Large-scale database modeling: Developing Xml schema

Zumreta Maslesa
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.
LARGE SCALE DATABASE MODELING:
DEVELOPING XML SCHEMA

by

Zumretta Maslesa

Bachelor of Science
University of Nevada, Las Vegas
1999

A thesis submitted in partial fulfillment of the requirements for the

Master of Science Degree
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2002
The Thesis prepared by

ZUMRETTA MASLESA

Entitled

LARGE SCALE DATABASE MODELING: DEVELOPING XML SCHEMA

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Graduate College Faculty Representative
ABSTRACT

Large Scale Database Modeling: Developing XML Schema

by

Zumreta Maslesa

Dr. Kazem Tagva, Examination Committee Chair
Professor of Computer Science
University of Las Vegas, Nevada

This thesis is concerned with the team efforts to develop a large database to track medical information. A large XML schema is developed from the Extended ER diagram to capture key and foreign key constraints. The strong types of XML schema were also used to assert patterns and domain constraints. These constraints will be used to aid the recognition of medical forms.
TABLE OF CONTENTS

ABSTRACT .............................................................................................................. iii
LIST OF FIGURES .................................................................................................. vi
ACKNOWLEDGEMENTS ........................................................................................ vii
CHAPTER 1  INTRODUCTION ....................................................................... 1

CHAPTER 2  XML SCHEMA BACKGROUND .............................................. 3
  XML and Database .......................................................................................... 4
  Overview of XML ............................................................................................ 5
    Elements ..................................................................................................... 6
    Attributes .................................................................................................. 6
    Entity References ......................................................................................... 7
    Comments .................................................................................................. 7
    Processing Instructions (Pls) .................................................................. 7
    CDATA Sections ......................................................................................... 7
  Document type definition (DTD) ................................................................. 8
  Namespace ........................................................................................................ 9
  XML Schema ..................................................................................................... 11
    Simple and complex types ........................................................................ 12
    Simple Type Derivation ............................................................................ 15
    Cardinality .................................................................................................. 16
    References .................................................................................................. 16
    Compositors ............................................................................................... 17
    Groups ........................................................................................................ 18
    Constraints .................................................................................................. 19
  XML Schema XPath Subset ............................................................................ 20
  Abstract types .................................................................................................. 21
  Documentation .................................................................................................. 21

CHAPTER 3  XML SCHEMA IMPLEMENTATION ...................................... 22
  XML Schema for complex types ..................................................................... 24
  Key constraints in XML Schema ................................................................. 28
  XML Schema for Binary Relationship ........................................................... 31
  XML Schema for Ternary Relationship ........................................................ 33
  XML Schema for Generalization ................................................................... 38
LIST OF FIGURES

2.1 Map of simple type ............................................................... 13
3.1 Binary relationship between Person and Signature ................. 31
3.2 Ternary Relationship between Patient, Physician and Referral .......... 34
3.3 Example of generalization ................................................... 39
3.4 Example of aggregation ....................................................... 42
3.5 Example of weak entity ....................................................... 46
ACKNOWLEDGEMENTS

I would like to thank all of the staff members at the Information Science Research Institute for their discussions about this work as it progressed, especially Julie Borisack. I am also grateful to Jeff Coombs for his help with \LaTeX. I would like to especially thank Dr. Kazem Taghva for advising this work and introducing me to the field of database. I would also like to specifically thank Dr. Tom Nartker, Dr. Wolfgang Bein, and Dr. Henry Selvaraj for serving on the committee.

I am grateful to my sister Zemira for their inspiration. Moreover, special thanks go to my parents for their encouragement.
CHAPTER 1

INTRODUCTION

In today's distributed healthcare environment, information is a key asset and getting access to that data is vital to the management of a patient's health. The need for electronic health records (EHRs) arises from the need to be able to record patient health information more accurately and in greater detail than is possible with paper [3][32]. The medical record is an important tool supporting quality in clinical care. Just as there will be many different situations in which it is accessed, the record plays many roles in the provision of care to individuals and to populations.

The Information Science Research Institute (ISRI) is engaged in the process to digitally archive a massive amount of medical records [28]. This work is specific to occupational medicine. Occupational medicine is medical specialty dedicated to prevention, diagnosis, and treatment of illnesses and injuries arising from work-related activity. Forms from patients' files are broken down to attributes and to entities, and an Extended Entity Relation (EER) diagram was created. Database created this way will be used by OCR for data recognition. ISRI's main goal is to design a system that takes as input hard copy medical data and produces queryble information.

With the growing popularity of XML (eXtensible Markup Language) [1], XML Schema is being widely used to describe data. This project seeks to build XML Schema from EER diagram, represented in the Unified Modeling Language (UML) notation. UML was chosen because it is a popular method for designing software and has proven to be valuable for data modeling. XML Schema was chosen as the method for describing the structure of XML instance documents. XML is intended
to be a self-describing data format, allowing authors to define a set of element and attribute names that describe the content of a document. This paper defines what element and attribute names are allowed to appear in a document in order to make that document useful, and what sort of content they are allowed to contain.

An XML Schema specifies the kinds of objects allowed in an XML document, as well as how the objects and their properties are to be organized and the types of values that can be assigned to the object attributes.

Chapter 2 provides an introduction to the concept of XML Schema and its types. Chapter 3 covers the concept of XML Schema and the approach taken to construct such a schema for this project. The EER diagram features are described and its representation in XML Schema. Finally, chapter 4 states the conclusion of the study and offers prospects for further research.
CHAPTER 2

XML SCHEMA BACKGROUND

Markup has been with us almost since the beginning of written language. We use it daily, underlining words, punctuating sentences, capitalizing proper names, and the starting characters of every sentence to make our thoughts clear on paper.

XML is a markup, meta-language (a language for describing other languages) that enables designers to create their own customized tags to provide functionality not available with HTML. Development of XML started in 1996 and has been a W3C Recommendation since February 1998. Before XML, there was SGML (Standard Generalized Markup Language), developed in the early '80s and widely used for large documentation projects. XML took the best parts of SGML, but it is more regular and simple to use. While SGML is mostly used for technical documentation and much less for other kinds of data, with XML it is exactly the opposite. Defining XML as an application profile of SGML means that any fully conformant SGML system will be able to read XML documents. However, using and understanding XML documents do not require a system that is capable of understanding the full generality of SGML [36].

Like HTML, XML makes use of tags (words bracketed by < and >) and attributes. While HTML specifies what each tag and attribute means, and often how the text between them will look in a browser, XML uses the tags only to delimit pieces of data and leaves the interpretation of the data completely to the application that reads it. A forgotten tag or an attribute without quotes makes an XML file unusable, while in HTML such practice is tolerated and is often explicitly allowed. XML is case sensitive.

3
The XML has been widely adopted for Web applications because of its affinity with HTML and its ability to serve as a repository for multiple types of data. Because XML describes a simple flat-file database, every application using any sort of database access can use XML as a lowest common denominator for transport, generating and translating from XML for transfer while using normalized or proprietary formats internally.

XML and Database

One of the computer's most powerful applications is the ability to store, to organize, and to retrieve large quantities of the data. An organized collection of data is referred to as a database, and the programs designed to manage database are known as Database Management Systems (DBMSs) [10][21].

Even though XML offers many advantages as a universal data-exchange format, using it with a DBMS can be problematic. Some sort of data model mapping must be used before XML can be retrieved from or stored in databases. XML is most similar to object modeling because it can be regarded as consisting of nodes, and nodes can contain heterogeneous data.

A data model is a conceptual representation of data structures. The data structures include the data objects, the relationships between the objects, and the rules that govern which operations can be performed on the objects. XML document can be categorized into two major types: data-centric and document-centric. Characterizing document as data-centric or document-centric helps deciding on the kind of database to use: relational, object-oriented, or hierarchical database [10].

The main characteristics of a data-centric XML document are a more organized and regular structure and a lack of mixed content. Data-centric XML documents are designed for application consumption and application-to-application data exchange.
Common examples include invoices, stock quotes, and product catalogs. Document-centric design represents a more liberal use of free-form text that is "marked-up" with elements. Document-centric XML documents are usually meant for human consumption and are characterized by a less regular structure with plenty of mixed content. Examples are books, letters, e-mails, and HTML/XHTML documents.

XML APIs (Application Programming Interfaces) generally fall into two categories: tree-based and event-based. The API was created by W3C and describes a set of platform- and language-neutral interfaces that can represent any well-formed XML or HTML document. DOM (Document Object Model) is a tree-based API for XML that provides an object-oriented view of the data. SAX (Simple API for XML) is an event-based, serial-access API for XML that uses callbacks to report parsing events to the application.

Overview of XML

The XML specification provides for two levels of document processing: well-formed and valid. An XML document that conforms to the structural and notational rules of XML is considered well-formed, and an XML document that is well-formed and also conforms to a DTD is considered valid [41].

XML is extensible, platform-independent, fully Unicode compliant, and it supports internationalization and localization. XML is a set of rules for designing text formats that let us structure our data. Since XML is a text format and it uses tags to delimit the data, XML files are nearly always larger than comparable binary formats. The advantages of a text format are that it allows users to look at the data without the program that produced it. The disadvantages can usually be compensated at a different level. Disk space is less expensive than it used to be, and compression programs like zip and gzip can compress files well and fast. In addition, communication protocols, such as modem protocols and HTTP/1.1 can compress data on the fly,
saving bandwidth as effectively as a binary format. XML is referred to as a 'write once, publish anywhere' language, with facilities, such as stylesheets that allow the same XML document to be published in different ways using a variety of formats and media.

An XML document consists of elements, attributes, entity references, comments, CDATA sections, and processing instructions [25][39].

Elements

Delimited by angle brackets, most elements identify the nature of the content they surround. Some elements may be empty (<applause></applause>), in which case they have no content. Since XML documents do not require a DTD, it could be impossible for an XML parser to determine which tags were intentionally empty and which had been left empty by mistake. Every tag has to be closed in the context in which it was opened.

Attributes

Attributes are name-value pairs that occur inside start-tags after the element name. For example, <sex gender="M"> is a sex element with the attribute gender having the value M. In XML, all attribute values must be quoted. Attributes cannot contain any child information items, and they are always simple types. The order in which attributes should appear on a parent element cannot be specified. Attribute declarations can either be local or global. If they are global declarations, they are direct children of the schema element, meaning that any complex type definition can make use of the attribute.

When we declare an element to carry an attribute, its presence in an instance document is optional. There is no provision for minOccurs and maxOccurs attributes on attribute declarations because an attribute can only appear once on any given element. To change this, we add an attribute called use to the attribute declaration, which takes value required when indicating that an attribute must appear, or optional
when it can either appear once or not at all (the default value), or prohibited when we want to explicitly indicate that it must not appear. For example,

<attribute name="isbn" type="isbnType" use="required"/>

Entity References

Every entity must have a unique name and is simply referenced by name. Entity references begin with the ampersand and end with a semicolon. For example, &lt; entity inserts a literal < into a document. A special form of entity reference, called a character reference, can be used to insert arbitrary Unicode characters into a document. This is a mechanism for inserting characters that cannot be typed directly on a keyboard. Character references take one of two forms: decimal references, &#8478;, and hexadecimal references, &amp;#x211E;.

Comments

Comments are enclosed in &lt;!-- and --&gt; tags and can contain any data except the literal string '—'. Comments can be placed between a markup anywhere in a document, although an XML processor is not required to pass them to an application.

Processing Instructions (PIs)

The XML document begins with a PI: &lt;?xml version="1.0"?&gt;[33]. While it is not required, its presence explicitly identifies the document as an XML document and indicates the version of XML to which it was authored. Like comments, they are not textually part of the XML document, but the XML processor is required to pass them to an application.

CDATA Sections

A CDATA section instructs the XML parser to ignore markup characters (< and &), for example). Between the start of the section, &lt;![CDATA[ and the end of the section, ] ]&gt;, all character data are passed directly to the application, without
interpretation. The only string that cannot occur in a CDATA section is | |> [38]. For example,

\[
<! [CDATA[ \*p = \&q; \ b = \ (i \ <= \ 3); \ ]] >.
\]

Document type definition (DTD)

A set of rules for structuring an XML document is called a DTD. A DTD is a formal grammar that specifies a legal XML document, based on the tags used in the document and its attributes. The DTD may consist of two parts, an internal subset and an external subset. DTDs have a number of limitations. DTDs are written in a non-XML syntax; they have no support for namespaces, and they only offer extremely limited data typing. The W3C XML Schema overcomes these limitations and is much more expressive than DTDs.

Unlike SGML, XML does not require a DTD. However, in some its declaration is required in order to be understood unambiguously:

- Most authoring environments need to read and to process the DTD in order to understand and to enforce the content models of the document.

- If an XML document relies on default attribute values, at least part of the declaration must be processed in order to obtain the correct default values.

- The semantics associated with white space in element content differs from the semantics associated with white space in mixed content. Without a DTD, the processor can not distinguish between these cases, and all elements are effectively mixed content.

In applications where a person composes or edits the data (as opposed to data that may be generated directly from a database, for example), a DTD is probably going to be required if any structure is to be guaranteed.
If present, the DTD must be the first thing in the document after optional processing instructions and comments [33]. In order to determine if a document is valid, the XML processor must read the entire DTD (both internal and external subsets). For performance reasons, many XML documents will be used without ever validating against the DTD, even if DTD is available. Over slow connections, reading in a DTD located external to user’s local machine may be slow, and because DTDs may contain references to other documents, resolving all the external references may take an inordinate amount of time even with a high speed connection.

Users are accustomed to seeing HTML documents load incrementally, so they can be read before the document finishes loading, but validating XML parsers aren’t allowed to display the document unless it is valid. Thus, the document won’t appear on the user’s screen until everything is loaded.

One of the greatest strengths of XML is that it allows user to create his/her own tag names. The four kinds of declarations in XML are as follows: element type declarations, attribute list declarations, entity declarations, and notation declarations [39].

Namespace

Namespaces allow element names and relationships in XML document to be qualified to avoid name collisions for elements that have the same name but are defined in different vocabularies. XML allows us to define a new document format by combining and reusing other formats. Since two formats developed independently may have elements or attributes with the same name, care must be taken when combining those formats, for example <name> means first and last name in one format and company name in other. To eliminate name confusion when combining formats, XML provides a namespace mechanism [34].

We want to have namespace and localname for every tag or attribute name
in order to fix above problem. Thus, we author a vocabulary for employee as 
\url{http://www.arc.com/JP#employee}, and \url{http://www.arc.com/JP#company} as vocab-
ulary for company. If a name tag has no prefix, then it belongs to default namespace; 
in our case, it is employee name. Consider the following snippet:

```xml
<item xmlns='http://www.arc.com/JP#employee' xmlns:comp='http://www.arc.com/JP#company'>
  <name>
    <first>John</first>
    <last>Doe</last>
  </name>
  <feature>
    <item>
      <comp:name>IBM</comp:name>
    </item>
  </feature>
</item>
```

Every schema starts with namespace declaration. We can define namespaces as 
one of the following:

1. \url{http://www.w3.org/2001/XMLSchema} 
   The names in this namespace are understood by all schema XML processors. 
   These are not names occurring in the document instance.

2. \url{http://www.w3.org/2001/XMLSchema_instance} 
   A small number of special names that are defined in the XML Schema specific-
   cation but are used in the instance document.

3. Target namespace

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
User defined names that are to be used in the instance document.

Each XML Schema document is bound to a specific namespace through the targetNamespace attribute or to the absence of namespace through the lack of such an attribute. We need at least one schema document per namespace we want to define.

```xml
<xsd:schema targetNamespace="http://example.org/ns/books/">
    <xs:schema targetNamespace="http://example.org/ns/books/">
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:bk="http://example.org/ns/books/
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">
    ...
</xsd:schema>
```

Line (1) says we have chosen to use the prefix xsd to identify the elements that will be W3C XML Schema instructions, and that we will prefix the W3C XML Schema predefined datatypes with xsd. We could have chosen any prefix instead of xsd. We could even make http://www.w3.org/2001/XMLSchema our default namespace, and in this case, we wouldn't have prefixed the W3C XML Schema elements or its datatypes. Let us assume that all examples in chapter 2 and chapter 3 will have above default namespace. We define http://example.org/ns/books/ namespace with a bk prefix. This means that we will now prefix the references to "objects" (datatypes, elements, attributes) belonging to this namespace with bk:

XML Schema

An XML Schema is the definition of a specific XML structure [35][39]. An XML schema uses the W3C XML Schema language to specify how each type of element in the schema is defined and what data type that element has associated with it.
The schema is itself an XML document, so it can be read by the same tools that read the XML it describes. XML Schema files are saved with .xsd extension. The XML Schema specification makes a clear distinction between element “definition” and element “declaration”. <element> is a declaration of an element that may appear in a valid document instance, but it doesn’t define that element type. <complexType> definitions include child elements that define the content model and/or attributes for this element type [8].

The XML tags make a document self-describing. The XML specification explicitly defines the use of schema-less XML processing. And since all XML documents must be well formed, they can be parsed without schema. But there remain some compelling reasons for creating an XML schema to accompany our application such as: defining and documenting vocabulary for all users, validating documents when using XML parsers, providing default attribute values, enumerated lists, and identifier declarations, etc. Using a schema, an XML vocabulary can also provide default attribute values and enumeration lists. For example, an address element type may include a county code attribute that defaults to “US” if the value is not provided in a document instance. We can supply a default value for text-only element content. If we specify a default value for an element and that element is empty in the instance document, an XML Schema aware processor would treat the document as though it had the default value when it parses the document. If we want the default content of the Subscribe element to be yes, we will write it as:

<element name = "Subscribe" type = "string" default = "yes"/>

Simple and complex types

The easiest way to create XML Schema is to use the Russian doll design, whereby we define each element as we encounter.
Figure 2.1: Map of simple type
Elements that contain other elements are of type complexType. Elements that have no subelements or attributes are of type simpleType. The current specification defines 43 simple datatypes that are built-in to XML Schema (see figure 2.1). W3C XML Schema gives us a mechanism to define data types and to use these types to define our attributes and elements [39]. We can define either simple types that will be used for PCDATA elements or attributes, or complex types that will be used only for elements. This is achieved by giving a name to the simpleType and complexType elements, and locating them outside of the definition of elements or attributes. For instance, to define a datatype named nameType, which is a string with a maximum of 32 characters, we will write:

```xml
<simpleType name="nameType">
  <restriction base="string">
    <maxLength value="32"/>
  </restriction>
</simpleType>
```

The enumeration element limits a simple type to a set of distinct values. For example, we can define identification for an employee as follows:

```xml
<simpleType name="identificationListType">
  <restriction base="string">
    <enumeration value="SSN"/>
    <enumeration value="employeeNo"/>
    <enumeration value="personID"/>
  </restriction>
</simpleType>
```
Another powerful facet is the *pattern* element, which defines a regular expression that must be matched. For instance, we can define an isbn datatype as 10 digits long, thus:

```xml
<simpleType name="isbnType">
  <restriction base="string">
    <pattern value="[0-9]{10}"/>
  </restriction>
</simpleType>
```

**Simple Type Derivation**

Simple datatypes are defined by derivation of other datatypes, either predefined and identified by the W3C XML Schema namespace or defined elsewhere in our schema. The different kinds of restrictions that can be applied on a datatype are called facets. Many facets allow constraints on the length of a value, an enumeration of the possible values, the minimal and maximal values, its precision and scale, etc.

The following definition uses *union* derivation type [39]. The union has been applied on the two embedded simple types to allow values from both datatypes. Our new datatype, named isbnType, will now accept the values from an enumeration with two possible values (TBD and NA) or any 10 digits long number:

```xml
<simpleType name="isbnType">
  <union>
    <simpleType>
      <restriction base="string">
        <pattern value="[0-9]{10}"/>
      </restriction>
    </simpleType>
  </union>
</simpleType>
```
<simpleType>
  <restriction base="NM_TOKEN">
    <enumeration value="NA"/>
    <enumeration value="TBD"/>
  </restriction>
</simpleType>

Cardinality

The W3C XML Schema allows the cardinality of an element to be represented using the attributes minOccurs (the minimum number of occurrences) and maxOccurs (the maximum number of occurrences). To represent an optional element, we set minOccurs to 0; to indicate that there is no maximum number of occurrences, we set maxOccurs to the term unbounded. If unspecified, each attribute defaults to 1.

References

The Russian doll design is relatively simple, but it does lead to a significant depth in embedded definitions, and the resulting schema can be difficult to read and to maintain. An alternative approach (flat catalog) is based on using references to elements and to attribute definitions that need to be within the scope of the referencer. This enables re-uses of element declarations and saves repeating the element declarations inside each element. For example, we could define staffno element as:

<element name="staffno" type="string"/>

and use this definition in the following way whenever a staffno element is required,
Compositors

XML Schema supports three compositors that can be mixed to allow various combinations. Each of these compositors can have \textit{minOccurs} and \textit{maxOccurs} attributes to define their cardinality. The \textit{sequence} compositor defines ordered groups of elements. The \textit{choice} compositor describes a choice between several possible elements or groups of elements. The following group can appear within groups, complex types, or other compositors and will accept either a single name element or a sequence of \texttt{firstName}, an optional \texttt{middleName} and a \texttt{lastName}:

\begin{verbatim}
<group name="nameTypes">
  <choice>
    <element name="name" type="string"/>
    <sequence>
      <element name="firstName" type="string"/>
      <element name="middleName" type="string" minOccurs="0"/>
      <element name="lastName" type="string"/>
    </sequence>
  </choice>
</group>
\end{verbatim}

The \textit{all} compositor defines an unordered set of elements. In order to avoid combinations that could become ambiguous or too complex, \textit{all} can appear only as a unique child at the top of a content model and their children can be only element definitions or references, and cannot have a cardinality greater than one. The following complex type definition allows its contained elements to appear in any order:

\begin{verbatim}
<complexType name="bookType">
  <all>
\end{verbatim}
Groups

The W3C XML Schema allows the definition of both groups of elements and groups of attributes. These groups are not datatypes but containers holding a set of elements or attributes that can be used to describe complex types.

<!-- definition of an element group -->
<group name="mainBookElements">
  <sequence>
    <element name="title" type="nameType"/>
    <element name="author" type="nameType"/>
  </sequence></group>

<!-- definition of an attribute group -->
<attributeGroup name="bookAttributes">
  <attribute name="isbn" type="isbnType" use="required"/>
  <attribute name="available" type="string"/>
</attributeGroup>

<!-- example of use in the definition of complex types -->
<complexType name="bookType">
  <sequence>
    <group ref="mainBookElements"/>
    <element name="character" type="characterType" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>
The W3C XML Schema also provides path expression (XPath) based features for specifying uniqueness constraints and corresponding references constraints that will hold within a certain scope. A key might be composed of a sequence of values located at different depths inside an element. XML key declarations are associated with collection of objects rather than types. A key does not need to be associated with a particular simple type. We can specify a key in XML Schema as candidate key (use the tag unique) or as primary key (use the tag key) [40][13]. We consider three types of constraints:

1. Unique constraints

To define uniqueness constraints, we specify a unique element that defines the element or attributes that are to be unique. The location of the unique element in the schema provides the context node in which the constraint holds. The following declaration indicates that the character has to be unique within the context of this element only:

```xml
<unique name="charName">
  <selector xpath="character"/>
  <field xpath="name"/>
</unique>
```

The two XPaths defined in the uniqueness constraint are evaluated relative to the context node. The selector defines which element has the uniqueness constraint, the node to which the selector points must be an element node. The
field element is evaluated relative to the element identified by the selector and can be an element or an attribute node.

2. Key

The only difference between key and unique key is that the value of keys can not be null.

```xml
<key name="charName">
  <selector xpath="character"/>
  <field xpath="name"/>
</key>
```

3. Keyref

The keyref is used to define a reference to a key or a unique. The keyref element defines a referential constraint made up of element and/or attribute fields that refer to a key. To indicate that friend-of needs to refer to a character from this same book, we will write the following:

```xml
<keyref name="charNameRef" refer="charName">
  <selector xpath="character"/>
  <field xpath="friend-of"/>
</keyref>
```

XML Schema XPath Subset

All values of the xpath attribute in the selector and field tags must be legal expression. XPath expressions are made up of paths, separated by vertical bars. Each path is made up of steps, separated by forward slashes. A path may begin with the .// literal, which means that the matching nodes may appear anywhere in the descendants of the current scooping element. For example, .//department | .//product selects all department elements and product elements appearing anywhere under the
current element. Literal @ selects attribute of the element; thus \textit{product/@effDate} selects the \textit{effDate} attribute of all product children.

\textbf{Abstract types}

A substitution group is not defined explicitly through a W3C XML Schema element, but through referencing a common element (called the head) using a \textit{substitutionGroup} attribute. The head element does not hold any specific declaration but must be global. All the elements within a substitution group need to have a type that is either the same type as the head element or can be derived from it. Then they can all be used in place of the head element. This concept is described in greater detail in chapter 3.

\textbf{Documentation}

W3C XML Schema provides an alternative to XML comments (for humans) and processing instructions (for machines) that might be easier to handle for supporting tools. Human readable documentation can be defined by \textit{documentation} elements, while information targeted to applications should be included in \textit{appinfo} elements. Both elements need to be included in an \textit{annotation} element and to accept optional \textit{xml:lang} and \textit{source} attributes and any content type.
CHAPTER 3

XML SCHEMA IMPLEMENTATION

This section defines mappings from EER (Enhanced or Extended ER) diagram to XML Schema. An EER diagram represents relationships between data entities. The EER model is an extension of the Entity Relationship (ER) model proposed by Chen [24]. We can represent almost any important modeling with ER and EER design methodologies. The standard set for the representation of an EER diagram does not exist, and here, we have chosen UML notation to represent its basic concepts. The UML model is independent of the implementation language(s) and has proven to be valuable for data modeling [16]. The direction of arrows in the UML diagram indicates dependency, not the sequence of development process flow.

An EER diagram is a graphical representation of the elements, relationships, and constraints of an XML vocabulary that make up a given design [19]. In this model, each entity is shown as rectangle label with name of entity where the first letter is upper case. Entities are described using attributes, and every attribute specifies a particular property for that entity. Attributes are listed within the associated entity rectangles, and relationships are shown as lines linking two or more entities. An attribute whose value identifies the entity is called a key. Sometimes, more than one attribute is needed to uniquely identify an entity; in that case, the combination of those attributes is called a key. Relationship multiplicity constraints are also represented using standard UML notation. Here, we used a number of non-standard annotations required to represent some common conceptual constraints, such as primary identification of entities (an attribute suffixed with PK) [15].

22

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The basic constructs of an EER diagram are as follows:

• Relationships

Relationships provide connections between two or more entities. Each relationship is given a name that describes its function. When two entities are involved in a relationship, it is known as a binary relationship, and it is referred to as being one-to-one (1:1), one-to-many (1:*), or many-to-many(*:*). A relationship between three entities is called a ternary relationship and so on. The degree of a relationship is the number of participating entities; thus a relationship of degree two is called binary. Multiplicities are placed on opposite ends of the relationship. Like entities, relationships can have attributes. To distinguish between a relationship with an attribute and an entity, the rectangle representing the attribute(s) is associated with the relationship using a dashed line. The participation constraint specifies whether the existence of an entity depends on its being related to another entity via the relationship. The participation can be total or partial. For example, in figure 3.1 while a person in our database may or may not have a signature, each signature must be associated with a person. Hence, the entity person has a partial participation in relationship Signs, whereas the entity signature has total participation [6].

• Generalization and Specialization

Specialization is the process of maximizing the differences between members of an entity by identifying their distinguishing features. It is a top-down approach to define a set of superclasses and their related subclasses. We may have several specialization of the same entity based on different distinguishing characteristics. Generalization is the process of minimizing the differences between members of an entity by identifying their common features. This is a bottom-up approach, which results in a generalized superclass. The relationship between a superclass and subclass is one-to-one. Generalization can be viewed
as being functionally the inverse of the specialization process. Generalization from one class to another is a fundamental concept in object-oriented analysis and design. The attributes of a superclass are inherited by its subclass. This is called attribute inheritance. The key of the superclass is also a key of all of its subclasses. This property is sometimes called ISA relationship because of the way we refer to the concept, for example “a secretary IS-AN employee” [10].

- Aggregation

Aggregation represents a “has-a” or “is-part-of” relationship between entities, where one represents the “whole” and the other the “part.” UML represents aggregation by placing an open diamond shape at one end of the relationship line, next to the entity representing the “whole.” The variation of aggregation called composition represents a strong ownership and coincidental lifetime between the “whole” and the “part.” Our database does not contain any composition [9][30].

- Weak Entity

An entity that is existence-dependent on some other entity is called a weak entity. Such an entity is identified as being related to another entity. A weak entity only has a partial key that uniquely identifies the weak entity related to the owner entity. An owner entity may itself be a weak entity [20].

**XML Schema for complex types**

Each entity creates a schema `complexType` definition containing a child element for each attribute within that entity [22]. The `complexType` definition includes child elements that define the content model and/or attributes for this element type. Both the child element definitions and attribute definitions specify the type of their content. The type may be one of the built-in primitive types or one of the complex or simple
types declared within the schema. For example, personName is of type personNameType. Here, we specify multiplicity using minOccurs and maxOccurs attributes in the element definition [12] [3].

Next, we define some complexTypes that will be used later in the chapter. Consider the definition of the complexType personNameType. Each complexType must define the content model for its child elements [26]. The content model for personNameType is any, determined by examining our forms. The personNameType consists of the lastName, firstName and middleInitial child elements that may be present, subject to their individual multiplicity constraints. We can define personNameType elements as simple types because they do not have attributes or non-text children. A first name can be written as a combination of two names, two names with a hyphen in between, or just one name. The same goes for last name. We captured this in simpleType definition called nameType. An initialType definition says that element middleInitial can have as value any capital letter possibly followed by a dot.

<complexType name="personNameType">
 <any>
   <element name="lastName" type="nameType"/>
   <element name="firstName" type="nameType"/>
   <element name="middleInitial" type="initialType" minOccurs="0"/>
 </any></complexType>

<simpleType name="nameType">
 <union>
   <complexType>
   <restriction base="string">
     <pattern values="[a-z]{20}"/>
   </restriction>
 </complexType>
 </union>
</simpleType>
A Person entity is identified by derived key, called personID. On some forms, we could capture their social security numbers and employee numbers. Thus, we will use for personID value of SSN if present, if not we would use their employee number. If employee numbers weren’t available, we would use generated personID. PersonID is of pattern: capital letter P followed by 9 digits. We capture this with following code:
<simpleType>
  <restriction base="string">
    <pattern value="P{1}\d{9}"/>
  </restriction>
</simpleType>

The primitive type *date* requires that the element contains a valid date when the document instance is validated against this schema. This date represents the Gregorian date, for example 2002-07-26. Throughout our forms, we noticed a few instances of writing a date as 07-26-2002 or 07-26-02. To capture this, we write the following:

<simpleType name="dateType">
  <union>
    <simpleType>
      <restriction base="date">
        </restriction>
    </simpleType>
    <simpleType>
      <restriction base="integer">
        <pattern value="\d{1,2}-\d{1,2}-\d{2} | \d{1,2}-\d{1,2}-\d{4}"/>
      </restriction>
    </simpleType>
  </union>
</simpleType>

Images of actual signatures are stored as *.jpg* files; thus, we define the following:
Key constraints in XML Schema

To explain concept of key constraints, we will examine the figure 3.1, namely the entity Signature [32][4]. Whenever we want to allow an element to carry an attribute or contain a child element, we have to define a complex type. The complex type defined for Signature entity is known as an anonymous complex type because it is nested within the element declaration. Entity Signature contains two child elements and one attribute. By definition, the value of an attribute signatureID is a simple type, and it is declared in the previous section. Attributes that have primitive data types are included in the schema as element content [37]. An attribute presence in an instance document is optional, thus we add an attribute use which takes value required to indicate that an attribute must appear. We want to make sure that our primary key exists, thus every attribute that participate in primary key constraint will have attribute use. The signatureIDType, signatureType and dateType types are defined in the previous section. We write the following XML Schema code to represent entity Signature:

```xml
<element name="Signature">
  <complexType>
    <all>
      <element name="personSignature" type="signatureType"/>
      <element name="signatureDate" type="dateType"/>
    </all>
  </complexType>
</element>
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<attribute name="signatureID" type="signatureIDType" use="required"/>
</complexType>
</element>

The keyref acts as a foreign key while key acts as a primary key in the XML Schema [18] [23]. The key/keyref/unique element may be placed anywhere in our schema, but where we place them determines the scope of the uniqueness. If we place them at the top level (direct child of the schema element), we are stating that in an instance document, the uniqueness is with respect to the entire document. In our example, they are placed as child of an element. Thus, over the entire instance document there may be repeats, but within any instance of the parent element, it will be unique.

The key definition declares that the signatureID attribute must be present for all Signature elements, and it must be unique across all signatureID attributes on Signature elements. The selector element selects all Signature elements, and the field element specifies that the signatureID of Signature entity must be unique; that is, no two signatures can have the same value for their signatureID attribute. The key has a name, signaturePK so that it can be referred to from a keyref element.

<key name="signaturePK">
  <selector xpath=".//Signature"/>
  <field xpath="@signatureID"/>
</key>

We now define foreign key. We want the attribute signatureID to refer to the fields that constitute the key. The reference is defined using the attribute refer, and its value must match the name of a key or unique constraint. In our case, the refer
attribute references the signaturePK key described earlier, and field element specifies that the signatureID children of the Signature must have a corresponding value in the signatureID element identified by the key. Definition of this foreign key will be used when explaining XML Schema code for binary relationship. The code for foreign key definition is as follows:

```xml
<keyref name="signatureFK" refer="signaturePK">
  <selector xpath=".//Signature"/>
  <field xpath="@signatureID"/>
</keyref>
```

If we define two fields in the key, then we must define two fields in the keyref and so on. Further, the fields in the keyref must match in type and position the key. Hence, complete XML Schema code to represent Signature entity including key constraints is in the following listing:

```xml
<element name="Signature" minOccurs="0">
  <complexType>
    <all>
      <element name="personSignature" type="signatureType"/>
      <element name="signatureDate" type="dateType"/>
    </all>
    <attribute name="signatureID" type="signatureIDType" use="required"/>
  </complexType>
  <key name="signaturePK" !--Signature Primary Key-->
    <selector xpath=".//Signature"/>
    <field xpath="@signatureID"/>
  </key>
</element>
```
XML Schema for Binary Relationship

All attributes within entities and relationships are mapped to XML elements or attributes. In order to create relationships, we must have at least one key defined in one of the associated entities [7]. Figure 3.1 represents two entities with a relationship Signs. An entity Person has a primary key called personID. The explanation of creating XML Schema code for Person entity is omitted for the sake of brevity. The only way to show a relationship between two entities is via a foreign key. For each relationship we generate schema with attributes consisting of the key(s) of each associated entity and associated attributes. The keys of the associated entities are foreign keys [29]. Each signature is signed by a single person, but not every person has a signature. We say that Sign is uniquely identified by signatureID. Thus the key
of the relationship Signs is signatureID. We use the primary key of the entity because the goal is to identify the entity involved in the relationship. An arrow symbol next to the relationship name indicates the direction of relationship. In order to describe XML code for relationship Signs, we will first define a complex type for Signs as follows:

```xml
<complexType name="signsType">
  <sequence>
  </sequence>
  <attribute name="signatureID" type="signatureIDType" use="required"/>
  <attribute name="personID" type="personIDType" use="required"/>
</complexType>
```

From figure 3.1, we see that the relationship Signs does not have its own attribute. By definition, we include primary keys of associated entities as Signs's attributes. To capture participation between Person and Signature entities, we have to create two foreign keys, named signsFK1 and signsFK2. The selector element selects all Signs elements/attributes. The refer attribute references personPK in the first foreign key and signaturePK in the second foreign key.

The XML code representing the figure 3.1 is as follows.

```xml
<!-- definition of Person entity -->
<element name="Person">
  <complexType>
    <all>
      <element name="personName" type="personNameType"/>
    </all>
  </complexType>
</element>
```
XML Schema for Ternary Relationship

The relationship between three entities is called ternary relationship [23][31]. Figure 3.2 shows a ternary relationship between Patient, Physician, and Referral entity.
The name of our ternary relationship is patientReferredBy. Here, every entity has its own primary key in addition to the foreign key that refers to primary key of the ternary relationship [19][2]. We define primary and foreign key for Patient entity as the following snippet:

```xml
<!-- Patient Primary Key-->  
<key name="patientPK">  
  <selector xpath=".//Patient"/>  
  <field xpath="@patientID"/>  
</key>  

<!-- Patient foreign Key-->  
<keyref name="patientFK" refer="patientReferredByPK">  
  <selector xpath=".//patientReferredBy"/>  
  <field xpath="@patientID"/>  
</keyref>
```

Figure 3.2: Ternary Relationship between Patient, Physician and Referral
We define ternary relationship similar as binary relationship. As in the previous section, we list all primary keys of associated entities as attributes of ternary relation. Thus, the ternary relationship has a primary key that is a combination of three attributes. To capture participation, we created three foreign keys pointing to associated entities.

The XML code representing the ternary relationship for figure 3.2 is as follows:

```xml
<!-- definition of Patient entity -->
<element name="Patient" type="PatientType">
  <key name="patientPK"> <!-- Patient Primary Key-->
    <selector xpath="./Patient"/>
    <field xpath="@patientID"/>
  </key>
  <keyref name="patientFK" refer="patientReferredByPK">
    <selector xpath="./patientReferredBy"/>
    <field xpath="@patientID"/>
    <field xpath="@physicianID"/>
    <field xpath="@referralID"/>
  </keyref></element>
  <complexType name="patientType">
    <all>
      <element name="patientDOB" Type="date"/>
      <element name="numberOfDependent" Type="integer"/>
    </all>
  </complexType>
</element>
```
<element name="Physician" type="physicianType">
    <key name="physicianPK"> <!-- Physician Primary Key-->
        <selector xpath=".//Physician"/>
        <field xpath="@physicianID"/>
    </key>
    <keyref name="physicianFK" refer="patientReferredByPK">
        <selector xpath=".//patientReferredBy"/>
        <field xpath="@patientID"/>
        <field xpath="@physicianID"/>
        <field xpath="@referralID"/>
    </keyref>
</element>

<complexType name="physicianType">
    <all>
        <element name="physicianDegree" Type="physicianDegreeType"/>
    </all>
</complexType>

<attribute name="physicianID" type="integer" use="required"/>

<!-- definition of Referral entity -->
<element name="Referral" type="referralType">
    <key name="referralPK"> <!-- Referral Primary Key-->
        <selector xpath=".//Referral"/>
</element>
<field xpath="@referralID"/>
</key>

<keyref name="referralFK" refer="patientReferredByPK">
    <selector xpath=".//patientReferredByPK"/>
    <field xpath="@patientID"/>
    <field xpath="@physicianID"/>
    <field xpath="@referralID"/>
</keyref></element>

<complexType name="referralType">
    <all>
        <element name="referralDate" Type="dateType"/>
        <element name="treatment" Type="string"/>
    </all>
    <attribute name="referralID" type="integer" use="required"/>
</complexType>

<!-- definition of ternary relationship -->
<element name="patientReferredByPK" type="patientReferredByPKType">
    <key name="patientReferredByPK"><!-- Primary key-->
        <selector xpath=".//patientReferredByPK"/>
        <field xpath="@patientID"/>
        <field xpath="@physicianID"/>
        <field xpath="@referralID"/>
    </key>
    <!-- Foreign Key1 -->
    <keyref name="patientReferredByPKFK1" refer="patientPK">
        <selector xpath=".//patientReferredByPK"/>
        <field xpath="@patientID"/>
        <field xpath="@physicianID"/>
        <field xpath="@referralID"/>
    </keyref>
<field xpath="@patientID"/>
</keyref>

<! -- Foreign Key2 -->
<keyref name="patientReferredByPKFK2" refer="physicianPK">
  <selector xpath=".//patientReferredByPK"/>
  <field xpath="@physicianID"/>
</keyref>

<! -- Foreign Key3 -->
<keyref name="patientReferredByPKFK3" refer="physicianPK">
  <selector xpath=".//patientReferredByPK"/>
  <field xpath="@referralID"/>
</keyref></element>

<complexType name="patientReferredByPKType">
  <sequence>
  </sequence>
  <attribute name="patientID" type="integer" use="required"/>
  <attribute name="physicianID" type="integer" use="required"/>
  <attribute name="referralID" type="integer" use="required"/>
</complexType>

XML Schema for Generalization

The subclass inherits attributes and relationships from all of its parent classes. In figure 3.3, LabMemo is a subclass of Correspondence, which means that LabMemo inherits the three attributes defined in its superclass.

There is a significant difference in the way inheritance structure is interpreted in EER versus in XML Schema. In EER, the order of attributes and relationship references within an entity are not specified, and the features inherited from parent classes
freely intermingle with locally defined attributes and relationships in a subclass. In XML Schema, inherited elements are treated as a group, so the three elements inherited from Correspondence are an unordered group in a LabMemo element, followed in sequence by another unordered group of the elements defined by the LabMemo entity. A single group of these five elements (three inherited and two local to the subclass) cannot be defined when using XML Schema extension. This capability of XML Schema enables schemas to be written in a much more object-oriented style than using DTDs. Inheritance improves maintainability and avoids duplicate declarations.

The `complexType` definition for labMemoType is an extension of the base `complexType` named correspondenceType. The LabMemo element includes substitution-group="Correspondence," which means that whenever the Correspondence element is required as an XML content element, LabMemo may be substituted in its place [11][7]. We use the `complexContent` tag to imply that we will extend existing definition of `complexType`. The `all` group in this definition specifies that the three elements within correspondenceType may appear in any order in a document instance. Types `limitOfDetectionType` and `bloodLeadTestUnitsType` are not defined in the following example, but they must be included in the schema.

The corresponding XML Schema code fragment for figure 3.3 is given in code below. In our example, we are extending the XML `complexType` correspondenceType

![Diagram of LabMemo and Correspondence elements](image.png)
to labMemoType. The LabMemo is extension of the correspondenceType and it inherits all elements from Correspondence. Thus, the primary key of LabMemo is correspondenceID.

```xml
<element name="Correspondence" type="correspondenceType">
  <key name="correspondencePK"> <!-- Primary Key-->
    <selector xpath="./Correspondence"/>
    <field xpath="@correspondenceID"/>
  </key>
</element>

<complexType name="correspondenceType">
  <all>
    <element name="sendDate" type="dateType" minOccurs="0"/>
    <element name="message" type="string"/>
  </all>

  <attribute name="correspondenceID" type="interger" use="required"/>
</complexType>

<!-- LabMemo is subclass of Correspondence -->
<element name="LabMemo" type="labMemoType"
  substitutionGroup="Correspondence">
  <key name="correspondencePK">
    <selector xpath="./Correspondence"/>
    <field xpath="@correspondenceID"/>
  </key>
</element>
```
<complexType name="labMemoType">
  <complexContent>
    <extension base="correspondanceType">
      <all>
        <element name="limitOfDetection"
          type="limitOfDetectionType" minOccurs="0"/>
        <element name="bloodLeadTestUnits"
          type="bloodLeadTestUnitsType" minOccurs="0"/>
      </all>
    </extension>
  </complexContent>
</complexType>

XML Schema for Aggregation

We will explain aggregation concept using figure 3.4. This is just a subset of bigger picture that contains every possible medical examination for a patient. Our code uses choice content to represent aggregation. We allow the choice group to be omitted by setting its minOccurs attribute to zero. This means that medicalExamination entity allows any of the children, in any order, to appear at most once. medicalExamination would also be valid if it were completely empty. Thus, we capture the fact that on the form we can either have medical examination listed or omitted. Looking from another direction, Smoke examination can have the same results for several patients. Thus,
Figure 3.4: Example of aggregation

we write for multiplicity 1..* on side of medicalExamination. We capture this in XML Schema setting maxOccurs attribute to unbounded (minOccurs=1 by default):

```xml
<element name="Smoke" type="smokeType" maxOccurs="unbounded" />
<element name="Vaccination" type="vaccinationType" maxOccurs="unbounded"/>
```

From figure 3.4 we see that there are two relationships between medicalExamination and associated entities, called IncludeSmokePart and IncludeVaccinationPart. An explanation of representing these relationships is omitted for the sake of brevity, but its code is included below. The combined values in two attributes, medicalExaminationDate and medicalExaminationPurpose, are a reference to a key defined by includesSmokePartFK1. For each value of includesSmokePartFK1 (the pair of two attributes), there must be a medicalExaminationPK with the same value [11][26]. The same code for the Smoke entity is applied to the Vaccination entity. Thus, it is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
We write the following code to represent aggregation from figure 3.4:

```xml
<element name="MedicalExamination" type="medicalExaminationType">
    <key name="medicalExaminationPK"> <!-- medicalExamination Primary Key -->
        <selector xpath="./MedicalExamination"/>
        <field xpath="@medicalExaminationDate"/>
        <field xpath="@medicalExaminationPurpose"/>
    </key>
</element>

<complexType name="medicalExaminationType">
    <choice minOccurs="0">
        <element name="Smoke" type="smokeType"/>
        <element name="Vaccination" type="vaccinationType"/>
    </choice>
    <attribute name="medicalExaminationDate" type="dateType"
        maxOccurs="unbounded" use="required"/>
    <attribute name="medicalExaminationPurpose" type="string"
        maxOccurs="unbounded" use="required"/>
</complexType>

<!-- Smoke entity -->
<element name="Smoke" type="smokeType">
    <key name="smokePK"> <!-- Smoke Primary Key -->
        <selector xpath="./Smoke"/>
        <field xpath="@smokeID"/>
</element>
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
</key>

<keyref name="smokeFK" refer="IncludesSmokePartPK"><!-- Foreign Key-->
   <selector xpath=".//IncludesSmokePart"/>
   <field xpath="@smokeID"/>
   <field xpath="@medicalExaminationDate"/>
   <field xpath="@medicalExaminationPurpose"/>
</keyref>
</element>

<complexType name="smokeType">
   <all>
      <element name="smokeStatus" type="boolean"/>
      <element name="smokeType" type="smokeTypeList"/>
   </all>
   <attribute name="smokeID" type="integer" use="required"/>
</complexType>

<element name="IncludesSmokePart" type="includesSmokePartType">
   <key name="includesSmokePartPK">
      <selector xpath=".//IncludesSmokePart"/>
      <field xpath="@smokeID"/>
      <field xpath="@medicalExaminationDate"/>
      <field xpath="@medicalExaminationPurpose"/>
   </key>

   <keyref name="includesSmokePartFK1" refer="medicalExaminationPK">
      <selector xpath=".//IncludesSmokePart"/>
      <field xpath="@medicalExaminationDate"/>
   </keyref>
</element>
XML Schema for a Weak Entity

Looking at figure 3.5, our weak entity does not have primary key by definition. We can only uniquely identify each dependent through relationship that a Dependent has with a Patient, who is uniquely identifiable using the primary key patientID. The weak entity does not inherit any of elements/attributes of owner entity. For our example, the weak entity Dependent only has one attribute: dependentAge. The local key of the weak entity Dependent cannot be modeled as key attribute since it may not be unique for the whole XML document. We need to generate a composite key as a key of the corresponding owner entity combined with the partial key of the weak entity. In our example, the primary key of the owner entity Patient is patientID, and the partial key of weak entity Dependent is dependentAge; thus, the key of Dependent is patientID and dependentAge.

We utilize a combination of keyref and key elements to specify the composite key [17][31]. To capture our composite primary key, we let selector field select all children of the root of the schema. We can do this because all elements and attributes have unique name in our database. We capture this in the following key definition:
Each weak entity is associated with a binary relationship that identifies the weak entity with its owner entity. This mapping is a 1:1 relationship. We write the following code to represent weak entity from figure 3.5:

```xml
<element name="Patient">
  <complexType>
    <all>
      <element name="numOfDependents" type="integer" minOccurs="0"/>
    </all>
  </complexType>
</element>
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<element name="patientDOB" type="date"/>
</all>

<attribute name="patientID" type="integer" use="required"/>
</complexType>

<key name="patientPK">
<!-- Patient Primary Key-->
  
  <selector xpath=".//Patient"/>

  <field xpath="@patientID"/>

</key>
</element>

<!-- Weak entity -->
<element name="Dependent">
  
  <complexType>
    
    <attribute name="dependentAge" type="integer" use="required"/>

  </complexType>

  <!-- partial key plus primary key from owner entity-->

  <key name="dependentPK">
    
    <selector xpath="/"

    <field xpath="@dependentAge"/>

    <field xpath="@patientID"/>

  </key>

  <!-- Dependent Foreign Key-->

  <keyref name="dependentFK" refer="patientPK">
    
    <selector xpath=".//Dependent"/>

    <field xpath="@patientID"/>

  </keyref>

</element>
CHAPTER 4

CONCLUSION AND FURTHER RESEARCH

The default mapping rules described in previous chapters can be used to generate a complete XML Schema from any EER diagram. Comprehensive example illustrating the algorithm is presented.

We are currently working on the implementation of XML Schema using the algorithm given above. Moreover, we will experiment with the use of a validating XML parser for constraint checking.

In this thesis, we examined several features provided by XML Schema. In particular, we examined how binary and ternary relationship can be represented using XML Schema. We also examined generalization, aggregation, and weak entity representation.

Future research plans are to create normalized form of our database and to investigate the reverse procedure of generating EER diagram from an XML Schema. We also need to incorporate more integrity constraints and dependencies, such as those defined in [14] and [27].
BIBLIOGRAPHY


[34] W3C. Extensible markup language (XML) 1.0 (second edition). URL: <http://www.w3.org/TR/REC-xml-names> (viewed May. 18, 2002).


VITA

Graduate College
University of Nevada, Las Vegas

Zumreta Maslesa

Local Address:
9353 Spellman CT
Las Vegas, NV 89123

Degrees:
Bachelor of Science, Computer Science, 1999
University of Nevada, Las Vegas

Thesis Title:
Large scale database modeling: Developing XML Schema

Thesis Examination Committee:
Chairperson, Kazem Taghva, Ph. D.
Committee Member, Dr. Thomas Nartker, Ph. D.
Committee Member, Dr. Wolfgang Bein, Ph. D.
Graduate Faculty Representative, Dr. Henry Selvaraj, Ph. D.