources to be linked into by another resources. An arbitrary region of an XML document may be referred to without respect to ownership or organization. XPointer uses XPath which provides a way of specifying well-bounded regions, such as entire elements or lists of elements. Xpointer uses XPath to define arbitrary regions by the users of spans (or regions). XPointer can refer to all the content between two points in a document, where the points are defined using XPath. Thus, an XPointer reference can refer to any contiguous region of any XML document accessible via a URL.

XSL

XSL [Wil01] is a language for expressing stylesheets. It consists of two parts:

1. XSL Transformations (XSLT), which is a language for transforming XML documents.

2. An XML vocabulary for specifying formatting semantics.

An XSL stylesheet specifies the presentation of XML documents by describing how those documents are transformed into XML documents that use the formatting vocabulary. XSL (or XSLT) can also be used to transform a document to a different XML document or an HTML document.

XML Namespaces

A single XML document may contain elements and attributes that are defined for and used by multiple applications. Such documents pose problems of recognition and conflict. Applications need to be able to recognize the tags and attributes that they are designed to process, even when some other applications use the same element type or attribute name. XML Namespaces [Wil01] provide a method for qualifying element and attribute names used in XML documents by associating them with namespaces.

This requires that document constructs have universal names where scope extends beyond the document. XML Namespaces Specification describes a mechanism for accomplishing this. An XML namespace is a collection of names, identified by a Uniform Resource Identifier (URI) reference, which is a string of characters to identify abstract or physical resources. A URI includes both Uniform Resource Locators (URLs) and Relative Uniform Resource Locators.

Within an XML document, a namespace, such as "http://www.w3.org/TR/WD-xsl", can be associated with an element type name prefix, such as "xsl:", which can be used to distinguish elements of that namespace from other elements. Thus, element type names from different namespaces can be distinguished, such as "xsl:if"

and “xperl:if”.

SOAP

SOAP [SOA03] is the Simple Object Access Protocol used to invoke code over the Internet using XML and HTTP. The mechanism is similar to Java Remote Method Invocation (RMI). In SOAP, method calls are converted to XML and transmitted over HTTP. SOAP was designed for compatibility with XML schemas; though their use is not mandatory. Being based on XML, XML schemas offer a seamless means to describe and transmit SOAP types.

2.3 XML Databases

2.3.1 What is an XML Database?

An XML database is a collection of XML documents that persist and can be manipulated. XML documents [Bou03a] tend to be either document-centric or data-centric. *Document-centric documents* are those in which XML is used for its ability to capture natural (human) languages, such as in user's manuals, static Web pages, and marketing brochures. They are characterized by complex or irregular structure and mixed content and their physical structure is important. The processing of the document is focused on the final presentation of the information to the user. *Data-centric documents* are those where XML is primarily for data transport. These include sales orders, patients records, flight schedules, stock quotes, and scientific data. The physical structure of data-centric documents, such as the order of elements or whether data is stored in attributes or subelements, is often unimportant. They are characterized by highly-regular structure with many repetitions of those data structures.

The processing of the document is usually focused on its use and exchange by applications.

The distinction between data-centric or document-centric XML documents can be subtle, and some documents, such as dynamic Web pages with descriptive text and data, could be viewed either way.

2.3.2 Why Do We Need to Use XML Databases?

XML is new standard format for exchanging data between databases and applications and between multiple pairs of applications over the Internet. Why do we use XML databases? To answer this question, consider the following reasons:

1. XML documents are traversing the Web enabling data exchange between E-commerce applications. Those applications need to log their communication and thus shortly end up with thousands of logged XML documents. The manager of a company might want to analyze the content of such XML collection.

2. Web sites can use XML documents to store the data to be published on the Web. The growth of the Web site can be accompanied with the growth of the number and/or size of XML documents. Additionally, some of those XML documents must be updated frequently.

3. News agencies can store all the news articles in XML documents. Editors might want to find the articles on previous elections in US that were the most referenced ones. In these cases large volumes of XML documents are used. Some software tools are needed to help achieving the required tasks.

4. One of the advantages of XML is that its structure is more flexible than the relations used in relational databases because XML can contain other elements, vary the order and number of attributes, and contain multiple elements with the same element type. This makes it easier to represent more complex data than the relations.

5. XML has some features that are similar to objects in object-oriented programming languages, such as element types, named attributes, and the ability to represent hierarchical tree structures. Thus, XML is easier to work with than objects for data exchange, but requires a different language for manipulating in applications.

If databases store the data as XML, there are several choices on the granularity of storage. The XML data can be stored as one document, divided into smaller sections and stored as fragments, or broken up into individual elements. Even if the data is not stored as XML, it can be exported as XML and presented to other applications or users. Data can be entered into the database as XML by parsing the document and storing the data in the structures of the database, such as relations.

2.3.3 Types of XML Databases

As XML becomes widely accepted, the need for systematic and efficient storage of XML documents increases. Some researchers consider storage in XML format as the desire solution. However, since relational database systems have such necessary database characteristics as transactions and administration tools, most E-business data is still stored in relational databases. Therefore, databases storing and retrieving

XML documents fall into two main categories: Enabled XML databases (XEDs) and Native XML databases (NXDs).

Enabled XML Databases

The good of Enabled XML databases is the development of middleware or an XML server, or extension of existing databases in such a way that would enable the publishing of relational data as XML documents, storing XML within relational databases, and querying stored data through XML views over such data. These types of XML databases are traditionally used for data-centric applications. In most cases, XML documents are first converted to relational form and then stored in the database. When the documents are queried, the data in the database is converted back to XML after query evaluation.

Typical of enabled XML databases is the SilkRoute system [FTS00]. The architecture of SilkRoute system is shown in Figure 2.2. Through SilkRoute the main principles of XML will be discussed.

(1) The underlying RDBMS contains the data that should be published and queried as XML. In our example we will assume that there is just one database named PEOPLE and one table in it named PERSON with columns as follows:

PERSON (ID, Name, Age, Employer)

(2) To define how this data will be exposed as XML the SilkRoute's proprietary relational to XML transformation Language (RXL) is used.

FROM PERSON $p

CONSTRUCT <people>

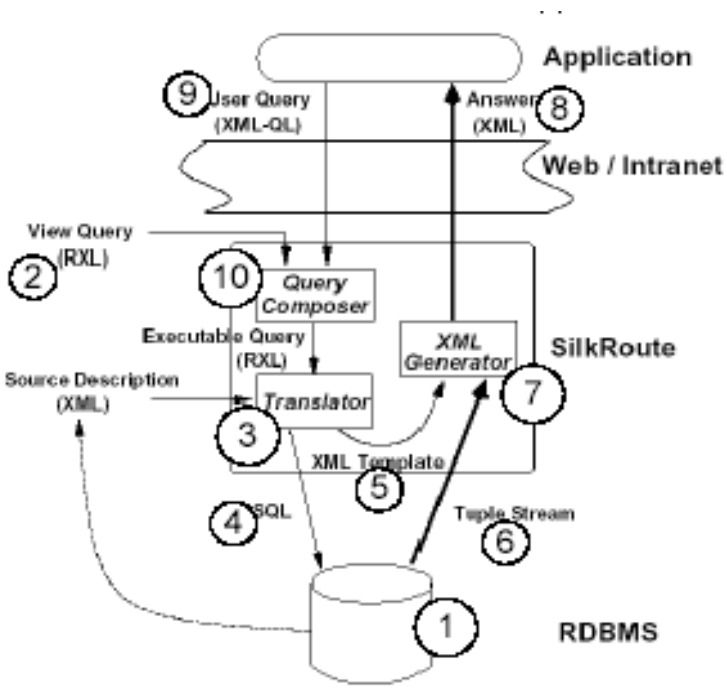


Figure 2.2: The architecture of SilkRoute

<person name=$p.Name>

<age>$p.Age</age>

</person>

. . . . . </people>

This RXL definition produces the output confirming to the following XML schema.

<xsd:element name="people">

<xsd:complexType>

<xsd:sequence>

```
<xsd:element name="person" minOcuures="0" maxOccurs="unbounded">
    <complexType>
        <sequence>
            <xsd:element name="age" type="xsd:int"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
</complexType>
</xsd:element>
```

(3) The Translator uses this RXL expression to:

- Generate a set of SQL queries (4) that will extract the necessary data from the underlying relational database. The task of RDBMS is to execute those queries and produce the answer in the form of the set of tuples (6),
- Extract the XML template (5) that will be used to structure the relational data in the final XML output.

(7) XML Generator fills the XML template (5) with received data (6), thus producing the final XML answer (8).

(9) SilkRoute also supports XML querying of the views defined in RXL. The querying language supported is XML-QL. Consider the following query (9).

CONSTRUCT <children>

WHERE <people>

<person name=$n>

<age>$a</age>

</person>

</people> IN “document.xml”, $a < 16

CONSTRUCT <child name=$n/>

</children>

(10) The Query composer is the most complex piece of SilkRoute’s architecture. The Composer has to integrate the view definition (2) and the user query (9) defined over that view to produce a new RXL expression. When executing this composed RXL expression, the answer (8) is the answer to the user query (9).

For our example, the composed RXL expression is:

FROM PERSON $p

WHERE $p.Age < 16

CONSTRUCT <children>

<child name=$p.Name/>

. . . . . </children>

We presented this simple example here only to show how SilkRoute system works. More detail about SilkRoute can be found elsewhere [FTS00, FMST01].

Oracle's Oracle9i is another example of an Enabled XML database system. Oracle is a relational database system that takes into account new requirements that emerged in the area of managing XML documents. It supports the storage and retrieval of XML documents. The essential building blocks are the XML SQL Utility (XSU), the new XMLType object type, and the OracleText Cartridge [CAZ03].

To store XML documents in relational database systems they typically add functionality to relational database thereby allowing for retrieval and storage of XML data. This is achieved using the existing functionality of the underlying relational database and the extensional bidirectional functionality for converting XML data to relational data and back. Therefore, issues such as concurrency control, scalability in multi-user environments, data integrity, transaction control, and security are maintained by the database. Using a relational database for storing XML data requires a mapping from the XML data to database relations. However, when the structure of XML documents is irregular, the result is either a large number of columns filled with null values or a large number of tables. A large number of null values waste space and a large number of tables will lead to slow read and write operations due to the larger number of joins needed for each database query.

Native XML Databases

Native XML databases are databases designed especially to store and retrieve XML documents. Like other databases, they support features like transactions, security, multiple users access, programmatic APIs, query languages, and so on. The only difference from other databases is that their internal model is based on XML. The term *native* means that documents are stored in data structures especially designed

for XML data, not in the form of relations as in traditional DBMSs. Many methods of storage and access for XML documents have been proposed for native storage, mainly based on labeled-directed graphs.

Native XML databases are most clearly useful for storing document-centric documents. This is because they preserve document order, processing instructions, comments, CDATA sections, and entity usage. Furthermore, Native XML databases support XML query languages, allowing users to ask questions such as: "Get me all documents in which the third paragraph after the start of the section contains a bold word." Such queries are clearly difficult to ask in a language like SQL.

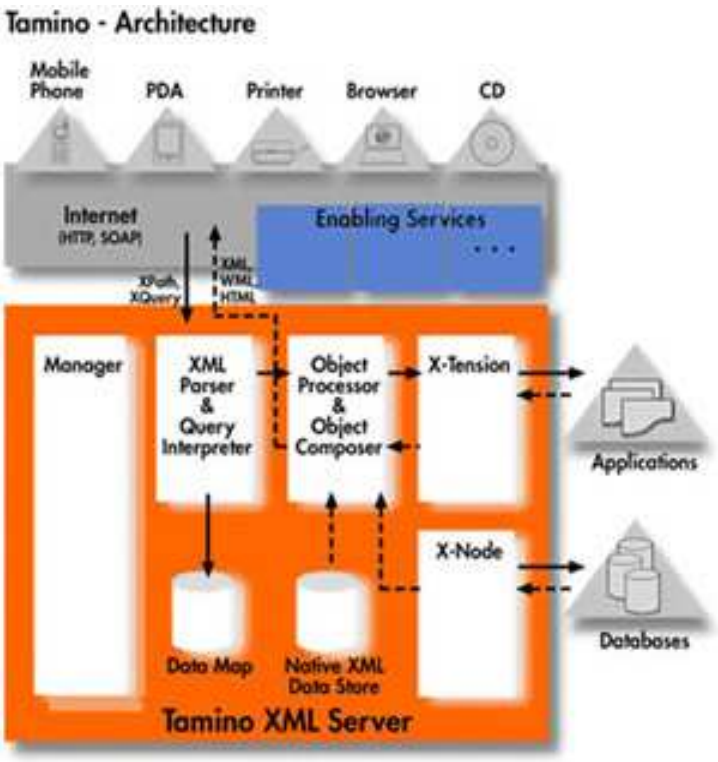


Figure 2.3: The architecture of Tamino system

Typical representative of Native XML databases is the Tamino XML server [TAM03], produced by Software AG in 1999. Tamino handles document-centric documents uniformly and is designed to process XML documents efficiently regardless of their structure. In addition, Tamino can store other types of data (e.g., images, sound files, MS Word documents, HTML pages, *etc.*) that are relevant in a Web context. Tamino XML server has a complete database system built in, providing transactions, security, multiuser access, scalability, and etc. In addition, Tamino is tailored to fit the needs of XML: It supports relevant XML standards and is optimized for XML processing. Tamino is available on Windows, UNIX, and OS/390. The architecture of Tamino XML server is shown in Figure 2.3. The general purpose of Timino is described next:

1. Native XML Data Store stores the XML data using proprietary "native" storage system.

2. Data Map is a metadata repository for all the data managed by the Tamino XML Server, including DTDs, XML Schemas, Style Sheets and even mappings between XML and external legacy databases.

3. XML Engine supports XQuery and full-text retrieval functionality.

4. X-Tension is used to access various external applications and for writing custom functionality enabling Tamino XML Server in meeting application specific needs.

5. X-Node acts as an interface between external applications and data sources. It provides access to existing heterogeneous databases with traditional data

structures, regardless of database type or location. X-Node maps this data to XML structures.

6. Enabling services surrounding Tamino XML Server include X-Port (direct communication with Web Servers) and a number of tools for development support like Tamino X-Plorer (querying tool for XML Server's data) etc.

7. Manager is Tamino's administration tool; it allows the Tamino administrator to manage the entire system over the Web (e.g., create database, start/stop server, back up, restore, load, etc). Tamino Manager allows for the installation of Tamino X-Tension server extensions for greater flexibility.

Software AG (1999) released the first version of its native XML server Tamino, which included a Native XML database. Tamino XML Server provides the full functionality required in a modern database system that has been thoroughly designed to handle XML.

eXist [CAZ03] is an Open Source effort to develop a Native XML database system, which is tightly integrated with existing XML development tools like Apache's Cocoon. eXist covers most of the basic Native XML database features as well as a number of advanced techniques such as keyword search on text, queries on the proximity of terms, and regular expression-based search patterns.

There are some others Native XML databases, such as TIMBER [JAKCL02] being developed at the University of Michigan, and Apache Xindice [Sta03] developed by the Apache Software Foundation.

Comparison between Enabled and Native XML Databases

In this section, we concentrate on an Enabled XML database and a Native XML database by comparing their performance. The Enabled XML database is a relational database that transfers data between XML documents and relational tables. It retrieves data for maintaining the relational properties between tables and fields, rather than to model XML documents. The Enabled XML database stores XML data in relations. The XML documents schema must be translated into the relational schema before accessing the corresponding tables. Similarly, the XML query language must be translated into SQL to access the relations. The Native XML database stores XML data directly. It maps the structure of XML documents to the database without any conversion. This is the main difference between an Enabled XML database and a Native XML database. This direct access in a Native XML database can reduce processing time and provide better performance. The XML engine stores XML documents in and retrieves them from their respective data sources. The storage and retrieval are based on schemas defined by the administrator. The Native XML database implements performance-enhancing technologies, such as compression and buffer pool management, and reduces the workload related to database administration.

As a database management system, an XML DBMS must have all of the standard DBMS properties, including support for transactions, concurrency, data integrity, efficient query processing, good backup and recovery utilities, *etc.*

What makes an XML database management system important for XML applications is its efficiency. A number of DBMS characteristics must be considered for efficient XML handling:

- Ability to understand and treat XML as data.
- Support of all XML syntax.
- Ability to manipulate documents, fragments and elements efficiently.
- Ability to search XML documents efficiently.
- Support for native XML-based APIs.
- Retention of all document content.
- Support for document round-tripping.
- Legacy data integration.

These characteristics suggest criteria by which an XML DBMS proves whether it offers fast and reliable performance across the full spectrum of data it stores. We compare the two types of XML DBMS in Table 2.1. To store XML data in an Enabled

Table 2.1: The comparison between enabled and native XML DBMS

Criterion	**Enabled XML DBMS**	**Native XML DBMS**
Ability to treat XML as data	Yes	No
Support all XML syntax	No	Yes
Document/Fragment Manipulation	Some	May be low performance
Search XML documents	Inefficient	Efficient
Support XML-based APIs	Inefficient	Inefficient
Retain all content	No	Yes
Round-tripping	No	Yes
Legacy data integration	Yes	No

XML database substantially limits the value that XML provides. An Enabled XML database is useful only for data-centric XML data, not for document-centric XML data, and therefore only works with a limited subset of XML data. Thus, using a relational database to store general XML data is not efficient [SXM02]. However, since the relational database systems are well defined and it has been used in industry for many years, all the major database vendors have implemented support for storing XML documents directly in the database.

A Native XML database is better to store and retrieve XML documents. Although Native XML databases and query technology are important capabilities, a common opinion is that they will be successful in the mainstream market only if they are efficiently and effectively combined with SQL and relational database technology in an overall system architecture.

There is another approach, which is to store XML data in object-oriented databases. Which is the best way to store XML documents: in text files, in relational databases, or in object-oriented databases? It is still an open question. Although storing XML documents in text files is now the standard approach, most XML documents are still stored in relational databases today because the effective and mature relational database techniques can be reused. Indeed, the relational database technology is well established and widespread, supporting efficient data management for even huge databases and providing scalability even for thousands of parallel users. Relational database systems offer powerful SQL query mechanisms, conforming to standards and providing effective query optimization. Therefore, most researchers and vendors of relational database systems are focusing on extending existing relational database systems to support the storage and retrieval of XML documents.

Chapter 3

Transformation of Legacy Relational Database into XML

3.1 Introduction

XML is a standard information exchange format over the Internet. How should XML documents be stored? The most popular way to achieve this storage is to transform XML documents into relational tabular formatted data and then make use of already existing relational database management systems to store this data in database. This also imposes the need to transform data from relational format back into XML format. This reverse transformation will be discussed in Chapter 4.

In this chapter, we consider how relational representations can be transformed into XML. Background and related work is discussed in Section 3.2. We give a brief introduction for catalog-based and legacy database systems in Section 3.3, relational conceptual model in Section 3.4, and flat or nested XML structures in Section 3.5. In Section 3.6, we present the approach to extract an ER model from a legacy database. In Section 3.7, we present the ER-to-XML transformation process.

3.2 Background and Related Work

The transformation from relational to XML has recently received considerable attention. It is an important problem for the following reasons:

- Users may want to publish their relational data in web applications. Since XML is the standard information exchange format over the Internet, users want to provide an XML view of their underlying relational data. In this case, the relational schema is transformed into XML schema, but the data still resides in relational databases.

- It is better to directly query on XML documents by using existing XML query language, rather than to translate XML query syntax to SQL and query on relational database and then translate the query result back to XML. Therefore, providing an XML view makes it easier for the end users or applications to access data.

- Web applications may want to exchange their data with other Web applications over the Internet. As XML is the standard information exchange format, the relational data need to be transformed into XML documents and then exchanged with other Web applications. In this case, the relational schema is transformed into XML schema and also relational data is transformed into XML data.

The importance of this transformation problem, has motivated by several researchers so different approaches have been proposed to handle this problem. The approach taken by XML Extender [CX00] from IBM is that users provide as input the relational schema as well as the target XML schema; also users provide the mapping between the relational and the XML schema. The tool provides the ability to convert operations from XML to relational and obtain the results as XML. The XML Extender is detailed in Section 4.2. In SilkRoute, XPERANTO, and Agora, users provide

the relational schema and the queries against the XML view. The XML schema is obtained from these queries, and they are also used to translate operations on the XML view to the relational data, as well as to translate results. In the three approaches mentioned above, the success of the conversion is closely related to the quality of the target XML schema onto which a given input relational schema is mapped. However, the mapping from the relational schema to the XML schema is specified by human experts. Therefore, when large amounts of relational schemas and data need to be translated into XML documents, a significant investment of human effort is required to initially design target schemas. More details about SilkRoute system is included in Section 2.3.

There is another approach, which is to map from non-relational models to XML models. The work described in [MLM01b, FPB01, KL01] study the conversion from XML to EER model and vice verse. Generation of an XML schema from a UML model is reported as well [BCFK, CSF]. Mani *et al.* [LMCC01, LMCC02] have investigated how to come up with a "good" XML schema. We will discuss them individually as follows:

- **XPERANTO**: In XPERANTO [CFI+00] (XML Publishing of Entities, Relationships, ANd Typed Objects), middleware solution for publishing XML, object-relational data can be published as XML documents. It can be used by developers who prefer to work in a "pure XML" environment. XPERANTO creates the XML view over the internal relational database, and provides an XML query facility, which translates XML queries into corresponding SQL queries. It then transforms the results back to XML. XPERANTO consists

of XML-QL Parser, Query Rewrite, SQL Translation, and XML Tagger. The input, for example an XQuery, is first parsed and translated to an internal representation called XQGM. XQGM stands for XML Query Graph Model it is the input to the SQL Translation that generates the query representation in SQL. The SQL query is executed in a database engine and returns the answer to the XML Tagger to create the resulting XML document for the users.

The mapping between a relational database and XML is not perfect. XML query languages and SQL do not have the same semantics. There is a mismatch between them. For example, metadata queries are supported by XML query language support and not by SQL. Therefore, these metadata queries cannot be passed to the relational database [CFI+00].

- **Agora**: Agora [MFK01] is a system that queries over virtual views of XML data. The most important feature of the Agora system is the lack of an algebra. Agora is a mediator between different heterogeneous data formats and their schemas. It is implemented on the top of the Le Setect Data Integration System [MFK01] which is a framework for publishing and querying relational data. The interface is pure XML, while the internal data is relational. The interface takes an XML query language as input and the output is XML. The XML query language is a subset of the Quilt language. A query is first simplified by normalization and then translated to SQL. The SQL query is optimized and executed over the view that has been created from the XML documents. Tuples that form the result are tagged into XML elements, thereby producing the final XML result.

The biggest problem of Agora is the complexity issues related to transforming XQuery over virtual views of XML to SQL queries over real data sources and its performance consequences.

- **EER-to-XML**: The conversion of EER-to-XML is described by Mani *et al.* [MLM01b]. XGrammar is used for the notation of XML schemas. It is an extension of the regular tree grammar definition presented by [MLM01a], which uses a six tuple notation to precisely describe content models of XML schema languages. Informally, XGrammar takes the structural specification feature from DTD and RELAX, and the data typing feature from XML-Schema. The basic idea of this conversion is to generate XGrammar from a given XML model, then convert XGrammar to EER model, or vice versa. There is another similar approach described by Fong *et al.* [FPB01], which adopts a database reverse engineering approach. It reconstructs the semantic model in the form of ER model from the logical schema capturing user's knowledge and then forward engineering to the XML document. However, the work described [FPB01] only deals with catalog-based databases. Similarly, the work described by Kleiner and Lipeck [KL01] proposes a way to translate a well-known ER schema into DTD. The authors describe a set of rules on how to translate constructs from the ER model into DTD. They claim that their translation preserves almost all semantic information. In addition, an extension of the translation from EER model is presented.

- **UML-to-XML**: The work by Booch *et al.* [BCFK] describes a graphical notation in UML (Unified Modeling Language) for designing XML schemas. UML

is a standard object-oriented design language. They map all elements and data types in XML schema to classes annotated with stereotypes that reflect the semantics of the related XML schema concept. They use a sequence number for content model elements to indicate the order of document types. XML schemas may contain anonymous groups. They introduce special stereotypes indicating that the class represents an anonymous grouping of elements in UML. Similarly, the work by Conrad *et al.* [CSF] proposes a way to transform UML class diagrams to DTDs. In this work, no comprehensive algorithm is given and associations in UML are only weakly considered since it is proposed to use XLinks which were still in the development stage at that time.

- **NeT and CoT**: The work described [LMCC01, LMCC02] studies how to come up with a "good" XML schema. It presents three algorithms for translating relational model to XML: FT (Flat Translation), NeT (Nesting-based Translation) and CoT (Constraint-based Translation). The native translation algorithm FT translates "flat" relational model to "flat" XML model in one-to-one manner. Thus, FT does not use the non-flat features of the XML model, possible through regular expression operators such as "*" and "+". To remedy this problem, they present NeT, which uses the nest operator to generate a more precise and intuitive XML schema from relational inputs. However, NeT is only applicable to a single table at a time, and cannot obtain a big picture of a relational schema where many tables are interconnected with each other. CoT addresses this problem; it uses semantic constraints to come up with a more intuitive XML schema for the entire relational schema. They have done some

testing on NeT and CoT algorithms. Comparing their experimental results with DTDs obtained by DB2XML, NeT and CoT can obtain better results in accuracy and size.

- **Publishing XML**: The work described by Shanmugasundaram *et al.* [SSB+01] thoroughly analyzes the techniques for publishing XML from relational data. They attempt to make existing relational database support XML publishing. The XML publishing task is separated into three subtasks: (1) Data extraction; (2) Data structuring; and (3) Data tagging.

Our approach is different from these others. We focus on the transformation for both catalog-based and legacy relational databases. To transform legacy database into XML, we first need to obtain all possible information to construct ER model which is possible by applying the reverse engineering technique. In our approach, we adopt some algorithms [Alh03] to extract ER model from the given legacy relational database and we convert the ER model to XML schema.

3.3 Catalog-based and Legacy Database Systems, and Reverse Engineering

A relational database is organized as a view consisting of tables. The rows of a table are called tuples and columns of a table are called attributes. The structure of the database is defined in the database schema; and the schema contains all information about tables, keys and constraints. We can divide existing database systems into two categories: catalog-based and legacy database systems. The former

are distinguished by well-defined catalog, and the latter are characterized by old-fashioned architecture, lack of the related documentation (missing or vague catalogs) and non-uniformity resulting from numerous extensions.

In general, reverse engineering can be defined as the process of discovering how a system work. It requires identifying and understanding all components of an existing system and the relationships between them. Database reverse engineering is necessary to semantically enrich and document a database, and to avoid throwing away the huge amounts of data stored in existing legacy databases if the organization of an existing database wants to reengineering, or maintain and adjust the database design.

3.4 The Conceptual Model

The goal of this section is to represent the modeling concepts of the ER model, which is a popular high-level conceptual data model. The basic elements of the ER model are entities, relationships, and their attributes. A relational schema in the ER model allows the designer to define entity types, relationship types, attributes for entity types and relationship types, and cardinality constraints for relationship types. An entity type defines a set of entities that have the same attributes. Each entity type in the database is described by a name and a list of attributes. Figure 3.1(a) shows two entity types, named DEPARTMENT and DEPT_LOCATIONS. A relationship type R among n entity types E_1, E_2, ..., E_n defines a set of associations among entities from these types. Figure 3.1(c) shows a relationship type WORKS_ON between the two entity types EMPLOYEE and PROJECT, which associates each employee with

the projects the employee works on. Each relationship instance in WORKS_ON associates one EMPLOYEE entity and one PROJECT entity. In this thesis, we will follow the notation from Elmasri and Navathe [EN94]. Entity types are shown by a rectangular box, relationship types by a diamond-shaped box, attributes are shown in ovals, and each attribute is attached to its entity type or relationship type by straight line. Key attributes have their names underlined. Cardinality constraints for entity types in a relationship type are specified using a pair (min, max). The minimum cardinality of an entity type E in a relationship type R is min; shows how many times any entity of type E must be present if we project the tuples of R on E (without removing duplicates). The definition of the maximum cardinality is similar: The maximum cardinality of an entity type E in a relationship type R is max; shows how many times at most any entity of type E may be present if we project the tuples of R on E. The degree of a relationship type is the number of participating entity types. Hence, WORKS_ON relationship type is of degree two in Figure 3.1(c). A relationship type of degree two is called binary and one of degree three is called ternary. Figure 3.1(d) illustrates TAKES as an example of ternary relationship type. In general, if the degree is equal or greater than 3, we say the relationship type is nary. The cardinality ratio is one constraint on relationship types. It specifies the number of relationship instances that an entity can participate in. The binary relationship type DEPARTMENT:DEPT_LOCATIONS has cardinality ratio *1:M*, meaning that each department can be located at numerous locations, but a location can be related to only one department. Common cardinality ratios for binary relationship types are *1:1*, *1:M*, and *M:N*. An example of a *1:1* binary relationship type is EMPLOYEE : DEPARTMENT shown in Figure 3.1(b),

which relates a department entity to the employee who manages that department. This represents the constraints that an employee can manage only one department and that a department has only one manager. The relationship type WORKS_ON represents *M:N* cardinality ratio, meaning an employee can work on several projects and a project can have several employees.

3.5 Flat and Nested XML Document Structures

In relational databases, primary and foreign keys define the relationship between two tables. However, in XML schema, there are two ways to represent a relationship between two parts of an XML document:

- Specify nested complex types.
- Specify "key" and "keyref" constraints in a way similar to the relational model.

Nested complex type definitions in the XML schema indicate the parent-child relationships of the elements. If we specify nested complex types to create a relationship between two parts of an XML document, we will create a nested XML document structure. For example, the following XML schema fragment shows that DEPT_LOCATIONS is a child element of the DEPARTMENT element; and DEPT_LOCATIONS element is nested under DEPARTMENT element.

```
<xs:complexType name="DEPARTMENT_Relation">
    <xs:sequence>
      <xs:element name="DEPARTMENT_Tuple"
          type="db:DEPARTMENT_Tuple" maxOccurs="unbounded"/>
```

```
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPARTMENT_Tuple">
  <xs:sequence>
    <xs:element name="DNAME" type="xs:string" />
    <xs:element name="DNUMBER" type="xs:int" />
    <xs:element name="MGRSSN" type="xs:string" />
    <xs:element name="MGRSTARTDATE" type="xs:string" />
    <xs:element name="DEPT_LOCATIONS_Relation"
        type="db:DEPT_LOCATIONS_Relation" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPT_LOCATIONS_Relation">
  <xs:sequence>
    <xs:element name="DEPT_LOCATIONS_Tuple"
    type="db:DEPT_LOCATIONS_Tuple" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPT_LOCATIONS_Tuple">
  <xs:sequence>
    <xs:element name="DNUMBER" type="xs:int" />
    <xs:element name="DLOCATION" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:key name="DEPARTMENT_PrimaryKey">
```

```
    <xs:selector xpath="db:DEPARTMENT_Relation/db:DEPARTMENT_Tuple"/>
    <xs:field xpath="db:dnumber" />
  </xs:key>
  <xs:key name="DEPT_LOCATIONS_PrimaryKey">
    <xs:selector xpath="db:DEPT_LOCATIONS_Relation/db:DEPT_LOCATIONS_Tuple"/>
    <xs:field xpath="db:dnumber" />
    <xs:field xpath="db:dlocation" />
  </xs:key>
```

A part of the XML document generated from the above XML schema fragment is given below. All DEPT_LOCATIONS that have the same DNUMBER are grouped as a unit. This structure of XML document is easy to read for users and also reduces search time (no need for joins-like operations). This will speed up the processing of queries raised on XML documents. Unfortunately, some data is repeated many times causing data redundancy, and the size of XML document is increase as well, which is the major disadvantage of the nested XML structure.

```
<db:DEPARTMENT_Relation>
  <db:DEPARTMENT_Tuple>
    <db:DNAME>Research</db:DNAME>
    <db:DNUMBER>5</db:DNUMBER>
    <db:MGRSSN>333445555</db:MGRSSN>
    <db:MGRSTARTDATE>1978-05-22</db:MGRSTARTDATE>
    <db:DEPT_LOCATIONS_Relation>
       <db:DEPT_LOCATIONS_Tuple>
           <db:DNUMBER>5</db:DNUMBER>
```

```
            <db:DLOCATION>Houston</db:DLOCATION>
        </db:DEPT_LOCATIONS_Tuple>
        <db:DEPT_LOCATIONS_Tuple>
            <db:DNUMBER>5</db:DNUMBER>
            <db:DLOCATION>Sugarland</db:DLOCATION>
        </db:DEPT_LOCATIONS_Tuple>
        <db:DEPT_LOCATIONS_Tuple>
            <db:DNUMBER>5</db:DNUMBER>
            <db:DLOCATION>Bellaire</db:DLOCATION>
        </db:DEPT_LOCATIONS_Tuple>
      </db:DEPT_LOCATIONS_Relation>
   </db:DEPARTMENT_Tuple>
   <db:DEPARTMENT_Tuple>
   . . . . . .
   </db:DEPARTMENT_Tuple>
 </db:DEPARTMENT_Relation>
```

The second way to create relationship between two parts of an XML document is to specify "key" and "keyref" constraints in the XML schema. For example, the following schema fragment specifies DEPARTMENT and DEPT_LOCATIONS "complexType" elements at the same level (not nested). The "key" and "keyref" constraints specify the parent-child relationship between two elements. This is similar to the primary/foreign keys link in relational databases.

```
  <xs:complexType name="DEPARTMENT_Relation">
    <xs:sequence>
```

```
    <xs:element name="DEPARTMENT_Tuple"
    type="db:DEPARTMENT_Tuple" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPARTMENT_Tuple">
  <xs:sequence>
    <xs:element name="DNAME" type="xs:string" />
    <xs:element name="DNUMBER" type="xs:int" />
    <xs:element name="MGRSSN" type="xs:string" />
    <xs:element name="MGRSTARTDATE" type="xs:string" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPT_LOCATIONS_Relation">
  <xs:sequence>
    <xs:element name="DEPT_LOCATIONS_Tuple"
    type="db:DEPT_LOCATIONS_Tuple" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPT_LOCATIONS_Tuple">
  <xs:sequence>
    <xs:element name="DNUMBER" type="xs:int" />
    <xs:element name="DLOCATION" type="xs:string" />
  </xs:sequence>
</xs:complexType>
<xs:key name="DEPARTMENT_PrimaryKey">
```

```
    <xs:selector xpath="db:DEPARTMENT_Relation/db:DEPARTMENT_Tuple" />
    <xs:field xpath="db:dnumber" />
  </xs:key>
  <xs:key name="DEPT_LOCATIONS_PrimaryKey">
    <xs:selector xpath="db:DEPT_LOCATIONS_Relation/db:DEPT_LOCATIONS_Tuple"/>
    <xs:field xpath="db:dnumber" />
    <xs:field xpath="db:dlocation" />
  </xs:key>
  <xs:keyref name="DEPARTMENT.mgrssn" refer="db:EMPLOYEE_PrimaryKey">
    <xs:selector xpath="db:department_Relation/db:department_Tuple" />
    <xs:field xpath="mgrssn" />
  </xs:keyref>
  <xs:keyref name="DEPT_LOCATIONS.dnumber" refer="db:DEPARTMENT_PrimaryKey">
    <xs:selector xpath="db:dept_locations_Relation/db:dept_locations_Tuple"/>
    <xs:field xpath="dnumber" />
  </xs:keyref>
```

The following is a part of the XML document obtained from a flat XML structure. XML data are grouped separately. Comparing flat XML structure to nested XML structure, they contain less data redundancy because of the minimum data repetition, and hence require less storage space. Unfortunately, the time for data retrieval increases because the related data may be stored in different fragment which slows XML querying.

```
  <db:DEPARTMENT_Relation>
    <db:DEPARTMENT_Tuple>
```

```
    <db:DNAME>Research</db:DNAME>
    <db:DNUMBER>5</db:DNUMBER>
    <db:MGRSSN>333445555</db:MGRSSN>
    <db:MGRSTARTDATE>1978-05-22</db:MGRSTARTDATE>
  </db:DEPARTMENT_Tuple>
  <db:DEPARTMENT_Tuple>
    <db:DNAME>Administration</db:DNAME>
    <db:DNUMBER>4</db:DNUMBER>
    <db:MGRSSN>987654321</db:MGRSSN>
    <db:MGRSTARTDATE>1985-01-01</db:MGRSTARTDATE>
  </db:DEPARTMENT_Tuple>
  . . . . . .
</db:DEPARTMENT_Relation>
<db:DEPT_LOCATIONS_Relation>
  <db:DEPT_LOCATIONS_Tuple>
    <db:DNUMBER>1</db:DNUMBER>
    <db:DLOCATION>Houston</db:DLOCATION>
  </db:DEPT_LOCATIONS_Tuple>
  <db:DEPT_LOCATIONS_Tuple>
    <db:DNUMBER>2</db:DNUMBER>
    <db:DLOCATION>Houston</db:DLOCATION>
  </db:DEPT_LOCATIONS_Tuple>
  . . . . . .
</db:DEPT_LOCATIONS_Relation>
```

3.6 Extracting ER model from Legacy Databases

In this section, an overview of the reverse engineering process described earlier is presented [Alh03]. We will show the results obtained for the following running example.

Example 3.6.1 Consider the COMPANY database shown in Figure 3.2. The database contains six tables: EMPLOYEE, DEPENDENT, PROJECT, DEPARTMENT, WORKS_ON, and DEPT_LOCATIONS.

The first step of the reverse engineering process is to extract the basic necessary information from a given legacy relational database. The information includes all candidate and foreign keys found within the relations. It is summarized in Table 3.1 and Table 3.2 as below:

CandidateKeys (relation name, attribute name, Candidate Key#)

ForeignKeys (PK relation, PK attribute, FK relation, FK attribut, Link#)

The *CandidateKeys* table contains all possible candidate keys of the relations. The *Candidate Keys#* can be used to track of the same attributes participating in more than one candidate key. The *CandidateKeys* table for the example COMPANY database is shown in Table 3.1.

The *ForeignKeys* table contains all pairs of attributes such that the first attribute is part of a candidate key in a certain relation and the second attribute is part of a foreign key, a representative of the first attribute within any of the relations. *Link#* differentiates different foreign keys in the same relation. Foreign keys are numbered so that all attributes within the same foreign key are assigned the same

sequence number. The *ForeignKeys* table for the example COMPANY database is shown in Table 3.2.

In general, a relation may have a set of candidate keys. One candidate key is chosen as the primary key by checking corresponding foreign keys. For relations that have multiple candidate keys, the primary key is selected to be the candidate key that appears in the first column of *ForeignKeys*. The *PrimaryKeys* table for the example COMPANY database is shown in Table 3.3.

The employed reverse engineering process decides on the presence of *candidate key(s)* of a given relation R as *foreign key(s)* within R itself or any other relation in the relational schema. This leads to unary and binary relationships because when translating from the ER model to the relational model, all relationships are mapped into these two types of relationships with some weak entities introduced and converted into relations.

The information in *ForeignKeys* is used to construct the *Relational Intermediate Directed (RID) Graph*, which present all possible binary and nary relationships between relations in the given relational schema. In the RID graph, each node represents a relation and two nodes are connected by a link to show that a foreign key in the relation that corresponds to the first node represents the primary key of the relation that corresponds to the second node. The RID graph is equivalent to the ER model of relational database. The initial RID of COMPANY is shown in Figure 3.3.

Alhajj described [Alh03] the cardinality of a relationship in the RID graph as follows. A link is directed from R_2 to R_1 to reflect the presence of the primary key of R_1 as a foreign key in R_2; so, its cardinality is:

- *1:1* if and only if at most one tuple from R_2 holds the value of the primary key of a tuple from R_1.

- *M:1* if more than one tuple from R_2 hold the value of the primary key of a tuple from R_1.

The employed process decides also on the minimum and maximum cardinalities at both sides of the link by investigating whether the link is optional or mandatory on each side.

Analyzing the information in Table 3.2 leads to the RID graph so it can be easily observed that it contains some extra information because a foreign key is allowed to play the role of a candidate key which leads to two *symmetric* and *transitive* references. Such extra information is then deleted [Alh03]. The $ForeignKeys$ table for the example COMPANY database after deleting *symmetric* and *transitive* references is shown in Table 3.4.

Eliminating symmetric and transitive references leads to an optimized RID graph. The optimized RID graph is analyzed further to identify relationships with attributes, *M:N* and nary relationships, if any. The remaining unary and binary relationships are without attributes so they are represented by direct connections between nodes in the optimized RID graph. They are all classified as *1:1*, or *1:M*. The optimized RID graph for the example COMPANY database is shown in Figure 3.4.

The last step of this process is to determine which relationship type is *M:N* or n-ary. In Figure 3.4, the WORKS_ON relationship type is *M:N*.

3.7 Transforming ER model into XML Schema

Existing approaches to deal with the conversion of relational databases into XML have mainly concentrated on *1:1* and *1:M* relationships; *M:N* and n-ary ($n > 2$) relationships have not received much attention. We argue that it is essential to equally consider all types of relationships for the two cases of flat and nested XML schemas. Thus, we consider all types of relationships in our conversion process as detailed in the rest of this chapter.

In this section, we first present transforming ER models to a flat XML schema algorithm (ER2FXML); and then transforming ER models to a nested XML schema algorithm (ER2NXML).

3.7.1 ER Model to Flat XML Schema Transformation

The ER2FXML in pseudo-code is depicted in Algorithm 3.1.

Algorithm 3.1 ER2FXML (ER Model to Flat XML Conversion)

Input: The ER model

Output: The corresponding flat XML schema

Step:

1. Transform each entity type, *M:N* or n-ary relationship (we call them objects hereafter) from the ER model into a "complexType" in the XML schema.

2. Map each attribute in an object transformed in Step (1) into a subelement within the corresponding "complexType".

3. Create a root element as the relational database schema name and insert each of

the three types of objects identified in Step (1) in the ER model as a subelement with the corresponding "complexType".

4. Define the primary key for each of the three types of objects identified in Step (1) by using "key" element.

5. Map each link, in the ER model, between the three types of objects identified in Step (1) by using "keyref" element.

EndAlgorithm 3.1

To understand the steps of Algorithm 3.1, we present more details with supporting examples.

- Each object E in the ER model is translated into an XML "complexType" of the same name E in the XML schema. In each "complexType" E, there is only one empty element, which includes several subelements. For example, PROJECT is translated into a "complexType" named PROJECT_Relation. The empty element is called PROJECT_Tuple.

```
<xs:complexType name="PROJECT_Relation">
    <xs:sequence>
      <xs:element name="PROJECT_Tuple"
      type="db:PROJECT_Tuple" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="PROJECT_Tuple">
    <xs:sequence>
      . . . . . .
```

```
    </xs:sequence>
  </xs:complexType>
```

The cardinality constraint in the ER model can be explicated by associating two XML built-in attributes (also called indicators), namely "minOccurs" and "maxOccurs", with subelements under the "complexType" element. The default value for both "maxOccurs" and "minOccurs" is 1. If specified, the value for "minOccurs" should be either 0 or 1 and the value for "maxOccurs" should be greater than or equal to 1. If both "minOccurs" and "maxOccurs" are omitted, the subelement must appear exactly once.

- Each attribute A_i in E is mapped into a subelement of the corresponding "complexType" E. For example, PROJECT is mapped into a "complexType" named PROJECT_Tuple, inside which there are several subelements such as PNAME, PNUMBER, PLOCATION, and DNUM. They are attributes of the PROJECT entity. The XML schema for PROJECT is:

```
<xs:complexType name="PROJECT_Tuple">
    <xs:sequence>
      <xs:element name="PNAME" type="xs:string" />
      <xs:element name="PNUMBER" type="xs:int" />
      <xs:element name="PLOCATION" type="xs:string" />
      <xs:element name="DNUM" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
```

The "sequence" specification in the XML schema captures the sequential se-

mantics of a set of subelements. For instance, in the "sequence" given above, the subelements appear in the order: PNAME, PNUMBER, PLOCATION, and DNUM. They must appear in instance documents in the same order as they are declared here. The XML schema also provides another constructor called "all", which allows elements to appear in any order, and each element must appear once or not at all.

- Each object in the ER model is mapped into the XML schema. We first need to create a root element that represents the entire given relational database. We create the root element as a "complexType" in XML schema and give it the same name as the relational database schema. It then inserts each object as a subelement of the root element. An example which contains the six objects DEPARTMENT, DEPENDENT, DEPT_LOCATIONS, EMPLOYEE, PROJECT, WORKS_ON is now presented. We give the root element the name COMPANY:

```
<xs:element name="COMPANY">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="DEPARTMENT_Relation"
                    type="db:DEPARTMENT_Relation" />
        <xs:element name="DEPENDENT_Relation"
                    type="db:DEPENDENT_Relation" />
        <xs:element name="DEPT_LOCATIONS_Relation"
                    type="db:DEPT_LOCATIONS_Relation" />
        <xs:element name="EMPLOYEE_Relation"
```

```
                    type="db:EMPLOYEE_Relation" />
        <xs:element name="PROJECT_Relation"
                    type="db:PROJECT_Relation" />
        <xs:element name="WORKS_ON_Relation"
                    type="db:WORKS_ON_Relation" />
      </xs:sequence>
    </xs:complexType>
    <!-- definition of keys and keyrefs -->
    . . . . . .
</xs:element>
```

- The elements "key" and "keyref" are used to enforce the uniqueness and referential constraints. They are among the key features introduced in the XML schema. Further, we can use "key" and "keyref" to specify the uniqueness scope and multiple attributes in creating composite keys. Consider this example:

```
<xs:key name="PROJECT_PrimaryKey">
      <xs:selector xpath="db:PROJECT_Relation/db:PROJECT_Tuple"/>
      <xs:field xpath="db:PNUMBER" />
</xs:key> <xs:key name="WORKS_ON_PrimaryKey">
      <xs:selector xpath="db:WORKS_ON_Relation/db:WORKS_ON_Tuple"/>
      <xs:field xpath="db:ESSN" />
      <xs:field xpath="db:PNO" />
</xs:key> <xs:keyref name="WORKS_ON.PNO"
refer="db:PROJECT_PrimaryKey">
      <xs:selector xpath="db:WORKS_ON_Relation/db:WORKS_ON_Tuple"/>
```

```
        <xs:field xpath="PNO" />
    </xs:keyref>
```

In this example, we first specify the primary key for each object in the ER model. From the *ForeignKeys* table, we have PNUMBER as the primary key of PROJECT; ESSN and PNO together form a composite primary key for WORKS_ON. PNO is a foreign key in WORKS_ON so we use "keyref" to specify the foreign key relationship between PROJECT and WORKS_ON. Compared to DTD, the XML schema provides a more flexible and powerful mechanism through "key" and "keyref", which share the same syntax as "unique" also make referential constraints possible in XML documents. The whole flat XML schema result generated from the example COMPANY database can be found in Appendix A.

In general, ER2FXML is a straightforward and effective transformation algorithm, but it is only applicable when generating a flat XML structure from an ER model of a relational database. As the name implies, ER2FXML cannot handle the nested features provided by XML. We remedy this problem in the ER2NXML algorithm which will be presented in the following section.

3.7.2 ER Model to Nested XML Schema Transformation

In XML schema, we can use nested complex type elements to define the relationship between two elements. The biggest advantage of the nested XML structure is to store all related information in one fragment of an XML document. This reduces the time for data retrieval when users query on the XML document. Algorithm 3.2

(ER2NXML) does the transformation from the ER model to a nested XML structure.

The nesting may be specified as a sequence of objects enclosed inside parentheses to indicate that the other instances (may be objects or tuples) in each tuple that are nested inside the first object. Formally: $(E(E_1, ..., E_n))$, $n \geq 1$, means that each E_i, $i \geq 1$ is nested inside E. Further, each E_i may be either an object or a tuple. This leads to a one level tree rooted at E and all the other nodes are direct children of E.

Note that the first instance inside a tuple must be an object and each other instance may be either an object or a tuple. For the object case, there must be a link in the ERD between it and the first object in the tuple. For the tuple case, the link in the ERD must connect the first object in the latter tuple with the first object in the former tuple.

For better understanding of the nesting process, consider the following cases:

1. The nesting specified using $(E_1(E_2, E_3, E_4, E_5))$ means that each of E_5, E_4, E_3, and E_2 are nested inside E_1. This is one level tree rooted at E_1 and each node E_i $(2 \leq i \leq 5)$ is at level one.

2. The nesting specified using $(E_1(E_2(E_3(E_4(E_5)))))$ means that E_5 is nested inside E_4, E_4 is nested inside E_3, E_3 is nested inside E_2, and E_2 is nested inside E_1. This is a chain, i.e., four levels tree rooted at E_1 and each node E_i $(2 \leq i \leq 5)$ is at level $i - 1$.

3. The nesting specified using $(E_1(((E_2(E_3)), E_4, (E_5(E_6(E_7))), E_8))$, is interpreted as follows: E_3 is nested inside E_2, E_7 is nested inside E_6 and E_6 is nested inside E_5; then E_2, E_4, E_5 and E_8 are all nested inside E_1.

The ER2NXML takes the nesting sequence specified by users as the input and generates an output of nested XML schema. The ER2NXML in pseudo-code is depicted in Algorithm 3.2.

Algorithm 3.2 ER2NXML (ER Model to Nested XML Conversion)

Input: The ER model and input nesting sequence as specified by the user

Output: The corresponding nested XML schema

Step:

If the user specified a non-empty nesting tuple then

Let the input nesting sequence be $(E_1(E_2))$; note that this may be generalized to a tuple with $n > 2$ tuples.

E_2 (whether object or tuple) is nested inside E_1 as multi-valued attribute.

If E_2 contains a foreign key that represents the primary key of E_1 then

Remove from E_2 the foreign key that represents the primary key of E_1.

If E_2 contains only foreign keys then

Replace E_2 inside E_1 by objects represented by foreign keys inside E_2.

For each of the objects (E_i) that replaced E_2 inside E_1 do

If E_i contains some instances not participating in the relationship then

Leave a representative element of E_i to hold its instances not participating in the relationship.

Else (If E_1 contains a foreign key that represents the primary key of E_2 then)

Remove from E_1 the foreign key that represents the primary key of E_2.

Leave a representative of E_2 to hold its instances not related to instances of E_1.

Else (the system will decide on the nesting)

For objects connected by *1:1* or *1:M* relationship, we nest the object that contains the foreign key inside the object that contains the primary key. Attributes of the relationship are added inside the latter object.

For objects connected by *M:N* or n-ary relationship do

Nest the other object(s) involved in the relationship inside object E_i which is involved in the relationship and has the smallest number of instances not participating in the relationship. Attributes of the relationship are added inside the latter object.

For each objects E_j nested inside E_i do

If E_j contains some instances not participating in the relationship then

Leave a representative of E_j to hold its instances not participating in the relationship.

EndAlgorithm 3.2

To illustrate the nesting process, consider the COMPANY database where users may write queries to retrieve information about employees and their projects. The users may specify the nesting sequence as: (EMPLOYEE(WORKS_ON(PROJECT))).

ER2NXML takes such sequence as input and generates as output the XML schema in a nested structure. The element of PROJECT is nested under the element of WORKS_ON. The nested element then moves under the element of EMPLOYEE to build two levels of nested XML structure.

In Figure 3.4, WORKS_ON has two foreign keys: ESSN refers to EMPLOYEE and PNO refers to PROJECT. Further, also WORKS_ON has an attribute HOURS. The user enters the nesting sequence as (EMPLOYEE(WORKS_ON(PROJECT))). The nested XML schema result generated based on ER2NXML is given next:

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:db="http://www.cpsc.ucaglary.ca/wangch/xml"
targetNamespace="http://www.cpsc.ucaglary.ca/wangch/xml"
elementFormDefault="qualified">
  <!--definition of simple and complex elements-->
  <xs:complexType name="EMPLOYEE_Relation">
    <xs:sequence>
      <xs:element name="EMPLOYEE_Tuple"
      type="db:EMPLOYEE_Tuple" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="EMPLOYEE_Tuple">
    <xs:sequence>
      <xs:element name="FNAME" type="xs:string" />
      <xs:element name="LNAME" type="xs:string" />
      <xs:element name="SSN" type="xs:string" />
```

```
    <xs:element name="BDATE" type="xs:string" />
    <xs:element name="ADDRESS" type="xs:string" />
    <xs:element name="SEX" type="xs:string" />
    <xs:element name="SALARY" type="xs:decimal" />
    <xs:element name="SUPERSSN" type="xs:string" />
    <xs:element name="DNO" type="xs:int" />
    <xs:element name="WORKS_ON_Relation"
                type="db:WORKS_ON_Relation" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="WORKS_ON_Relation">
  <xs:sequence>
    <xs:element name="WORKS_ON_Tuple"
    type="db:WORKS_ON_Tuple" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="WORKS_ON_Tuple">
  <xs:sequence>
    <xs:element name="HOURS" type="xs:decimal"/>
    <xs:element name="PROJECT_Relation"
                type="db:PROJECT_Relation"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PROJECT_Relation">
  <xs:sequence>
```

```
    <xs:element name="PROJECT_Tuple"
    type="db:PROJECT_Tuple" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PROJECT_Tuple">
  <xs:sequence>
    <xs:element name="PNAME" type="xs:string"/>
    <xs:element name="PNUMBER" type="xs:int"/>
    <xs:element name="PLOCATION" type="xs:string"/>
    <xs:element name="DNUM" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
<!--definition of key and keyref-->
<xs:element name="EMP_WOK_PRO_Nested">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="EMPLOYEE_Relation"
                  type="db:EMPLOYEE_Relation" />
      <xs:element name="PROJECT_Relation"
                  type="db:PROJECT_Relation" />
    </xs:sequence>
  </xs:complexType>
  <xs:key name="EMPLOYEE_PrimaryKey">
    <xs:selector xpath="db:EMPLOYEE_Relation/db:EMPLOYEE_Tuple"/>
    <xs:field xpath="db:SSN" />
```

```
    </xs:key>
    <xs:key name="PROJECT_PrimaryKey">
      <xs:selector xpath="db:PROJECT_Relation/db:PROJECT_Tuple"/>
      <xs:field xpath="db:PNUMBER" />
    </xs:key>
    <!--keyref-->
    <xs:keyref name="EMPLOYEE.superssn"
               refer="db:EMPLOYEE_PrimaryKey">
      <xs:selector xpath="db:employee_Relation/db:employee_Tuple"/>
      <xs:field xpath="superssn" />
    </xs:keyref>
  </xs:element>
</xs:schema>
```

In the result of nested XML schema above, WORKS_ON.ESSN is the foreign key of EMPLOYEE, and WORKS_ON.PNO is the foreign key of PROJECT, which are both removed according to Algorithm 3.2. Therefore, only WORKS_ON.HOURS attribute is left in WORKS_ON. If WORKS_ON does not contain the attribute HOURS, then it will be removed. In the XML document generated, all records in PROJECT which relate to WORKS_ON are nested under a particular WORKS_ON record and all records in WORKS_ON which relate to a particular EMPLOYEE are nested under that EMPLOYEE record. This forms two levels of nested XML structure.

N-ary Relationships

Figure 3.5 shows an example n-ary relationship TAKES, which connects three objects: STUDENTS, COURSES, and PROFESSORS. In addition to the attribute GRADE, there are three foreign keys in TAKES: SID, CID and PID. Assume users often query for information related to students taking courses and their professors. Thus, users may choose to nest both COURSES and PROFESSORS under TAKES and then nest TAKES under STUDENTS. The result of the two levels nested XML schema is given below.

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:db="http://www.cpsc.ucaglary.ca/wangch/xml"
targetNamespace="http://www.cpsc.ucaglary.ca/wangch/xml"
elementFormDefault="qualified">
 <xs:complexType name="STUDENTS_Relation">
    <xs:sequence>
      <xs:element name="STUDENTS_Tuple"
          type="db:STUDENTS_Tuple" maxOccurs="unbounded"/>
    </xs:sequence>
 </xs:complexType>
  <xs:complexType name="STUDENTS_Tuple">
    <xs:sequence>
      <xs:element name="SID" type="xs:string"/>
      <xs:element name="SNAME" type="xs:string"/>
      <xs:element name="MAJOR" type="xs:string"/>
      <xs:element name="TAKES_Relation"
```

```
                type="db:TAKES_Relation"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="TAKES_Relation">
    <xs:sequence>
      <xs:element name="TAKES_Tuple"
          type="db:TAKES_Tuple" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="TAKES_Tuple">
    <xs:sequence>
      <xs:element name="GRADE" type="xs:string"/>
      <xs:element name="COURSES_Relation"
                type="db:COURSES_Relation"/>
      <xs:element name="PROFESSORS_Relation"
                type="db:PROFESSORS_Relation"/>
    </xs:sequence>
  </xs:complexType>
<xs:complexType name="COURSES_Relation">
    <xs:sequence>
      <xs:element name="COURSES_Tuple"
          type="db:COURSES_Tuple" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="COURSES_Tuple">
```

```
  <xs:sequence>
    <xs:element name="CID" type="xs:string"/>
    <xs:element name="CNAME" type="xs:string"/>
    <xs:element name="DEPT" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PROFESSORS_Relation">
  <xs:sequence>
    <xs:element name="PROFESSORS_Tuple"
        type="db:PROFESSORS_Tuple" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PROFESSORS_Tuple">
  <xs:sequence>
    <xs:element name="PID" type="xs:string"/>
    <xs:element name="PNAME" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
 <xs:element name="STUDENTS">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="STUDENTS_Relation"
                  type="db:STUDENTS_Relation"/>
      <xs:element name="COURSES_Relation"
                  type="db:COURSES_Relation"/>
```

```
        <xs:element name="PROFESSORS_Relation"
                    type="db:PROFESSORS_Relation"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="STUDENTS_PrimaryKey">
      <xs:selector xpath="db:STUDENTS_Relation/db:STUDENTS_Tuple"/>
      <xs:field xpath="db:SID" />
    </xs:key>
    <xs:key name="COURSES_PrimaryKey">
      <xs:selector xpath="db:COURSES_Relation/db:COURSES_Tuple"/>
      <xs:field xpath="db:CID" />
    </xs:key>
    <xs:key name="PROFESSORS_PrimaryKey">
      <xs:selector xpath="db:PROFESSORS_Relation/db:PROFESSORS_Tuple"/>
      <xs:field xpath="db:PID" />
    </xs:key>
  </xs:element>
</xs:schema>
```

Compared to the flat XML structure, the nested XML structure improves the response of certain queries because there is no need to use keyrefs to perform additional scans to find tuples in other parts of the referenced XML documents. However, nested XML structure has data redundancy because some records will repeat several times. It is important to emphasize that the main purpose of facilitating nested structures to allow users to construct XML document most suitable for efficient information retrieval. Thus, it is reasonable to leave it to users to make the nesting

decision.

Generating XML Documents

After the XML schema is obtained, the next step is to generate XML document(s) from the considered relational database. Algorithm 3.3 (GenXMLDoc) checks top-down through the list of selected objects and generates an element for each object.

Algorithm 3.3 GenXMLDoc (Generating XML Document)

Input: XML schema and Relational database

Output: The corresponding XML Document

Step:

Create XML document and set its namespace declaration

Create a root element of the XML document with the same name as the root name of the XML schema

For each relation R in the relational database do

If R is selected and does not contain any nested relations

Create R_Relation element for R

Let queryString = "select * from R"

ResultSet = execute(queryString)

For each tuple T in ResultSet do

Create R_Tuple element for tuple T

Create an element for each attribute in R and insert it into R_Tuple element

else if R is selected and contains a nested relation R_c then

Create R_Relation element for R and R_c_Relation for R_c

Let queryString = "select selectedAttrs from R, R_c"

ResultSet = execute(queryString)

For each tuple T in ResultSet do

Create R_Tuple element for the tuple of R, and R_c_Tuple element for the tuple of R_c

Create an element for each selected attribute in R and insert it to R_Tuple element, and do same for R_c

EndAlgorithm 3.3

Algorithm 3.3 can generate flat XML documents as well as nested XML documents, depending on the processed XML schema. In Algorithm 3.3, a query is executed to obtain all tuples that satisfy the constraints so one element is created to store data of each tuple in the result set. The following shows a fragment of the XML document output.

```
<?xml version="1.0" encoding="UTF-8"?> <test
xmlns="http://www.cpsc.ucaglary.ca/wangch/xml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cpsc.ucaglary.ca/wangch/xml test_2.xsd">
  <db:EMPLOYEE_Relation xmlns:db="http://www.cpsc.ucaglary.ca/wangch/xml">
    <db:EMPLOYEE_Tuple>
      <db:FNAME>James</db:FNAME>
      <db:LNAME>Borg</db:LNAME>
```

```
    <db:SSN>888665555</db:SSN>
    <db:BDATE>1927-11-10</db:BDATE>
    <db:ADDRESS>450 Stone, Houston, TX</db:ADDRESS>
    <db:SEX>M</db:SEX>
    <db:SALARY>55000.00</db:SALARY>
    <db:SUPERSSN />
    <db:DNO>1</db:DNO>
    <db:WORKS_ON_Relation>
      <db:WORKS_ON_Tuple>
        <db:HOURS />
        <db:PROJECT_Relation>
          <db:PROJECT_Tuple>
            <db:PNAME>Reorganization</db:PNAME>
            <db:PNUMBER>20</db:PNUMBER>
            <db:PLOCATION>Houston</db:PLOCATION>
            <db:DNUM>1</db:DNUM>
          </db:PROJECT_Tuple>
        </db:PROJECT_Relation>
      </db:WORKS_ON_Tuple>
    </db:WORKS_ON_Relation>
  </db:EMPLOYEE_Tuple>
  . . . . . .
<db:EMPLOYEE_Relation
```

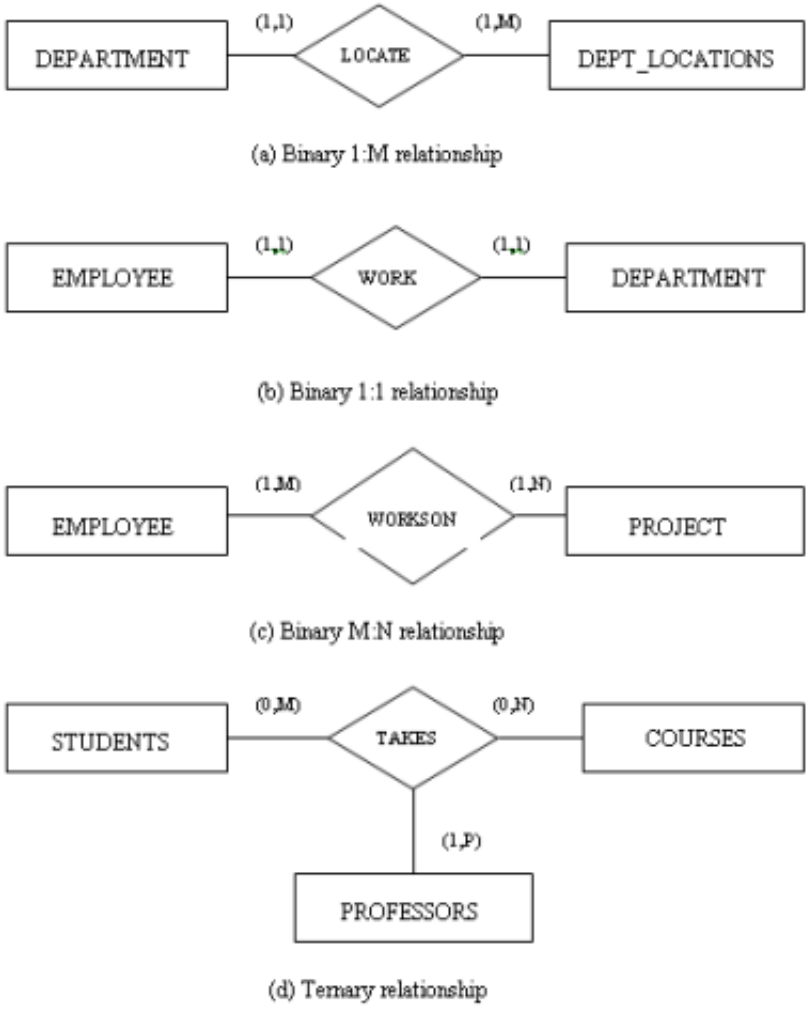


Figure 3.1: An example of relationship types in the ER model

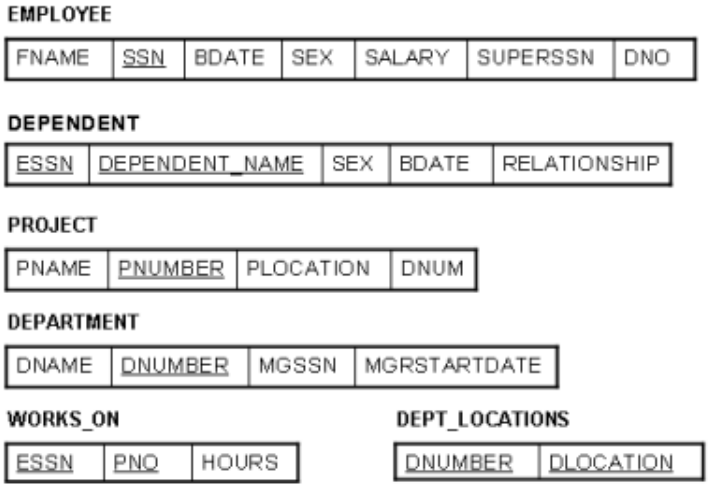


Figure 3.2: An example: the COMPANY relational database

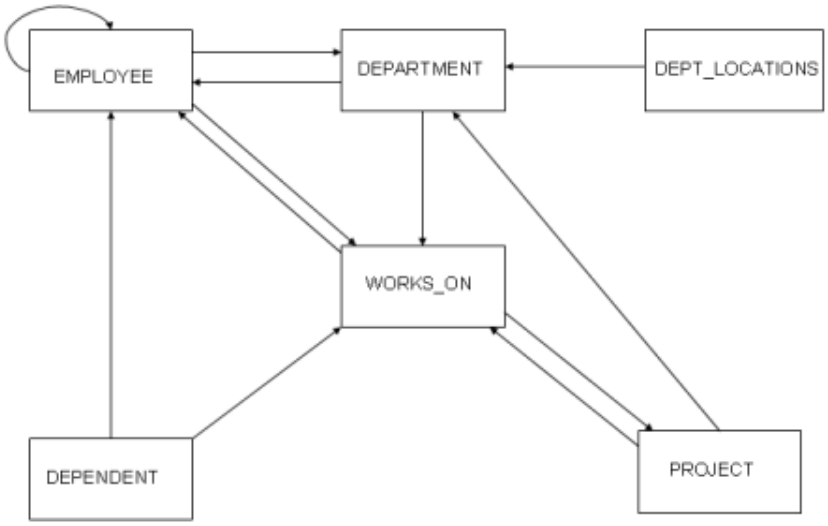


Figure 3.3: The initial RID graph of the COMPANY database

Table 3.1: Candidate keys: a list of possible candidate keys of all relations

Relation Name	**Attribute Name**	**Candidate Key#**
DEPARTMENT	DNAME	1
DEPARTMENT	DNUMBER	2
DEPARTMENT	MGRSSN	3
DEPARTMENT	MGRSTARTDATE	4
DEPENDENT	BDATE	1
DEPENDENT	ESSN	2
DEPENDENT	DEPENDENT_NAME	2
DEPENDENT	ESSN	3
DEPENDENT	RELATIONSHIP	3
DEPT_LOCATIONS	DNUMBER	1
DEPT_LOCATIONS	DLOCATION	1
EMPLOYEE	ADDRESS	1
EMPLOYEE	BDATE	2
EMPLOYEE	FNAME	3
EMPLOYEE	LNAME	4
EMPLOYEE	SSN	5
EMPLOYEE	SEX	6
EMPLOYEE	SALARY	6
EMPLOYEE	DNO	6
EMPLOYEE	SEX	7
EMPLOYEE	SALARY	7
EMPLOYEE	SUPERSSN	7
PROJECT	PNAME	1
PROJECT	PNUMBER	2
WORKS_ON	ESSN	1
WORKS_ON	PNO	1

Table 3.2: Foreign keys: a list of attributes in candidate keys and their corresponding foreign keys

Candidate Key Attributes		***Foreign Key Attributes***		
R_Name	**Attr_Name**	**R_Name**	**Attr_Name**	**Link#**
DEPARTMENT	DNUMBER	DEPT_LOCATIONS	DNUMBER	1
DEPARTMENT	DNUMBER	EMPLOYEE	DNO	1
DEPARTMENT	DNUMBER	PROJECT	DNUM	1
DEPARTMENT	MGRSSN	EMPLOYEE	SUPPERSSN	1
EMPLOYEE	SSN	DEPARTMENT	MGRSSN	1
EMPLOYEE	SSN	DEPENDENT	ESSN	1
EMPLOYEE	SSN	WORKS_ON	ESSN	1
EMPLOYEE	SSN	EMPLOYEE	SUPERSSN	1
PROJECT	PNUMBER	WORKS_ON	PNO	1
WORKS_ON	ESSN	DEPARTMENT	MGRSSN	1
WORKS_ON	ESSN	DEPENDENT	ESSN	1
WORKS_ON	ESSN	EMPLOYEE	SSN	1
WORKS_ON	ESSN	EMPLOYEE	SUPERSSN	1
WORKS_ON	PNO	PROJECT	PNUMBER	1

Table 3.3: Primary keys: a list of the primary keys for the COMPANY database

Relation Name	**Attribute Name**
DEPARTMENT	DNUMBER
DEPENDENT	ESSN
DEPENDENT	DEPENDENT_NAME
DEPT_LOCATIONS	DNUMBER
DEPT_LOCATIONS	DLOCATION
EMPLOYEE	SSN
PROJECT	PNUMBER
WORKS_ON	ESSN
WORKS_ON	PNO

Table 3.4: The optimized ForeignKeys table

Candidate Key Attributes		***Foreign Key Attributes***		
R_Name	**Attr_Name**	**R_Name**	**Attr_Name**	**Card**
DEPARTMENT	DNUMBER	DEPR_LOCATIONS	DNUMBER	M:1
DEPARTMENT	DNUMBER	EMPLOYEE	DNO	M:1
DEPARTMENT	DNUMBER	PROJECT	DNUM	M:1
EMPLOYEE	SSN	DEPARTMENT	MGRSSN	1:1
EMPLOYEE	SSN	DEPENDENT	ESSN	M:1
EMPLOYEE	SSN	EMPLOYEE	SUPERSSN	M:1
EMPLOYEE	SSN	WORKS_ON	ESSN	M:1
PROJECT	PNUMBER	WORKS_ON	PNO	M:1

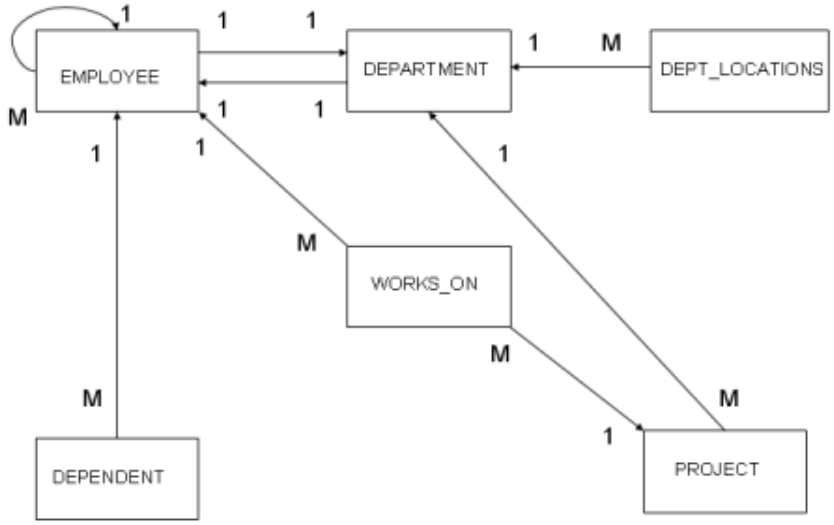


Figure 3.4: The optimized RID graph of the COMPANY database

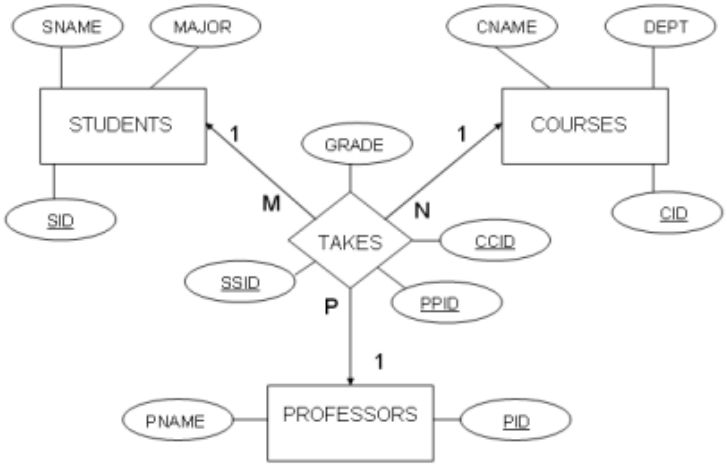


Figure 3.5: An sample of ternary relationship

Chapter 4

Transformation from XML to Relational Schema

4.1 Introduction

In this chapter, transforming XML representations into relational representations is considered. The XML-to-Relational transformation algorithm is presented. XML is a logical model and end users/applications view the data in XML. However, the data can be physically stored in different ways: (a) as direct-labeled graph in Native XML databases; (b) in relational tables using products provided by vendors such as IBM, Microsoft, or Oracle; or (c) some applications may store portion of the data which is structured in relational format while the rest of data which is unstructured in native databases. It is believed that a large amount of XML data will be stored in relational databases and the XML-to-Relational transformation problem is an important topic in XML research area. In Section 4.2 related work is described. The XML-to-Relational transformation algorithm is then detailed in Section 4.3.

4.2 Related Work

More recently, much research have addressed the particular issues of the transformation from XML to relational data. On the commercial side, database vendors are busy extending their database products to adopt XML type. In the next two sections, an overview of the XML-to-Relational schema mapping specification language

currently available from two selected commercial DBMSs: IBM DB2 and Microsoft SQL Server are provided.

4.2.1 IBM DB2

IBM DB2 [CX00]offers the XML Extender for storing XML documents. DAD (Document Access Definition) is used for XML-to-Relational schema mapping. A DAD uses two elements, *element_node* and *attribute_node*, to describe the structure of XML documents; and one subelement *RDB_node* to describe its mapping to the relational schema. *Element_node* specifies the elements in an XML document and *attribute_node* specifies the attributes. Both *element_node* and *attribute_node* have *RDB_node* as their subelement. *RDB_node* in turn has the subelements *table*, *column*, and *condition*, which respectively, specify a table, a column, and a relationship between tables in a relational schema.

In the DAD mapping, XML elements are mapped to relational tables or columns, XML attributes are mapped to database columns, and relationships between XML elements are mapped to relationships between relational tables. The rules for this mapping are as follows.

1. specify the structure of the XML documents using *element_node* and *attribute_node*,
2. specify the mapping to a relational schema using *RDB_node*,
3. specify all relationships between tables in the *RDB_node* subelement of the root *element_node*, and
4. specify the table and column, to which an element or an attribute is mapped, in the *RDB_node* subelement of each non-root *element_node* and *attribute_node*.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:annotation>
   <xsd:documentation>
     Book schema for www.book.com.
     Copyright 2001 www.book.com. All rights reserved.
   </xsd:documentation>
 </xsd:annotation>
 <xsd:element name="book">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="author" minOccurs="0" maxOccurs="unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="name" type="xsd:string"/>
             <xsd:element name="email" type="xsd:string"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:integer"/>
   </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figure 4.1: An example of a XML schema.

Figure 4.1 shows an example of an XML schema and Table 4.1 shows an example of a relational schema. Figure 4.2 shows a DAD specifying the mapping from the XML document structure in Figure 4.1 to the relational schema in Table 4.1. The table *book*, the table *author* and the relationship between the two tables are specified in the *RDB_node* subelement of the root *element_node* named "book". The columns id and title of the table book are specified in the *RDB_node* subelements of the *attribute_node* named "id" and the *element_node* named "title", respectively. Likewise,

Table 4.1: An example of a relational schema

book	*name*	id	title
	type	integer	varchar(100)

author	*name*	bookid	name	email
	type	integer	varchar(100)	varchar(100)

the columns name and email of the table author are specified in the *RDB_node* subelements of the *element_node* named "name" and the *element_node* named "email".

4.2.2 Microsoft SQL Server

Microsoft SQL Server offers the XML Bulk Load utility for storing XML documents. Its XML-to-Relational mapping language is the annotated XDR (XML-Data Reduced) schema. XDR uses four elements, such as *ElementType*, *AttributeType*, *element*, and *attribute* to specify the structure of XML documents; and two attributes, *relation* and *field*, as well as one element *relationship* to specify the mapping to a relational schema. Specifically, *ElementType* and *AttributeType* are used, respectively, to declare XML elements and attributes, whereas *element* and *attribute* are used to refer to the declared *ElementType* and *AttributeType*. In addition, the attributes *relation* and *field*, respectively, specify a table and a column, and the element *relationship* specifies the relationship between two tables in the relational schema.

Like the DAD, the annotated XDR schema maps XML elements and attributes to relational tables or columns, and relationships between XML elements to relationships between relational tables. The principles for specifying the mapping are similar to that of DAD.

1. specify the structure of XML documents using the elements *ElementType*, *AttributeType*, *element*, and *attribute*,

2. specify the table and column, to which an element or an attribute is mapped, using the attributes *relation* and *field*, and

3. specify the relationship between tables using the element *relationship*.

Figure 4.3 shows an annotated XDR schema corresponding to the DAD in Table 4.1. The element *book* is mapped to the table *book*, and the attribute *author* and the element *title* are, respectively, mapped to the columns *id* and *title* of the table *book*. Likewise, the element *author* is mapped to the table *author*, and the attribute *name* and the element *email* are, respectively, mapped to the columns *name* and *email* of the table *author*. Besides, the relationship between two elements *book* and *author* is mapped to the relationship between two tables *book* and *author*.

Researchers have built systems that serve as middleware between users and database systems, including SilkRoute, XPERANTO, and Agora. In these systems, users provide the relational schema and queries which give the XML view. They do operations on the XML view to translate the XML data to relational database. The XML schema is also obtained from these queries. More details about these systems are included in Section 3.2.

4.3 Transforming XML Schema into Relational Schema

Our approach to this problem is different from existing approaches described earlier. IBM DB2, Microsoft SQL Server and SilkRoute, all require users to use a new lan-

guage, such as DAD, XDR, and RXL. Our conversion from XML to relational schema is straightforward and handles all constraint mappings. In Chapter 3, the algorithms to transform the relational data to flat and nested XML schema are presented . How do we transform the resulting XML schema back to the relational schema? The XML-to-Relational transformation in pseudo-code is depicted in Algorithm 4.1 to accomplish this task.

Algorithm 4.1 XML2R (XML to Relational Schema Conversion)

Input: XML schema file

Output: The corresponding relational schema

Step:

1. Generate relational structure from a given XML schema by:
    - Converting each "complexType" in XML schema into a relational table.
    - Converting each sequence element in the "complexType" into an attribute of that relational table.
2. Transform XML schema constraints into relational schema constraints by:
    - Converting "unique" XML schema constraints into relational constraints.
    - Converting "key" XML schema constraints into relational constraints.
    - Converting "keyref" XML schema constraints into relational constraints.

EndAlgorithm 4.1

Each step is now considered in detail. We then give an example to demonstrate how the algorithm works.

4.3.1 Generate Relational Structure from a Given XML Schema

This section provides an overview of how the relational schema is built from an XML schema. In general, we map each "complexType" in the XML schema to a relational table. The relational table structure is determined by the definition of the complex type. We map each sequence element in the "complexType" to an attribute in the relational table. Tables are created for top-level elements in the XML schema. However, a table is only created for a top-level "complexType" element when the "complexType" element is nested inside another "complexType" element; in which case, the nested "complexType" element is mapped to another relational table in the relational schema.

The following example demonstrates an XML schema where DEPARTMENT is the "complexType" element.

```
<xs:complexType name="DEPARTMENT_Relation">
    <xs:sequence>
      <xs:element name="DEPARTMENT_Tuple" type="db:DEPARTMENT_Tuple"
                  maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPARTMENT_Tuple">
    <xs:sequence>
      <xs:element name="DNAME" type="xs:string" />
      <xs:element name="DNUMBER" type="xs:int" />
      <xs:element name="MGRSSN" type="xs:string" />
      <xs:element name="MGRSTARTDATE" type="xs:string" />
```

```
    </xs:sequence>
</xs:complexType>
```

According to Algorithm 4.1, we obtain the DEPARTMENT relational table from the "complexType" element DEPARTMENT. The produced DEPARTMENT table contains the four attributes: DNAME, DNUMBER, MGRSSN, and MGRSTART-DATE as follows:

DEPARTMENT(DNAME, DNUMBER, MGRSSN, MGRSTARTDATE)

The data type of each attribute in the DEPARTMENT table is derived from the XML schema type of the corresponding element specified. This example shows a flat XML schema structure fragment. In the next example, a nested XML schema structure follows.

```
<xs:complexType name="DEPARTMENT_Relation">
    <xs:sequence>
      <xs:element name="DEPARTMENT_Tuple" type="db:DEPARTMENT_Tuple"
                  maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType> <xs:complexType name="DEPARTMENT_Tuple">
    <xs:sequence>
      <xs:element name="DNAME" type="xs:string" />
      <xs:element name="DNUMBER" type="xs:int" />
      <xs:element name="MGRSSN" type="xs:string" />
      <xs:element name="MGRSTARTDATE" type="xs:string" />
      <xs:element name="DEPT_LOCATIONS_Relation"
                  type="db:DEPT_LOCATIONS_Relation" />
```

```
    </xs:sequence>
</xs:complexType> <xs:complexType name="DEPT_LOCATIONS_Relation">
    <xs:sequence>
      <xs:element name="DEPT_LOCATIONS_Tuple" type="db:DEPT_LOCATIONS_Tuple"
                  maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType> <xs:complexType name="DEPT_LOCATIONS_Tuple">
    <xs:sequence>
      <xs:element name="DNUMBER" type="xs:int" />
      <xs:element name="DLOCATION" type="xs:string" />
    </xs:sequence>
</xs:complexType>
```

In this example, the "complexType" element DEPT_LOCATIONS_Relation is nested inside the "complexType" element DEPARTMENT_Relation. Therefore, the mapping process produces two tables: DEPARTMENT and DEPT_LOCATIONS.

DEPARTMENT (DNAME, DNUMBER, MGRSSN, MGRSTARTDATE)

DEPT_LOCATIONS (DNUMBER, DLOCATION)

4.3.2 Convert XML Schema Constraints to Relational Constraints

Now let us convert XML schema constraints to relational constraints. In an XML schema, the uniqueness constraint is specified using the "unique" element; the key constraint is specified using the "key" element; the relationship constraint between tables is specified using the "keyref" element.

In an XML schema, we can indicate that an attribute or element value must be unique within a specific document. The "unique" element is used to identify a set of elements and then to identify the attribute or "field" element relative to each selected element that must be unique. In the process of transforming an XML schema into a relational schema, we map the unique constraint specified on an element in the XML schema to a relational unique constraint. The example of the unique element XML schema statement is:

```
<xs:unique>
      <xs:selector xpath="db:DEPARTMENT_Relation/db:DEPARTMENT_Tuple" />
      <xs:field xpath="db:DNAME" />
</xs:unique>
```

In this XML schema, the "unique" element specifies that the value of the DNAME child element must be unique for all DEPARTMENT_Tuple elements in a XML document instance.

In an XML schema, we can specify a key constraint on an element using the "key" element. The element on which a key constraint is specified must have unique values in any schema instance, cannot be null valued, and can be referenced from elsewhere. This constraint is similar to the unique constraint, except that the column on which a key constraint is defined cannot have null values.

In the mapping process, we convert the "key" element to an attribute which is the primary key of a relational schema. In the following XML schema example, the "key" element specifies the key constraint on the element DNUMBER; the "key" element specifies that the values of the DNUMBER child element of the DEPARTMENT el-

ement must have unique values and cannot be null valued. In transforming the XML schema, we map the element DNUMBER to the primary key of the DEPARTMENT table.

```
<xs:key name="DEPARTMENT_PrimaryKey">
     <xs:selector xpath="db:DEPARTMENT_Relation/db:DEPARTMENT_Tuple"/>
     <xs:field xpath="db:DNUMBER" />
</xs:key>
```

The XML schema uses the "keyref" element to build links between elements within a specific XML document. This is similar to a primary key and foreign key relationship in a relational schema. If a XML schema specifies the "keyref" element, the "keyref" element is transformed during the mapping process to a corresponding foreign key constraint on the attributes in the relational tables.

```
<xs:keyref name="DEPT_LOCATIONS.DNUMBER" refer="db:DEPARTMENT_PrimaryKey">
     <xs:selector xpath="db:DEPT_LOCATIONS_Relation/db:DEPT_LOCATIONS_Tuple"/>
     <xs:field xpath="DNUMBER"/>
</xs:keyref>
```

In the above example XML schema, the DEPT_LOCATIONS.DNUMBER child element of the DEPT_LOCATIONS element refers to the DNUMBER key child element of the DEPARTMENT element. In the mapping process, DEPT_LOCATION. DNUMBER "keyref" element is transformed to a foreign key attribute in DEPT_ LOCATIONS table. Further, the relationship between the table DEPARTMENT and the table DEPT_LOCATIONS is built.

In the mapping process, the relationship between two relational tables is obtained from the "keyref" element in the XML schema. In addition to this, the relationships can be formed from the specified nested complex types. Nested complex type definitions in a schema indicate the parent-child relationships of the elements.

```
<element_node name="book">
    <RDB_node>
        <table name="book" key="id"/>                                  ]
        <table name="author"/>                                         ] mapping the relationship between
        <condition> book.id=author.bookId </condition>                 ] the elements book and author
    </RDB_node>
    <attribute_node name="id">
        <RDB_node>
            <table name="book"/>                                       ]
            <column name="id" type="integer"/>                         ] mapping the attribute id
        </RDB_node>
    </attribute_node>
    <element_node name="title">
        <text_node>
            <RDB_node>
                <table name="book"/>                                   ]
                <column name="title" type="varchar(100)"/>             ] mapping the attribute title
            </RDB_node>
        </text_node>
    </element_node>
    <element_node name="author" multi_occurrence="YES">
        <element_node name="name">
            <text_node>
                <RDB_node>
                    <table name="author"/>                             ]
                    <column name="name" type="varchar(100)"/>          ] mapping the element author
                </RDB_node>
            </text_node>
        </element_node>
        <element_node name="email">
            <text_node>
                <RDB_node>
                    <table name="author"/>                             ]
                    <column name="email" type="varchar(100)"/>         ] mapping the element email
                </RDB_node>
            </text_node>
        </element_node>
    </element_node>
</element_node>
```

Figure 4.2: An example of a DAD.

```
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql">

    <ElementType name="book" sql:relation="book">                          ] mapping the elements
        <element type="title" sql:field="title"/>                          ]   book and title
        <element type="author" minOccurs="0" maxOccurs="*">
            <sql:relationship key-relation="book" key="id"                 ] mapping the relationship between
                              foreign-relation="author" foreign-key="bookId"/> ] the elements book and author
        </element>
        <attribute type="id" sql:field="id"/>                              ] mapping the attribute id
    </ElementType>

    <ElementType name="title" dt:type="string"/>
    <AttributeType name="id" dt:type="int"/>

    <ElementType name="author" sql:relation="author">                      ] mapping the elements author,
        <element type="name" sql:field="name"/>                            ]   name, and email
        <element type="email" sql:field="email"/>                          ]
    </ElementType>

    <ElementType name="name" dt:type="string"/>
    <ElementType name="email" dt:type="string"/>
</Schema>
```

Figure 4.3: An example of annotated XDR schema.

Chapter 5

COCALEREX: A Engine for Converting Relational Databases into XML

5.1 Introduction

XML is gradually being accepted as the standard format for publishing and exchanging data over the Internet. Therefore, it is important to automatically generate XML documents containing information from existing databases and preserve as much information as possible during the transformation process. Since a large number of the existing relational databases are classified as legacy and the transformation of legacy relational databases to XML has received little attention, so COCALEREX has been developed to successfully handle the conversion process for both legacy and catalog-based databases. It leads to identifying and understanding all components of an existing database and the relationships between them. It also provides an efficient algorithm for transforming relational database content into XML document(s).

The balance of this chapter is organized as follows. Related work is discussed in Section 5.2. The implementation and development environment for COCALEREX are discussed in Section 5.3. Section 5.4 presents COCALEREX architecture and reports experimental results.

5.2 Related Work and Our Approach

There exist several applications that enable the composition of XML documents from relational data, such as IBM DB2 XML Extender [CX00], SilkRoute [FTS00], XPERANTO [CFI+00], and Agora [MFK01]. These systems are detailed in Section 2.3, Section 3.2, and Section 4.2. If we consider IBM DB2 XML Extender, SilkRoute, XPERANTO, and Agora systems as one category of the related work, described next is another category. The work described in [LMCC01] requires knowing the catalog contents of the given relational database in order to extract the relational schema. The conversion of Relational-to-ER-to-XML has been proposed in [FPB01]. This reconstructs the semantic model, in the form of ER model, from the logical schema model capturing user's knowledge, and then transforms the ER model into the XML document. However, *M:N* and n-ary relationships are not considered. Finally, VXE-R [LVLG03, LLG] is an engine for transforming an underlying relational schema into equivalent XML schema. Then, XML queries are issued directly against the XML schema. VXE-R is used only for a certain type of the catalog-based relational databases; it does not work for legacy databases. VXE-R does not consider n-ary relationship; and also does not provide a visualized interface to users.

Our research takes an approach similar to the one described by Liu *et al.* [LVLG03, LLG]. In SilkRoute and XPERANTO, users cannot see the integrity constraints buried in the relational schema from the defined XML views. VXE-R translates relational data to XML schema while preserving integrity constraints defined in a relational database schema. COCALEREX provides the following functionalities:

- Supports a user-friendly visualized interface.

- Can be used for catalog-based relational databases build using any of the known RDBMS, including FirstSql, DB2, MySql, Oracle, and MS Access.
- Users are given the option to choose between legacy database and a catalog-based database.
- Keeping a list of the converted/available databases, and users can add/delete any database from the list.
- Users can specify the nesting sequence.
- Extracting information from legacy as well as catalog-based relational databases, and display the result with the GUI.
- Building an ER model for the given relational database.
- Converting the ER model to the corresponding XML schema and actual XML document in either flat or nested structure.
- Can properly and equally deal with binary and n-ary relationships mapping.
- The results can be saved to files.

COCALEREX has been developed based on the framework described by Wang *et al.* [WLAB04]. It allows users to view the underlying relational data in either flat or nested XML structure depend on their desire and needs. It utilizes the nesting sequence specified by the user in producing the XML schema. It allows users to directly view the result of each phase of the process. COCALEREX is composed of four main modules:

1. EELRR - **E**xtracting **E**R Model from **L**egacy **R**elational Database by **R**everse Engineering Module;
2. EECR - **E**xtracting **E**R Model from **C**atalog-based **R**elational Database Module;
3. ER2X - **ER** model to **XML** Module; and
4. X2R - **X**ML schema to **R**elational schema Module.

More detailed description of COCALEREX is provided in the following sections.

5.3 Program Language and Development Environment

COCALEREX is implemented in Java using JDBC/ODBC interface to connect to RDBMSs that hold the relational database to be processed. The reasons for these choices are:

- Java is an Object-Oriented language and is easy to implement.
- Java is compatible with many platforms, including Window XP/2000, UNIX and Linux; it is easy to setup.
- SQL objects in Java can be used to make implementation easier.
- We can use JDOM to obtain the XML schema.
- Java supports user-friendly visual graphical interface. It is easy for users to use the system and to see the output on the GUI.

- To use JDBC/ODBC interface connection, users can download the JDBC/ODBC driver which they need online easily.

Our implementation in Java runs on the Eclipse Platform. The Eclipse Platform is designed to build integrated development environments (IDEs) that can be used to create applications as diverse as web sites, embedded JavaTM programs, C++ programs, and Enterprise JavaBeansTM.

Eclipse was originally developed by IBM as a common, easy-to-use interface that provides a consistent "look and feel", regardless of which vendor's tool you prefer. In 2001 IBM donated this open source code to a new community of software tool vendors and eclipse.org was born.

The Eclipse Platform has been designed and built to meet the following requirements:

- Supports the construction of a variety of tools for application development.
- Supports an unrestricted set of tool providers, including independent software vendors.
- Supports tools to manipulate arbitrary content types (e.g. HTML, Java, C, JSP, EJB, XML, and GIF).
- Facilitates seamless integration of tools within and across different component types and tool providers.
- Supports both GUI and non-GUI-based application development environments.
- Runs on a wide range of operating systems, including Windows and Linux.

- Capitalizes on the popularity of Java programming for writing tools.

The principal role of the Eclipse Platform is to provide tool providers with mechanisms to use, and rules to follow, that lead to seamlessly-integrated tools. These mechanisms are exposed via well-defined API interfaces, classes, and methods. The Platform also provides useful building blocks and frameworks to facilitate developing new tools.

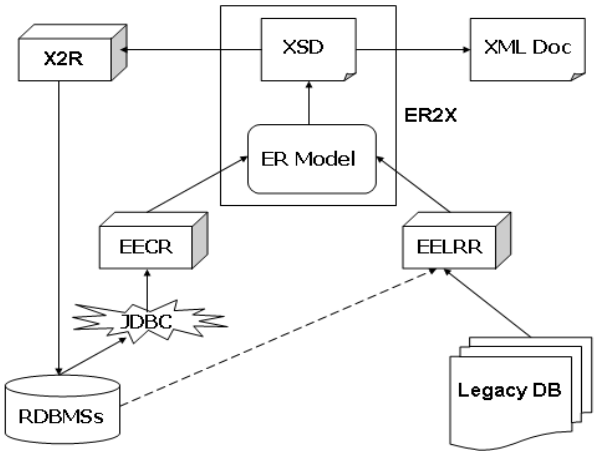


Figure 5.1: The architecture of COCALEREX system

5.4 COCALEREX System Architecture and Experimental Results

The architecture of COCALEREX as depicted in Figure 5.1 consists of four main modules: EELRR, EECR, ER2X, and X2R.

Users can use the main GUI of COCALEREX shown in Figure 5.2 to specify the category of the database to be converted into XML as either legacy or catalog-based.

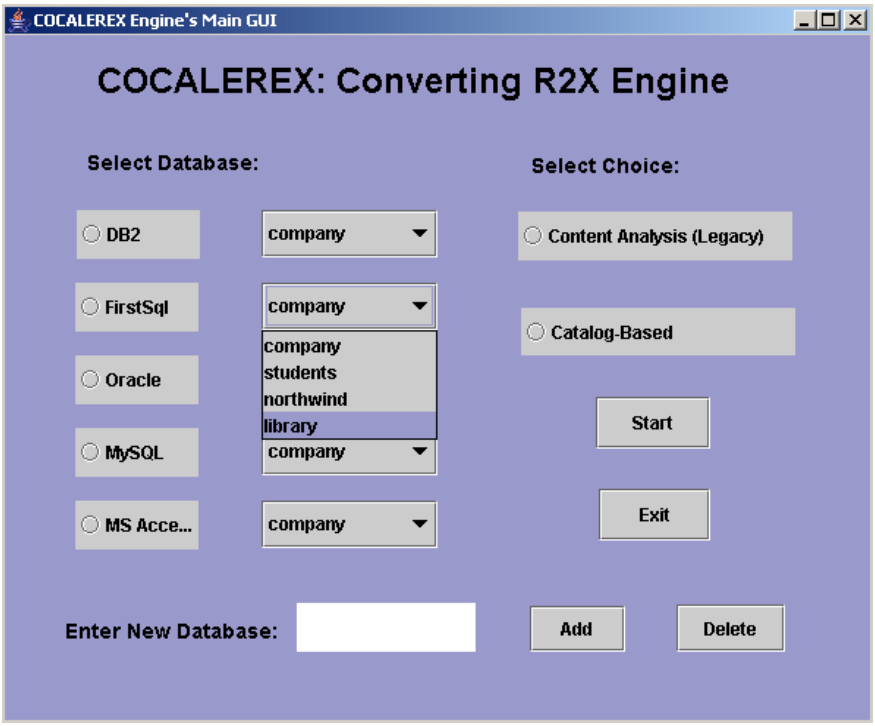


Figure 5.2: The main GUI of COCALEREX system

For the former category, the system calls the set of functions built inside EELRR to extract all possible information from the legacy database. The extracted information is equivalent to the catalog information required for constructing the ER model. For the latter category, the system connects to the corresponding RDBMS by using JDBC/ODBC connection, and calls the set of functions built inside EECR to get the catalog information necessary for constructing the ER model. Such information includes primary and foreign keys. After the ER model of a given database is generated, ER2X takes the ER model as input, convert the ER model to XML schema according to the transformation algorithms defined in Chapter 3. COCALEREX can generate flat as well as nested XML depending on users' specifications. If users

wish to nest some entities at different levels, they can specify the nesting sequence string in the input window. The system also can convert XML schema to relational schema. This conversion is completed inside X2R.

The four basic modules of COCALEREX are described in more details in the following sections.

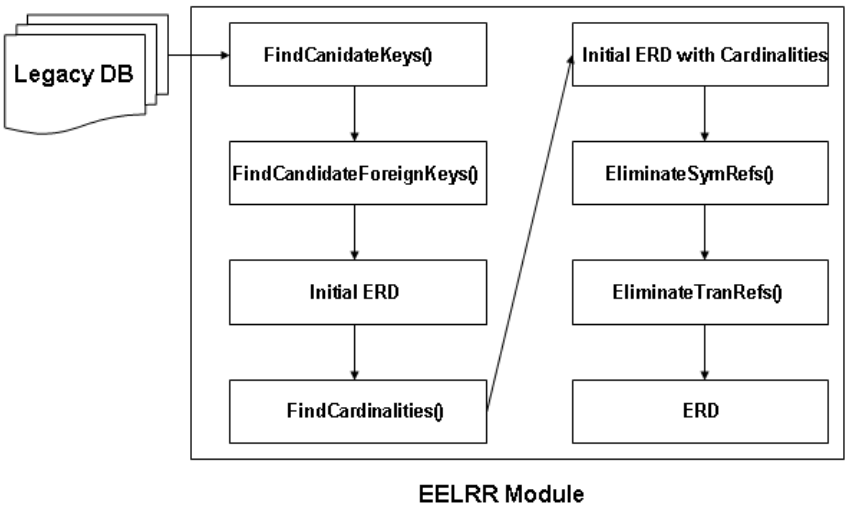


Figure 5.3: The flowchart of EELRR module

5.4.1 EELRR Module

EELRR is the most complicated and important module in COCALEREX. The main purpose of EELRR is to extract all possible information (foreign and candidate/primary keys) from the given legacy relational database, and then generates and displays an ER diagram (ERD) on the screen. For this task, a set of algorithms proposed by Alhajj [Alh03] are implemented. This is [Alh03] a method for extracting a conceptual schema from a legacy database through reverse engineering technique.

He developed a set of algorithms that investigate characteristics of an existing legacy database to identify candidate keys of all relations in the relational schema, to locate foreign keys, and to decide on the appropriate links between the given relations. Based on this analysis, a graph consistent with the entity-relationship diagram is derived to contain all possible binary and n-ary relationships between the given relations. The minimum and maximum cardinalities of each link in the mentioned graph are determined and extra links within the graph are eliminated. Finally, *M:N* and n-ary relationships are identified. According to the flowchart shown in Figure 5.3, the ER model extraction process can be divided into the following main steps:.

1. For each table in the relational database: the powerset of its set of attributes is the input to the "Find Candidate Keys" algorithm, which finds all possible candidate keys in the database.

2. Run the "Find Candidate Foreign Keys" algorithm on the result obtained from Step 1 to find all attributes in foreign keys, which are simply representatives of Candidate Keys.

3. Use the result from Step 2 to construct the initial ERD.

4. Run the "Find Cardinalities" algorithm to determine the cardinalities of the relationships in the initial ERD.

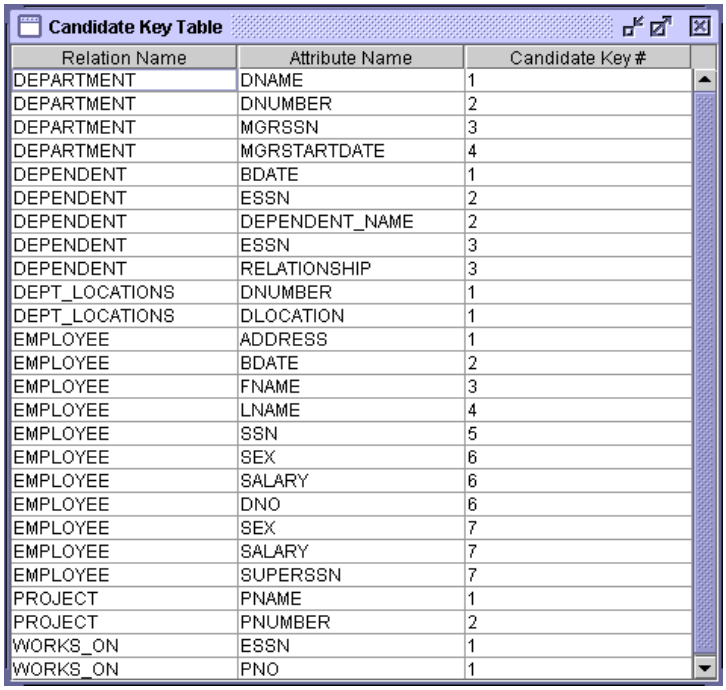
5. Remove the extra information (if it exists) from the initial ERD.

   (a) Run the "Eliminate Symmetric References" algorithm to eliminate symmetric references.

(b) Run the "Eliminate Transitive References" algorithm to eliminate transitive references.

(c) The optimized ERD is generated and displayed as output using the GUI.

6. Run the "Identify Relationships" algorithm to identify all *M:N* and n-ary relationships in the database, if any.

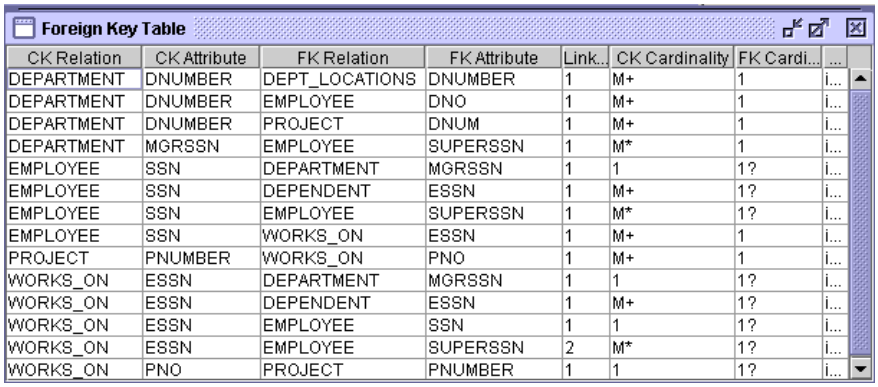


Candidate Key Table

Relation Name	Attribute Name	Candidate Key #
DEPARTMENT	DNAME	1
DEPARTMENT	DNUMBER	2
DEPARTMENT	MGRSSN	3
DEPARTMENT	MGRSTARTDATE	4
DEPENDENT	BDATE	1
DEPENDENT	ESSN	2
DEPENDENT	DEPENDENT_NAME	2
DEPENDENT	ESSN	3
DEPENDENT	RELATIONSHIP	3
DEPT_LOCATIONS	DNUMBER	1
DEPT_LOCATIONS	DLOCATION	1
EMPLOYEE	ADDRESS	1
EMPLOYEE	BDATE	2
EMPLOYEE	FNAME	3
EMPLOYEE	LNAME	4
EMPLOYEE	SSN	5
EMPLOYEE	SEX	6
EMPLOYEE	SALARY	6
EMPLOYEE	DNO	6
EMPLOYEE	SEX	7
EMPLOYEE	SALARY	7
EMPLOYEE	SUPERSSN	7
PROJECT	PNAME	1
PROJECT	PNUMBER	2
WORKS_ON	ESSN	1
WORKS_ON	PNO	1

Figure 5.4: The candidate keys table of the COMPANY legacy database

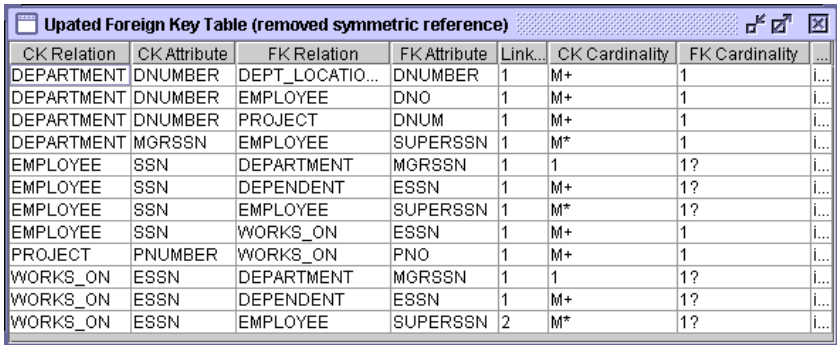
The process outlined above is detailed in Section 3.6. Here, we use the same example COMPANY database to illustrate this process in COCALEREX. The candidate keys table generated from the COMPANY database is depicted in Figure 5.4. The resulting foreign keys table is shown in Figure 5.5. The foreign keys table, after



Foreign Key Table

CK Relation	CK Attribute	FK Relation	FK Attribute	Link...	CK Cardinality	FK Cardi...	...
DEPARTMENT	DNUMBER	DEPT_LOCATIONS	DNUMBER	1	M+	1	i...
DEPARTMENT	DNUMBER	EMPLOYEE	DNO	1	M+	1	i...
DEPARTMENT	DNUMBER	PROJECT	DNUM	1	M+	1	i...
DEPARTMENT	MGRSSN	EMPLOYEE	SUPERSSN	1	M*	1	i...
EMPLOYEE	SSN	DEPARTMENT	MGRSSN	1	1	1?	i...
EMPLOYEE	SSN	DEPENDENT	ESSN	1	M+	1?	i...
EMPLOYEE	SSN	EMPLOYEE	SUPERSSN	1	M*	1?	i...
EMPLOYEE	SSN	WORKS_ON	ESSN	1	M+	1	i...
PROJECT	PNUMBER	WORKS_ON	PNO	1	M+	1	i...
WORKS_ON	ESSN	DEPARTMENT	MGRSSN	1	1	1?	i...
WORKS_ON	ESSN	DEPENDENT	ESSN	1	M+	1?	i...
WORKS_ON	ESSN	EMPLOYEE	SSN	1	1	1?	i...
WORKS_ON	ESSN	EMPLOYEE	SUPERSSN	2	M*	1?	i...
WORKS_ON	PNO	PROJECT	PNUMBER	1	1	1?	i...

Figure 5.5: The foreign keys table of the COMPANY legacy database



Upated Foreign Key Table (removed symmetric reference)

CK Relation	CK Attribute	FK Relation	FK Attribute	Link...	CK Cardinality	FK Cardinality	...
DEPARTMENT	DNUMBER	DEPT_LOCATIO...	DNUMBER	1	M+	1	i...
DEPARTMENT	DNUMBER	EMPLOYEE	DNO	1	M+	1	i...
DEPARTMENT	DNUMBER	PROJECT	DNUM	1	M+	1	i...
DEPARTMENT	MGRSSN	EMPLOYEE	SUPERSSN	1	M*	1	i...
EMPLOYEE	SSN	DEPARTMENT	MGRSSN	1	1	1?	i...
EMPLOYEE	SSN	DEPENDENT	ESSN	1	M+	1?	i...
EMPLOYEE	SSN	EMPLOYEE	SUPERSSN	1	M*	1?	i...
EMPLOYEE	SSN	WORKS_ON	ESSN	1	M+	1	i...
PROJECT	PNUMBER	WORKS_ON	PNO	1	M+	1	i...
WORKS_ON	ESSN	DEPARTMENT	MGRSSN	1	1	1?	i...
WORKS_ON	ESSN	DEPENDENT	ESSN	1	M+	1?	i...
WORKS_ON	ESSN	EMPLOYEE	SUPERSSN	2	M*	1?	i...

Figure 5.6: The foreign keys table after removing symmetric references

removing symmetric and transitive references, is separately displayed in Figure 5.6 and Figure 5.7. The primary keys table is shown in Figure 5.8. Figure 5.9 shows TAKES_ON as *M:N* relationship. Shown in Figure 5.10 are all relational tables contained in COMPANY and the corresponding ERD extracted by EELRR is provided in Figure 5.11.

Updated Foreign Key Table (removed transitive reference)

CK Relation	CK Attribute	FK Relation	FK Attribute	Link...	CK Cardinality	FK Cardinality	...
DEPARTMENT	DNUMBER	DEPT_LOCATIONS	DNUMBER	1	M+	1	i...
DEPARTMENT	DNUMBER	EMPLOYEE	DNO	1	M+	1	i...
DEPARTMENT	DNUMBER	PROJECT	DNUM	1	M+	1	i...
EMPLOYEE	SSN	DEPARTMENT	MGRSSN	1	1	1?	i...
EMPLOYEE	SSN	DEPENDENT	ESSN	1	M+	1?	i...
EMPLOYEE	SSN	EMPLOYEE	SUPERSSN	1	M*	1?	i...
EMPLOYEE	SSN	WORKS_ON	ESSN	1	M+	1	i...
PROJECT	PNUMBER	WORKS_ON	PNO	1	M+	1	i...

Figure 5.7: The foreign keys table after removing transitive references

Primary Key Table

Relation Name	Attribute Name	Primary Key #
DEPARTMENT	DNUMBER	2
DEPENDENT	ESSN	2
DEPENDENT	DEPENDENT_NAME	2
DEPT_LOCATIONS	DNUMBER	1
DEPT_LOCATIONS	DLOCATION	1
EMPLOYEE	SSN	5
PROJECT	PNUMBER	2
WORKS_ON	ESSN	1
WORKS_ON	PNO	1

Figure 5.8: The primary keys table of the COMPANY legacy database

5.4.2 EECR Module

EECR provides a function similar to EELRR but it generates the ER model from an existing database catalog that employs EELRR in case some catalog information is missing. The latter step is depicted by the dotted line in Figure 5.1. As a requirement of EECR, COCALEREX connects to a relational database by using JDBC/ODBC connection. The required foreign and primary keys information is obtained from RDBMS that support getPrimaryKeys() and getImportedKeys() functions; otherwise, the system uses EELRR to extract the not-supported information. Finally, EECR has been tested for different RDBMS include: DB2, FirstSql, and Oracle.

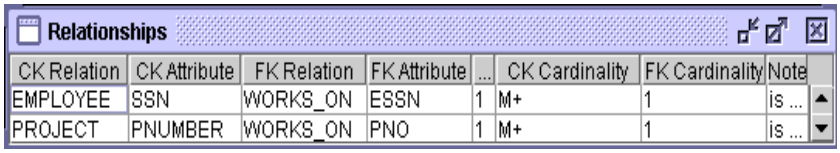


Figure 5.9: The *M:N* relationship generated from the COMPANY legacy database

EECR connects to RDBMS and calls two functions, getAllPrimaryKeys() and getAllForeignKeys() to extract all primary and foreign keys from the given relational database (see Figure 5.12). It then generates the ER model of the database. The primary keys table of COMPANY generated by EECR is presented in Figure 5.13 and the foreign keys table is shown in Figure 5.14. The ERD of COMPANY is shown in Figure 5.15. When we compare the ERD in Figure 5.15 with the one in Figure 5.11, it is clear they are identical. Therefore, this result supports the correctness of the algorithms which this work is based [Alh03].

5.4.3 ER2X Module

COCALEREX can properly handle *1:1*, *1:M*, *M:N* and n-ary relationships during the conversion into XML. It provides some functions that allow users to partially convert selected portion of a relational database into XML schema and corresponding virtual XML document. For example, users who want to view only the three relations: EMPLOYEE, DEPARTMENT, and DEPT_LOCATIONS converted into XML can do so by selecting them from the ERD displayed on the screen. The desired XML schema and documents will be displayed as output.

The ER2X module is responsible for transforming the ER model of the given database into the corresponding XML schema, which is more comprehensive and

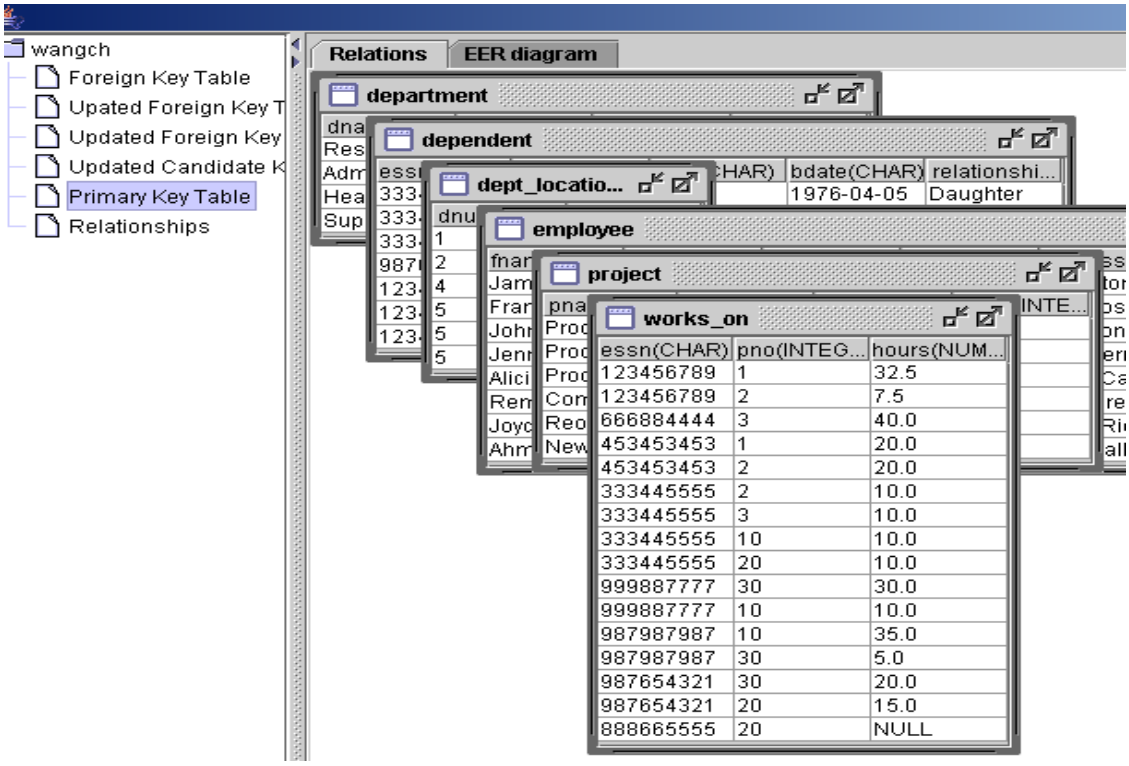


Figure 5.10: The relations generated from the COMPANY legacy database

rigorous for defining the content model of an XML document. The schema itself is an XML document and can be processed by the same tools that read the XML documents it describes. The XML schema supports rich built-in types and allows the construction of complex types based on built-in types. It also supports key, keyref and unique constraints, which are important for transforming relational schema into XML schema.

We also consider mapping all different types of relational schema constraints to the XML schema, including: primary keys (PKs), foreign keys (FKs), null/not-null, unique, *etc.* Basically, the null/not-null constraint can be easily represented by

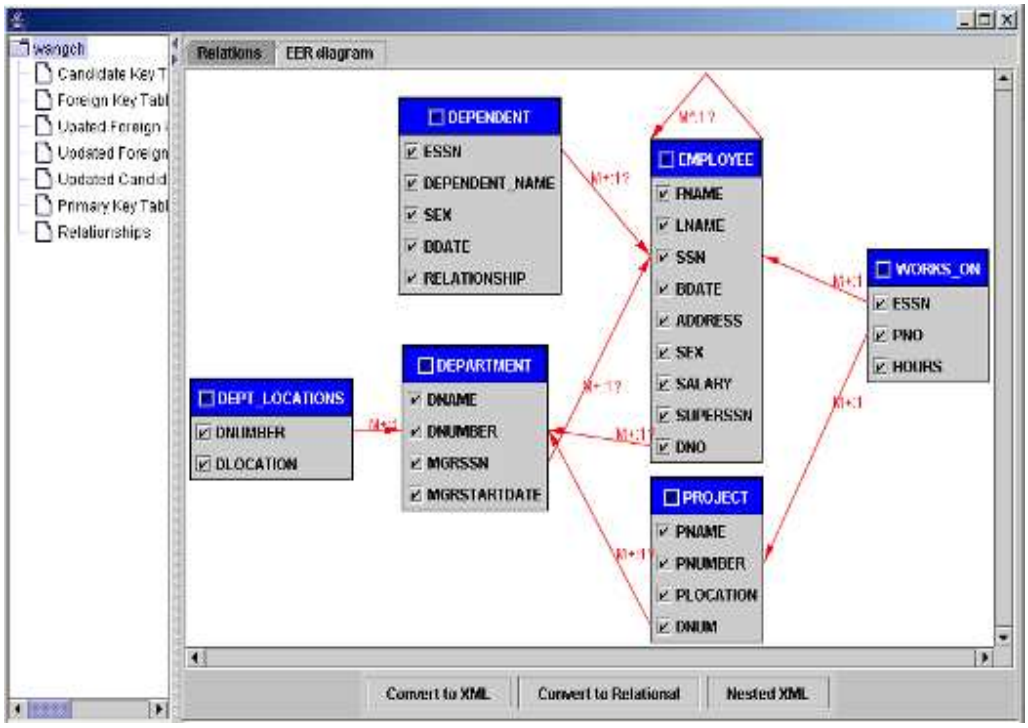


Figure 5.11: The ERD generated from the COMPANY legacy database

properly setting "minOccurs" of the XML element transformed from the relational attribute. The unique constraint can also be represented by the unique mechanism in the XML schema in a straightforward manner.

The ER2X module by default generates a flat structure of the XML schema. However, users may specify a nested structure in a way to improve the performance of querying XML documents. Chapter 3 we defines ER2FXML and ER2NXML algorithms for transforming ER model into XML in either flat or nested structure. Therefore, in this section, we only demonstrate and discuss the results generated from ER2X.

Figure 5.16 shows a flat XML schema of the example COMPANY database gener-

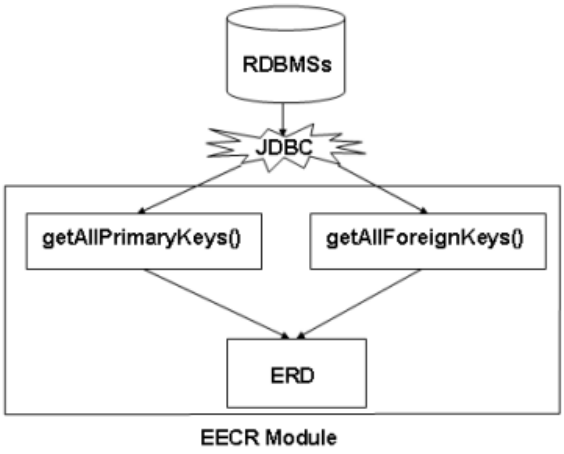


Figure 5.12: The steps diagram of EECR module

ated by employing ER2FXML algorithm. ER2X is also capable of producing nested XML schema by employing ER2NXML algorithm, given next in this section. ER2X may derive a nested XML schema if nesting is specified by users with a particular nesting request. Further, we provide an interface where users may specify the required nesting of different objects from the ERD model. This facility gives power to users who are familiar with the most commonly raised types of queries and who may specify a nested XML structure to speed up the processing of such queries. The nesting may be specified as a sequence of objects enclosed inside parentheses to indicate that the other instances (may be objects or tuples) in each tuple are nested inside the first object. For example, a nesting specified as $(E(E_1, ..., E_n))$, $n \geq 1$, means that each E_i, $i \geq 1$ is nested inside E. Further, each E_i may be either an object or a tuple. This leads to a tree rooted at E and all the other nodes/subtrees are direct children of E. Some other nesting cases have been already discussed in Chapter 4.

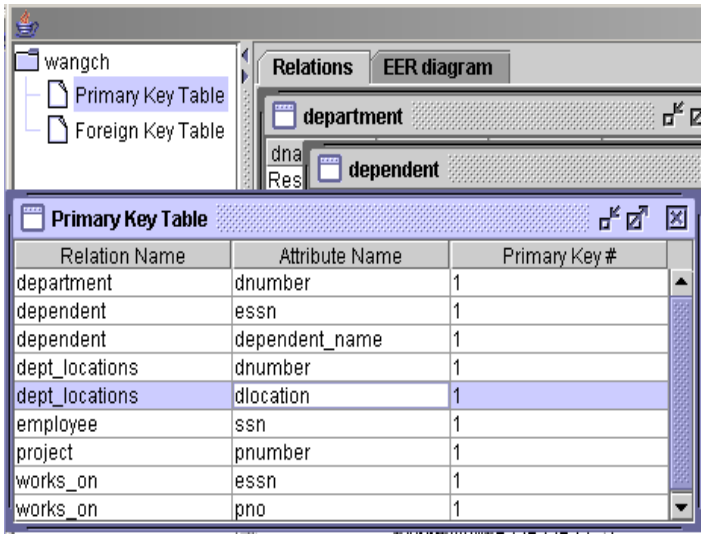


Figure 5.13: The primary keys table generated from EECR module

To illustrate the nesting process, consider the COMPANY database where users may write queries to retrieve information about employees and their projects. For this case, users can specify the nesting sequence in the Nested Input GUI as: (EMPLOYEE(WORKS_ON(PROJECT))). The ER2X module takes such nesting sequence as input and generates an output XML schema in nested structure. The element PROJECT is nested under the element WORKS_ON and then the nested element moves under the element EMPLOYEE to build two levels of nested XML structure. In Figure 5.17, WORKS_ON is *M:N* relationship which has two foreign keys: ESSN refers to EMPLOYEE and PNO refers to PROJECT. ER2X allows users to specify the objects they want to nest as shown in Figure 5.18 where users enter the nesting sequence as (EMPLOYEE(WORKS_ON(PROJECT))). After the user clicks on the "Start to Nest" button, only EMPLOYEE, WORKS_ON and PROJECT are selected. When users click on the "Convert to XML" button, the nested XML

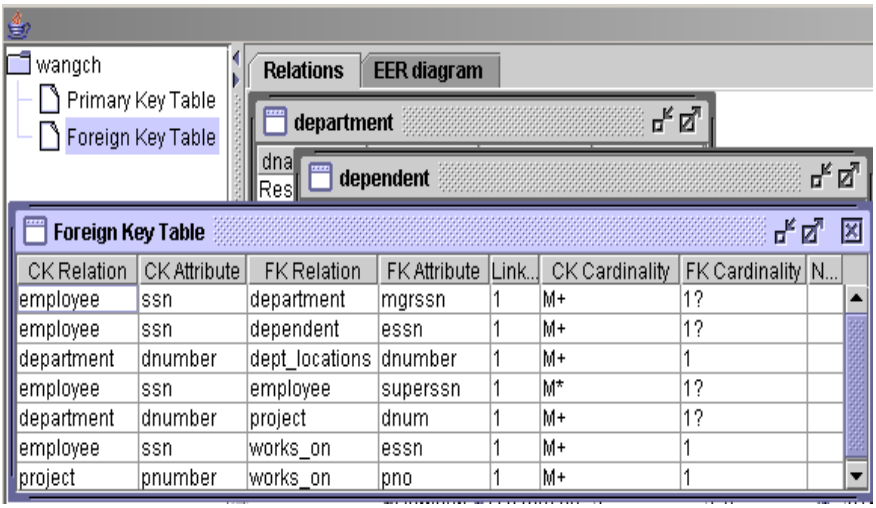


Figure 5.14: The foreign keys table generated from EECR module

schema output generated by ER2X is displayed as shown in Figure 5.19. Figure 5.20 shows the nested XML document output.

ER2X can handle binary relationships as well as n-ary relationships. In Section 3.7.2, we discussed how to nest the sample of ternary relationship in the example STUDENTS database. At this point, we use the same example to demonstrate the result generated from ER2X module. The output ERD of the STUDENTS database is shown in Figure 5.21. It contains four objects: STUDENTS, TAKES, COURSES, and PROFESSORS. TAKES represents a ternary relationship that has four attributes: SSID, CCID, PPID, and Grade. The first three attributes are foreign keys; SSID references SID, the primary of STUDENTS; CCID references CID, the primary key of COURSES; and PPID references PID, the primary key of PROFESSORS. There are a few different ways to nest a ternary relationship based on user's decision. For example, users want to frequently query on "all students are

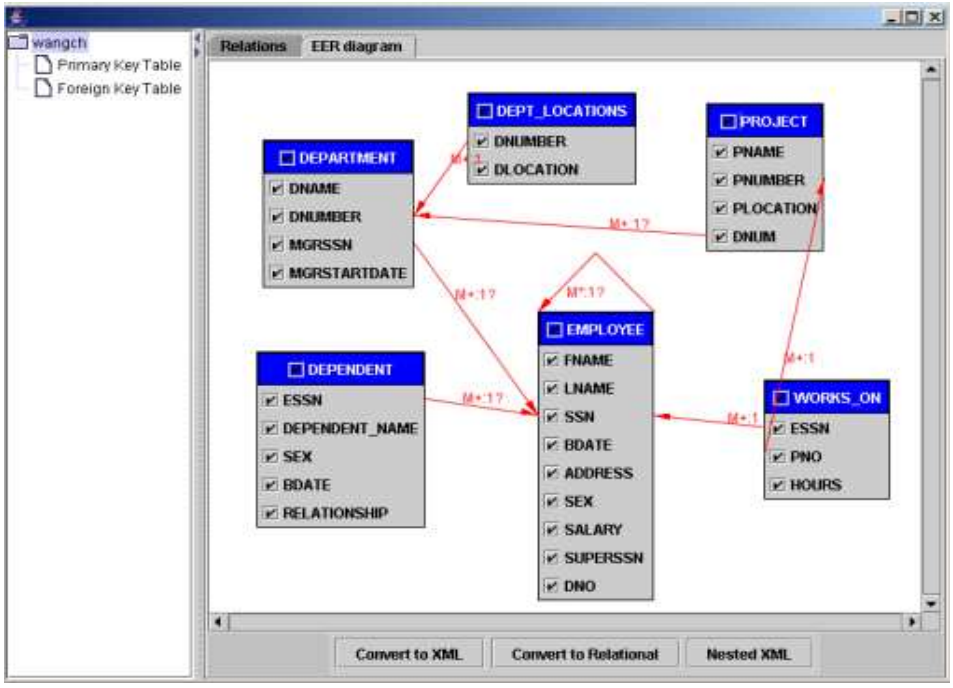


Figure 5.15: The ER diagram of catalog-based COMPANY database

taking courses taught by certain professor". Therefore, users may specify the nesting sequence as: (STUDENTS(TAKES(COURSES, PROFESSORS))) as shown in Figure 5.22. ER2X generates the nested XML schema output shown in Figure 5.23 and the nested XML document output is displayed in Figure 5.24. COURSES and PROFESSORS are nested under TAKES, which is nested under STUDENTS. The COURSES and PROFESSORS elements can be moved to under TAKES and STUDENTS if every COURSES and PROFESSORS are related to a particular student.

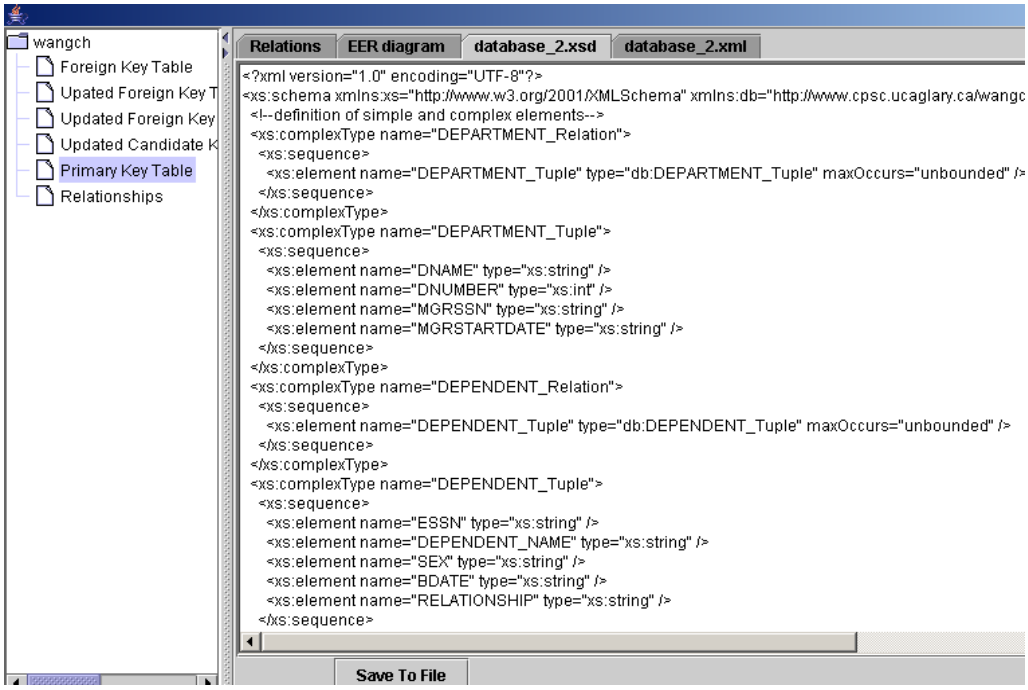


Figure 5.16: The flat XML schema output generated from ER2X module

5.4.4 X2R Module

After an XML document is generated from ER2X it must be restored to a relational database format. The X2R module's goal is to handle this transformation from XML to a relational schema. It is capable of producing both flat and nested XML structure. The transformation process implements the XML2R algorithm described in Section 4.3. The functions implemented in X2R module are depicted in Figure 5.25.

The process of transforming an XML schema file to a relational schema can be divided into two steps:

Step 1: Generate a relational schema file from the input XML schema file.

Step 2: Translate the resulted relational schema file to a corresponding SQL script file

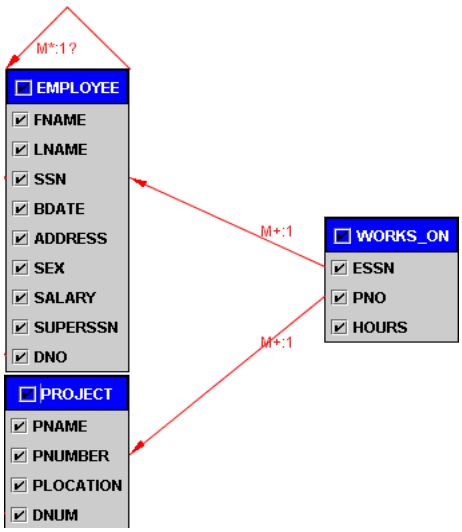


Figure 5.17: Many-to-many relationship

and then runs the script file to create a database in RDBMS.

We have implemented the first step in our current COCALEREX system. The second step is considered as a possible future work. The output relational schema for the COMPANY XML schema file produced by X2R module is shown in Figure 5.26 and the complete relational schema can be found in Appendix B.

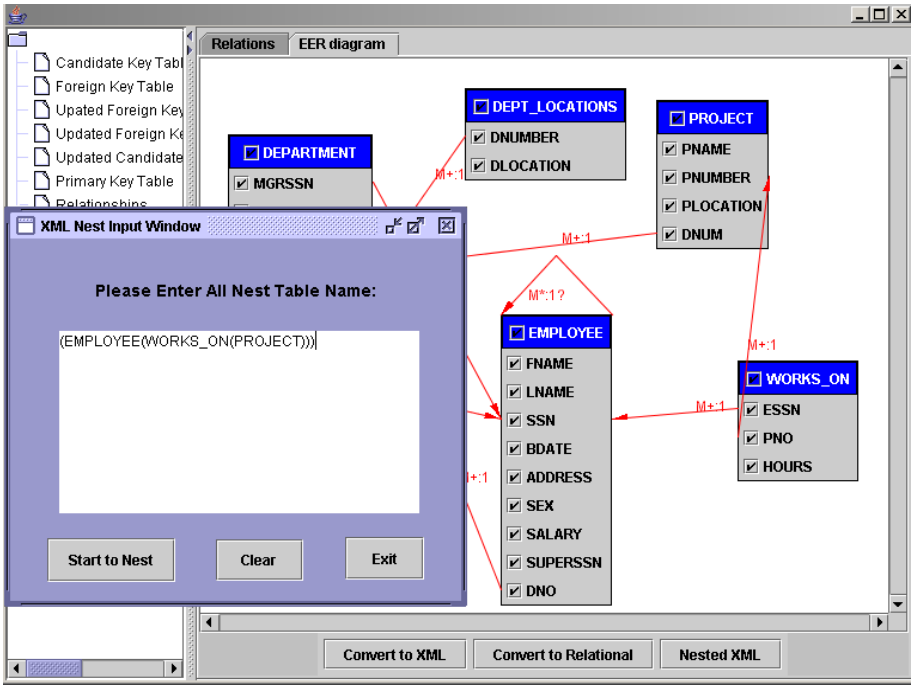


Figure 5.18: The nesting input GUI

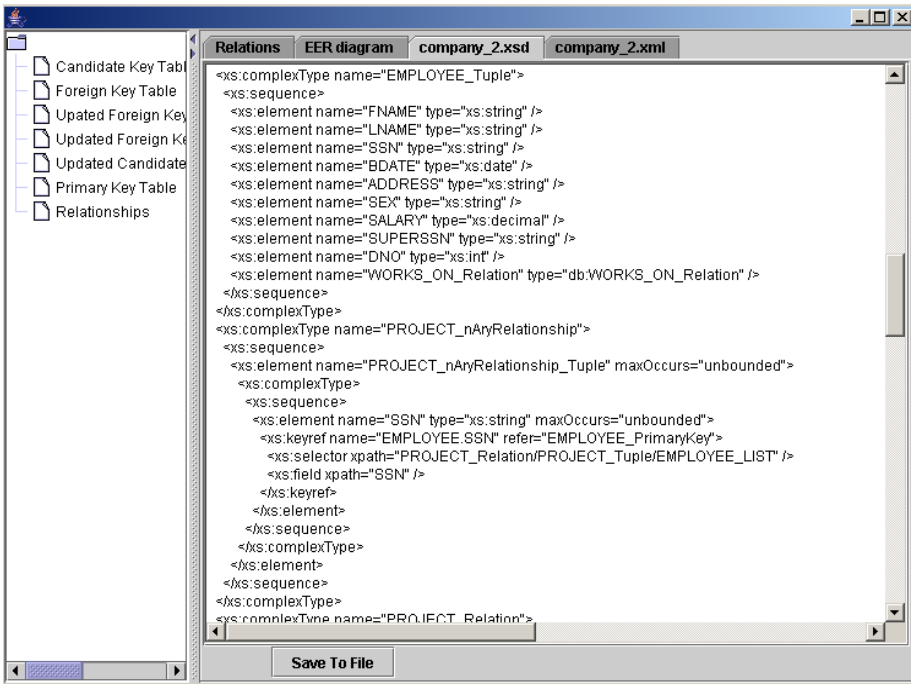


Figure 5.19: EMPLOYEE, WORKS_ON, and PROJECT nested XML schema output

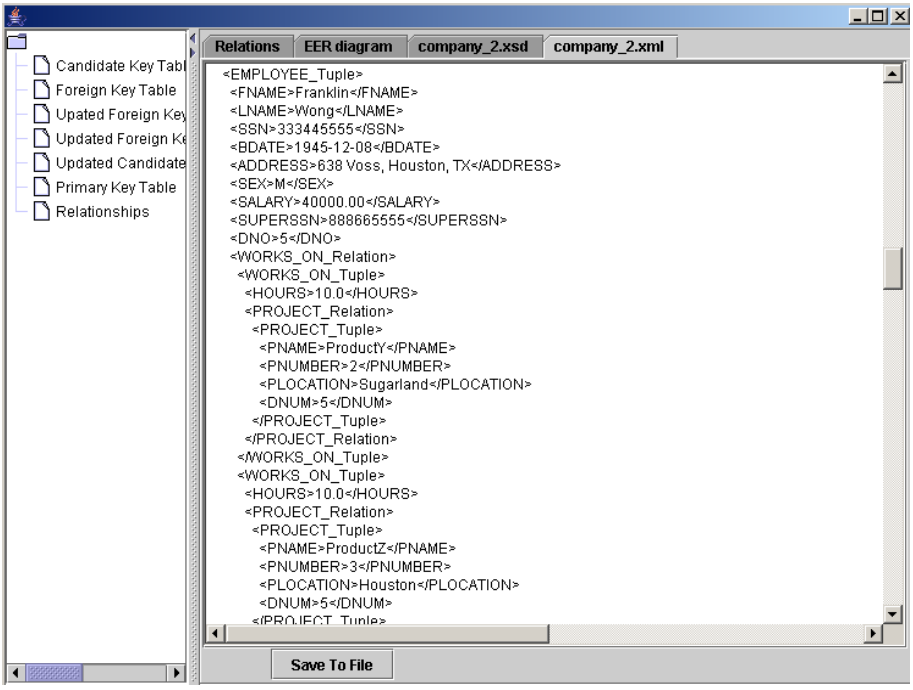


Figure 5.20: EMPLOYEE, WORKS_ON, and PROJECT nested XML document output

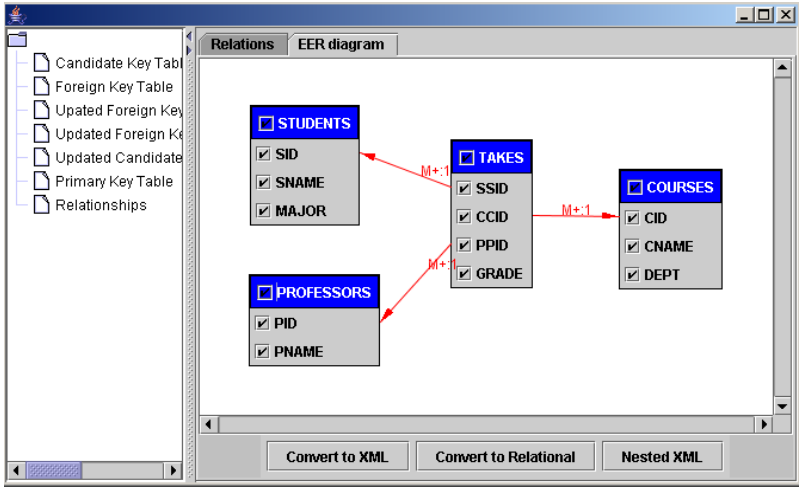


Figure 5.21: The ERD of the STUDENTS database

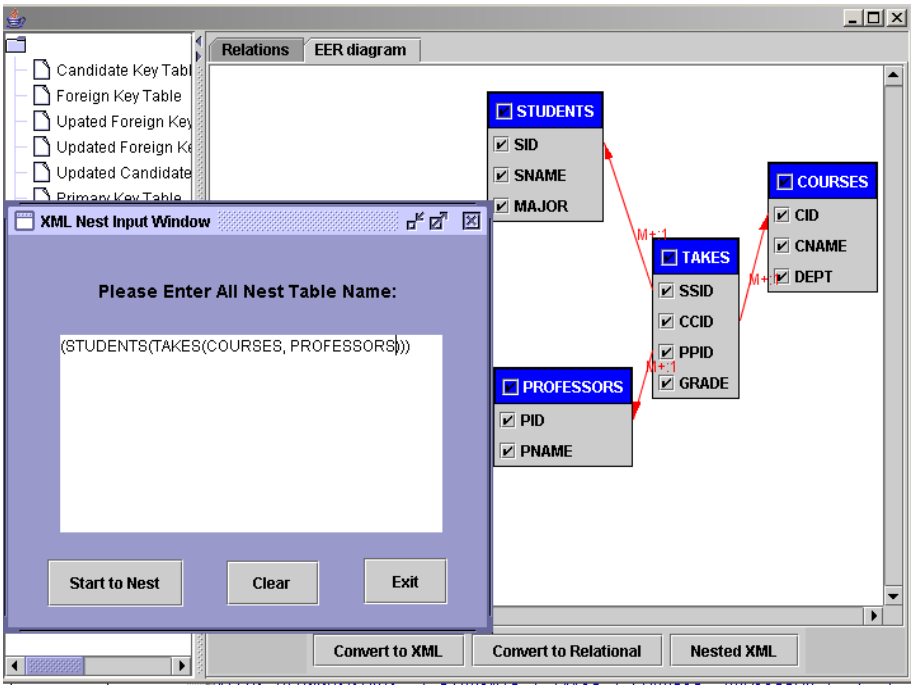


Figure 5.22: The nesting input sequence of the STUDENTS database

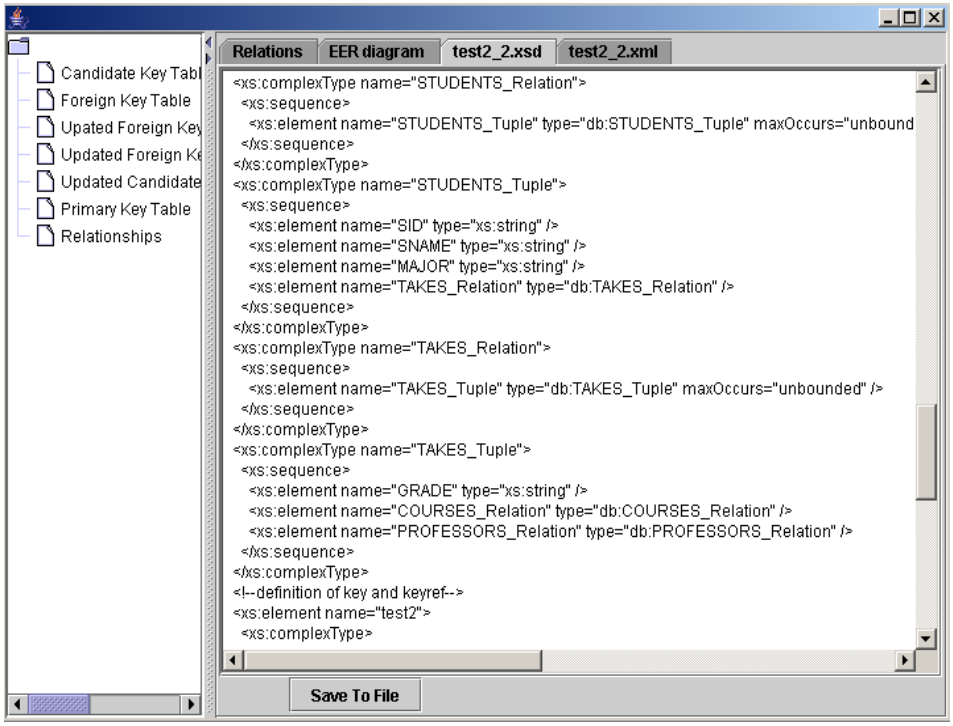


Figure 5.23: The output of ternary nesting XML schema

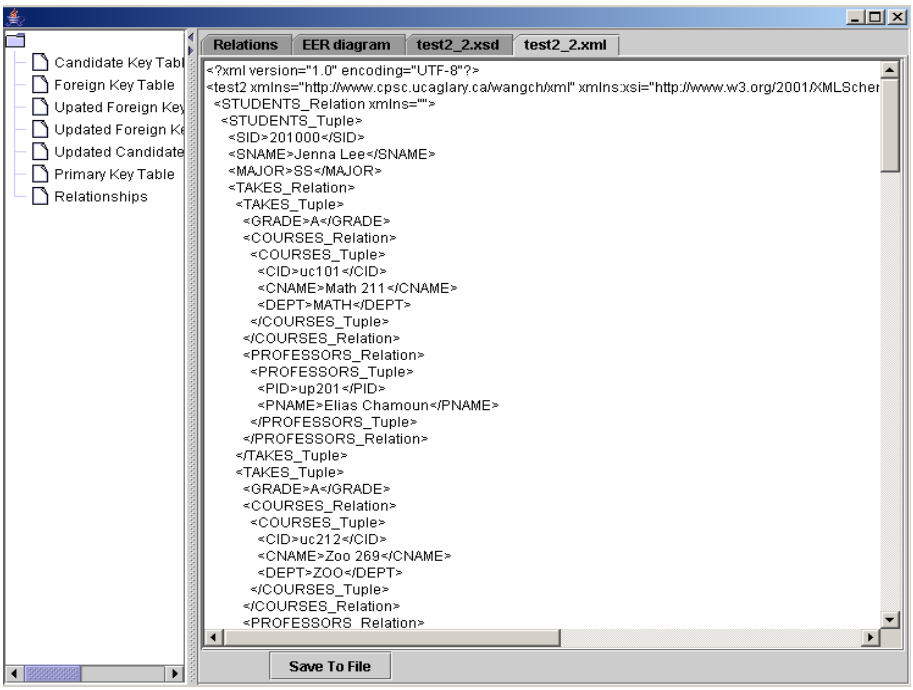


Figure 5.24: The output of ternary nesting XML document

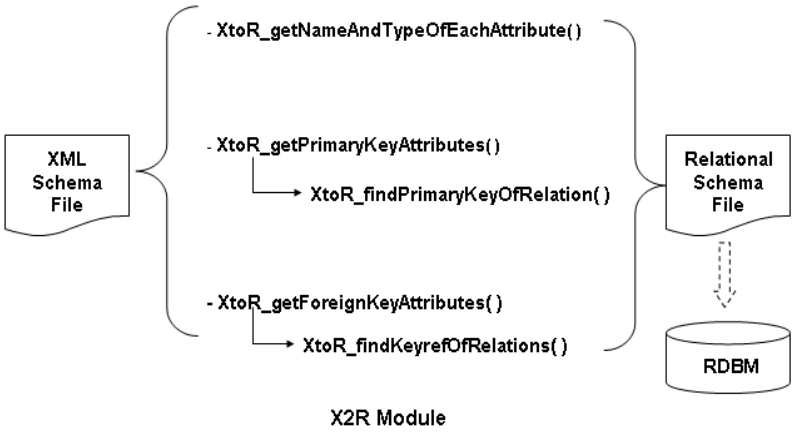


Figure 5.25: The functions implemented in X2R module

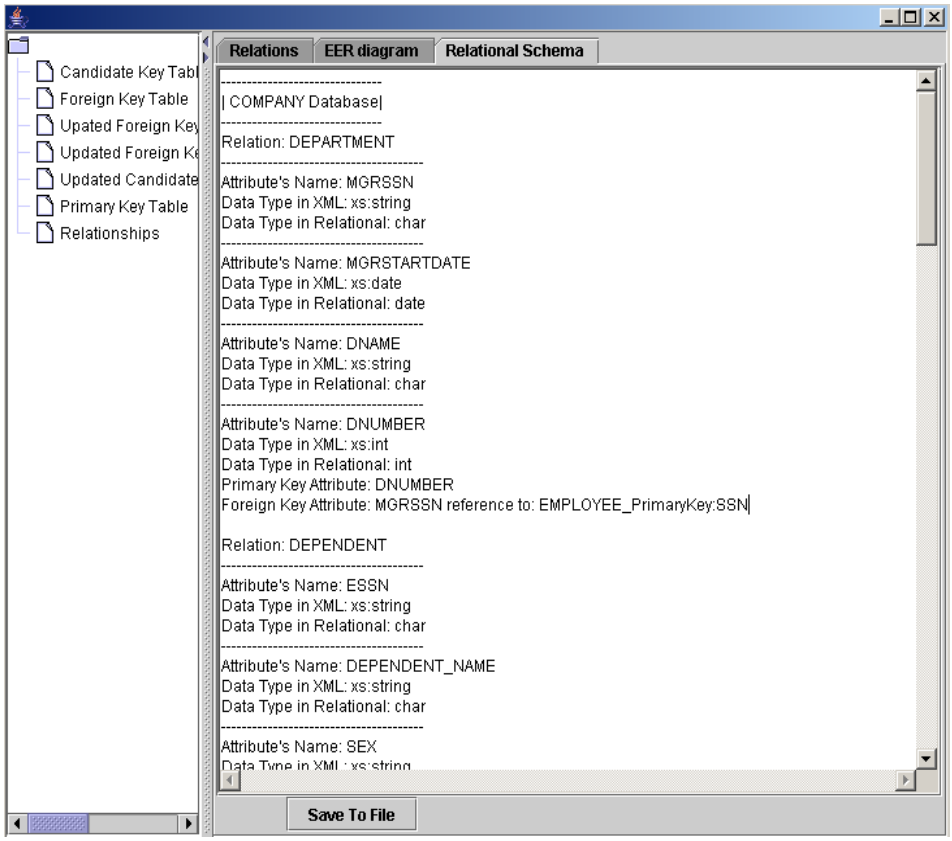


Figure 5.26: The relational schema output generated from X2R

Chapter 6

Conclusions and Future Work

In this thesis, the COCALEREX is presented as a user-friendly engine for converting catalog-based as well as legacy relational databases into corresponding XML schemas and documents. We introduced the algorithms developed for the transformation between relational and XML in Chapters 3 and 4. These algorithms handle the mapping of the semantic constraints as much as possible during the transformation process. Further, COCALEREX can properly and equally deal with *1:1*, *1:M*, *M:N*, and n-ary relationships in the conversion into XML schema. It provides essential functionalities that allow users to partially convert selected portion of a relational database into XML schema and corresponding virtual XML documents. The ER2X module by default generates a flat structure of the XML schema. However, users may specify a nested structure in some ways to improve the performance of querying XML documents. In Chapter 5, we tested COCALEREX on an example COMPANY relational database. We compared the obtained ERD generated by COCALEREX to Figure 6.7 in [EN94] and found them to be identical. This supports and demonstrates the correctness of the approach for reconstructing ER model from the given database. It also proves that COCALEREX is highly applicable, effective and efficient. We also tested the proposed approach on the SAMPLE DB2 database, Mysql STUDENTS and LIBRARY databases, as well as on the MS Access North-Wind database. We tested the conversion of these databases by considering their corresponding catalogs; and we tested them as legacy databases, i.e., by neglecting

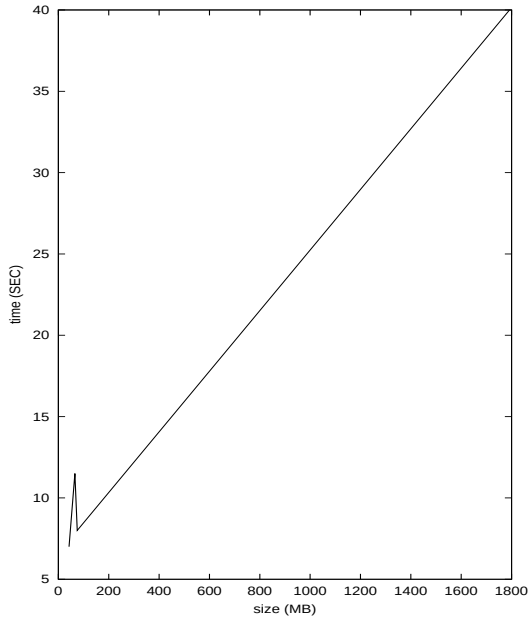


Figure 6.1: The comparison between running time and database size

their catalogs and hence derived the required catalog information by analyzing the content. Testing these databases provided rough feedback and estimate about the relationship between the size of a database and the time required to convert it into XML. The result of the comparison between the run time required and the database size is plotted in Figure 6.1. It roughly shows that as the size of a database increases, the run time to generate ERD from a database increases as well. Comparing NorthWind to other databases, it takes much longer time than others. The reason is that the NorthWind database contains 8 tables, a large number of attributes in some of the tables, and huge number of records in each table. It has much bigger size than the other databases. Therefore, when COCALEREX analyzes NorthWind as a legacy databases it takes longer to derive the information necessary to produce

the ER model. Particularly, deciding on candidate and foreign keys is time consuming because it requires analyzing almost all elements in the powerset of the set of attributes in each relation. For example, if a table contains $n = 12$ attributes, there are $2^n - 1 = 2^{12} - 1 = 4095$, elements in the power set of this table. Of course, not all of them are analyzed in general because we consider some heuristics inspired from the definition of candidate key. Therefore, the job of reconstructing an ERD from a legacy database is undoubtedly heavy and tedious, especially for a large real application. Users could be relieved from this heavy load by using COCALEREX. On the other hand, the users' knowledge could also be involved in this system. However, compared to reconstructing an ERD from scratch, it is almost impossible to manually reengineer a given relational database into XML. As a result, the human beings' mental workload is greatly reduced with COCALEREX. Further, COCALEREX is a very useful tool for designers to update and redesign existing databases. Finally, COCALEREX provides a user-friendly graphical interface which makes it attractive even to naive users to try the conversion process. It gives users a direct visualization for each phase of the process.

As the future work is concerned, COCALEREX could be expanded in the following directions. Query Translator may be added on the top of COCALEREX. It will run query transforming algorithm to translate XQuery queries to corresponding SQL queries. Also, more research may be conducted to improve the performance of the conversion from legacy relational databases to XML. It may be expanded into a Web-based application to be used over the Internet. Finally, the implementation of COCALEREX may be adjusted to minimize keyboard usage and bypass JDBC utilized in the conversion from catalog-based relational databases to XML.

Bibliography

[Alh03] R. Alhajj. Extracting the extended entity-relationship model from a legacy relational database. In *Information Systems*, volume 28, pages 591–618. Elsevier Science Ltd., 2003.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. In *International Journal on Digital Libraries*, volume 1, pages 68–88, Stanford, CA, 1997.

[BCFK] G. Booch, M. Christerson, M. Fuchs, and J. Koistinen. UML for XML schema mapping specification. `http://www.rational.com/media/uml/uml_xmlschema33.pdf`.

[BCN92] C. Batini, S. Ceri, and S. B. Navathe. Conceptual database design: An entity-relationship approach. In *Int'l Conference, on Conceptual Modeling (ER)*, 1992.

[BKKM00] S. Banerjee, V. Krishnamuthy, M. Krishnamrasad, and R. Murthy. Oracle8i: The XML Enabled Data Management System. In *16th IEEE International Conference on Data Engineering (ICDE)*, pages 561–569, Washington, DC, USA, 2000. IEEE Computer Society.

[BL01] A. Bonifati and D. Lee. Technical survey of XML schema and query languages. Technical report, UCLA Computer Science Dept., 2001.

[Bou03a] R. Bourret. XML and Databases.

http://www.rpbourret.com/xml/XMLAndDatabases.htm, April 2003.

[Bou03b] R. Bourret. Data transfer strategies: Transferring data between XML documents and relational databases. http://www.rpbourret.com/xml/DataTransfer.htm, March 2003.

[CAZ03] A. B. Chaudhri, Rashid A., and R. Zicari. *XML Data Management - Native XML and XML-Enabled Database Systems.* Addison-Wesley, 2003.

[CFI+00] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Schanmugasundaram, E. Shekita, and S. Subramanian. XPERATO: Publishing object-relational data as XML. In *Int'l Workshop on the Web and Databases (WebDB)*, pages 105–110, Dallas, TX, USA, May 2000.

[CSF] R. Conrad, D. Scheffner, and J. C. Freytag. XML conceptual modeling using UML. In *19th Intern. Conf. on Conceptual Modeling*, pages 558–571, Salt Lake City, Utah, USA. Springer.

[CSK+00] M. Carey, J. Schanmugasundaram, J. Kiernan, E. Shekita, and S. Subramanian. XPERATO: A middleware for publishing object-relational data as XML documents. In *Proc. of the 26th Intern. Conf. on VLDB*, pages 646–648, Cairo, Egypt, 2000.

[CX00] J. Cheng and J. Xu. XML and DB2. In *Proc. of the* 16th *IEEE Int. Conf. on Data Engineering*, pages 569–573, SanDiego, USA, 2000.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proc. of 8th Intern. World Wide Web Conference (WWWC)*, pages 77–91, Toronto, Canada, 1999.

[DFS99] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of the Int. ACM SIGMOD Conf.*, pages 431–442, Philadephia, USA, 1999.

[DOM03] Document Object Model DOM. `http://www.w3.org/DOM/`, June 2003.

[DT03] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *Proc. of 29th Intern. Conf. on VLDB*, pages 201–212, Berlin, Germany, September 2003. Morgan Kaufmann.

[DTD02] Document Type Definition DTD. `http://www.w3schools.com/DTD/`, November 2002.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 1994.

[FK99] D. Florescu and D. Kossman. Storing and querying XML data using an RDBMS. In *IEEE Data Engineering Bulletin*, volume 22, pages 27–34, September 1999.

[FMS01] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *Proc. of ACM SIGMOD*, pages 103–114, Santa Barbara, CA, USA, 2001.

[FMST01] M. Fernandez, A. Morishima, D. Suciu, and W. Tan. Publishing relational data in XML: The SilkRoute approach. In *IEEE Data Engineering Bulletin*, volume 24, pages 12–19, 2001.

[FPB01] J. Fong, F. Pang, and C. Bloor. Converting relational database into XML document. In *12th International Conference on Data Engineering*, pages 61–65. IEEE Computer Society Press, 2001.

[FTS00] M. Fernandez, W. Tan, and D. Suciu. SilkRoute: Trading between relational and XML. In *9th Int'l World Wide Web Conf. (WWW)*, volume 33, pages 723–745, Amsterdam, Netherlands, 2000.

[GA02] G. Guardalben and S. Atre. Integrating XML and relational database technologies: A position paper. In *White Paper*, October 2002.

[Gra02] M. Graves. *Designing XML Databases*. Prentice Hall PTR, 2002.

[JAKCL02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, and L. V. S. Lakshmanan. TIMBER: A native XML database. In *The VLDB Journal The International Journal on Very Large Databases*, volume 11, AT&T Labs Research, Florham Park, NJ, USA, December 2002.

[KKRSR00] G. Kappel, E. Kapsammer, S. Rausch-Schott, and W. Retschitzegger. X-Ray - towards integrating XML and relational database systems. In *19th Int'l Conference, on Conceptual Modeling (ER)*, pages 339–353, Salt Lake City, UT, 2000.

[KL01] C. Kleiner and U. W. Lipeck. Automatic generation of XML DTDs from

conceptual database schemas. In *Workshop of the Annual Conference of the German and Austrian Computer Societies in Web-Databases*, Hannover, Germany, 2001. University of Hannover, Institute of Information.

[Lan03] XML Query (XQuery) 1.0: An XML Query Language. `http://www.w3.org/XML/Query`, January 2003.

[LC00] D. Lee and W. W. Chu. Constraints-preserving transformation from XML document type definition to relational schema. In *19th International Conference on Conceptual Modeling (ER)*, pages 339–353, Salt Lake City, Utah, USA, October 2000. Springer.

[LC01] D. Lee and W. W. Chu. CPI: Constraints-Preserving inlining algorithm for mapping XML DTD to relational schema. In *J. Data & Knowledge Engineering (DKE)*, volume 39, pages 3–25, Amsterdam, Netherlands, October 2001. Elsevier Science Publishers B. V.

[LLG] C. Liu, J. Liu, and M. Guo. On transformation to redundancy free xml schema from relational database schema. In *5th Asian-Pacific Web Conference (APWeb), on XML and Database Design.*

[LMCC01] D. Lee, M. Mani, F. Chiu, and W. W. Chu. Nesting based relational-to-XML schema translation. In *Int'l Workshop on the Web and Databases (WebDB)*, pages 61–66, Santa Barbara, CA, USA, May 2001.

[LMCC02] D. Lee, M. Mani, F. Chiu, and W. W. Chu. Net & CoT: Translating relational schemas to XML schemas using semantic constraints. In *11th*

ACM Int'l Conf. on Information and Knowledge Management (CIKM), pages 282–291, McLean, VA, USA, November 2002.

[LVLG03] C. Liu, M. W. Vincent, J. Liu, and M. Guo. A virtual XML database engine for relational databases. Springer-Verlag, 2003.

[MFK01] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *Proc. of 27th Int'l Conf. on VLDB*, pages 241–250, Rome, Italy, 2001. Morgan Kaufmann.

[MLM01a] M. Mani, D. Lee, and R. R. Muntz. Taxonomy of XML schema language using formal language theory. In Philip S. Yu and Arbee S. P. Chen, editors, *Extreme Markup Languages*, Montreal, Canada, August 2001. IEEE Computer Society Press.

[MLM01b] M. Mani, D. Lee, and R. R. Muntz. Semantic data modeling using XML schemas. In *20th Int'l Conf. on Conceptual Modeling (ER)*, pages 149–163, Yokohama, Japan, November 2001. Springer.

[Mur03] M. Murata. RELAX (REgular LAnguage) description for XML. `http://www.xml.gr.jp/relax/hedge_nice.html`, July 2003.

[NDMC03] J. Naughton, D. DeWitt, D. Maier, and J. Chen. The Niagara Internet Query System. `http://www.cs.wisc.edu/niagara`, January 2003.

[Sch02] XML Schema. `http://www.w3.org/TR/xmlschema-0/`, November 2002.

[SLR98] D. Schach, J. Lapp, and J. Robie. Querying and transforming XML. In *WWW The Query Language Workshop (QL)*, Cambridge, MA, USA, December 1998.

[SOA03] SOAP. `http://www.w3schools.com/SOAP/`, July 2003.

[SSB+01] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *The VLDB Journal*, 10(2–3):133–154, 2001.

[SSK+01] J. Shanmugasundaram, E. Shekita, J. Kiernan, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proc. of 27th VLDB*, pages 261–270, Rome, Italy, September 2001.

[ST01] A. Salminen and F. W. Tompa. Requirements for XML document database systems. In *Proceedings of the 2001 ACM Symposium on Document Engineering*, pages 85–94, Atlanta, Georgia, USA, 2001.

[Sta03] K. Staken. Apache Xindice. `http://xml.apache.org/xindice/`, February 2003.

[STH+99] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of 25th VLDB*, pages 302–314, Edinburgh, Scotland, 1999.

[SXM02] Socrates XML: Choosing an XML database solution. In *White Paper*, 2002.

[TAM03] Tamino: The Native XML Management System, http://www.softwareag.com/tamino, March 2003.

[TUK03] The Tukwila Data Integration System, http://data.cs.washington.edu/integration, March 2003.

[Wil01] H. Williamson. *XML: The Complete Reference*. Osborne/McGraw-Hill, 2001.

[WLAB04] C. Wang, A. Lo, R. Alhajj, and K. Barker. Converting legacy relational database into XML database through reverse engineering. In *6th International Conference on Enterprise Information Systems (ICEIS)*, pages 216–221, Porto, Portugal, April 2004.

[XML02] Extensible Markup Language XML 1.0. http://www.w3.org/TR/REC-xml/, November 2002.

[XML03] XML-QL: A Query Language for XML. http://www.w3.org/TR/NOTE-xml-ql/, January 2003.

[XPa03] XML Path Language XPath. http://www.w3.org/TR/xpath20/, June 2003.

Appendix A

The Flat XML Schema Output for the COMPANY Database

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:db="http://www.cpsc.ucaglary.ca/wangch/xml"
targetNamespace="http://www.cpsc.ucaglary.ca/wangch/xml"
elementFormDefault="qualified">
  <!--definition of simple and complex elements-->
  <xs:complexType name="DEPARTMENT_Relation">
    <xs:sequence>
      <xs:element name="DEPARTMENT_Tuple" type="db:DEPARTMENT_Tuple"
                  maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="DEPARTMENT_Tuple">
    <xs:sequence>
      <xs:element name="DNAME" type="xs:string" />
      <xs:element name="DNUMBER" type="xs:int" />
      <xs:element name="MGRSSN" type="xs:string" />
      <xs:element name="MGRSTARTDATE" type="xs:string" />
    </xs:sequence>
```

```
</xs:complexType>
<xs:complexType name="DEPENDENT_Relation">
  <xs:sequence>
    <xs:element name="DEPENDENT_Tuple" type="db:DEPENDENT_Tuple"
                maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPENDENT_Tuple">
  <xs:sequence>
    <xs:element name="ESSN" type="xs:string" />
    <xs:element name="DEPENDENT_NAME" type="xs:string" />
    <xs:element name="SEX" type="xs:string" />
    <xs:element name="BDATE" type="xs:string" />
    <xs:element name="RELATIONSHIP" type="xs:string" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPT_LOCATIONS_Relation">
  <xs:sequence>
    <xs:element name="DEPT_LOCATIONS_Tuple" type="db:DEPT_LOCATIONS_Tuple"
                maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DEPT_LOCATIONS_Tuple">
  <xs:sequence>
    <xs:element name="DNUMBER" type="xs:int" />
```

```
    <xs:element name="DLOCATION" type="xs:string" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="EMPLOYEE_Relation">
  <xs:sequence>
    <xs:element name="EMPLOYEE_Tuple" type="db:EMPLOYEE_Tuple"
                maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="EMPLOYEE_Tuple">
  <xs:sequence>
    <xs:element name="FNAME" type="xs:string" />
    <xs:element name="LNAME" type="xs:string" />
    <xs:element name="SSN" type="xs:string" />
    <xs:element name="BDATE" type="xs:string" />
    <xs:element name="ADDRESS" type="xs:string" />
    <xs:element name="SEX" type="xs:string" />
    <xs:element name="SALARY" type="xs:decimal" />
    <xs:element name="SUPERSSN" type="xs:string" />
    <xs:element name="DNO" type="xs:int" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PROJECT_Relation">
  <xs:sequence>
    <xs:element name="PROJECT_Tuple" type="db:PROJECT_Tuple"
```

```
                maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PROJECT_Tuple">
  <xs:sequence>
    <xs:element name="PNAME" type="xs:string" />
    <xs:element name="PNUMBER" type="xs:int" />
    <xs:element name="PLOCATION" type="xs:string" />
    <xs:element name="DNUM" type="xs:int" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="WORKS_ON_Relation">
  <xs:sequence>
    <xs:element name="WORKS_ON_Tuple" type="db:WORKS_ON_Tuple"
                maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="WORKS_ON_Tuple">
  <xs:sequence>
    <xs:element name="ESSN" type="xs:string" />
    <xs:element name="PNO" type="xs:int" />
    <xs:element name="HOURS" type="xs:decimal" />
  </xs:sequence>
</xs:complexType>
<!--definition of key and keyref-->
```

```
<xs:element name="database">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="DEPARTMENT_Relation"
                  type="db:DEPARTMENT_Relation" />
      <xs:element name="DEPENDENT_Relation"
                  type="db:DEPENDENT_Relation" />
      <xs:element name="DEPT_LOCATIONS_Relation"
                  type="db:DEPT_LOCATIONS_Relation" />
      <xs:element name="EMPLOYEE_Relation"
                  type="db:EMPLOYEE_Relation" />
      <xs:element name="PROJECT_Relation"
                  type="db:PROJECT_Relation" />
      <xs:element name="WORKS_ON_Relation"
                  type="db:WORKS_ON_Relation" />
    </xs:sequence>
  </xs:complexType>
  <xs:key name="DEPARTMENT_PrimaryKey">
    <xs:selector xpath="db:DEPARTMENT_Relation/db:DEPARTMENT_Tuple" />
    <xs:field xpath="db:dnumber" />
  </xs:key>
  <xs:key name="DEPENDENT_PrimaryKey">
    <xs:selector xpath="db:DEPENDENT_Relation/db:DEPENDENT_Tuple" />
    <xs:field xpath="db:essn" />
    <xs:field xpath="db:dependent_name" />
```

```
</xs:key>
<xs:key name="DEPT_LOCATIONS_PrimaryKey">
  <xs:selector xpath="db:DEPT_LOCATIONS_Relation/db:DEPT_LOCATIONS_Tuple" />
  <xs:field xpath="db:dnumber" />
  <xs:field xpath="db:dlocation" />
</xs:key>
<xs:key name="EMPLOYEE_PrimaryKey">
  <xs:selector xpath="db:EMPLOYEE_Relation/db:EMPLOYEE_Tuple" />
  <xs:field xpath="db:ssn" />
</xs:key>
<xs:key name="PROJECT_PrimaryKey">
  <xs:selector xpath="db:PROJECT_Relation/db:PROJECT_Tuple" />
  <xs:field xpath="db:pnumber" />
</xs:key>
<xs:key name="WORKS_ON_PrimaryKey">
  <xs:selector xpath="db:WORKS_ON_Relation/db:WORKS_ON_Tuple" />
  <xs:field xpath="db:essn" />
  <xs:field xpath="db:pno" />
</xs:key>
<!--keyref-->
<xs:keyref name="DEPARTMENT.mgrssn" refer="db:EMPLOYEE_PrimaryKey">
  <xs:selector xpath="db:department_Relation/db:department_Tuple" />
  <xs:field xpath="mgrssn" />
</xs:keyref>
<xs:keyref name="DEPENDENT.essn" refer="db:EMPLOYEE_PrimaryKey">
```

```
      <xs:selector xpath="db:dependent_Relation/db:dependent_Tuple" />
      <xs:field xpath="essn" />
    </xs:keyref>
    <xs:keyref name="DEPT_LOCATIONS.dnumber" refer="db:DEPARTMENT_PrimaryKey">
      <xs:selector xpath="db:dept_locations_Relation/db:dept_locations_Tuple" />
      <xs:field xpath="dnumber" />
    </xs:keyref>
    <xs:keyref name="EMPLOYEE.superssn" refer="db:EMPLOYEE_PrimaryKey">
      <xs:selector xpath="db:employee_Relation/db:employee_Tuple" />
      <xs:field xpath="superssn" />
    </xs:keyref>
    <xs:keyref name="PROJECT.dnum" refer="db:DEPARTMENT_PrimaryKey">
      <xs:selector xpath="db:project_Relation/db:project_Tuple" />
      <xs:field xpath="dnum" />
    </xs:keyref>
    <xs:keyref name="WORKS_ON.essn" refer="db:EMPLOYEE_PrimaryKey">
      <xs:selector xpath="db:works_on_Relation/db:works_on_Tuple" />
      <xs:field xpath="essn" />
    </xs:keyref>
    <xs:keyref name="WORKS_ON.pno" refer="db:PROJECT_PrimaryKey">
      <xs:selector xpath="db:works_on_Relation/db:works_on_Tuple" />
      <xs:field xpath="pno" />
    </xs:keyref>
  </xs:element>
</xs:schema>
```

Appendix B

The Relational Schema Generated from XML File

```
------------------
|COMPANY Database|
------------------
Relation: DEPARTMENT
---------------------------------------
    Attribute's Name: DNAME
              Require: unique
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: DNUMBER
    Data Type in XML: xs:int
    Data Type in Relational: int
---------------------------------------
    Attribute's Name: MGRSSN
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: MGRSTARTDATE
    Data Type in XML: xs:string
    Data Type in Relational: char
```

```
---------------------------------------

    Primary Key Attribute: dnumber

    Foreign Key Attribute: mgrssn

    reference to: EMPLOYEE_PrimaryKey

---------------------------------------

---------------------------------------

Relation: DEPENDENT

---------------------------------------

    Attribute's Name: ESSN

    Data Type in XML: xs:string

    Data Type in Relational: char

---------------------------------------

    Attribute's Name: DEPENDENT_NAME

    Data Type in XML: xs:string

    Data Type in Relational: char

---------------------------------------

    Attribute's Name: SEX

    Data Type in XML: xs:string

    Data Type in Relational: char

---------------------------------------

    Attribute's Name: BDATE

    Data Type in XML: xs:string

    Data Type in Relational: char

---------------------------------------

    Attribute's Name: RELATIONSHIP
```

```
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Primary Key Attribute: essn
    Primary Key Attribute: dependent_name
    Foreign Key Attribute: essn
    reference to: EMPLOYEE_PrimaryKey
---------------------------------------
---------------------------------------
Relation: DEPT_LOCATIONS
---------------------------------------
    Attribute's Name: DNUMBER
    Data Type in XML: xs:int
    Data Type in Relational: int
---------------------------------------
    Attribute's Name: DLOCATION
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Primary Key Attribute: dnumber
    Primary Key Attribute: dlocation
    Foreign Key Attribute: dnumber
    reference to: DEPARTMENT_PrimaryKey
---------------------------------------
---------------------------------------
```

```
Relation: EMPLOYEE
---------------------------------------
    Attribute's Name: FNAME
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: LNAME
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: SSN
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: BDATE
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: ADDRESS
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: SEX
    Data Type in XML: xs:string
    Data Type in Relational: char
```

```
--------------------------------------

    Attribute's Name: SALARY

    Data Type in XML: xs:decimal

    Data Type in Relational: decimal

--------------------------------------

    Attribute's Name: SUPERSSN

    Data Type in XML: xs:string

    Data Type in Relational: char

--------------------------------------

    Attribute's Name: DNO

    Data Type in XML: xs:int

    Data Type in Relational: int

--------------------------------------

    Primary Key Attribute: ssn

    Foreign Key Attribute: superssn

    reference to: EMPLOYEE_PrimaryKey

--------------------------------------

--------------------------------------

Relation: PROJECT

--------------------------------------

    Attribute's Name: PNAME

    Data Type in XML: xs:string

    Data Type in Relational: char

--------------------------------------

    Attribute's Name: PNUMBER
```

```
    Data Type in XML: xs:int
    Data Type in Relational: int
---------------------------------------
    Attribute's Name: PLOCATION
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: DNUM
    Data Type in XML: xs:int
    Data Type in Relational: int
---------------------------------------
    Primary Key Attribute: pnumber
    Foreign Key Attribute: dnum
    reference to: DEPARTMENT_PrimaryKey
---------------------------------------
---------------------------------------
Relation: WORKS_ON
---------------------------------------
    Attribute's Name: ESSN
    Data Type in XML: xs:string
    Data Type in Relational: char
---------------------------------------
    Attribute's Name: PNO
    Data Type in XML: xs:int
    Data Type in Relational: int
```

```
---------------------------------------

    Attribute's Name: HOURS

    Data Type in XML: xs:decimal

    Data Type in Relational: decimal

---------------------------------------

    Primary Key Attribute: essn

    Primary Key Attribute: pno

    Foreign Key Attribute: essn

    reference to: EMPLOYEE_PrimaryKey

    Foreign Key Attribute: pno

    reference to: PROJECT_PrimaryKey

---------------------------------------

---------------------------------------
```