HETEROGENEOUS DATABASE TRANSFORMATION USING XML

THESIS

Presented to the Graduate Council of

Southwest Texas State University

in Partial Fulfillment of

the Requirements

For the Degree

Master of SCIENCE

By

Kalyan Pydipati

San Marcos, Texas

August, 2003

DEDICATION

This thesis is dedicated to my family, friends, and the Computer Science department-Southwest Texas State University.

ACKNOWLEDGEMENTS

I am very thankful to Dr. Furman Haddix for his invaluable guidance and support. I am also thankful to Dr. Carol Hazlewood and Dr. Gregory Hall for supporting my work and for being on the thesis committee.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract

HETEROGENEOUS DATABASE TRANSFORMATION USING XML

By

KALYAN PYDIPATI, M.S

Southwest Texas State University

August 2003

SUPERVISING PROFESSOR: Furman Haddix

A framework for heterogeneous database transformation using XML as intermediate database exchange format is proposed. The framework consists of two DTDs (Data Type Definition), the structure DTD and the data DTD; and two stages, forward transformation and reverse transformation. The structure DTD defines a set of XML tags for embedding structural information of a heterogeneous database independent of its theoretical format, and the data DTD defines set of XML tags for embedding data of a heterogeneous database independent of its data representation format. A forward transformation transforms the database structure and data of source heterogeneous database to hierarchical structure of XML documents specified by structure DTD and data DTD. A reverse transformation transforms the database structure and data represented by the XML documents to the destination heterogeneous database structure and data. The framework developed is very general and effective, as it provides a database transformation facility between existing heterogeneous database formats. It can be easily extended to support any new database formats with only minor changes in the design of structure DTD and data DTD. The framework has been implemented as a database transformation tool using Java. The database transformation tool successfully

transformed customer database from SQL Server 2000 (Relational database) to Ozone (Object-oriented database), IPEDO (Native XML database) and MYSQL (Relational database); customer database from Ozone to SQL Server 2000 and IPEDO; customer database from IPEDO to Ozone and SQL Server 2000.

CHAPTER I

INTRODUCTION TO THE STUDY

1.1 Motivation

Today's widespread use of the World Wide Web has led to a proliferation of web accessible databases. Often, these databases not only come from different vendors, but from a varied theoretical base. Let us look at some of these heterogeneous databases.

1.1.1 Relational databases

Relational databases are structured as two-dimensional tables in which each item appears as a row. Data represented by tables is related using columns of the tables. Query languages such as SQL allow the user to query, manipulate, create, and update tables in the relational database. Some benefits of relational databases are good performance for huge amount of data; data independence; easy-to-use, declarative, query language; solid theoretical base; efficient optimization techniques and fast querying and retrieval of data.

In general, relational databases do not provide enough support for data manipulation, handling complex data structures, and expressing complex relationships. Relational databases are widely used in education, finance and small businesses. Application areas not well supported by relational databases are multimedia, scientific databases, and CAD and GIS applications.

1.1.2 Object-oriented databases

The object-oriented model consists of object-oriented programming languages, which allow the expression of structures that combine related code and data. The benefit

of using object oriented programming languages includes better support for expressing complex relationships among objects and data manipulation. Some areas of applications where object-oriented database provide better support are multimedia applications, GIS applications and scientific data.

1.1.3 Hierarchical databases

A hierarchical database stores data in hierarchical format; for example native XML databases use XML documents as their primary unit of storage. The structure of XML documents can be specified using XML Schema or DTD. Native XML databases are widely used in applications such as electronic data exchange, catalog data and medical information storage.

1.2 Problem

Mappings, required for database exchange among heterogeneous databases on the above three theoretical bases, include:

- Mapping from Hierarchical database (Native XML database) to relational database and vice-versa
- Mapping from Hierarchical database (Native XML database) to object-oriented database and vice-versa
- Mapping from relational database to object-oriented database and vice-versa

1.3 Proposed solution

In this research, we developed structure DTD, data DTD and proposed mapping techniques for mapping database structure and data between heterogeneous databases and hierarchical structure of XML documents specified by structure DTD and data DTD.

Structure DTD defines a set of XML tags to hold the structural information of a heterogeneous database. Data DTD defines a set of XML tags to hold the heterogeneous database data. The transformation process consists of two parts: forward transformation and reverse transformation. In forward transformation, structure of the heterogeneous database is embedded into the hierarchical structure of XML document using tags defined by structure DTD, and the data of the heterogeneous database is embedded into the hierarchical structure of XML document using tags defined by data DTD. In reverse transformation, database structure represented by XML document is mapped to heterogeneous database structure and the database data represented by XML document is mapped to heterogeneous database data. Some of the issues that are addressed for database transformation are:

- Components.
- Attributes of each component.
- Data type of attributes.
- Size of attributes.
- Associations between components.
- Inheritance between components.
- Keys defined on attributes of components.

1.4 Implementation

The database transformation tool has been developed using Java based on algorithms explained in this research. The database transformation tool provides database transformation facility between relational databases (SQL Server 2000, MYSQL), object-

oriented database (Ozone- a java based database) and hierarchical database (IPEDO- a native XML database).

Three case studies are presented at the end of this thesis demonstrating the transformation of the customer database from SQL Server 2000 (Relational database) to Ozone (Object-oriented database), IPEDO (Native XML database) and MYSQL (Relational database); Ozone to SQL Server 2000 and IPEDO; IPEDO to Ozone and SQL Server 2000.

1.5 Organization of thesis

Chapter 2 describes highlights of research work done in data model mappings and data transformation.

In Chapter 3, discusses heterogeneous databases with regard to their logical structure for data representation, query languages, and visual modeling.

Chapter 4 explains the structure DTD, data DTD and Mapping techniques used for achieving heterogeneous database transformation.

Chapter 5 gives details of database transformation tool and case study.

Chapter 6 highlights conclusions and potential future work.

CHAPTER II

RELATED WORK

There have been numerous prior research efforts in mapping between different data models. Some of the research work done in each area of mapping are:

- Relational model to XML data model [LeMaChCh '02], [FoPaBl '01], [Baru '99].
- Relational model to object-oriented model [Hohenstein '00], [JaScZü '96].
- XML schemas to relational model [MaLe '02],[BoFrRoSi '02]
- Object-oriented model to XML model [Bierman '00]

Lee *et al.* [LeMaChCh '02] specify two algorithms, NET and COT to translate relational schemas to XML schemas using various semantic constraints. They used XSchema (XML representation), a language independent formalism. Their proposed algorithms consist of following characteristics:

1) NeT derives a nested structure from a flat relational model by repeatedly applying the *nest* operator on each table so that the resulting XML schema becomes hierarchical.
2) CoT considers not only the structure of relational schemas, but also semantic constraints, such as inclusion dependencies during translation. CoT takes as input a relational schema where multiple tables are interconnected through inclusion dependencies and converts into a good XSchema.

Fong *et al.* [FoPaBl '01] provides a methodology of translating the conceptual schema of a relational database into an XML schema through EER (extended entity relationship) model. Physical data are then translated from relational table to an XML document. The semantics of a relational database, captured in EER diagram, are mapped to an XML schema using stepwise procedures. The physical data are then mapped to an XML document under the definitions of the XML schema.

Baru [Baru '99] proposes an X-Database system to support import and export of XML documents from relational repositories. The base of this system is an XML-Schema file that describes the logical model of interchanged information. Initially, the system analyses the syntax of the XML-Schema file and generates relational database. Then it handles the decomposition of XML documents from the information in the database. Finally, the system offers a flexible mechanism for modifying and querying database contents using only valid XML documents, which are validated over the XML-Schema file's rules.

Hohenstein [Hohenstein '00] proposes data migration between relational and object-oriented database systems using the Federation Approach. Given any relational database, a migration program filing an object-oriented database is generated. The other direction is automated the same way.

Jahnke *et al.* [JaScZü '96] describe an integrated design environment that supports the migration process and overcomes major drawbacks of comparable approaches. They employed structure-oriented editors for the representation and

manipulation of the SQL and the ODMG schema. These structure-oriented editors internally store an abstract syntax tree representation of the edited schemas.

Mani and Lee [MaLe '02] studied various steps for translating XML to relational models while maintaining semantic constraints. Their work is based on the theory of regular tree grammars, which proves a useful formal framework for understanding various aspects of XML schema languages. They first studied two normal form representations for regular tree grammars. The first normal form representation, called NF1, was used in the two scenarios: (a) Several document validation algorithms use the NF1 representation as the first step in the validation process for efficiency reasons, and (b) NF1 representation can be used to check whether a given schema satisfies the structural constraints imposed by the schema language. The second normal form representation, called NF2, forms the basis for conversion of a set of type definitions in a schema language L1 that supports union types (e.g., XML-Schema), to a schema language L2 that does not support union types (e.g., SQL) and is used as the first step in XML to relational conversion algorithm.

Bohannon *et al.* [BoFrRoSi '02] propose LegoDB, a cost-based XML-to-relational mapping engine that addresses the problem of storing XML documents in relational databases. LegoDB explores a space of possible mappings and selects the best mapping for a given application (defined by an XML Schema, XML data statistics, and an XML query workload). LegoDB leverages existing XML and relational technologies it represents the target application using XML standards and constructs the space of configurations using XML-specific operations, and uses a traditional relational optimizer to obtain accurate cost estimates of the derived configurations.

Bierman [Bierman '00] proposes OIFML, a XML based language defined to dump and load the current state of ODMG-complaint databases. In this paper, he defined a new XML document type, OIFML, and showed how it can be used to specify ODMG-objects.

CHAPTER III

OVERVIEW OF HETEROGENEOUS DATABASES

In this chapter, we will discuss heterogeneous databases in regards to their logical structure, query languages, and visual modeling.

3.1 Relational databases

3.1.1 Logical structure

3.1.1.1 Tables

A relational database uses two-dimensional tables made up of rows and columns for representing the logical view of the data stored in a relational database. Each table represents some real world object; each row in the table represents one instance or entity of the object, and each column in a table models the attributes of the object. A column is associated with a data type, and size can be defined on the column depending on its data type. Each row of the table can be uniquely identified using primary key and unique keys defined on the column. Tables can be associated using foreign keys defined on the columns of the tables. *Figure 3.1* illustrates the *Address* table, which consists of four columns and two rows.

CustomerID	City	State	Country
1	San Marcos	TX	USA
2	Austin	TX	USA

Figure 3.1: Address Table

3.1.1.2 Keys

We can define three types of keys on columns of a table.

- *Primary key*

A column defined as a primary key will only allow unique values in all rows and cannot have null values. Only one primary key is allowed on a table. Two columns can be set as one primary key. In *Figure 3.2, OrderID* is the primary key.

- *Unique key*

A column defined as a unique key will only allow unique values in all rows. A column defined as unique key allows null values. A table can have multiple unique keys. In *Figure 3.2, ItemNo* is the unique key.

- *Foreign key*

A foreign key is a combination of columns with values based on the primary key value from another table. A foreign key constraint specifies that the values of the foreign key correspond to actual values of the primary key in the other table. In *Figure 3.2, CustomerID* is the foreign key.

OrderID (Primary Key)	CustomerID (Foreign Key)	Description	ItemNo (Unique)	Quantity
1	1	Desktop		1

Figure 3.2: Order Details Table

3.1.1.3 Data types

Each column is associated with a particular data type such as int, float or text. The size of column is determined from the data type of the column and the contents to be stored. In *Figure 3.3,* three columns are defined for table *Order.*

OrderID: data type of this column is *int* (i.e. integer) of length 4, which means it can allow four digit numbers.

Description: data type of this column is *varchar* (i.e. text) of length 20, which means it will allow 20 characters.

Price: data type of this column is float (i.e. fractional numbers) with length 4, which means it can allow four digit numbers with decimal at the second place.

In SQL Server (Relational DBMS), we can define following data types on the columns: ntext, nvarchar, integer, real, image, binary, bit, money, small int, and small datetime.

OrderID Int (4)	Description Varchar (20)	Price Float (4,2)
1	Desktop	1

Figure 3.3: Order Table

3.1.1.4 Associations

The number of ways two tables can be associated is called cardinality. Cardinality can be of the following types:

One to One: a column in table A is associated with at most one column in table B, and a column in table B is associated with at most one column in table A.

One to Many: a column in table A is associated with any number (zero or more) columns in table B, and a column in table B is associated with at most one column in table A.

Many to One: a column in table A is associated with at most one column in table B, and a column in table B is associated with any number (zero or more) columns in table A.

Many to Many: a column in table A is associated with any number (zero or more) columns in table B and a column in table B is associated with any number (zero or more) columns in table A.

3.1.2 Query languages

SQL is the most widely used query language for querying relational databases. We will discuss SQL with respect to three key issues:

- How to create tables using SQL
- How to alter existing table structure using SQL
- How to extract, modify, insert and update data in the tables using SQL

3.1.2.1 How to create tables using SQL

SQL uses the CREATE statement for creating tables. The CREATE statement allows users to specify the table name, column names, data type and size for each column, NON NULL columns, primary key, foreign keys and unique keys.
Example of CREATE statement:

```
CREATE TABLE Person (
name varchar(20) PRIMARY KEY,
address varchar(20)
);
```

This statement creates a table with name *Person* with two columns, *name* and *address.* Both columns are declared as of data type *varchar* with length *20.* Column *name* is the primary key of the table *Person.*

Example of CREATE statement:

```
CREATE TABLE Person (
name varchar(20) PRIMARY KEY,
address varchar(20) NOT NULL
);
```

This statement creates table *Person* with columns *name* and *address.* Column *name* is the primary key of the table *Person,* and the column *address* is set to NOT NULL, which means it can contain null values.

Example of CREATE statement:

```
CREATE TABLE Employee (
employeeID varchar(20) REFERENCES Employer (employeeID) ,
SSN varchar(15) UNIQUE
);
```

This statement creates table *Employee* with columns *employeeID* of data type varchar and size 20, and *SSN* of data type varchar and size 15. Column *employeID* is declared as a foreign key, which references column *employeeID* of table *Employer.*

Column *SSN* is declared as a UNIQUE key it will only allow unique values and null values.

REFERENCES keyword is used to declare foreign keys in an SQL CREATE statement or update column definitions using the SQL ALTER statement to relate two tables based on their column definitions.

3.1.2.2 How to alter table structure using SQL

The *ALTER* statement can be used to alter the table structure in one of the two ways: either by changing the definition of columns, or by adding columns to the existing table Consider the following table definitions:

Table Employee

Name VARCHAR (20)

Address VARCHAR (20)

Table Employer

EmployeeName VARCHAR (20)

The column definition of table *Employee* is modified using the following ALTER statement:

```
ALTER TABLE Employee MODIFY (
  Name varchar (20) REFERENCES Employer (EmployeeName) );
```

This statement updates the definition of column *Name* of table *Employee* to a foreign key, and the REFERENCES keyword indicates that the foreign key refers to column *EmployeeName* in table *Employer.*

Similarly, a new column *Position* can be added to the table *Employer* using the following ALTER statement:

```
ALTER table Employer ADD (
Position varchar (35));
```

3.1.2.3 How to extract, insert, delete, and update data in tables using SQL

SQL provides a SELECT statement to select table data, an INSERT statement to insert data into table, a DELETE statement to delete data from the table and an UPDATE statement to modify table data. We will look at each SQL statement and their usage:

SELECT statement: is used to select rows of data from any number of tables based on either a condition or without condition. For example, "SELECT * FROM TABLE Person;" retrieves all rows from table *Person.* The operator "*" is used as an alternative to retrieve all columns from the table instead of specifying each column. The SQL SELECT statement "SELECT name FROM TABLE Person;" retrieves all rows for column *name* from the table *Person.*

INSERT statement: is used to insert rows of data into a table. For example consider the *Person* table shown in *Figure 3.4.*

Name	Address
SAM	AUSTIN, TX

Figure 3.4: Person Table

The statement "INSERT INTO Person (name, address) VALUES ("Tim", "San Marcos, TX");" adds a new row to the table *Person*, and the result is shown in *Figure 3.5*.

Name	Address
SAM	AUSTIN,TX
TIM	SAN MARCOS,TX

Figure 3.5: Person Table

UPDATE statement: is used to update the row values of a table. For example the statement "UPDATE Person SET name= "JOHN";" updates the rows of table *Person* as shown in *Figure 3.6*.

Name	Address
JOHN	AUSTIN,TX
JOHN	SAN MARCOS,TX

Figure 3.6: Person Table

DELETE statement: deletes the rows from a table. For example, the statement "DELETE name, address FROM Person WHERE name= "JOHN";" will delete all rows from the table *Person*, whose [row, column] value is equal to "JOHN"

3.1.3 Visual modeling

ER diagrams are well suited for a visual representation of logical structures of relational databases. ER notation uses different symbols for representing various components of a relational database; for example, a table is represented by rectangle, and

relationships between tables are represented using lines ending with or without arrows. Consider the tables *Customer* and *Store* shown in *Figure 3.7* and *Figure 3.8*.

CustomerID (Primary key)	Name	Address	Phone

Figure 3.7: Customer Table

StoreID (Primary key)	CustomerID (Foreign key)	Location	ProductID

Figure 3.8: Store Table

In *Figure 3.9,* table *Customer* is represented with a rectangle; the primary key *CustomerID* is shown on the upper block of the rectangle, while non-primary keys are on the lower block of the rectangle.

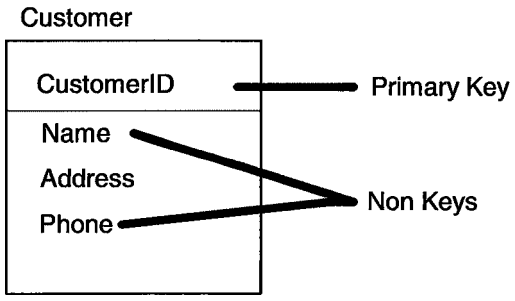

Figure 3.9: ER representation of Customer Table

In *Figure 3.10, Customer* and *Store* tables are related using their *CustomerID* column. *CustomerID* is the foreign key of table *Store* and the primary key of table

Customer. *Figure 3.10,* displays the ER diagram representing the relationship between tables *Customer* and *Store*.

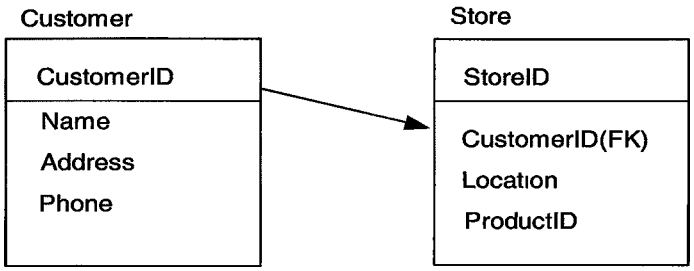

Figure 3.10: ER representation of relationship between tables

The line connecting two tables can have an arrow at its end depending on the type of relationship that exists between the columns of the tables.

One to One: ◄——►

One to Many: ◄——

Many to One: ——►

Many to Many: ——

3.2 Object-oriented databases

3.2.1 Logical structure

Object-oriented databases use class as the main component for specifying the structure of data storage. Classes are declared using object-oriented programming languages such as Java or C++. A class consists of method and variable declarations. Variables are used to hold the data and methods are used to access the variables of the class. The type of data stored in the variables depends on the data type of the variable. Classes specify the structure of data and the way of accessing data stored in the structure.

Objects are instantiated from classes, objects hold the actual data, and they are written to the persistent storage of an object-oriented database.

In the Java class declaration shown in *Figure 3.11,* a class is declared with name *Person*, and private variable *name* which is of data type *String* and two public methods *getName()* and *setName()*. The method *getName()* returns the value of variable *name* as *String*. The method *setName()* can be used to set the value of variable *name.*

```
public class Person {
    String name;
     public String getName() {
            return name;
     }
     public void setName(String name){
             this.name=name;
     }
}
```

Figure 3.11: Class Person

3.2.1.2 Keys

Unlike relational databases, programming languages do not provide facilities to declare primary key, foreign keys or unique keys on the variables. Keys for object database depend on the DBMS.

3.2.1.3 Data types

All variables of a class are associated with a particular data type. Depending on the programming language used a different naming convention is used for declaring the data type of a variable. Generic representations of data types are:

- Text: String, Char

- Integer: int, long
- Real: float, double

3.2.1.4 Cardinality

Classes are associated with each other using references. In the class declaration shown in *Figure 3.12*, a class with name *Employer* is declared along with member variable *emp.* Variable *emp* is an object of class *Employee.* Class *Employer* can access public methods of class *Employee* using object *emp.*

```
public class Employer {
        Employee emp; //reference
}
```

Figure 3.12: Class Employer

3.2.1.5 Inheritance

Programming languages allow users to specify inheritance between classes using keywords in their class declarations. Programming languages like Java use keywords *extends* and *implements* for inheritance. In *Figure 3.13,* class *Employer* is inheriting class *Person.* Class *Employer* can use the public methods *getName()* and *setName()* and variable *name* of class *Person* as its own methods and variable.

```
public class Person {
    public String name;
    public void setName(String name) {
     this.name=name;
  }
  public String getName() {
     return name;
      }
     }
public class Employer extends Person {
   String employerID;
  public void setEmployerDescription(String employerID, String name){
     this.employerID=employerID;
      this.name=name;
   }
}
```

Figure 3.13: Inheritance in Java

3.2.2 Query languages

Unlike relational databases, object-oriented databases do not rely on any other language such as SQL for accessing the database. A programming language used for creating objects of an object-oriented database is used as the query language. Programming languages are used to declare classes, instantiate objects, and manipulate data stored in the variables of objects. Some operations such as inserting, deleting and updating objects from the object-oriented database require importing libraries specific to the object-oriented database. In the next section, we will look at how to create classes, instantiate objects, and manipulate objects.

3.2.2.1 Creating classes

Creating a class depends on the programming languages used. The class declaration shown in *Figure 3.14* creates a class *Person* with variable *name.* The data type of variable *name* is *String.* Two methods are declared; one is for extracting the value stored in the variable *name* and another for changing the value stored in variable *name.*

```
public class Person {
  String name;
  public void setName(String name){
    this.name=name;
  }
   public String getName(){
    return name;
  }
}
```

Figure 3.14: Simple Java class structure

3.2.2.2 Creating objects

Objects are created using the class declaration. In *Figure 3.15,* an object *p* of class *Person* is created.

```
public class Test {
    public static void main (String args [ ]) {
        Test t=new Test ();
        Person p=new Person ();
     }
}
```

Figure 3.15: Object creation example

3.2.3 Extracting data from object

Data held in variables of objects are extracted using the methods defined in the class definition of the object. In the *Figure 3.17,* in class *Test* at line 3 we are creating

object *p* for class *Person* shown in *Figure 3.16*, and at line 4 we are retrieving the value of variable *name* using the *getName()* method.

```
public class Person {
    public String name;
    public Person(String name){
            this.name=name;
    }
    public void setName(String name){
            this.name=name;
    }
    public String getName(){
            return name;
    }
}
```

Figure 3.16: Class Person

```
Public class Test {
   Public static void main (String args [ ]) {
            Person p=new Person( "TOM");
            String name=p.getName();
   }
}
```

Figure 3.17: Object data extraction example

3.2.2.4 Updating objects

Methods are used for updating values of the variable of an object. In *Figure 3.18,* in class *Test* at line 4, an object *p* of class *Person* is instantiated. Using the *setName ( )* method of class *Person,* the value of the variable *name* of object *p* is set to "BOB."

```
public class Test extends ObjectDatabaseClass{
    public static void main (String args [ ]) {
       Test t=new Test();
        Person p= new Person();
        p.setName("BOB");
    }
}
```

Figure 3.18: Example for updating object data

3.2.3 Visual modeling

UML class diagrams are used to represent the logical design of object-oriented databases. The UML representation of class *Person* is illustrated in *Figure 3.19.*

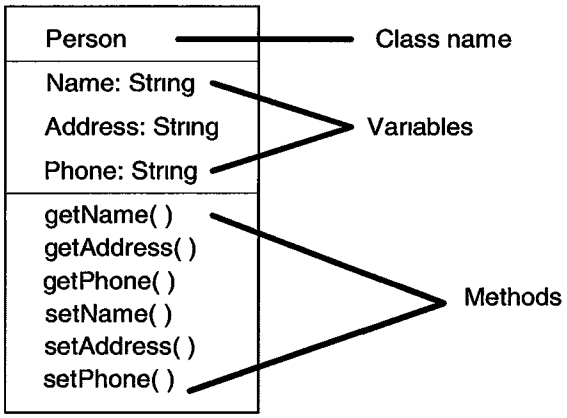

Figure 3.19: UML class diagram for class Person

The UML representation of association between two classes is illustrated in *Figure 3.20.*

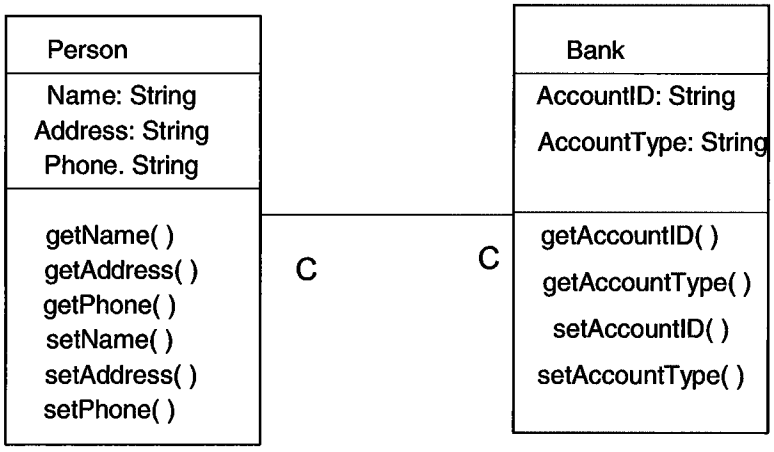

Figure 3.20: UML class diagram representing

Where *C* denotes the type of cardinality, *C* can will take one of following values: "0", "1", "*", or "+".

3.3 Hierarchical databases

A native XML database stores data in the format of XML documents. XML is defined an "Extensible mark up language (XML) describing class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]. By construction XML documents are conforming SGML documents" [w3c '00].

Before discussing the basics of XML, we will look at some of the definitions associated with XML, such as what constitutes a well-formed XML document and a valid XML document, in the next section.

3.3.1 Documents

A data object is an XML document if it is well formed [w3c '00].

3.3.1.1 Well-Formed XML

Documents that satisfy the XML 1.0 syntax specification [w3c '00] are known as well-formed XML documents. A well-formed document can be used without a validating schema such as DTD or XML schema. Basic features of a well-formed XML document are: it contains a root element called the document element; no part of which appears in any content of another element; it contains one or more elements and all components displaying the tree structure are in a parent child relationship. A well-formed XML document can be comprised of one to three parts: an optional prolog, which may contain important information about the rest of the data; the body, which consists of one or more elements in the form of a hierarchical tree, and an optional "miscellaneous" epilog that follows the element tree.

3.3.1.2 Valid XML

A XML document is considered as a valid XML document if it is well formed and satisfies the constraints imposed by schemas such as DTD or XML schema. Validation using a DTD or XML schema ensures that the element parent-child relationships are respected, that attributes have valid values, that all referenced entities have been properly defined, and that numerous other specific validity constraints are obeyed.

3.3.2 Basics of XML

Let us break up the discussion of XML in the following subsections:

- XML Processing
- XML Validation

3.3.2.1 XML Processing

Hierarchical structure of XML documents is accessed using parsers. There are two types of parsers:

- Non-validating parser: the parser merely ensures that a data object is a well-formed XML
- Validating parser: the parser uses a DTD or XML Schema to ensure the validity of a well-formed data object's form and content.

Both parsers search for illegal elements, improper nesting of tags, or any kind of illegal constructs. When they encounter an error, they should either terminate the parsing or report the error to the application. It is up to the implementation of the parser whether it should terminate or proceed with the processing whenever it encounters any errors. Implementation of the parser is done in two ways:

Event driven parser (SAX): a top down approach, in which the parser processes one individual component at a time while starting sequentially from the top of XML document. XML data is processed sequentially and events are signaled for each individual node; these events are be used by an application to process the data. This event-driven approach can handle immense XML documents without requiring massive amounts of memory. SAX (Simple API for XML) is a standard interface for event driven XML parsers that was developed by members of the XML-DEV email list [Megginson '02].

Tree-based parser (DOM): Constructs a tree representation of the entire document and provides access to individual nodes in the tree. The entire tree is usually constructed in memory, which makes it easier to use tree traversal algorithms to process the data, thus improving random access. However, if the XML document is large, a huge amount of virtual memory is required.

The standard tree object model used to access XML data is the Document Object Model (DOM) [w3c '97]. This neutral interface allows dynamic access and modification of structured data, such as XML from different languages and platforms.

3.3.2.3 XML Validation

As explained earlier an XML document will be valid if it satisfies the constraints specified by a DTD or XML schema. A DTD or XML schema defines a set of rules which can be incorporated within XML document or exist as a separate document, and which describe the structure and allowed content of the XML data. In addition, the presence of DTD or XML schema helps:

- To ensure that all necessary elements and attributes are present in the document
- To ensure that unnecessary elements or data are not present
- To enforce proper parent-child or other relationship between elements

3.3.2.3.1 DTD (Document type declaration)

Document Type Definitions use formal grammar (EBNF) to describe the structure and syntax of an XML document, including the permissible values for much of that document's content. DTD is associated with an XML document using the DOCTYPE

declaration, and an XML parser uses this declaration to access the DTD associated with the XML document.

Issues related with the DOCTYPE declaration are: this declaration may appear only once in an XML document; it must follow the document's XML declaration, if any, and precede any elements or character data content. Although this declaration is optional for simple well-formed XML documents any document that needs to be validated using a DTD must have a DOCTYPE declaration.

There are two forms of the DOCTYPE declaration (SYSTEM and PUBLIC sources):

<! DOCTYPE doc_element SYSTEM location [internal_subset] >

And

<! DOCTYPE doc_element PUBLIC identifier location [internal_subset] >

The SYSTEM keyword is used to explicitly specify the location of the DTD. It uses the URL form of a URI reference.

Example of using keyword SYSTEM

<! DOCTYPE Book SYSTEM "http://www.bookdealership.com/DTDs/books.dtd">

The PUBLIC identifiers are limited to internal systems and SGML legacy applications.

Example of using keyword PUBLIC

<! DOCTYPE Book PUBLIC "BookDealer/DTDs/books.dtd">

The first parameter of any DOCTYPE declaration is the document element. For example,

<?xml version ='1.0'?>

<! DOCTYPE Book SYSTEM

"http://www.bookdealership.com/DTDs/books.dtd">

After DOCTYPE, *Book* is the document element and this element is the root of the XML document. A sample XML document will look like this:

<Book>

……

</Book>

There can be only one <Book> element for a well-formed XML document. Basic components of DTD are:

1. ELEMENT: declares an XML element type name and its permissible sub-elements ("children").
2. ATTLIST: declares XML element attribute names, plus permissible and/or default attribute values.
3. ENTITY: declares special character references, text macros (much like C/C++ #define statement), and other repetitive content from external sources (like a C/C++ #include).
4. NOTATION: declares the external non-XML content (for example, binary image data) and an external application that handles that content.

3.3.2.3.1.1 Element type declarations

Elements are responsible for forming the logical structure of an XML document. To declare an ELEMENT, use:

<! ELEMENT name content_category>

<! ELEMENT name (content_model)>

The *name* parameter corresponds to the name of the element and it must be a legal XML name. The *content_category* and *content_model* parameters describe what kind of contents may appear within the elements of the given name. Possible values of *content_category* are:

- Text only: contains any text (character data), but no child elements. For example, for the element declaration *<! ELEMENT Customer (#PCDATA)>* we have,

  <elementName> text</elementName>.

- Element only: contains child elements, and no text outside of those children. For example, for the element declaration *<! ELEMENT elementName (childName) >*, we have *<elementName><childName>child text</childName></elementName>*.

- Mixed content: may contain a mixture of child elements and/or text data. For example, for the element declaration *<! ELEMENT elementName (#PCDATA | childName) >*, we have *<elementName>text <childName> child text </childName> </elementName>*.

- Any content: any element or text may appear inside the element when defined this way, and in any order. For the element declaration *<!ELEMENT Customer ANY>*, we can have *<Customer> text here </Customer>* or *<Customer><Customer><Customer>text here</Customer><Customer> <Customer>*.

- None (or empty): may not contain any text or child elements- only element attributes are permitted. For the element declaration *<! ELEMENT elementName EMPTY>*, we have

  <elementName/>

Two kinds of element lists may appear within content models:

- Sequence lists: operator (,) is used to separate child elements within the ELEMENT declaration; the result of this is the child elements must appear in the order specified. For the element declaration *<! ELEMENT elementName (childElement1, childElement2)>*, we have*<elementName><childElement1/> <childElement2/></elementName>*.
- Choice lists: operator (|) is used to separate child elements within the ELEMENT declaration; the result of this is that only one child element exists out of all child elements. For the element declaration *<!ELEMENT elementName (childElement1 | childElement2)>*, we have *<elementName><childElement2/></elementName>*.

3.3.2.3.1.2 Cardinality

The cardinality operator defines how many child elements may appear in a content model. There are four types of cardinality operators:

- None: The absence of a cardinality operator character indicates that one, and only one, instance of the child element is allowed (and it is required)
- ?: Zero or one instance- optional singular element
- *: Zero or more instances- optional element(s).
- +: One or more instances-required element(s).

Consider the example of an element declaration having cardinatility:

*<! ELEMENT elementName (childElement1 *, childElement2) >*

We have,

<elementName><childElement1/><childElement1/><childElement2/>

</elementName>

In the above XML document, *childElement1* occurs more than once and *childElement2* occurs only once.

3.3.2.3.1.3 Attribute (ATTLIST) declaration

Attributes associate meta-data or properties with the elements. An XML document can be built using attributes only. To declare attributes of an element, we use the ATTLIST declaration as shown in the following example:

<!ELEMENT elementName EMPTY>

<!ATTLIST elementName

attrName1 attrType1 attrDefault1 defaultValue1

attrName2 attrType2 attrDefault2 defaultValue2

...

attrNameN attrTypeN attrDefaultN defaultValueN>

attrDefault: represent the option used for specifying the default values.

attrType: represent the type of attribute

defaultValue1: represents the default values that can be specified for the element.

3.3.2.3.1.3.1 Attribute Defaults (*attrDefault)*

There are four attribute default values

1. #REQUIRED: imposes the constraint on the specified attribute that it must appear in every instance of the element.
2. #IMPLIED: the attribute can be omitted from the element instance.
3. #FIXED (plus default value): the attribute can be omitted; if the attribute appears in the element instance, it should take the default value specified. If the attribute does not appear, then default value is taken.
4. Default value (only, no keyword supplied): the attribute can be omitted from the element instance. If it does appear, it may be any value conforming to its attribute type. If the attribute does not appear, the parser may supply the default value.

3.3.2.3.1.3.2 Attribute Types (*attrType)*

The available attribute types are:

- CDATA: Most attribute values are nothing more than plain text. These attributes are declared using the CDATA type.

  Example:

  <!ATTLIST elementName attr CDATA #REQUIRED>

  This example states that an element type (named elementName) has a required single attribute (named attr) that has a text string for its value.

- Enumerated values: One of a series that is explicitly defined in the DTD.

  Example:

  <!ELEMENT PersonName (FirstName, LastName)>

<!ATTLIST PersonName

title (Mr | Ms) #IMPLIED

suffix (Jr | Sr) #IMPLIED >

Both attributes are name char text strings, but each must have a value that exactly matches one of the values shown in above declarations.

- ID: A unique identifier for each instance of this element type; it must be valid XML name. Attributes using the ID type provide a unique identifying name for a given instance of an element. Each element type may use only one ID attribute.
- IDREF: We can use an IDREF attribute to establish a link from an element to a different element using the ID attribute specified for that element.
- NMTOKEN: A name token that is a text string that conforms to the XML name rules, except that the first character of the name may be any valid name char.
- NMTOKENS: A list of NMTOKEN values separated by white space character(s).
- ENTITY: The name of a pre-defined entity.
- ENTITIES: A list of ENTITY names separated by white space character(s).
- NOTATION: A notation type that is explicitly declared elsewhere in the DTD.

3.3.2.3.1 Limitations of DTDs

Some limitations of DTDs are: non-XML syntax; DTDs are not extensible; only one DTD may be associated with each document; DTDs do not properly support XML Namespaces; very weak data typing; no OO-type object inheritance; data can ignore the external DTD by using the internal subset; no DOM support; relatively few, older more expensive, tools.

3.3.2.3.2 XML schema

XML schemas are based on XML syntax. An XML schema provides the following benefits over DTDs: it can use fragments of other XML schemas; it can define complex structures that can be reused in other schemas, one can derive their own new data types from existing ones; multiple XML schemas can be referred from one schema document; and schemas are closer to object-oriented languages.

The Main components of XML schemas are:

- Namespaces
- Element declaration
- Attribute declaration
- Complex type declaration
- Simple type declaration
- Global elements and attributes

3.3.2.3.2.1 Namespaces

Namespaces provide the ability to re-use already existing schemas; in order to use the built in types available in XML schema namespace, we use following declaration:

`xmlns:xsd="http://www.w3.org/2001/XMLSchema"`

In the above declaration, xmlns represents the XML namespace, and "http://www.w3.org/2001/XMLSchema" points to the location of the XML schema document which is having declaration for built in types. To use the built in types, we need to add the prefix "xsd" in front of each built in type used, which indicates that the

built in type belongs to that particular namespace. Similarly, we can use declarations of other XML schemas in our XML schema using namespace declarations.

3.3.2.3.2.2 Element declarations

Elements provide the logical structure for XML schemas. Let us take a look at various ways of declaring ELEMENT.

The element declaration *<xsd:element name="FirstName" type="xsd:string" />* declares an element with name *FirstName* of data type "string".

The element declaration shown in *Figure 3.21* declares the element *Customer* with two child elements, *FirstName* and *LastName,* both of data type "string".

```
<xsd:element name="Customer">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="FirstName" type="xsd:string" />
                        <xsd:element name="LastName" type="xsd:string" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 3.21: Customer element declaration

The element declaration shown in *Figure 3.22* declares an element *quantity* of data type integer, which can hold a maximum value of 100.

```
<xsd:element name="quantity">
        <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                        <xsd:maxExclusive value="100" />
                </xsd:restriction>
        </xsd:simpleType>
</xsd:element>
```

Figure 3.22: Quantity element declaration

Attributes that can be used in element declarations are:

type: can be complex type, user defined simple type or built-in type.

ref: refers to the global element declaration

minOccurs: determines the minimum number of times an element can appear in an instance document. The default value is 1.

maxOccurs: determines the maximum number of times an element can appear in an instance document. The default value is 1.

3.3.2.3.2.3 Attribute declarations

Attributes for elements can be declared in various ways, and some declaration types are explained below. The attribute declaration shown in *Figure 3.23* declares attribute *customerID* for element *Customer*.

```
<xsd:element name="Customer">
        <xsd:attribute name="customerID" />
        ...
</xsd:element>
```

Figure 3.23: Customer element declaration

The instance document for the element declaration above is shown in *Figure 3.24*.

```
<Customer customerID="21">
...
</Customer>
```

Figure 3.24: XML instance of Customer element

Some issues related to attribute declarations:

- An *attribute* declaration always appears at the end of corresponding element declaration.

- An *attribute* declaration cannot be of type complex.

Attributes that can be used in attribute declarations are:

- *use*: if this attribute is specified, then the attribute of the element is required to appear in the instance document.
- *type:* the attribute declaration can use this attribute to use the in built or pre-declared simple type definition.

3.3.2.3.2.4 Complex type declarations

Complex type declarations are well suited for declaring reusable structures. Complex type definitions mainly contain a set of element declarations, element references, and attribute declarations. Complex types provide a convenient way to define the content models for elements. In *Figure 3.25,* complexType *Person* is declared with two child elements, *FirstName* and *LastName.*

```
<xsd:complexType name="Person">
      <xsd:sequence>
            <xsd:element name="FirstName" type="xsd:string" />
            <xsd:element name="LastName" type="xsd:string" />
      </xsd:sequence>
</xsd:complexType>
```

Figure 3.25: Person element declaration

Attributes that can be used in complex type declarations are:

- *block:* prevents other complex types, and elements to inherit properties of complex type declarations.
- *mixed:* allows mixed contents in instance of element declaration.
- *name:* name of the complex type declaration.

3.3.2.3.2.5 Simple types

Simple types refer to atoms of data (whether element content or attribute values). A simple type is declared using the <simpleType> tag; it can be derived from existing simple types using keywords *restriction* and *extension,* or it can be derived from built-in simple types. There are two varieties of data types: built-in types and user-derived types. Built-in types are made available for use by including the XML schema namespace. There are two types of built-in types: Primitive types like string, integer and float, for example the statement "*<xsd:element name= "address" type= "xsd:string"/>*" declares an element with name *address* and data type as string obtained from namespace denoted by xsd; Derived types are derived from definitions of other data types, and are created by restricting existing data types. The type of derived types is same as that of their base type. The simple type declaration shown in *Figure 3.26* declares a derive type *Street* from *string.*

```
<xsd:simpleType name="Street">
        <xsd:restriction base="xsd:string">
        .....
        </xsd:restriction>
</xsd:simpleType>
```

Figure 3.26: Street simple type declaration

User-derived types are derived by the author of the schema, and are particular to that schema.

3.3.3 Query Languages

An XML document can be queried using XPATH or XQUERY.

3.3.3.1 XPath

The XML Path Language (XPath) is a querying language used to address specific parts of an XML data object as nodes within a tree. This language handles XML data as paths within an abstract hierarchical tree structure of nodes and a current context node, rather than using the superficial syntax of tags and attributes.

3.3.3.2 XQuery

XQuery is the proposed new query language that is expressed in non-XML syntax. It is used to make queries against XML data, using the proposed XPath 2.0 expressions, plus many expressions that are similar to SQL query expressions.

3.3.4 Visual modeling

Figure 3.27 displays the XML SPY representation for XML Schema element *Customer*.

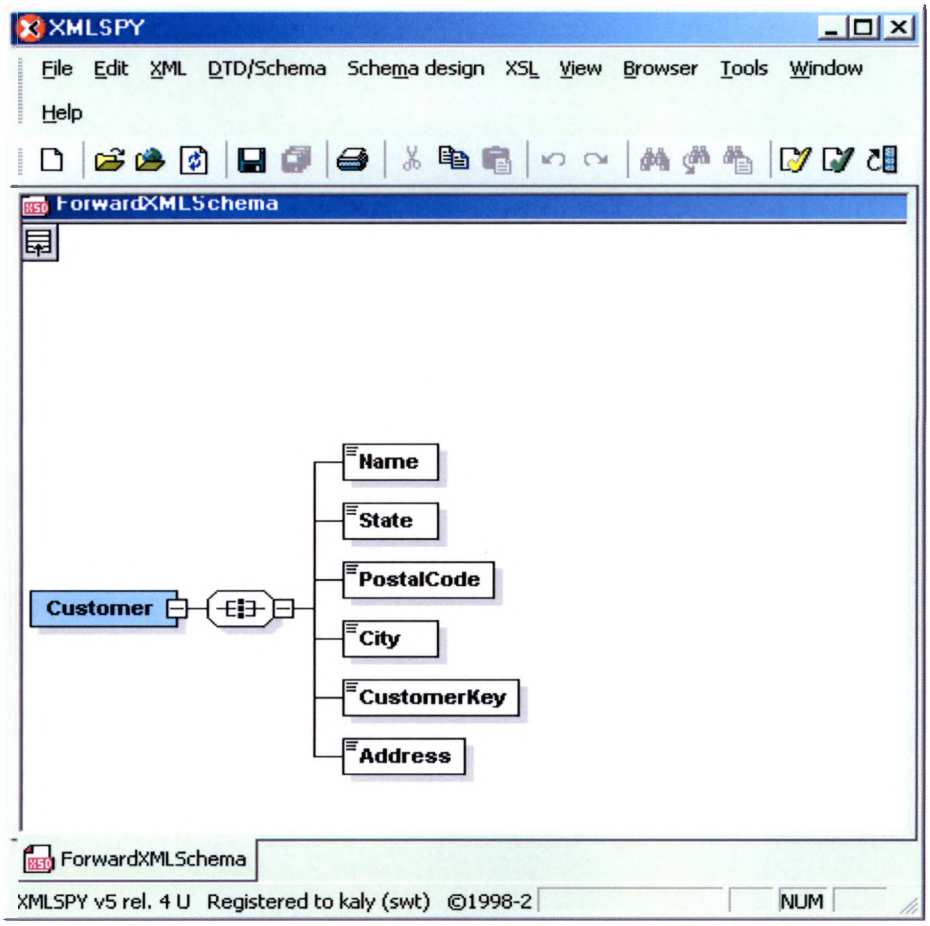

Figure 3.27: XML SPY representation of Customer element

CHAPTER IV

RESEARCH METHOD

In this chapter we will discuss the framework designed for database transformation between heterogeneous databases using XML. The framework consists of structure DTD, data DTD, forward transformation, and reverse transformation. We will first look at the high level overview of database transformation process, followed by an explanation of structure DTD and data DTD used for defining the XML tags. These tags hold the heterogeneous database structure and data in a hierarchical structure of XML documents. Then we will discuss the forward transformation process where heterogeneous databases structure and data are mapped to the hierarchical structure of XML documents specified by structure DTD and data DTD. Also we will discuss the reverse transformation in which we will explain mapping techniques for mapping XML documents to heterogeneous database structure and data.

4.1 Overview of database transformation process

In this section, we will look at basic steps involved in the database transformation process between heterogeneous databases. The transformation process consists of two stages: forward transformation and reverse transformation.

In forward transformation, heterogeneous database structure and data are mapped to the hierarchical structure of XML documents specified by the structure DTD and the data DTD. The hierarchical structure for storing database structure information is generated according to the tag constraints specified in the structure DTD. In general a

heterogeneous database structure consists of a main component representing some real world object, inheritance between components, and association between components. The usual structure of the main component consists of its name, attributes, keys and other structural information. The attributes consist of name, data type, and size, which actually depend on the database theoretical base. The hierarchical structure for storing database data is generated according to the tag constraints specified in the data DTD. A heterogeneous database data consists of actual values, which are embedded in the structure of the database.

In reverse transformation, database structure and data represented by XML documents are mapped to the database structure and data according to the destination heterogeneous database theoretical base. The first step is to generate the heterogeneous database structure which includes data type mapping of source heterogeneous data type naming conventions to destination heterogeneous database data type naming conventions. The second step is to generate the heterogeneous database data.

Figure 4.1 outlines the complete transformation process. Descriptions of each component involved in the transformation are as follows:

Relational database: represents relational databases such as SQL Server 2000 and MYSQL.

Object-oriented database: represents an object-oriented database such as Ozone.

Hierarchical database: represents a hierarchical database such as IPEDO.

Relational DB Structure & Data Forward Transformer: represents the transformation logic for transforming relational database structure and data to XML documents.

Object-oriented DB Structure & Data Forward Transformer: represents the transformation logic for transforming object-oriented database structure and data to XML documents.

Native DB Structure & Data Forward Transformer: represents the transformation logic for transforming native XML database structure and data to XML documents.

XML Documents: represents the XML documents. The XML structure document represents the database structure and XML data document represents the database data.

Data type Transformer: represents the mapping of data type naming convention of source heterogeneous database to data type naming convention of destination heterogeneous database.

XML documents to Relational DB Transformer: represents the transformation logic for transforming XML documents to relational database structure and data.

XML documents to Object-oriented DB Transformer: represents the transformation logic for transforming XML documents to object-oriented database structure and data.

XML documents to Native XML DB Transformer: represents the transformation logic for transforming XML documents to native XML database structure and data.

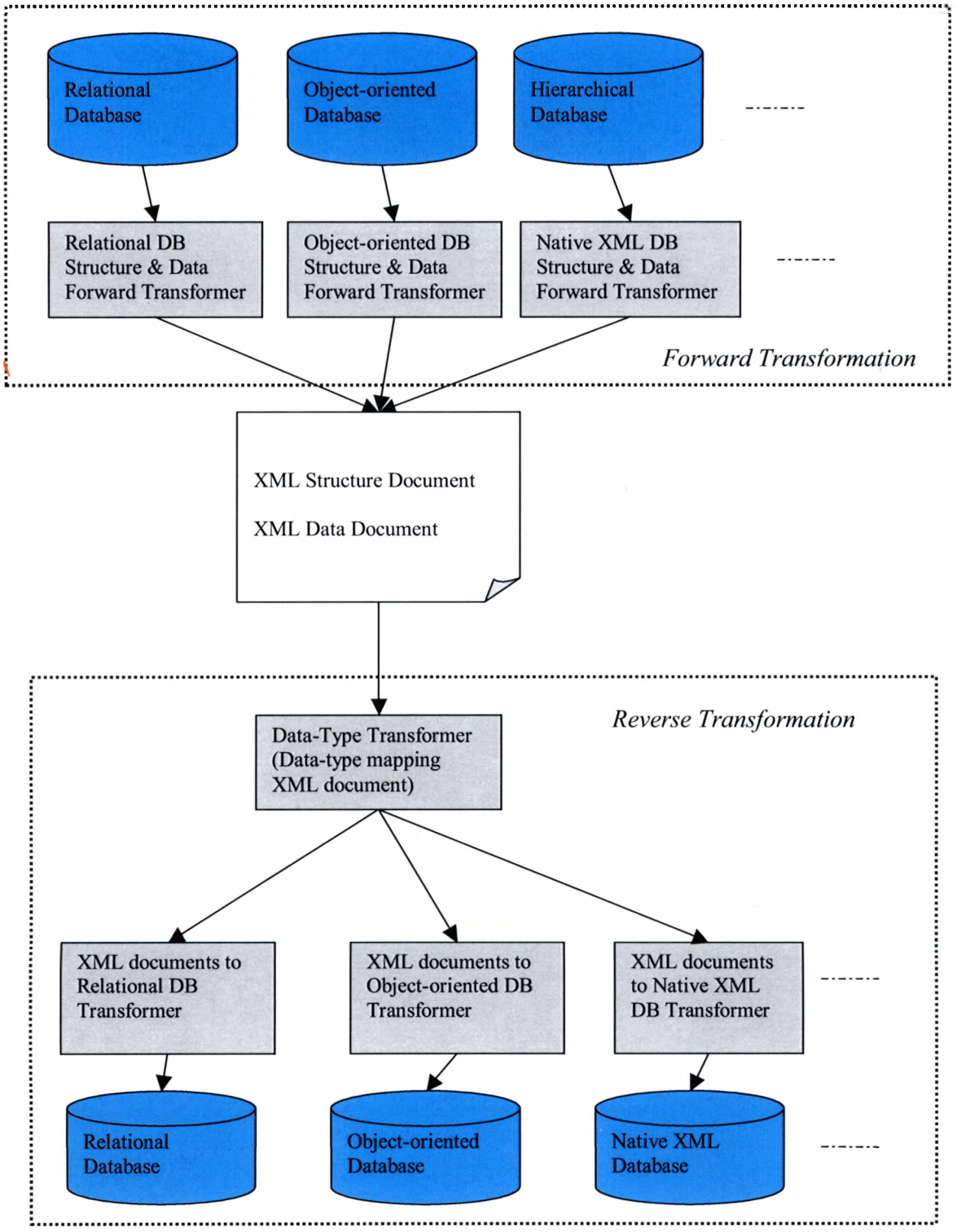

Figure 4.1: Basic overview of database transformation process

4.2 DTDs

The backbones of the database transformation framework are structure DTD and data DTD. Structure DTD holds the database structure information. Structural information consists of a main component description such as its attribute descriptions, inheritance between the components, and association between components. Data DTD holds the actual data of the database independent of the database structure.

4.2.1 Structure DTD

Structure DTD defines set of tags to constrain the database structural information in a hierarchical order of an XML document independent of the database theoretical base. Structure DTD can be extended to support new database formats by adding new tag definitions according to the new database theoretical base, thus enabling less overload for transforming the database between the existing database formats and the new database format. At present the tag definitions of structure DTD are capable of representing relational databases, object-oriented databases and native XML databases structural information. Structure DTD tags are shown in *Figure 4.2, Figure 4.3, Figure 4.4,* and *Figure 4.5* and statements included within <!-- --> are comments describing the element declaration above it.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT DatabaseStructure (ComponentList, RelationshipList?)>
    <!--- DatabaseStructure is the root element of the XML structure document. It
          includes as children the ComponentList element and the Relationship element.
          The ComponentList element holds the main components structural information
          of a database. The Relationship element holds relationship information between
          the main components -->
<!ATTLIST DatabaseStructure
              databaseName CDATA #IMPLIED
              databaseType CDATA #IMPLIED>
    <!-- databaseName attribute represents the name of the heterogeneous database to be
         transformed. databaseType attribute represents the source database theoretical
         base, whether it is a relational database, an object-oriented database or a
         hierarchical database -->
<!ELEMENT ComponentList (Component)+>
     <!-- ComponentList encloses all the Component tags which represent the main
          components of the database -->
<!ELEMENT Component (PrimaryKey?, ForeignKey?, UniqueKey?, AttributeList?,
                                 InheritanceList?)>
     <!—The Component element represents the main component of the database. For
          relational databases it represents a table, a class for an object-oriented database
          and a schema element for a native XML database. The Component element consists of
          children as PrimaryKey, ForeignKey, UniqueKey, AttributeList and
          InheritanceList -->
<!ATTLIST Component
              name CDATA #REQUIRED>
     <!--The name attribute of the element Component holds the name of the component. For a
          relational database, it represents the table name. For an object-oriented database it
          represents the class name and for a native XML database it represents the XML schema
          element name -->
```

Figure 4.2: Structure DTD Part-1

```
<!ELEMENT PrimaryKey (PK)+>
    <!—The PrimaryKey element holds the list of primary keys of the Component. It is an
        optional element. The PrimaryKey element has one child element PK. -->

<!ELEMENT PK EMPTY>
    <!—The element PK represents the primary key of the component. It can occur multiple
        times for each primary key of the component.-->
<!ATTLIST PK
            name CDATA #REQUIRED>
      <!--The name attribute holds the name of the primary key. -->

<!ELEMENT ForeignKey (FK)+>
      <!—The ForeignKey element represents the list of foreign keys of the component. It is an optional
      element and has one child element FK -->

<!ELEMENT FK EMPTY>
    <!--The element FK holds information from a foreign key of the component and can occur multiple
        times for each foreign key of the component. -->
<!ATTLIST FK
            name CDATA #REQUIRED
            referComponent CDATA #REQUIRED
            referField CDATA #REQUIRED >
    <!-- The name attribute holds the name of the foreign key, referComponent represents the referring
        component to which this key is indexing, and referField attribute represents the field in
        the referring component. -->

<!ELEMENT UniqueKey (UK)>
      <!--The UniqueKey element refers to the list of unique keys of the component and has one child
          element UK. -->
```

Figure 4.3: Structure DTD Part-II

```
<!ELEMENT UK EMPTY>
    <!--The element UK represents list of unique keys of the component. It can occur multiple
      for each unique key of the component. -->
<!ATTLIST UK
        name CDATA #REQUIRED>
    <!-- The name attribute represents the name of the unique key of the component -->

<!ELEMENT AttributeList (Attribute+)>
    <!--AttributeList represents the list of attributes of the component and has one child element
      Attribute.-->

<!ELEMENT Attribute EMPTY>
    <!—The Attribute element represents an attribute of the component -->
<!ATTLIST Attribute
        name CDATA #REQUIRED
        datatype CDATA #REQUIRED
        size CDATA #REQUIRED
        isNull CDATA #REQUIRED
        isAutoIncrement CDATA #REQUIRED>
    <!-- The name attribute holds the name of the attribute, datatype holds the data type of
    the attribute, size refers to the size of the attribute, isNull holds information whether or
    not the attribute can posses NULL values, and isAutoIncrement specifies the property of
    the attribute: whether or not it will be automatically increased with insertion of new
    values.
      -->

<!ELEMENT InheritanceList (Parent)+>
    <!-- InheritanceList represents the list of components inherited by this component. It has
    one child element Parent. -->
```

Figure 4.4: Structure DTD Part-III

```
<!ELEMENT Parent EMPTY>
    <!--The Parent element represents the parent of this component -->
<!ATTLIST Parent
            name CDATA #REQUIRED>
    <!-- The name attribute holds the name of the parent of this Component -->

<!ELEMENT RelationshipList (Relation)+>
    <!-- The RelationshipList element represents list of the relationships that exist between the
        components of the database.-->

<!ELEMENT Relation EMPTY>
    <!-- The Relation element represents the relationship information that exists between two
        components. -->
<!ATTLIST Relation
            name CDATA #REQUIRED
            relationType CDATA #REQUIRED>
        <!-- The name attribute represents the name of the Relation, relationType attribute
        represents  the type of relationship between two components, relationType takes one of
        the following values ONETOONE, ONETOMANY, MANYTOONE, MANYTOMANY
            -->
```

Figure 4.5: Structure DTD Part-IV

4.2.2 Data DTD

Data DTD defines set of tags to constrain the database data in a hierarchical order of a XML document independent of the database theoretical base. In the data DTD shown in *Figure 4.6* and *Figure 4.7,* sentences starting with <!-- --> enclose tag definition above it.

```
<!ELEMENT DatabaseData (ComponentList)>
    <!-- DatabaseData is the root element of the XML document and encloses all the database
        data within its child tags in the appropriate hierarchy. It has one child element
        ComponentList -->
<!ATTLIST DatabaseData
            databaseName CDATA #REQUIRED
            databaseType CDATA>
      <!-- The databaseName attribute holds the name of the database in the source DBMS, and
      the  databaseType attribute is holds the database type whether it is a relational database, a
      object-oriented database or hierarchical database-->

<!ELEMENT ComponentList(Component+)>
     <!-- The ComponentList element represents the list of components. -->

<!ELEMENT Component (AttributeDataList)>
    <!--The Component element represents individual components of the database and has one
    child AttributeDataList-->
<!ATTLIST Component
            name CDATA #REQUIRED>
            <!--  The name attribute holds the name of the component.-->

<!ELEMENT AttributeDataList (ObjectData)+>
     <!-- AttributeDataList represents the list of attributes of the component and has one child
        ObjectData.-->

<!ELEMENT ObjectData (Contains)+>
     <!--  ObjectData represents a particular set of  data of the component. For a relational
        database, it represents a row of the table; for an object-oriented database it represents a
        object; and for a native XML database it represents an element-->
```

Figure 4.6: Data DTD Part-I

```
<!ELEMENT Contains EMPTY>
    <!-- The Contains element holds the name and value of each attribute of the component.
    For a row of table in a relational database it holds the [row, column] value. For an object of
    an object-oriented database it holds the variable name and its value. For a native XML
    database it holds the element name and its value-->
<!ATTLIST Contains
            attributeName CDATA #REQUIRED
            value CDATA #REQUIRED>
    <!-- attributeName represents the name of the attribute of the component, and value
        attribute holds the value of the attribute -->
```

Figure 4.7: Data DTD Part-II

4.3 Forward transformation

In this section, we will discuss generic mapping of heterogeneous database structure and data to the hierarchical structure of XML documents. The tag definitions for the XML documents are defined in structure DTD and data DTD. We will first look at the database structure mapping and then database data mapping.

4.3.1 General database structure mapping

In database structure mapping, we will disuss the basic concept involved in the mapping of a heterogeneous database structure to the hierarchical structure of an XML document specified by tags of structure DTD. Depending on the database theoretical format, the structure of the component varies; for example, in relational databases tables are the component, while classes are the structural component in object-oriented

databases and elements for XML schema. Usually a component consists of a name, attributes and keys; if the name of the component is *myname,* its equivalent hierarchical representation is:

<ComponentDescription name = *"myname"* >

Each attribute of the component consists of a name and data type. Depending on the database theoretical base, each attribute can have other properties such as size, whether it can hold NULL values, and whether it can be used as a key. The hierarchical representation of the attribute description is shown in *Figure 4.8.*

```
<ComponentDescription name="myname">
        <AttributeList>
                <Attribute name="attributename" datatype="attributedatatype"
isNull="yes/no" isAuto="yes/no" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.8: Attribute hierarchical representation

Components can have keys defined on their attributes. Commonly defined keys are primary keys, unique keys and foreign keys. The hierarchical representation of component keys description is shown in *Figure 4.9.*

```
<ComponentDescription name="myname">
        <PrimaryKey>
                <PK name="primarykey" />
        </PrimaryKey>
        <ForeignKey>
                <FK name="foreignkey" referComponent="referComp" referField="referField" />
        </ForeignKey>
        <UniqueKey>
                <UK name="uniquekey" />
        </UniqueKey>
</ComponentDescription>
```

Figure 4.9: Keys hierarchical representation

Components are associated with each other in relationships such as one to one, one to many, many to one and many to many. The hierarchical representation of association between components is shown in *Figure 4.10*.

```
<RelationshipList>
        <Relation from="component1" to="component2" relationtype="ONETOONE" />
</RelationshipList>
```

Figure 4.10: Relationship hierarchical representation

Components can inherit the structure of other components. The hierarchical representation of inheritance between components is shown in *Figure 4.11*. In *Figure 4.11 component2* is inheriting *component1*.

```
<ComponentDescription name="component2">
        <InheritanceList>
                <Parent name="component1" />
        </InheritanceList>
</ComponentDescription>
```

Figure 4.11: Inheritance hierarchical representation

4.3.2 General database data mapping

In database data mapping, data stored in components is extracted and embedded within the tags specified by the data DTD. The hierarchical representation of the component data is shown in *Figure 4.12*.

```
<ComponentData name="componentname">
        <AttributeDataList>
                <ObjectData>
                        <Contains name="attributename" value="attributevalue" />
                </ObjectData>
        </AttributeDataList>
</ComponentData>
```

Figure 4.12: Database data hierarchical representation

In the *Figure 4.12,* the *ComponentData* tag represents the component of the database such as a table for relational database, class for object-oriented database and schema element for native XML database; the *componentname* attribute represents the name of the component. *AttributeDataList* tag represents the list of data sets. The *ObjectData* tag represents each set of data; for example, for tables it represents a row, objects for a class and element for a schema. The *Contains* tag represents the name value pair for example [row column] value for a table, variable and its value for an object, element and its value for a schema.

After looking at the basic mapping of heterogeneous database structure and data to XML documents, in the following sections we will discuss mapping for relational database, object-oriented database and native XML database.

4.3.3 Relational database to XML documents mapping

In this section we will discuss mapping of relational database structure and data to XML documents.

4.3.3.1 Structural Mapping

Structural mapping involves mapping of table structure information. Table structure includes table descriptions, relationship between tables, and keys defined on tables. Table description consists of table name and column description of the tables. Column descriptions consists of data type of columns, size of columns, and other properties of columns like whether columns allow NULL values, default values of

columns, and auto increment property of the columns. As explained in 3.1.3, SQL can be used to define the table structure.

CREATE TABLE *table1* (

Column1 data-type1 (size1) NOT NULL

Column2 data-type2 (size1) PRIMARY KEY

Column3 data-type2 (size1) UNIQUE KEY

Column4 data-type1 (size2) REFERENCES *table2 (column)*

Column5 data-type2 (size1)

...

Columnn data-type2 (size1)

);

Figure 4.13: SQL CREATE statement

The SQL CREATE TABLE statement shown in *Figure 4.13* creates a table with name *table1* and defines columns for the table. *Column1* is of data type *data-type1*, size is *size1* and it does not allow null values; *Column2* is of data type *data-type2*, size is *size1* and it is the primary key of *table1; Column3* is of data type *data-type2*, size is *size1* and it's a unique key; *Column4* is of data type *data-type1*, size is *size2* and it is a foreign key referencing column *column* in table *table2; Column5* thru *Columnn* are other columns of data type *datatype2* and size *size1* and they all allow null values.

The table mapping for SQL CREATE statement shown in *Figure 4.13* is as follows:

The name of the table is mapped to the *name* attribute of the *ComponentDescription* tag; each column is mapped to the *Attribute* tag, the name of the

column is mapped to the *name* attribute of the *Attribute* tag, the data type of *column* is mapped to the *data-type* attribute of *Attribute* tag and the size of column is mapped to the *size* attribute of *Attribute* tag. Other properties are similarly mapped; primary keys of the table are mapped to the *PK* tags, unique keys of the table are mapped to the *UK* tags, foreign keys are mapped to *FK* tags. The hierarchical representation of the table structure shown in *Figure 4.13* is shown in *Figure 4.14*.

```
<ComponentDescription name="table1">
        <PrimaryKey>
                <PK name="Column2" />
        </PrimaryKey>
        <UniqueKey>
                <UK name="Column3" />
        </UniqueKey>
        <ForeignKey>
                <FK name="Column4" referComponent="table2" referField="Column" />
        </ForeignKey>
        <AttributeList>
                <Attribute name="Column1" datatype="datatype1" size="s1" isNull="No" isAutoIncrement="No" />
                <Attribute name="Column2" datatype="datatype1" size="s1" isNull="No" isAutoIncrement="No" />
                <Attribute name="Column3" datatype="datatype1" size="s1" isNull="Yes" isAutoIncrement="No" />
                <Attribute name="Column4" datatype="datatype1" size="s1" isNull="Yes" isAutoIncrement="No" />
                <Attribute name="Column5" datatype="datatype1" size="s1" isNull="Yes" isAutoIncrement="No" />
                ...
                <Attribute name="Columnn" datatype="datatypen" size="sn" isNull="Yes" isAutoIncrement="No" />
        </AttributeList>
</ComponentDescription>
```

Figure 4. 14: Hierarchical representation of table structure

In structure DTD, associations are represented using the *Relationship* tags. Two tables can be associated with each other using their columns; the different categories of association are one to one, one to many, many to one and many to many. We will look at each possible association description and its equivalent hierarchical representation.

ONE TO ONE

Consider following table declarations

For Table1

Column1 PRIMARY KEY

Column1 REFERENCES Table2 (Column1)

For Table2

Column1 PRIMARY KEY

In the above table definitions, the primary key of table1 refers the primary key of table2, where the primary key of table1 is column1 and of table 2 is column1. The hierarchical structure representation for a one to one relationship is shown in *Figure 4.15.*

```
<RelationshipList>
  <Relation from="Table1" to="Table2" type="ONETOONE"/>
</RelationshipList>
```

Figure 4.15: Hierarchical representation of ONE TO ONE relationship between two tables

ONE TO MANY

Consider the following table declarations

For Table1

Column1 PRIMARY KEY

Column1 REFERENCES Table2 (Column1)

For Table2

Column1 (is not a key)

In the above table definitions, the primary key of *table1* refers to *column1* of *table2*, where the primary key of *table1* is *column1*. The hierarchical representation of ONE TO MANY relationships is shown in *Figure 4.16.*

```
<RelationshipList>
  <Relation from="Table1" to="Table2" type="ONETOMANY"/>
</RelationshipList>
```

Figure 4.16: Hierarchical representation of ONE TO MANY relationship between two tables

MANY TO ONE

Consider following table declarations

For Table1

Column1 REFERENCES Table2 (Column1)

For Table2

Column1 PRIMARY KEY

In the above table definitions, *column1* of *table1* refers to primary key (*column1*) of *table2*. The hierarchical representation of the MANY TO ONE relationship is shown in *Figure 4.17.*

```
<RelationshipList>
  <Relation from="Table1" to="Table2" type="MANYTOONE"/>
</RelationshipList>
```

Figure 4.17: Hierarchical representation of MANY TO ONE relationship between two tables

MANY TO MANY

Consider following table declarations

For Table1

Column1 REFERENCES Table2 (Column1)

For Table2

Column1

In the above table definitions, *column1* of *table1* refers to *column1* of *table2*. The hierarchical representation of a MANY TO MANY relationship is shown in *Figure 4.18*.

```
<RelationshipList>
  <Relation from= "Table1" to= "Table2" type= "MANYTOMANY"/>
</RelationshipList>
```

Figure 4.18: Hierarchical representation of MANY TO MANY relationship

4.3.3.2 Data mapping

In data mapping, data contained within rows and columns of a relational table are mapped to the hierarchical structure of an XML document specified by data DTD. Consider the abstract data view of relational table shown in *Figure 4.19*.

Column1	Column2	....	Columnn
Data1	Data2	...	Datan
....	....	...	...
Data1n	Data2n	...	Datann

Figure 4.19: Relational table representing data view

In this table, *Column1* to *Columnn* represent columns of the table. The table has n rows and each row consists of *data1* to *datan* values. This row column information is stored in corresponding tags specified by data DTD. The mapping is done as follows: the name of the table is mapped to the *name* attribute of the *ComponentData* tag; the *ObjectData* tags are created for each row and for each *ObjectData* tag, the *Contains* tag is created to represent each row-column value. The hierarchical representation is shown in *Figure 4.20*.

```
<?xml version="1.0" encoding="UTF8" ?>
<DatabaseData databaseName="RelationalDatabase">
        <ComponentList>
                <ComponentData name="table1">
                        <AttributeDataList>
                                <ObjectData>
                                        <Contains name="column1" value="data1" />
                                        .....
                                        <Contains name="columnn" value="datan" />
                                </ObjectData>
                                .....
                                <ObjectData>
                                        <Contains name="column1" value="data1" />
                                        .....
                                        <Contains name="columnn" value="datan" />
                                </ObjectData>
                        </AttributeDataList>
                </ComponentData>
                .....
                <ComponentData name="tablen">
                        <AttributeDataList>
                                <ObjectData>
                                        <Contains name="column1" value="data1" />
                                        .....
                                        <Contains name="columnn" value="datan" />
                                </ObjectData>
                                .....
                                <ObjectData>
                                        <Contains name="column1" value="data1" />
                                        .....
                                        <Contains name="columnn" value="datan" />
                                </ObjectData>
                        </AttributeDataList>
                </ComponentData>
        </ComponentList>
</DatabaseData>
```

Figure 4.20: Hierarchical representation of relational database data

4.3.4 Object-oriented Database to XML documents mappings

In this section, we will discuss mapping of object-oriented database structure and data to the hierarchical structure of XML documents specified by structure DTD and data DTD.

4.3.4.1 Structure mapping

As explained in section 3, class represents the main component of object-oriented database structure. Classes are declared using object-oreinted programming language. In structural mapping, we will map the class structure to the hierarchical structure of XML document specified by structure DTD. Class structure consists of a class name, variable names, and data type of variables, default values for variables, inheritance and associations between classes. For the purpose of understanding how a class structure is mapped to XML document, consider the Java class declaration shown *Figure 4.21.*

```
public class class1 extends class2{
        public datatype variable1;
        public datatype variablen;
        public datatype getVariable1() {
                return variable1;
          }
        public void setVariable1(datatype variable1) {
                this.variable1=variable1;
          }
        public datatype getVariablen() {
                return variablen;
          }
        public void setVariablen(datatype variablen) {
                this.variablen=variablen;
```

```
        }
}
```

Figure 4.21: General Java class structure

In the above class declaration, class with name *class1* is declared; it consists of 1..n variables; each variable is associated with a particular data type. Mapping is done according to the following rules: Name of the class is mapped to the *name* attribute of *ComponentDescription* tag and the variables of the class are mapped to *Attribute* tags, where each *Attribute* tag holds the name of the variable and its data type. The hierarchical representation is shown in *Figure 4.22.*

```
<ComponentDescription name="class1">
        <AttributeList>
                <Attribute name="variable1" datatype="datatype" />
                .....
                <Attribute name="variablen" datatype="datatype" />
        </AttributeList>
        <Inheritance>
                <Parent name="class2" />
        </Inheritance>
</ComponentDescription>
```

Figure 4.22: Hierarchical representation of class structure

4.3.2.2 Data mapping

In object-oriented database, objects are the containers of data. These objects are written to persistent storage. The structure of object depends on the class they are derived from. A class declaration defines variables and methods to access those variables. Data stored in the variables of objects are extracted using methods associated with the variables.

```
public class class1 {
        datatype variable1;
        ....
```

```
        datatype variablen;
}
```

Figure 4.23: Simple Java class declaration

In the class declaration shown in *Figure 4.23,* class with the name *class1* is declared. This class defines 1..n variables of data type *datatype.* Mapping of *class1* is done according to the following rules: for each variable; a *Contains* tag is generated. The *Contains* tag consists of two attributes: *name* and *value.* The *Name* attribute holds the name of a variable, and *value* attribute holds the value of a variable. For each object stored in object-oriented database, an *ObjectData* tag is generated. So the top tag is the *ComponentData* tag; its children are *ObjectData* tags, which have *Contains* tags as children. The hierarchical representation is shown in *Figure 4.24.*

```
<DatabaseData databaseName="ObjectOrientedDatabase">
    <ComponentList>
        <ComponentData name="class1">
            <AttributeDataList>
                <ObjectData>
                    <Contains name="variable1" value="data1" />
                    .....
                    <Contains name="variablen" value="datan" />
                </ObjectData>
                  .....
                <ObjectData>
                    <Contains name="variable1" value="data1" />
                    .....
                    <Contains name="variablen" value="datan" />
                </ObjectData>
            </AttributeDataList>
        </ComponentData>
    </ComponentList>
</DatabaseData>
```

Figure 4.24: Hierarchical representation of object-oriented database objects.

4.3.5 Hierarchical database to XML documents mappings

A hierarchical database (Native XML database) consists of XML documents as the storage unit. Data is embedded within tags of an XML document and the tag hierarchy defined in the XML schema. In this section, we will discuss mapping of XML schema and XML documents to hierarchical structure of XML documents specified by structure DTD and data DTD.

4.3.5.1 Structure mapping

As explained in Chapter 3.3.2, XML schema consists of elements as its main component. But there are various ways to declare elements we will discuss mapping strategies for each element declaration.

4.3.5.1.1 Element mapping

XML schema elements can be declared various ways. Each declaration affects the way data will be presented in the XML document. In order to map element structure, we have to traverse through the element declaration and resolve the structure to the hierarchical structure of the XML document specified by structure DTD. In each case below, we will look at the element declaration and its corresponding hierarchical representation.

Case 1: Element declaration consists of a built-in data type and derived data type.

In this case, an element consists of simple contents. An element having simple content can be declared in two ways; one way is declaring simple content using a built-in, type and another way is using a derived types.

Case 1a. In *Figure 4.25* an element is declared using a built-in type. In this element declaration, *element1* is declared with one child element called *variable1*. Data type of *variable1* is *string,* where *string* is a built-in data type.

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="variable1" type="xsd:string" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 4.25: Element declaration using built-in type

Mapping Logic

Element1 mapping: the value of *name* attribute of *element1* is represented by the *name* attribute of the *ComponentDescription* tag. For each child element a corresponding *Attribute* tag is created.

Variable1 mapping: the value of *name* attribute of *variable1* is represented by the *name* attribute of the *Attribute* tag. The value of *type* attribute is represented by the *data-type* attribute of *Attribute* tag.

The hierarchical representation of element declaration is shown in *Figure 4.26.*

```
<ComponentDescription name="element1">
        <AttributeList>
                <Attribute name="variable1" datatype="string" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.26: Hierarchical representation of element declaration

Case1b. In *Figure 4.27, element1* is declared using a user-defined simple type, *simpleType1*.

```
<xsd:simpleType name="simpleType1">
        <xsd:restriction base="string">
                <xsd:enumeration value="enumValue1" />
                ......
                <xsd:enumeration value="enumValuen" />
        </xsd:restriction>
</xsd:simpleType>

<xsd:element name="element1" type="simpleType1" />
```

Figure 4.27: User defined simple type element declaration

An alternative formulation of the above shapes example is to inline the simpleType definition as shown in *Figure 4.28.*

```
<xsd:element name="element1">
        <xsd:simpleType>
                <xsd:restriction base="string">
                        <xsd:enumeration value="enumValue1" />
                        .....
                        <xsd:enumeration value="enumValuen" />
                </xsd:restriction>
        </xsd:simpleType>
</xsd:element>
```

Figure 4.28: User defined inline-simple type element declaration

Mapping logic

Top element declaration mapping: the value of the *name* attribute of top element shown in *Figure 4.27* and *Figure 4.28* is mapped to the *name* attribute of the *ComponentDescription* tag, which is *element1*.

Enumeration tag mapping: for each *enumeration* tag a corresponding *Attribute* tag is created. The value of the *value* attribute of the *enumeration* tag is mapped to the *name* attribute of the *Attribute* tag. The value of the *base* attribute of the *restriction* tag is mapped to the *datatype* attribute of the *Attribute* tag.

The equivalent hierarchical representation is shown in *Figure 4.29.*

```
<ComponentDescription name="element1">
        <AttributeList>
                <Attribute name="enumValue1" datatype="datatype" />
                .....
                <Attribute name="enumValuen" datatype="string" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.29: Hierarchical representation of inline-simple type element declaration

Case 2: Element contains child elements. The immediate child of top element declaration is an unnamed complex type. Child elements enclosed in the complex type structure are simple types.

In *Figure 4.30, element1* is having unnamed complex type which encloses child elements *variable1* through *variablen,* each of built-in data type *string.*

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="variable1" type="xsd:string" />
                        .....
                        <xsd:element name="variablen" type="xsd:string" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 4.30: Nested simple type element declaration example

Mapping Logic

Top element declaration mapping: the value of the *name* attribute of the top element shown in *Figure 4.30* is mapped to the *name* attribute of the *ComponentDescription* tag, which is *element1.*

Unnamed complex type mapping: All child elements are mapped to a corresponding *Attribute* tag. The value of the *name* attribute of the child *element* tag is mapped to the *name* attribute of the *Attribute* tag. The value of the *type* attribute of the *element* tag is mapped to the *datatype* attribute of the *Attribute* tag.

The equivalent hierarchical representation is shown in *Figure 4.31.*

```
<ComponentDescription name="element1">
        <AttributeList>
                <Attribute name="variable1" datatype="string" />
                        .....
                <Attribute name="variablen" datatype="string" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.31: Hierarchical representation of nested, simple-type element declarations

Case 3: An alternate formulation for the element declaration shown in *Figure 4.30* uses named complexTypes as shown in *Figure 4.32.*

```
<xsd:complexType name="complexType1">
        <xsd:sequence>
                <xsd:element name="variable1" type="xsd:string" />
                ....
                <xsd:element name="variablen" type="xsd:string" />
        </xsd:sequence>
</xsd:complexType>
<xsd:element name="element1" type="complexType1" />
```

Figure 4.32: Element declaration using named complex type

In *Figure 4.32, element1* is of type *complexType1.ComplexType1* represents complex structure.

Mapping logic

Top element declaration mapping: The value of the *name* attribute of the top element shown in *Figure 4.32* is mapped to the *name* attribute of the *ComponentDescription* tag. The top level element is the element tag with the *name* attribute value *element1*.

Named complex type mapping: The complex type declaration is resolved and its structure is embedded as the structure of the element declaration referring it. All child elements are mapped to a corresponding *Attribute* tag. The value of the name attribute of the child *element* tag is mapped to the *name* attribute of the *Attribute* tag. The value of the *type* attribute of the *element* tag is mapped to the *datatype* attribute of the *Attribute* tag.

The equivalent hierarchical structural representation is shown in *Figure 4.33.*

```
<ComponentDescription name="element1">
        <AttributeList>
                <Attribute name="variable1" datatype="string" />
                ....
                <Attribute name="variablen" datatype="string" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.33: Hierarchical representation of named complex-type representation

Case 4: The element declaration is having an unnamed complexType as its immediate child, which in turn consists of the combination of element declaration with the complex structure and element declaration with the simple structure. In *Figure 4.34, element1* is the top element declaration. Element *element2* is having a complex structure. Element *variablen* is having a simple structure.

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="element2">
                                <xsd:complexType>
                                        <xsd:element name="variable1" type="xsd:string" />
                                        .....
                                        <xsd:element name="variablen" type="xsd:string" />
                                </xsd:complexType>
                        </xsd:element>
                        <xsd:element name="variablen" type="xsd:string" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 4. 34: Element declaration with unnamed complex type

Mapping logic

Top element declaration mapping: The value of the *name* attribute of the top element shown in *Figure 4.34* is mapped to the *name* attribute of the *ComponentDescription* tag.

Child element with complex structure mapping: We will separate the child with thecomplex structure to form a separate *ComponentDescription* tag. We create an *Attribute* tag corresponding to the child in its *ComponentDescription* tag and the *ComponentDescription* tag of the top element. The top element *ComponentDescription* tag *Attribute* is made the foreign key, which refers to the *Attribute* tag of the child elements *ComponentDescription* tag. The child elements *Attribute* tag is made the primary key. The data type of the *Attribute* is set to *string.*

Child element with simple structure mapping: Each child element with the simple structure of top element forms the *Attribute* tag for the top elements the *ComponentDescription* tag.

The equivalent hierarchical representation is shown in *Figure 4.35.*

```
<ComponentDescription name="element1">
        <ForeignKey>
                <FK name="element2IDREF" referComponent="element2" referField="element2ID" />
        </ForeignKey>
        <AttributeList>
                <Attribute name="element2IDREF" datatype="string" />
                <Attribute name="variable1" datatype="string" />
                <Attribute name="variablen" datatype="string" />
        </AttributeList>
</ComponentDescription>
<ComponentDescription name="element2">
        <PrimaryKey>
                <PK name="element2ID" />
        </PrimaryKey>
        <AttributeList>
                <Attribute name="element2ID" datatype="string" />
                <Attribute name="variable1" datatype="string" />
                <Attribute name="variablen" datatype="string" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.35: Hierarchical representation of unnamed complex type element declaration.

Case 5: Element declaration is having child elements with simple structures and child elements of type complex structure, while the complex structure is declared somewhere else. In *Figure 4.36*, *element2* is the child of *element1,* where *element2* is of type *complexType1*.

```
<xsd:complexType name="complexType1">
        <xsd:sequence>
                <xsd:element name="variable1" type="xsd:string" />
                ....
                <xsd:element name="variablen" type="xsd:datatype" />
        </xsd:sequence>
</xsd:complexType>
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="element2" type="complexType1" />
                        <xsd:element name="variable1" type="xsd:datatype" />
                        ....
                        <xsd:element name="variablen" type="xsd:datatype" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 4.36: Element declarations with child elements are of type complex type.

Mapping logic

Top element tag mapping: same as case 4

Child element of type complex type structure mapping: the child elements of type complex structure form separate *ComponentDescription* tags. An *Attribute* tag is created representing the child in the top elements *ComponentDescription* tag. This attribute is declared as the foreign key in the top elements the *ComponentDescription* tag. An *Attribute* tag is created in the child element's *ComponentDescription* tag reflecting its corresponding *Attribute* tag in the parent's *ComponentDescription* tag.

Child element with simple structure mapping: same as case 4

The equivalent hierarchical representation is shown in *Figure 4.37.*

```
<ComponentDescription name="element1">
        <ForeignKey>
                <FK name="element2IDREF" referComponent="element2"
referField="element2ID" />
        </ForeignKey>
        <AttributeList>
                <Attribute name="element2IDREF" datatype="datatype" />
                <Attribute name="variable1" datatype="datatype" />
                .....
                <Attribute name="variablen" datatype="datatype" />
        </AttributeList>
</ComponentDescription>
<ComponentDescription name="element2">
        <PrimaryKey>
                <PK name="element2ID" />
        </PrimaryKey>
        <AttributeList>
                <Attribute name="element2ID" datatype="datatype" />
                <Attribute name="variable1" datatype="datatype" />
                ....
                <Attribute name="variablen" datatype="datatype" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.37: Hierarchical representations of element declaration with children are of type complex.

Case 6: Element contains child elements, and child elements are referring other elements. In the *Figure 4.38, element3,* child of *element2,* is referring to *element1.*

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="variable1" type="xsd:datatype" />
                        <xsd:element name="variablen" type="xsd:datatype" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="element2">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="element3" ref="element1" />
                        <xsd:element name="variable1" type="xsd:datatype" />
                        <xsd:element name="variablen" type="xsd:datatype" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 4.38: Element declarations with children are referring to other complex types.

Mapping logic

Top element tag mapping: same as case 4

Child element referring complex type structure mapping: same as case 4

Child element with simple structure mapping: same as case 4

The equivalent hierarchical representation is shown in *Figure 4.37.*

Case 7: The element contains child elements, where the child elements have *maxOccurs* set to *unbounded.* Consider the element declaration shown in *Figure 4.39*; here *element2,* child of *element1,* is having the *maxOccurs* value set to *unbounded.* Thus *element2* can occur more than once in the XML document.

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="element2" type="xsd:datatype" maxOccurs="unbounded"
/>
                        <xsd:element name="variable1" type="xsd:datatype" />
                        <xsd:element name="variablen" type="xsd:datatype" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 4.39: Element declaration with children occurring more than once.

Mapping logic

Top element mapping: same as case 4

Child elements with maxOccur attribute set to unbounded mapping: child element's, which occur more than once, are separated to form the *ComponentDescription* tag. The primary key is associated with the child elements the *ComponentDescription* tag, and the foreign key is associated with the *ComponentDescription* tag of the parent element.

Equivalent hierarchical representation is shown in *Figure 4.40.*

```
<ComponentDescription name="element1">
        <AttributeList>
                <Attribute name="element2IDREF" datatype="datatype" />
                <Attribute name="variable1" datatype="datatype" />
                <Attribute name="variablen" datatype="datatype" />
        </AttributeList>
</ComponentDescription>
<ComponentDescription name="element2">
        <AttributeList>
                <Attribute name="element2ID" datatype="datatype" />
                <Attribute name="element2" datatype="datatype" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.40: Hierarchical representation of element declaration with child declaration having maxOccurs attribute set to unbounded.

Case 8: The element is having a complexType as its child, which inherits another complexType using the *extension* keyword of XML schema. The declarations are shown in *Figure 4.41.*

```
<xsd:complexType name="complexType1">
        <xsd:sequence>
                <xsd:element name="variable1" type="xsd:datatype" />
                .....
                <xsd:element name="variablen" type="xsd:datatype" />
        </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="complexType2">
        <xsd:complexContent>
                <xsd:extension base="complexType1">
                        <xsd:sequence>
                                <xsd:element name="variable_n1" type="xsd:datatype" />
                                .....
                                <xsd:element name="variable_nn" type="xsd:datatype" />
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:element name="element1" type="complexType2" />
```

Figure 4.41: Element declaration with complex type inheritance.

Mapping logic

Inherited complex structure mapping: all child elements of the parent complex structure are added to the existing list of child elements of child complex structure. All child elements of resultant complex structure are added to the attribute list of *element,* which is the type of inheriting complex structure. In the *Figure 4.41, element1* will have all child attributes of *complexType2* and *complexType1* in its *ComponentDescription* tag.

The equivalent hierarchical representation is shown in *Figure 4.42.*

```
<ComponentDescription name="element1">
        <AttributeList>
                <Attribute name="variable1" datatype="datatype" />
                ........
                <Attribute name="variablen" datatype="datatype" />
                <Attribute name="variable_n1" datatype="datatype" />
                .....
                <Attribute name="variable_nn" datatype="datatype" />
        </AttributeList>
</ComponentDescription>
```

Figure 4. 42: Hierarchical representation of element declaration with complex type inheritance.

Case 9: The element contains a child with complex structure which inherits another complex structure using the *restriction* keyword of XML schema. Declarations are shown in *Figure 4.43.*

```
<xsd:complexType name="complexType1">
        <xsd:sequence>
                <xsd:element name="variable1" type="xsd:datatype" />
                ......
                <xsd:element name="variablen" type="xsd:datatype" />
        </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="complexType2">
        <xsd:complexContent>
                <xsd:restriction base="complexType1">
                        <xsd:sequence>
                                <xsd:element name="variable1" type="xsd:datatype" />
                                .....
                                <xsd:element name="variablen" type="xsd:datatype" />
                        </xsd:sequence>
                </xsd:restriction>
        </xsd:complexContent>
</xsd:complexType>
<xsd:element name="element1" type="complexType2" />
```

Figure 4.43: Element declaration using *restriction* keyword

Mapping logic

Inherited complex structure mapping: since it is an inheritance using keyword *restriction,* all child elements of the inheriting complex structure are added to the attribute list of *element,* which is the type of inheriting complex structure. In *Figure 4.43, element1* will have all child attributes of *complexType1* in its *ComponentDescription* tag.

The equivalent hierarchical representation is shown in *Figure 4. 44.*

```
<ComponentDescription name="element1">
        <AttributeList>
                <Attribute name="variable1" datatype="datatype" />
                .....
                <Attribute name="variablen" datatype="datatype" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.44: Hierarchical representation of element declaration using *restriction* keyword

4.3.3.1.2 Key Mapping

XML schema provides support for the three types of keys: the primary key, unique key and foreign key. In this section we will discuss mapping of schema key declarations to hierarchical structure specified by structure DTD.

4.3.3.1.2.1 *Primary Key*

Consider the element declaration shown in *Figure 4.45.* The tag *<xsd:key>* defines the primary key with the name *primaryKey* on field the *element1ID* of element *element1.*

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="element1ID" type="datatype" />
                        <xsd:element name="variable1" type="datatype" />
                        .....
                        <xsd:element name="variablen" type="datatype" />
                </xsd:sequence>
        </xsd:complexType>
        <xsd:key name="primaryKey">
                <xsd:selector xpath="element1" />
                <xsd:field xpath="element1ID" />
        </xsd:key>
</xsd:element>
```

Figure 4.45: Element declaration with primary key

Primary key mapping

Top element mapping is done as in previous cases. Tag *PK* is created for each primary key in the element declaration; here we will create *PK* tag with its *name* attribute value set to name of primary key *primaryKey*.

The equivalent hierarchical structural representation is shown in *Figure 3.46.*

```
<ComponentDescription name="element1">
        <PrimaryKey>
                <PK name="elementID" />
        </PrimaryKey>
        <AttributeList>
                <Attribute name="variable1" datatype="datatype" />
                ....
                <Attribute name="variablen" datatype="datatype" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.46: Hierarchical representation of element declaration with primary key

4.3.3.1.2.2 *Unique key*

In the element declaration shown in *Figure 4.47,* the tag <*xsd:unique*> defines the unique key for element *element1*.

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="variable1" type="datatype" />
                        ....
                        <xsd:element name="variablen" type="datatype" />
                </xsd:sequence>
        </xsd:complexType>
        <xsd:unique name="uniqueKey1">
                <xsd:selector xpath="element1" />
                <xsd:field xpath="variable1" />
        </xsd:unique>
        .....
        <xsd:unique name="uniqueKeyn">
                <xsd:selector xpath="element1" />
                <xsd:field xpath="variablen" />
        </xsd:unique>
</xsd:element>
```

Figure 4.47: Element declaration with unique key

Unique key mapping

For each unique key the corresponding *UK* tag is created in the *ComponentDescription* tag for the element. Equivalent hierarchical representation is shown in *Figure 4.48.*

```
<ComponentDescription name="element1">
        <UniqueKey>
                <UK name="variable1" />
                ....
                <UK name="variablen" />
        </UniqueKey>
        <AttributeList>
                <Attribute name="variable1" datatype="datatype" />
                ....
                <Attribute name="variablen" datatype="datatype" />
        </AttributeList>
</ComponentDescription>
```

Figure 4.48: Hierarchical representation of element declaration with unique key.

4.3.3.1.2.3 *Foreign key*

Consider the element declaration shown in *Figure 4.49*. In this declaration, the *<xsd:keyref>* defines the foreign key with the name *foreignKey* which refers to the primary key *primaryKey1* in the element *element1*. This foreign key is defined in the element *element2* on the field *variable1*.

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="variable1" type="datatype" />
                        .....
                        <xsd:element name="variablen" type="datatype" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="element2">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="variable1" type="datatype" />
                        .....
                        <xsd:element name="variablen" type="datatype" />
                </xsd:sequence>
        </xsd:complexType>
        <xsd:key name="primaryKey1">
                <xsd:selector xpath="element1" />
                <xsd:field xpath="variable1" />
        </xsd:key>
        <xsd:keyref name="foreignKey" refer="primaryKey1">
                <xsd:selector xpath="element2" />
                <xsd:field xpath="variable1" />
        </xsd:keyref>
</xsd:element>
```

Figure 4.49: Element declaration with foreign key

Foreign key mapping

For each foreign key the corresponding *FK* tag is created with all relevant information embedded within the attributes of the *FK* tag. The equivalent hierarchical representation is shown in *Figure 4.50*.

```
<ComponentDescription name="element1">
        <PrimaryKey>
                <PK name="variable1" />
        </PrimaryKey>
</ComponentDescription>
<ComponentDescription name="element2">
        <ForiegnKey>
                <key name="variable1" referComponent="element1" referField="variable1" />
        </ForiegnKey>
</ComponentDescription>
```

Figure 4.50: Hierarchical representation of element declaration with foreign key.

4.3.3.1.3 Associations Mapping

Associations represent the number of ways XML schema elements can be associated with other elements. In an element declaration, elements can be associated with each other by using *minOccurs* and *maxOccurs,* and using <*key*> tag and <*keyref*> tag declarations. The possible associations are shown in *Figure 4.51* :

minOccurs	maxOccurs	Cardinality
0	0	ZEROTOONE
0	Unbounded	ZEROTOMANY
1	1	ONETOONE
1	Unbounded	ONETOMANY

Figure 4. 51 XML schema cardinality table

4.3.3.1.3.1 *Association using <key> and <keyref> tag*

Consider the primary key and foreign key declarations shown in *Figure 4.52.* In these declarations, foreign key of *element2* refers to the primary key of *element1.* The

field *variable1* is not the primary key of *element2*. So we have a MANYTOONE relationship between *element2* and *element1*.

```
<xsd:key name="primaryKey1">
        <xsd:selector xpath="element1" />
        <xsd:field xpath="variable1" />
</xsd:key>
<xsd:keyref name="foreignKey" refer="primaryKey1">
        <xsd:selector xpath="element2" />
        <xsd:field xpath="variable1" />
</xsd:keyref>
```

Figure 4.52: Primary key and foreign representation in schema

The equivalent hierarchical representation is shown in *Figure 4.53.*

```
<RelationshipList>
        <Relation from="element2" to="element1" cardinality="MANYTOONE" />
</RelationshipList>
```

Figure 4.53: Hierarchical representation of many to one association between schema elements

If we make *variable1* of *element2* as the primary key, then we have a ONETOONE relationship between *element2* and *element1*. The equivalent hierarchical representation is shown in *Figure 4.54.*

```
<RelationshipList>
        <Relation from="element2" to="element1" cardinality="ONETOONE" />
</RelationshipList>
```

Figure 4.54: Hierarchical representation of one to one association between schema elements

4.3.3.1.3.2 *Association using element declarations*

Elements can be associated from the way they refer to each other in their declarations.

Case 1: The top element contains child elements, and child elements have their own children. This is a nested element declaration.

```
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="element2">
                                <xsd:complexType>
                                  ......
                                </xsd:complexType>
                        </xsd:element>
                        ....
                        <xsd:element name="variable1" type="xsd:datatype" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 4.55: One to one association between XML schema elements

In *Figure 4.55, element1* has one child *element2* whose *minOccur* and *maxOccur* values are 1, so we have a ONETOONE relationship between *element1* and *element2.*

The equivalent hierarchical representation is shown in *Figure 4.56.*

```
<RelationshipList>
        <Relation from="element2" to="element1" cardinality="ONETOONE" />
</RelationshipList>
```

Figure 4.56: Hierarchical representation of XML schema elements one to one relationship

Case 2: The element contains child elements, and child elements are the type of other complex type.

```
<xsd:complexType name="complexType1">
        <xsd:sequence>
                ....
        </xsd:sequence>
</xsd:complexType>
<xsd:element name="element1">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="element2" type="complexType1" minOccurs="cardinality"
maxOccurs="cardinality" />
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 4.57: Association example 2 using element declaration

In *Figure 4.57, element2* is child of *element1* and its occurrence depends on the value of *cardinality*. Equivalent hierarchical representation is shown in *Figure 4.58.*

```
<RelationshipList>
        <Relation from="element1" to="element2" cardinality="cardinality" />
</RelationshipList>
```

Figure 4.58: Association between elements using *minOccurs* and *maxOccurs*

4.3.3.2 Data Mapping

Native XML database data is represented by the XML document. The structure of this XML document is hierarchical. We will traverse through this hierarchical structure and sort out elements, which have children and are not parents themselves of any other element.

The important issue here is the structure of the XML document as specified in the XML schema associated with it, so the data extracted from the XML document should match the *ComponentDescription* tags of the XML schema.

Case 1: In *Figure 4.59*, *element1* is having 1..n children each with specific data values.

```
<element1>
        <child1>data1</child1>
        ....
        <childn>datan</childn>
</element1>
```

Figure 4.59: Sample XML data document 1

Mapping logic

The structure of *element1* will be mapped to the *ComponentData* tag; the *name* attribute of *ComponentData* holds the name of element *element1;* for each child of *element1* the *Contains* tag is created whose *name* attribute takes the name of the child and the *value* attribute takes the value of the child. The equivalent hierarchical representation is shown in *Figure 4.60.*

```
<ComponentData name="element1">
        <AttributeDataList>
                <ObjectData>
                        <Contains name="child1" value="data1" />
                        ....
                        <Contains name="childn" value="datan" />
                </ObjectData>
        </AttributeDataList>
</ComponentData>
```

Figure 4.60: Hierarchical representation of XML data document 1

Case 2: In *Figure 4.61, element1* is having *child1* and *element2* as children. *Element2* is having its own child *child1*.

```
<element1>
        <child1>data1</child1>
        <element2>
                <child1> datan </child1>
        </element2>
</element1>
```

Figure 4. 61: Sample XML data document 2

Mapping logic

We will create *ComponentData* for *element1* and *element2.* For *element2* we will create a *Contains* tag in both *element1* and *element2 ComponentData* tags. The equivalent hierarchical representation is shown in *Figure 4.62.*

```
<ComponentData name="element1">
      <AttributeDataList>
            <ObjectData>
                  <Contains name="child1" value="data1" />
            </ObjectData>
      </AttributeDataList>
</ComponentData>
<ComponentData name="element2">
      <AttributeDataList>
            <ObjectData>
                  <Contains name="child1" value="datan" />
            </ObjectData>
      </AttributeDataList>
</ComponentData>
```

Figure 4.62: Hierarchical representation of XML data document 2

4.4 Datatype mapping

Heterogeneous databases have their own naming conventions for representing their data types. During structural transformation from source database to XML documents, we use the same data type naming as used in the source database. But since the destination heterogeneous database may have different naming conventions, we have to map the naming conventions of the source heterogeneous database data types to the

destination heterogeneous database data type naming convention. The mapping between various naming conventions is specified in the data type mapping the XML document. Consider the following classifications of data types:

- Text: stores characters and words
- Integer: stores numbers such as 1, 2 … 9
- Real: stores fractional values such as 1.1 , 1.2, 0.2
- Binary: stores binary data
- Blob: stores binary large object
- Image: used to store picture files such as jpeg, gif
- Byte: used to hold positive integer numbers ranging from 0–255.
- Boolean: represents true or false values

4.4.1 Relational database data types

The relational database (SQL Server 2000) equivalent representation of basic data types:

A text data type is represented can be represented by any of the keywords *char, nchar, varchar, nvarchar* or *text.* An integer data type can be represented by using any of the keywords *int, integer, long, short int, tiny int* or *number.* A Real data type can be represented using any of the keywords *float, double, decimal or number.* A Blob data type can be represented by using *blob.* A Binary data type is represented by using *binary.* An Image data type can be represented using *image.* A Byte data type can be represented using *byte.*

The above data types representations are for SQL Server 2000. In order to support other relational databases such as MYSQL or ORACLE data types, we can extend the tag structure of data type mapping XML document.

4.4.2 Object-oriented database data types

A Text data type can be represented by using *Char, char, String* or *Char [].* An Integer data type can be represented by using *int, Integer, long* or *short.* A Real data type can be represented by using *float, double, Float* or *Double.* A Binary data type can be represented by using *binary.* An Image data type can be represented using *image.* A Byte data type can be represented by using *byte.* A Boolean data type can be represented by using *bool* or *Boolean.*

4.4.3 XML Schema data types

A text data type can be represented by using the keyword *string.* An Integer data type can be represented be using keywords *int, Integer, long* or *short.* A real data type can be represented by using keywords *float, double, Float* or *Double.* A blob datatype can be represented by using the keyword *blob.* A Binary data type can be represented by using the keyword *binary.* A byte data type can be represented by using the keyword *byte.* An image data type can be represented by using the keyword *NOTATION.*

For the above databases, we specified their naming conventions in the data type-mapping XML document. In this document mapping, information of data types between heterogeneous databases is specified using XML tags. The structure of the data type mapping the XML document is specified in data type DTD.

4.4.4 Data type DTD

Data type DTD defines the set of tags for enclosing the data type naming conventions of the heterogeneous database in the data type mapping the XML document. The data type DTD tag definitions as shown in *Figure 4.63* and *Figure4.64*:

```
<!ELEMENT datatype-mapping (rdbms, oodbms, xdbms)>
    <!-- datatype-mapping is the root element of the data type mapping document, it consists of
        three children rdbms representing the relational database, oodbms representing the
       object- oriented database and xdbms representing native xml database-->

<!ELEMENT rdbms (rdbmsdatatype*)>
    <!-- rdbms consist of one child rdbmsdatatype which holds the data type mapping
        information for a relational database -->

<!ELEMENT rdbmsdatatype(rdbms_oodbms, rdbms_xdbms)>
    <!-- rdbmsdatatpe consists of two children rdbms_oodbms which represents a relational
        database to object-oriented database data type mapping information. rdbms_xdbms
        represents a relational database to XML schema data type mapping information -->

<!ELEMENT rdbms_oodbms (datatype)>
    <!-- rdbms_oodbms consists of one child datatype, which represents the equivalent object-
        oriented data type for the relational database data type. -->

<!ELEMENT rdbms_xdbms (datatype)>
    <!-- rdbms_xdbms consists of one child datatype, which represents the equivalent XML
        schemas data type for the relational database data type. -->

<!ELEMENT oodbms (oodbmsdatatype*)>
    <!-- oodbms consists of one child oodbmsdatatype, which holds the data type mapping
        information for the object-oriented database -->

<!ELEMENT oodbmsdatatype(oodbms_rdbms, oodbms_xdbms)>
    <!-- oodbmsdatatpe consists of two children oodbms_rdbms, which represents the object-
        oriented database to the relational database data type mapping information, and
        oodbms_xdbms representing object-oriented database to the XML schema data type
       mapping information -->
```

Figure 4.63: Data type mapping DTD Part-I

```
<!ELEMENT oodbms_rdbms (datatype)>
    <!-- oodbms_rdbms consists of one child datatype, which represents the equivalent
        relational data type for the object-oreinted database data type. -->

<!ELEMENT oodbms_xdbms (datatype)>
    <!-- oodbms_xdbms consists of one child datatype, which represents the equivalent XML
        schema data type for an object-oriented database data type. -->

<!ELEMENT xdbms (xdbmsdatatype*)>
    <!-- xdbms consists of one child xdbmsdatatype, which holds the data type mapping
        information for XML schemas -->

<!ELEMENT xdbmsdatatype(xdbms_oodbms, xdbms_rdbms)>
    <!-- xdbmsdatatpe consists of two children xdbms_rdbms, which represents XML schema
      to the relational database data type mapping information, and xdbms_oodbms represents
      XML schema to an object-oriented database data type mapping information -->

<!ELEMENT xdbms_oodbms (datatype)>
    <!-- xdbms_oodbms consists of one child datatype, which represents the equivalent object-
        oriented database data type for the XML schema data type. -->

<!ELEMENT xdbms_rdbms (datatype)>
    <!-- xdbms_rdbms consists of one child datatype which represents the equivalent relational
        database data type for the XML schema data type. -->

<!ELEMENT datatype EMPTY>
    <!—the datatype element represents the tag which holds actual data type information -->
<!ATTLIST datatype
            name CDATA #REQUIRED>
    <!--  the datatype consists of one attribute, name, which holds the name of the datatype --
>
```

Figure 4. 64: Data type mapping DTD Part-II

A sample data type mapping XML document is shown in *Figure 4.65.*

```
<datatype-mapping>
        <rdbms>
                <rdbms_datatype name="char">
                        <rdbms_oodbms>
                                <datatype name="char" />
                        </rdbms_oodbms>
                        ....
                        <rdbms_xdbms>
                                <datatype name="string" />
                        </rdbms_xdbms>
                </rdbms_datatype>
                ....
                <rdbms_datatype name="int">
                        <rdbms_oodbms>
                                <datatype name="int" />
                        </rdbms_oodbms>
                        ....
                        <rdbms_xdbms>
                                <datatype name="int" />
                        </rdbms_xdbms>
                </rdbms_datatype>
        </rdbms>
        <oodbms>
                <oodbms_datatype name="String">
                        <oodbms_rdbms>
                                <datatype name="varchar" />
                        </oodbms_rdbms>
                        <oodbms_xdbms>
                                <datatype name="string" />
                        </oodbms_xdbms>
                </oodbms_datatype>
                ....
                <oodbms_datatype name="float">
                        <oodbms_rdbms>
                                <datatype name="float" />
                        </oodbms_rdbms>
                        <oodbms_xdbms>
                                <datatype name="float" />
                        </oodbms_xdbms>
                </oodbms_datatype>
        </oodbms>
        <xdbms>
                <xdbms_datatype name="string">
                        <xdbms_oodbms>
                                <datatype name="String" />
                        </xdbms_oodbms>
                        <xdbms_rdbms>
                                <datatype name="varchar" />
                        </xdbms_rdbms>
                </xdbms_datatype>
                <xdbms_datatype>
                        <xdbms_datatype name="decimal">
                                <xdbms_oodbms>
                                        <datatype name="double" />
                                </xdbms_oodbms>
                                <xdbms_rdbms>
                                        <datatype name="double" />
                                </xdbms_rdbms>
                        </xdbms_datatype>
        </xdbms>
</datatytpe-mapping>
```

Figure 4.65: Sample data type mapping XML document

4.5 Reverse transformation

In this section, we will map the database structure and data information stored in XML documents to the heterogeneous database structure and data. Reverse transformation consists of two stages: structure mapping and data mapping.

4.5.1 Structure mapping

In structure mapping, the database structure information represented by the XML structure document is interpreted and processed to generate destination heterogeneous database structure. In the XML document shown in *Figure 4.66,* we have three main components:

Component1 consists of three attributes: *variable1, variable2* and *variable3* of data type *datatype; variable1* is the primary key of *Component1*, and *variable2* is the foreign key, which refers to the field *variable1* in *Component2.*

Component2 consists of three attributes: *variable1, variable2* and *variable3* of data type *datatype; variable1* is the primary key of *Component2*, and *variable2* is the foreign key refers to the field *variable1* of *Component1. Component3* is the parent of *Component2*

Component3 consists of one attribute *variable1* of data type *datatype.*

Relationship: There exists the ONETOONE relationship between component1 and component2.

```
<DatabaseStructure name="Xdatabase" databaseType="HeterogeneousDatabase">
        <ComponentDescription name="component1">
                <PrimaryKey>
                        <PK name="variable1" />
                </PrimaryKey>
                <ForeignKey>
                        <FK name="variable2" referComponent="component2" referField="variable1" />
                </ForeignKey>
                <AttributeList>
                        <Attribute name="variable1" datatype="datatype" />
                        <Attribute name="variable2" datatype="datatype" />
                        <Attribute name="variable3" datatype="datatype" />
                </AttributeList>
        </ComponentDescription>
        <ComponentDescription name="component2">
                <PrimaryKey>
                        <PK name="variable1" />
                </PrimaryKey>
                <ForeignKey>
                        <FK name="variable2" referComponent="component1" referField="variable1" />
                </ForeignKey>
                <AttributeList>
                        <Attribute name="variable1" datatype="datatype" />
                        .....
                        <Attribute name="variablen" datatype="datatype" />
                </AttributeList>
                <Inheritance>
                        <Parent name="Component3" />
                </Inheritance>
        </ComponentDescription>
        <ComponentDescription name="component3">
                <AttributeList>
                        <Attribute name="variable1" datatype="datatype">
                </AttributeList>
        </ComponentDescription>
        <RelationshipList>
                <Relation from=" component1" to="component2" cardinality="ONETOONE" />
        </RelationshipList>
</DatabaseStructure>
```

Figure 4.66: XML structure document representing structure of heterogeneous database.

In the next section, we will discuss mapping of each tag in the above XML structure document to its corresponding database structure.

4.5.1.1 *ComponentDescription* tag attributes mapping

With the *Component* tag *name* attribute mapping for relational databases, table names take the value of the *name* attribute of the *Component* tag; for object-oriented databases, class name represents the value of the *name* attribute of the *Component* tag; for hierarchical databases, XML schema element *name* attribute represents the value of the *name* attribute of the *Component* tag.

4.5.1.2 *Attribute* tag mappings

For relational databases, table columns are created for each *Attribute* tag of the *AttributeList* tag. Mapping of the *Attribute* tag attributes is as follows: the name of the column represents the value of the *name* attribute of the *Attribute* tag; the data type of the column takes the value of the *datatype* attribute of the *Attribute* tag, the size of the column is determined based on the datatype and the other properties are set according to *isNull* and *autoIncrement* attribute values.

For object-oriented databases, for each class, variables are declared for each *Attribute* tag of the *AttributeList* tag. Mapping of the *Attribute* tag attributes is as follows: name of the variable is the value of *name* attribute of the *Attribute* tag, data type of the variable is the value of *datatype* attribute of *Attribute* tag, and get and set methods are defined for each variable.

For the hierarchical database, each top-level element corresponds to the *ComponentDescription* tag, child elements are created for each *Attribute* tag of the *AttributeList* tag. Mapping of the *Attribute* tag attributes is as follows: the name of child element is the value of *name* attribute of *Attribute* tag and the type of child element is the value of the *datatype* attribute of the *Attribute* tag.

4.5.1.3 Key mappings

The XML structure document provides three tags for representing primary keys, foreign keys and unique keys. These tags are *PrimaryKey* for primary keys, *ForeignKey* for representing foreign keys, and *UniqueKey* for unique keys.

For relational databases, we will define primary keys, foreign keys and unique keys on the columns of relational tables according to each tag description. In primary key tag mapping, column definitions of tables are updated to primary keys using the value of the *name* attribute of the *PK* tags. In foreign key tag mapping, column definitions of tables are updated to foreign keys using the value of *name*, *referComponent* and *referField* attributes of the *FK* tag. In unique key tag mapping, column definitions of tables are updated to unique keys using the value of the *name* attribute of the *UK* tag.

For object-oriented databases, programming languages do not have appropriate constructs for defining keys, unless provided by the DBMS vendor.

For the hierarchical database, we will transform the key definitions to appropriate tags of XML schema. In primary key tag mapping, for each PK tag, a *key* tag is defined in *topelement* declaration of the XML schema, the *name* attribute of *key* tag is set by appending "PK" to the value of the *name* attribute of PK tag, the *field* attribute of *xpath* tag within the *key* tag is set to the value of the *name* attribute of the PK tag. In foreign key tag mapping, for each FK tag, the corresponding *keyref* tag is defined in the topelement declaration of the XML schema. The *name* attribute of the *key* tag is set by appending "FK" to the value of the *name* attribute of the FK tag. The *refer* attribute of *keyref* tag is set to the value of the primary key defined for the element represented by the

value of *referComponent*, the *field* attribute of the *xpath* tag within the *keyref* tag is set to the value of the *name* attribute of the FK tag. In unique key tag mapping, for each UK tag, a *unique* tag is defined in the topelement declaration of the XML schema; the *name* attribute of the *unique* tag is set by appending "UK" to the value of the *name* attribute of the UK tag, and the *field* attribute of the *xpath* tag within the *unique* tag is set to the value of the *name* attribute of the UK tag.

4.5.1.1.4 Inheritance tag mapping

Relational databases do not provide support for inheritance. Therefore we resolve the inheritance structure defined in the XML structure document as follows: for each child table we will add column definitions of its parent table.

Object-oriented databases provide inheritance through the use of keywords. Depending on programming language used, parent classes are inherited using language-specific keywords.

XML schema provides inheritance support through the use of keywords like *extension* and *restriction*. All child tags use these keywords to inherit their parents.

4.5.1.1.5 Relationship tag mapping

The *Relationship* tag holds information about how each component is related to the other component. For relational databases, relationships are automatically created when we update the column definitions using the key definitions provided by XML structure document. For object-oriented databases, class references are created for each

class existing in the relationship tag. For a hierarchical database, corresponding *key* and *keyref* tags are created.

4.5.2 Data mapping

Database data is represented using tags defined by data DTD. Database data mapping is done after generating the database structure from the XML structure document. Consider the XML data document shown in *Figure 4.67*.

```
<DatabaseData databaseName="heterogeneousDatabase">
 <ComponentData name="component1">
                <AttributeDataList>
                        <ObjectData>
                                <Contains name="variable1" value="value1" />
                                <Contains name="variable2" value="value2" />
                        </ObjectData>
                        ....
                        <ObjectData>
                                <Contains name="variable1" value="value1" />
                                <Contains name="variable2" value="value2" />
                        </ObjectData>
                </AttributeDataList>
  </Component>
</DatabaseData>
```

Figure 4.67: XML data document representing data of heterogeneous database.

This data document holds the data for *component1*. Tag mapping for the XML data document is described below.

4.5.2.1 ComponentData tag mapping

In the case of the relational database, each *ComponentData* tag maps to the relational table, and the name of the relational table is the value of the *name* attribute of

the *ComponentData* tag. Each *ObjectData* tag forms the rows of the relational table, and each *Contains* tag attribute values are mapped to corresponding row-column position.

For the object-oriented database, each *ComponentData* tag maps to class definition, the name of the class is the value of the *name* attribute of the *ComponentData* tag. For each *ObjectData* tag, an object for the class is instantiated whose variable values are set according to the attribute of the *Contains* tag. The name attribute corresponds to the variable of the class, and the *value* attribute to the value of variable.

For the hierarchical database, XML elements are generated for each *ObjectData* tag with data embedded in the child tags of the elements.

CHAPTER V

IMPLEMENTATION AND CASE STUDY

In this chapter, we will discuss the design of the database transformation tool and a case study, which demonstrates the database transformation process between heterogeneous databases.

5.1 Design of database transformation tool

The database transformation tool is developed using Java using framework explained in the previous chapter. The database transformation tool consists of two parts: forward transformation and reverse transformation. *Figure 5.1* displays the database transformation tool.

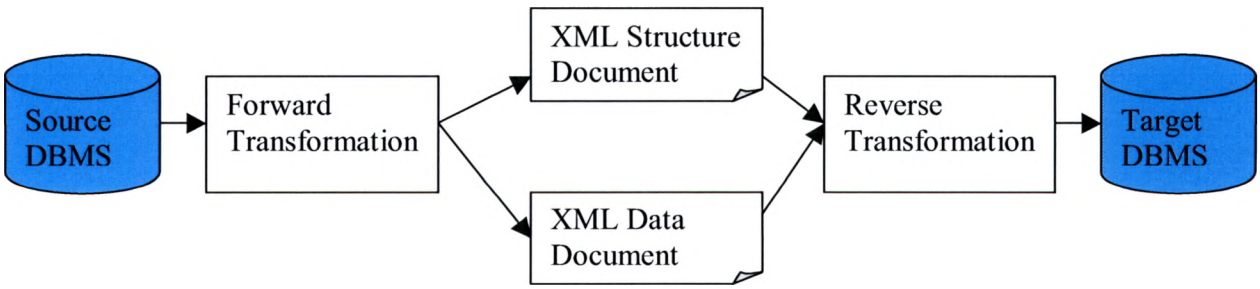

Figure 5.1: Overview of Database Transformation Tool

5.1.1 Source DBMS

Source DBMS represents the source heterogeneous database. Current implementation supports SQL Server 2000, MYSQL, Ozone and IPEDO.

5.1.2 Target DBMS

Target DBMS represents the target heterogeneous database. Current implementation supports SQL Server 2000, MYSQL, Ozone and IPEDO.

5.1.3 XML structure document

XML structure document is generated by forward transformation. It represents the source database structure.

5.1.4 XML data document

XML data document is generated by forward transformation. It represents the source database data.

5.1.5 Forward transformation

Forward transformation represents the transformation logic used for transforming the database structure and data to the XML structure document and XML data document. *Figure* 5.2 displays the architecture of the forward transformation stage of the database transformation tool. It consists of the following parts:

- Front End GUI
- Structure Extractor
- Data Extractor
- GUI Builder

- Visual GUI
- Location Informer
- Structure Transformer
- Data Transformer
- Source DBMS

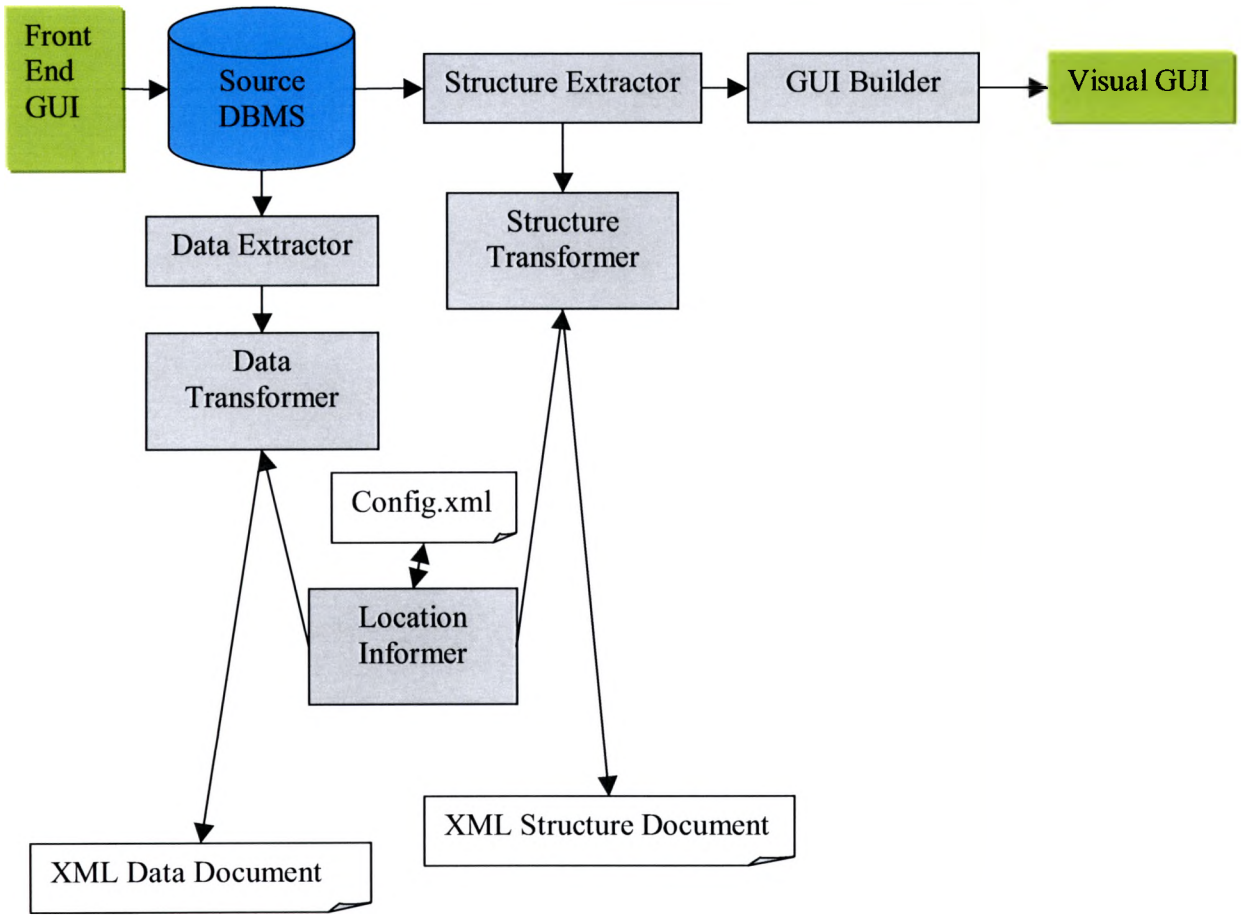

Figure 5.2: Architecture of Forward Transformation

5.1.5.1 *Front End GUI*

Front End GUI consists of a series of java swing windows; each window provides a set of options. The user can specify options whether he wants to transform the database to XML documents or vice versa, and for each option he can choose the DBMS type and provide related information.

5.1.5.2 *Structure Extractor*

The function of *Structure Extractor* depends on the type of source database. If the source database is the relational database, *Structure Extractor* uses JDBC API to connect the RDBMS; it uses various function calls to retrieve information such as table names, and column information including the names of columns, size of columns, the data type of columns, whether the columns are NOT NULL, and whether the columns have their values generated automatically, key information such as primary keys, foreign keys and unique keys. Extracted information is stored in the Hash table so that *Structure Transformer*, *GUI Builder*, and *Data Extractor* can use the Hash table in later parts of execution.

If the source database is the object-oriented database, the structure extractor parses java class files, and extracts class names, names of variables, data type of variables, object references, and method definitions. Extracted information is stored in the Hash table so that *Structure Transformer* and *GUI Builder* can use it in the later part of execution.

If the source database is the hierarchical database, *Structure Extractor* extracts required XML schemas from the hierarchical database. It parses the XML schema and extracts the XML schema ELEMENTS, attributes of the ELEMENTS, child elements of the ELEMENTS, data type of the attributes, data type of the child elements and keys

defined on the ELEMENTS. Extracted information is stored in the Hash table so that *Structure Transformer*, *GUI Builder*, and *Data Extractor* can use the data in later parts of execution.

5.1.5.3 *GUI Builder*

The function of *GUI Builder* is to generate logical diagrams to represent the structure of the source database. For the relational database, it generates ER diagram to represent the relational database table structures and their relationships. For the object-oriented database, it generates the UML class diagram to represent the class structure and their relationships. For the hierarchical database it generates the UML class diagram to represent different elements of schema and their relationship.

5.1.5.4 *Data Extractor*

Data Extractor extracts the data associated with each component of the database such as tables, objects, and XML documents. For the relational database, it extracts all rows related to the tables and stores in Hash table. For the object-oriented database, it retrieves all required objects for the transformation and stores data extracted from objects in the Hash table. For the hierarchical database, it retrieves the XML document, parses it and stores the element information in the Hash table.

5.1.5.5 *Visual GUI*

Visual GUI displays the logical diagrams built by *GUI Builder*. *Visual GUI* is built on top of JGraphpad [Jg '01]

5.1.5.6 *Location Informer*

The function of *Location Informer* is to provide information about the location of XML documents and the data type-mapping XML document. This API uses an XML document called *config.xml*. *Config.xml* stores information about file locations and database connection string information.

5.1.5.7 *Database Structure Ttransformer*

This API implements the structural transformation logic explained in the previous chapter for generating the XML documents from database structure. It uses the information stored in the Hash table and generates the XML structure document and stores the generated file at the location provided by *Location Informer.*

5.1.5.8 *Database Data Transformer*

This API implements the data transformation logic explained in the previous section for generating XML document from database data. It uses the information stored in the Hash table for generating the XML data document and stores the generated file at the location provided by *Location Informer.*

5.1.5.9 *Source DBMS and Target DBMS*

Both represent the DBMS types supported by the transformation tool

- Relational databases: SQL Server, Oracle, and My SQL
- Object-oriented database: Ozone
- Native XML database: IPEDO

5.1.6 Reverse transformation

Reverse transformation implements the logic explained in section 4.5. *Figure* 5.3 displays the reverse transformation model of the database transformation tool. It consists of the following parts:

- Reverse Structure Transformer
- Data type transformer
- Reverse Data Transformer
- Structure Inserter
- Data Inserter
- Target DBMS
- Location Informer

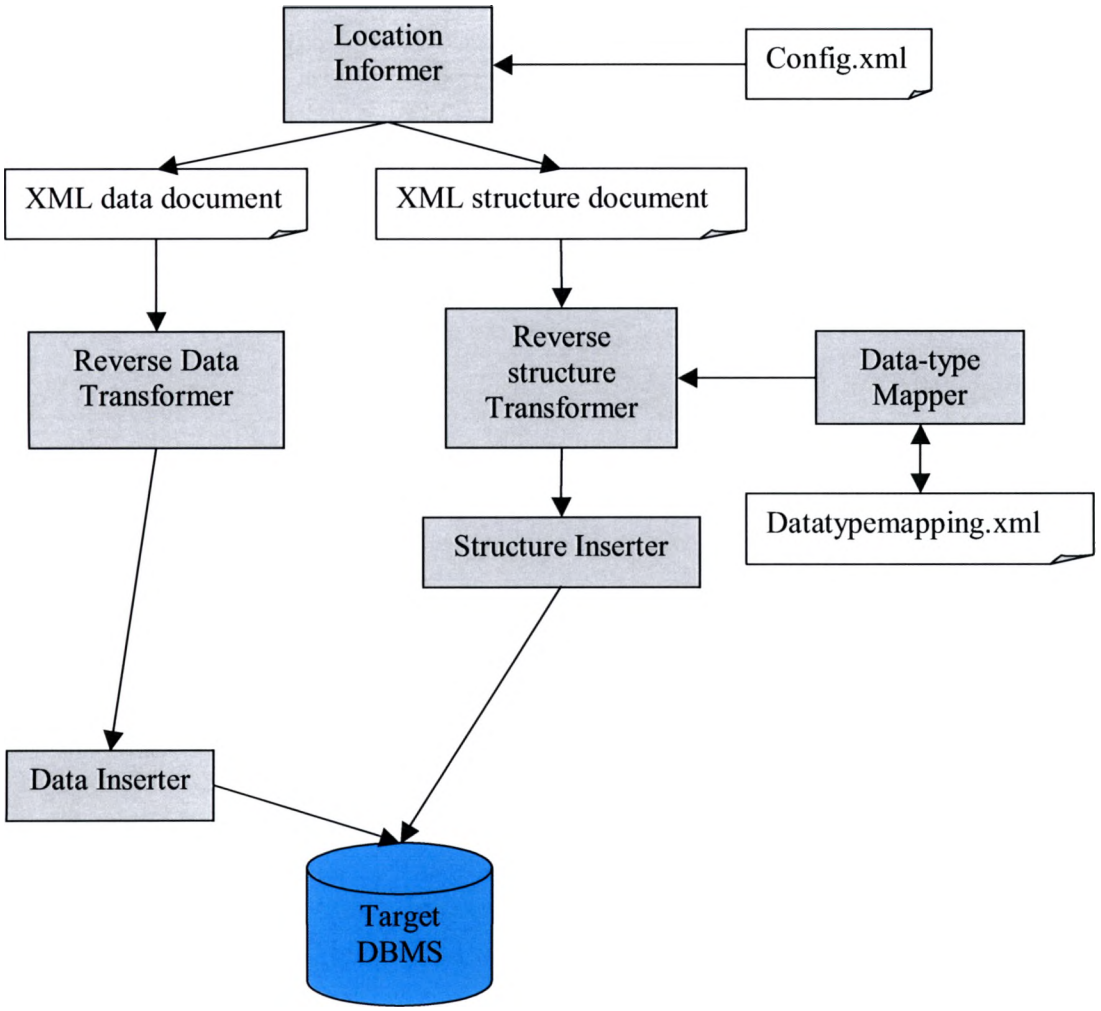

Figure 5.3: Architecture of Reverse Transformation

5.1.6.1 *Datatype Transformer*

This is built on top of the data type mapping XML document; it provides the mapping information of data types between heterogeneous databases.

5.1.6.2 *Reverse Structure Transformer*

This is built on logic explained in the previous chapter for generating the structure of DBMS from structure DTD. For relational database, it generates the XML file containing SQL CREATE statements embedded within XML tags. For the object-oriented database, it generates the XML file containing class declarations, variables, methods embedded within XML tags. For the hierarchical database, it generates the XML Schema document.

5.1.6.3 *Structure Inserter*

Its purpose is to generate the database structure in the target DBMS. For the relational database it executes the SQL CREATE statements embedded within the tags of XML document to create table structures in relational database. For the object-oriented database it executes the driver program, which inserts the class structures into the Ozone database. For the hierarchical database, it inserts the XML Schema into the IPEDO database.

5.1.6.4 *Reverse Data Transformer*

This is built on logic explained in the previous chapter for generating the structure of DBMS from data DTD. For the relational database, it generates the XML file containing SQL INSERT statements embedded within XML tags. For the object-oriented database, it generates the XML file containing objects embedded within XML tags; also, it generates the driver program to insert those objects into the Ozone database. For the hierarchical database, generates the XML data document.

5.1.6.5 *Data Inserter*

Its purpose is to insert the data into the database structure generated in target DBMS. For the relational database, it executes the SQL INSERT statements included in the XML file to insert data into the tables. For the object-oriented database, it executes the driver program to insert objects into the Ozone database. For the hierarchical database, it inserts the XML documents into the IPEDO database.

5.2 Case studies

In this section we will discuss three case studies related to database transformation between heterogeneous databases management systems using the database transformation tool. In the first case study we will discuss the transformation of customer database from SQL Server 2000 (Relational database) to Ozone (Object-oriented database), MYSQL (Relational database) and IPEDO (Native XML database). In the second case study we will discuss the customer database transformation from Ozone to SQL Server 2000 and IPEDO. In third case study we will discuss the customer database transformation from IPEDO to Ozone and SQL Server 2000.

5.2.1 Case Study: Relational database to Heterogeneous databases

In this section, we will discuss the customer database transformation from SQL Server 2000 to OZONE, IPEDO and MYSQL. As explained earlier, the first step in the transformation process is the forward transformation, and the next step is the reverse transformation.

5.2.1.1 Forward Transformation

In forward tranformation, the customer database structure and data are extracted from SQL server 2000 and transformed to XML documents. Customer database information is extracted using JDBC API. The ER diagram of the customer database is shown in *Figure 5.4*.

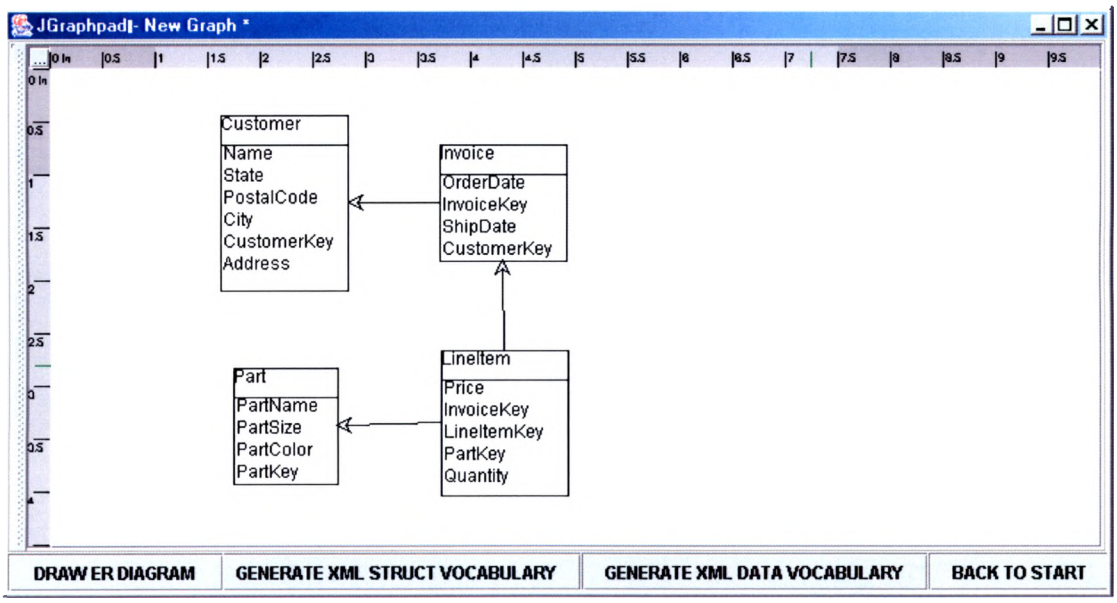

Figure 5. 4: ER diagram representing customer database

Customer database consists of four tables: Customer table, Invoice table, Part table, and LineItem table. Table structures are shown in *Figures 5.5, 5.6, 5.7* and *5.8.*

Column Name	Data type	Size	Nulls	Primary keys	Foreign keys
CustomerKey	int	4		Yes	
Name	varchar	30	Yes		
Address	varchar	30	Yes		
City	varchar	30	Yes		
State	char	2	Yes		
PostalCode	varchar	10	Yes		

Figure 5.5: Customer Table Structure

Column Name	Data type	Size	Nulls	Primary keys	Foreign keys	ReferTable
InvoiceKey	Int	4		Yes		
CustomerKey	Int	4	Yes		Yes	Customer
OrderDate	Datetime	8	Yes			
ShipDate	Datetime	8	Yes			

Figure 5.6: Invoice Table structure

Column Name	Data type	Size	Nulls	Primary keys	Foreign keys	ReferTable
LineItemKey	Int	4		Yes		
InvoiceKey	Int	4	Yes		Yes	Invoice
PartKey	Int	4	Yes		Yes	Parts
Quantity	Int	4	Yes			
Price	Float	8	Yes			

Figure 5.7: LineItem Table structure

Column Name	Data type	Size	Nulls	PrimaryKey	Foreign Key	ReferTable
PartKey	int	4		Yes		
PartName	varchar	20	Yes			
PartColor	varchar	10	Yes			
PartSize	varchar	10	Yes			

Figure 5.8: Part table structure

Using the forward transformation logic explained in section 4.3.3.1, table structures are mapped to the hierarchical structure specified by structure DTD. The resultant XML structure document generated by the database transformation tool is shown in *Figure 5.9* and *Figure5.10.*

```
<DatabaseStructure databaseName="Customer">
        <ComponentList>
                <ComponentDescription name="Part">
                        <PrimaryKey>
                                <Key name="PartKey" />
                        </PrimaryKey>
                        <AttributeList>
                                <Attribute name="PartName" datatype="varchar" size="20" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="PartSize" datatype="varchar" size="10" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="PartColor" datatype="varchar" size="10" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="PartKey" datatype="int" size="11" isNull="NO" isAutoIncrement="NO"/>
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="Invoice">
                        <PrimaryKey>
                                <Key name="InvoiceKey" />
                        </PrimaryKey>
                        <ForeignKey>
                                <Key name="CustomerKey" referTable="Customer" />
                        </ForeignKey>
                        <AttributeList>
                                <Attribute name="OrderDate" datatype="datetime" size="23" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="InvoiceKey" datatype="int" size="11" isNull="NO" isAutoIncrement="NO" />
                                <Attribute name="ShipDate" datatype="datetime" size="23" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="CustomerKey" datatype="int" size="11" isNull="YES" isAutoIncrement="NO" />
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="Customer">
                        <PrimaryKey>
                                <Key name="CustomerKey" />
                        </PrimaryKey>
                        <AttributeList>
                                <Attribute name="Name" datatype="varchar" size="30" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="State" datatype="char" size="2" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="PostalCode" datatype="varchar" size="10" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="City" datatype="varchar" size="30" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="CustomerKey" datatype="int" size="11" isNull="NO" isAutoIncrement="NO" />
                                <Attribute name="Address" datatype="varchar" size="30" isNull="YES" isAutoIncrement="NO" />
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="LineItem">
                        <PrimaryKey>
                                <Key name="LineItemKey" />
                        </PrimaryKey>
                        <ForeignKey>
                                <Key name="InvoiceKey" referTable="Invoice" />
                                <Key name="PartKey" referTable="Part" />
                        </ForeignKey>
                        <AttributeList>
                                <Attribute name="Price" datatype="float" size="24" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="InvoiceKey" datatype="int" size="11" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="LineItemKey" datatype="int" size="11" isNull="NO" isAutoIncrement="NO" />
                                <Attribute name="PartKey" datatype="int" size="11" isNull="YES" isAutoIncrement="NO" />
                                <Attribute name="Quantity" datatype="int" size="11" isNull="YES" isAutoIncrement="NO" />
                        </AttributeList>
```

Figure 5. 9: Hierarchical representation of customer database structure Part-I

```
                </ComponentDescription>
        </ComponentList>
        <RelationshipList>
                <Relation from="Invoice" to="Customer" cardinality="ManyToOne" />
                <Relation from="LineItem" to="Invoice" cardinality="ManyToOne" />
                <Relation from="LineItem" to="Part" cardinality="ManyToOne" />
        </RelationshipList>
</DatabaseStructure>
```

Figure 5.10: Hierarchical representation customer database structure Part-II

5.2.1.2 Data Mapping

XML data document is generated in the forward transformation process using logic explained in section 4.3.3.2; it is shown in *Figure 5.11, Figure 5.12* and *Figure 5.13.*

```
<DatabaseData name="Customer">
        <ComponentList>
                <ComponentData name="Part">
                        <AttributeDataList>
                                <ObjectData>
                                        <Data name="PartKey" value="1" />
                                        <Data name="PartName" value="football" />
                                        <Data name="PartColor" value="white" />
                                        <Data name="PartSize" value="5" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="PartKey" value="2" />
                                        <Data name="PartName" value="baseball" />
                                        <Data name="PartColor" value="grey" />
                                        <Data name="PartSize" value="2" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="PartKey" value="3" />
                                        <Data name="PartName" value="speakers" />
                                        <Data name="PartColor" value="black" />
                                        <Data name="PartSize" value="5" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="PartKey" value="4" />
                                        <Data name="PartName" value="computer case" />
                                        <Data name="PartColor" value="white" />
                                        <Data name="PartSize" value="8" />
                                </ObjectData>
                        </AttributeDataList>
                </ComponentData>
                <ComponentData name="Invoice">
                        <AttributeDataList>
```

Figure 5. 11: Hierarchical representation of customer database data Part-I

```
                <ObjectData>
                        <Data name="InvoiceKey" value="1" />
                        <Data name="CustomerKey" value="1" />
                        <Data name="OrderDate" value="2001-01-01 00:00:00.000" />
                        <Data name="ShipDate" value="2001-01-10 00:00:00.000" />
                </ObjectData>
                 <ObjectData>
                        <Data name="InvoiceKey" value="2" />
                        <Data name="CustomerKey" value="2" />
                        <Data name="OrderDate" value="2001-02-01 00:00:00.000" />
                        <Data name="ShipDate" value="2001-02-12 00:00:00.000" />
                </ObjectData>
                <ObjectData>
                        <Data name="InvoiceKey" value="3" />
                        <Data name="CustomerKey" value="3" />
                        <Data name="OrderDate" value="2001-05-10 00:00:00.000" />
                        <Data name="ShipDate" value="2001-05-15 00:00:00.000" />
                        </ObjectData>
                <ObjectData>
                        <Data name="InvoiceKey" value="4" />
                        <Data name="CustomerKey" value="4" />
                        <Data name="OrderDate" value="2001-06-20 00:00:00.000" />
                        <Data name="ShipDate" value="2001-06-25 00:00:00.000" />
                </ObjectData>
        </AttributeDataList>
</ComponentData>
<ComponentData name="Customer">
        <AttributeDataList>
                <ObjectData>
                        <Data name="CustomerKey" value="1" />
                        <Data name="Name" value="jim" />
                        <Data name="Address" value="1200 A drive" />
                        <Data name="City" value="San Marcos" />
                        <Data name="State" value="TX" />
                        <Data name="PostalCode" value="78666" />
                </ObjectData>
                <ObjectData>
                        <Data name="CustomerKey" value="2" />
                        <Data name="Name" value="tom" />
                        <Data name="Address" value="1300 B drive" />
                        <Data name="City" value="Austin" />
                        <Data name="State" value="TX" />
                        <Data name="PostalCode" value="78752" />
                </ObjectData>
                <ObjectData>
                        <Data name="CustomerKey" value="3" />
                        <Data name="Name" value="bob" />
                        <Data name="Address" value="1400 C drive" />
                        <Data name="City" value="Austin" />
                        <Data name="State" value="TX" />
                        <Data name="PostalCode" value="78575" />
```

Figure 5.12: Hierarchical representation of customer database data Part-II

```
                    </ObjectData>
                    <ObjectData>
                        <Data name="CustomerKey" value="4" />
                        <Data name="Name" value="mat" />
                        <Data name="Address" value="200 springs drive" />
                        <Data name="City" value="San Marcos" />
                        <Data name="State" value="TX" />
                        <Data name="PostalCode" value="78666" />
                    </ObjectData>
                </AttributeDataList>
            </ComponentData>
            <ComponentData name="LineItem">
                <AttributeDataList>
                    <ObjectData>
                        <Data name="LineItemKey" value="1" />
                        <Data name="InvoiceKey" value="1" />
                        <Data name="PartKey" value="1" />
                        <Data name="Quantity" value="2" />

                        <Data name="Price" value="10.0" />
                    </ObjectData>
                    <ObjectData>
                        <Data name="LineItemKey" value="2" />
                        <Data name="InvoiceKey" value="2" />
                        <Data name="PartKey" value="2" />
                        <Data name="Quantity" value="3" />
                        <Data name="Price" value="20.0" />
                    </ObjectData>
                    <ObjectData>
                        <Data name="LineItemKey" value="3" />
                        <Data name="InvoiceKey" value="3" />
                        <Data name="PartKey" value="3" />
                        <Data name="Quantity" value="10" />
                        <Data name="Price" value="200.0" />
                    </ObjectData>
                    <ObjectData>
                        <Data name="LineItemKey" value="4" />
                        <Data name="InvoiceKey" value="4" />
                        <Data name="PartKey" value="4" />
                        <Data name="Quantity" value="10" />
                        <Data name="Price" value="300.0" />
                    </ObjectData>
                </AttributeDataList>
            </ComponentData>
        </ComponentList>
</DatabaseData>
```

Figure 5.13: Hierarchical representation of customer database data Part-III

5.2.1.2 Reverse Transformation

In reverse transformation the XML structure document and XML data document obtained from forward transformation are transformed to the structure and data format of target databases.

5.2.1.2.1 XML documents to MYSQL (relational database)

The first step is to generate the database structure in MYSQL and then to insert data.

5.2.1.2.1.1 MYSQL structure generation

From the XML structure document, the database transformation tool generates the XML document with SQL CREATE statements embedded within XML tags using logic explained in section 4.5.1. The resultant XML document is shown in *Figure 5.14.*

```
<SQLCREATE>
    <SQLStatement>CREATE TABLE Part (PartName varchar(20) , PartSize
    varchar(10) , PartColor varchar(10) ,  PartKey int primary key );
   </SQLStatement>
    <SQLStatement>CREATE TABLE Invoice (OrderDate datetime, InvoiceKey int
    primary key, ShipDate datetime, CustomerKey int);
  </SQLStatement>
    <SQLStatement>CREATE TABLE Customer ( Name varchar(30) , State  char(2) ,
    PostalCode varchar(10) , City varchar(30) , CustomerKey int  primary key , Address
    varchar(30) );
    </SQLStatement>
    <SQLStatement>CREATE TABLE LineItem (Price float(24) , InvoiceKey int ,
    LineItemKey int primary key ,   PartKey int , Quantity int );
  </SQLStatement>
</SQLCREATE>
```

Figure 5. 14: XML document with SQL CREATE statements

5.2.1.2.1.2 MYSQL data generation

From the XML data document the database transformation tool generates the XML document with SQL INSERT statements embedded within tags, as shown in *Figure 5.15.*

```
<SQLINSERT>
    <SQLstatement>insert into Part ( PartKey, PartName, PartColor, PartSize) values ('1','football','white','5');
    </SQLstatement>
    <SQLstatement>insert into Part (PartKey, PartName, PartColor, PartSize) values ('2','baseball','grey','2');
    </SQLstatement>
    <SQLstatement>insert into Part (PartKey, PartName, PartColor, PartSize) values ('3','speakers','black','5');
    </SQLstatement>
    <SQLstatement>insert into Part (PartKey, PartName, PartColor, PartSize) values ('4','computer case','white','8');
    </SQLstatement>
    <SQLstatement>insert into Invoice (InvoiceKey, CustomerKey, OrderDate, ShipDate) values ('1','1','2001-01-01
    00:00:00.000','2001-01-10 00:00:00.000');
    </SQLstatement>
    <SQLstatement>insert into Invoice (InvoiceKey, CustomerKey, OrderDate, ShipDate) values ('2','2','2001-02-01
    00:00:00.000','2001-02-12 00:00:00.000');
    </SQLstatement>
    <SQLstatement>insert into Invoice (InvoiceKey, CustomerKey, OrderDate, ShipDate) values ('3','3','2001-05-10
    00:00:00.000','2001-05-15 00:00:00.000');
    </SQLstatement>
    <SQLstatement>insert into Invoice (InvoiceKey, CustomerKey, OrderDate, ShipDate) values ('4','4','2001-06-20
    00:00:00.000','2001-06-25 00:00:00.000');
    </SQLstatement>
    <SQLstatement>insert into Customer (CustomerKey, Name, Address, City, State, PostalCode) values ('1','jim','1200 A
    drive','San Marcos','TX','78666');
    </SQLstatement>
    <SQLstatement>insert into Customer (CustomerKey, Name, Address, City, State, PostalCode) values ('2','tom','1300 B
    drive','Austin','TX','78752');
    </SQLstatement>
    <SQLstatement>insert into Customer (CustomerKey, Name, Address, City, State, PostalCode) values ('3','bob','1400 C
    drive','Austin','TX','78575');
    </SQLstatement>
    <SQLstatement>insert into Customer (CustomerKey, Name, Address, City, State, PostalCode) values ('4','mat','200
    springs drive','San Marcos','TX','78666');
    </SQLstatement>
    <SQLstatement>insert into LineItem (LineItemKey, InvoiceKey, PartKey, Quantity, Price) values ('1','1','1','2','10.0');
    </SQLstatement>
    <SQLstatement>insert into LineItem (LineItemKey, InvoiceKey, PartKey, Quantity, Price) values ('2','2','2','3','20.0');
    </SQLstatement>
    <SQLstatement>insert into LineItem (LineItemKey, InvoiceKey, PartKey, Quantity, Price) values ('3','3','3','10','200.0');
    </SQLstatement>
    <SQLstatement>insert into LineItem (LineItemKey, InvoiceKey, PartKey, Quantity, Price) values ('4','4','4','10','300.0');
    </SQLstatement>
</SQLINSERT>
```

Figure 5.15: XML document with SQL INSERT statements

5.2.1.2.2 XML documents to Ozone

The database represented by XML documents is transformed to Java classes and objects.

5.2.1.2.2.1 Structure generation

From the XML structure document, the database transformation tool generates Java class files specific to Ozone using logic explained in section 4.5.1. Class files are shown in *Figures 5.16, 5.17, 5.18, 5.19, 5.20, 5.21, 5.22 and 5.23.*

```
import org.ozoneDB.OzoneRemote;

public interface Customer extends OzoneRemote {
        public void setName(String Name);
        public String getName();
        public void setState(String State);
        public String getState();
        public void setPostalCode(String PostalCode);
        public String getPostalCode();
        public void setCity(String City);
        public String getCity();
        public void setCustomerKey(int CustomerKey);
        public int getCustomerKey();
        public void setAddress(String Address);
        public String getAddress();

}
```

Figure 5.16: Customer Interface for Customer table

```
import org.ozoneDB.OzoneRemote;

public interface Invoice extends OzoneRemote {
        public void setOrderDate(String OrderDate);
        public String getOrderDate();
        public void setInvoiceKey(int InvoiceKey);
        public int getInvoiceKey();
        public void setShipDate(String ShipDate);
        public String getShipDate();
        public void setCustomerKey(int CustomerKey);
        public int getCustomerKey();

}
```

Figure 5.17: Invoice interface for Invoice Table

```
import org.ozoneDB.OzoneObject;
import java.util.*;

public class CustomerImpl extends OzoneObject implements Customer {

        String Name;
        public void setName(String Name) { this.Name=Name; }

        public String getName(){ return Name; }
        String State;
        public void setState(String State) { this.State=State; }

        public String getState(){ return State; }
        String PostalCode;
        public void setPostalCode(String PostalCode) { this.PostalCode=PostalCode; }

        public String getPostalCode(){ return PostalCode; }
        String City;
        public void setCity(String City) { this.City=City; }

        public String getCity(){ return City; }
        int CustomerKey;
        public void setCustomerKey(int CustomerKey) { this.CustomerKey=CustomerKey; }

        public int getCustomerKey(){ return CustomerKey; }
        String Address;
        public void setAddress(String Address) { this.Address=Address; }

        public String getAddress(){ return Address; }
}
```

Figure 5. 18: Customer class for Customer table

```
import org.ozoneDB.OzoneRemote;

public interface LineItem extends OzoneRemote {
        public void setPrice(float Price);
        public float getPrice();
        public void setInvoiceKey(int InvoiceKey);
        public int getInvoiceKey();
        public void setLineItemKey(int LineItemKey);
        public int getLineItemKey();
        public void setPartKey(int PartKey);
        public int getPartKey();
        public void setQuantity(int Quantity);
        public int getQuantity();

}
```

Figure 5.19: LineItem interface for LineItem table

```
import org.ozoneDB.OzoneObject;
import java.util.*;

public class InvoiceImpl extends OzoneObject implements Invoice {

        CustomerImpl customer;
        String OrderDate;
        public void setOrderDate(String OrderDate) { this.OrderDate=OrderDate; }

        public String getOrderDate(){ return OrderDate; }
        int InvoiceKey;
        public void setInvoiceKey(int InvoiceKey) { this.InvoiceKey=InvoiceKey; }

        public int getInvoiceKey(){ return InvoiceKey; }
        String ShipDate;
        public void setShipDate(String ShipDate) { this.ShipDate=ShipDate; }

        public String getShipDate(){ return ShipDate; }
        int CustomerKey;
        public void setCustomerKey(int CustomerKey) { this.CustomerKey=CustomerKey; }

        public int getCustomerKey(){ return CustomerKey; }
}
```

Figure 5.20: Invoice class for Invoice Table

```
import org.ozoneDB.OzoneRemote;

public interface Part extends OzoneRemote {
        public void setPartName(String PartName);
        public String getPartName();
        public void setPartSize(String PartSize);
        public String getPartSize();
        public void setPartColor(String PartColor);
        public String getPartColor();
        public void setPartKey(int PartKey);
        public int getPartKey();

}
```

Figure 5. 21: Part Interface for Part Table

```
import org.ozoneDB.OzoneObject;
import java.util.*;

public class PartImpl extends OzoneObject implements Part {


        String PartName;
        public void setPartName(String PartName) { this.PartName=PartName; }

        public String getPartName(){ return PartName; }
        String PartSize;
        public void setPartSize(String PartSize) { this.PartSize=PartSize; }

        public String getPartSize(){ return PartSize; }
        String PartColor;
        public void setPartColor(String PartColor) { this.PartColor=PartColor; }

        public String getPartColor(){ return PartColor; }
        int PartKey;
        public void setPartKey(int PartKey) { this.PartKey=PartKey; }

        public int getPartKey(){ return PartKey; }
}
```

Figure 5.22: PartImpl class for Part table

```
import org.ozoneDB.OzoneObject;
import java.util.*;

public class LineItemImpl extends OzoneObject implements LineItem {

        InvoiceImpl invoice;
        PartImpl part;

        float Price;
        public void setPrice(float Price) { this.Price=Price; }
                                          -
        public float getPrice(){ return Price; }
        int InvoiceKey;
        public void setInvoiceKey(int InvoiceKey) { this.InvoiceKey=InvoiceKey; }

        public int getInvoiceKey(){ return InvoiceKey; }
        int LineItemKey;
        public void setLineItemKey(int LineItemKey) { this.LineItemKey=LineItemKey; }

        public int getLineItemKey(){ return LineItemKey; }
        int PartKey;
        public void setPartKey(int PartKey) { this.PartKey=PartKey; }

        public int getPartKey(){ return PartKey; }
        int Quantity;
        public void setQuantity(int Quantity) { this.Quantity=Quantity; }

        public int getQuantity(){ return Quantity; }
}
```

Figure 5.23: LineItemImpl class for LineItem table

5.2.1.2.2.2 Data generation

From the XML data document, the database transformation tool generates a Java class file with the method that calls for object creation and insertion into the Ozone database using logic explained in section 4.5.2. The tool generates XML documents with object names in it, shown in *Figure 5.25*, which we will use in the second case study. The generated driver program to insert objects is shown in *Figure5.24*.

```
import java.util *,
import org.ozoneDB *,
public class OODBMSDataDriver {
        public static ExternalDatabase     db,
        public static void main( String[] args ) throws Exception {
                db = ExternalDatabase.openDatabase( "ozonedb.remote //localhost 3333" ),
                System.out.println( "Connected    " ),
                db.reloadClasses(),
                OODBMSDataDriver od=new OODBMSDataDriver(),
                od.start();
                db.close(),
        }
        public void start(){
                createPart("objPart1","football","5","white",1),
                createPart("objPart3","baseball","2","grey",2),
                createPart("objPart5","speakers","5","black",3),
                createPart("objPart7","computer case","8","white",4),
                createInvoice("objInvoice1","2001-01-01 00:00:00.000",1,"2001-01-10 00:00:00.000",1),
                createInvoice("objInvoice3","2001-02-01 00:00:00.000",2,"2001-02-12 00:00:00.000",2);
                createInvoice("objInvoice5","2001-05-10 00:00:00.000",3,"2001-05-15 00:00:00.000",3),
                createInvoice("objInvoice7","2001-06-20 00:00:00.000",4,"2001-06-25 00:00:00.000",4),
                createCustomer("objCustomer1","jim","TX","78666","San Marcos",1,"1200 A drive"),
                createCustomer("objCustomer3","tom","TX","78752","Austin",2,"1300 B drive"),
                createCustomer("objCustomer5","bob","TX","78575","Austin",3,"1400 C drive"),
                createCustomer("objCustomer7","mat","TX","78666","San Marcos",4,"200 springs drive"),
                createLineItem("objLineItem1", (float)10.0,1,1,1,2),
                createLineItem("objLineItem3", (float)20.0,2,2,2,3),
                createLineItem("objLineItem5", (float)200.0,3,3,3,10),
                createLineItem("objLineItem7", (float)300.0,4,4,4,10),
        }

        public void createPart(String objName,String var5,String var7,String var9,int var11)
        {
                Part partObj = (Part )(db.createObject( PartImpl.class.getName(), 0, objName )),
                partObj.setPartName(var5),
                partObj.setPartSize(var7),
                partObj.setPartColor(var9),
                partObj.setPartKey(var11),

        }
        public void createInvoice(String objName,String var5,int var7,String var9,int var11)
        {
                Invoice invoiceObj = (Invoice )(db.createObject( InvoiceImpl.class.getName(), 0, objName )),
                invoiceObj.setOrderDate(var5),
                invoiceObj.setInvoiceKey(var7),
                invoiceObj.setShipDate(var9),
                invoiceObj.setCustomerKey(var11),

        }
        public void createCustomer(String objName,String var5,String var7,String var9,String var11,int var13,String var15)
        {
                Customer customerObj = (Customer )(db.createObject( CustomerImpl.class.getName(), 0, objName )),
                customerObj.setName(var5),
                customerObj.setState(var7),
                customerObj.setPostalCode(var9),
                customerObj.setCity(var11),
                customerObj.setCustomerKey(var13),
                customerObj.setAddress(var15),

        }
        public void createLineItem(String objName,float var5,int var7,int var9,int var11,int var13)
        {
                LineItem lineitemObj = (LineItem )(db.createObject( LineItemImpl.class.getName(), 0, objName )),
                lineitemObj.setPrice(var5),
                lineitemObj.setInvoiceKey(var7),
                lineitemObj.setLineItemKey(var9),
                lineitemObj.setPartKey(var11),
                lineitemObj.setQuantity(var13),
        }
}
```

Figure 5. 24: Java driver class

```
<objectnames>
      <class name="Part">
            <objectName value="objPart1" />
            <objectName value="objPart3" />
            <objectName value="objPart5" />
            <objectName value="objPart7" />
      </class>
      <class name="Invoice">
            <objectName value="objInvoice1" />
            <objectName value="objInvoice3" />
            <objectName value="objInvoice5" />
            <objectName value="objInvoice7" />
      </class>
      <class name="LineItem">
            <objectName value="objLineItem1" />
            <objectName value="objLineItem3" />
            <objectName value="objLineItem5" />
            <objectName value="objLineItem7" />
      </class>
      <class name="Customer">
            <objectName value="objCustomer1" />
            <objectName value="objCustomer3" />
            <objectName value="objCustomer5" />
            <objectName value="objCustomer7" />
      </class>
</objectnames>
```

Figure 5.25: XML document with object names for objects stored in Ozone

5.2.1.2.3 XML documents to IPEDO

The database structure and data represented by XML documents are transformed to XML schema and related XML documents. The XML schema and associated XML documents are inserted into IPEDO.

5.2.1.2.3.1 IPEDO XML Schema generation

The database transformation tool generates XML schema from the XML structure document using logic explained in section 4.5.1. In this XML schema, shown in *Figures 5.26, 5.27* and *5.28,* a top element is created called *topstructure*; the purpose of this element is to hold all the top-level elements of the schema.

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <xsd:element name="topstructure">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="Part" maxOccurs="unbounded" />
                                <xsd:element ref="Invoice" maxOccurs="unbounded" />
                                <xsd:element ref="Customer" maxOccurs="unbounded" />
                                <xsd:element ref="LineItem" maxOccurs="unbounded" />
                        </xsd:sequence>
                </xsd:complexType>
                <xsd:key name="PartPartKeyPK">
                        <xsd:selector xpath="Part" />
                        <xsd:field xpath="PartKey" />
                </xsd:key>
                <xsd:key name="InvoiceInvoiceKeyPK">
                        <xsd:selector xpath="Invoice" />
                        <xsd:field xpath="InvoiceKey" />
                </xsd:key>
                <xsd:key name="CustomerCustomerKeyPK">
                        <xsd:selector xpath="Customer" />
                        <xsd:field xpath="CustomerKey" />
```

Figure 5. 26: XML schema for customer database Part-I

```
        <xsd:keyref name="LineItemPartKeyFK" refer="PartPartKeyPK">
                <xsd:selector xpath="LineItem" />
                <xsd:field xpath="PartKey" />
        </xsd:keyref>
</xsd:element>
 <xsd:element name="Part">
        <xsd:complexType>
                <xsd:all>
                        <xsd:element name="PartName" type="xsd:string" />
                        <xsd:element name="PartSize" type="xsd:string" />
                        <xsd:element name="PartColor" type="xsd:string" />
                        <xsd:element name="PartKey" type="xsd:int" />
                        </xsd:all>

</xsd:complexType>
</xsd:element>
<xsd:element name="Invoice">
        <xsd:complexType>
                <xsd:all>
                        <xsd:element name="OrderDate" type="xsd:string" />
                        <xsd:element name="InvoiceKey" type="xsd:int" />
                        <xsd:element name="ShipDate" type="xsd:string" />
                        <xsd:element name="CustomerKey" type="xsd:int" />
                </xsd:all>
        </xsd:complexType>
</xsd:element>
<xsd:element name="Customer">
        <xsd:complexType>
                <xsd:all>
                        <xsd:element name="Name" type="xsd:string" />
                        <xsd:element name="State" type="xsd:string" />
                        <xsd:element name="PostalCode" type="xsd:string" />
                        <xsd:element name="City" type="xsd:string" />
                        <xsd:element name="CustomerKey" type="xsd:int" />
                        <xsd:element name="Address" type="xsd:string" />
                </xsd:all>
        </xsd:complexType>
</xsd:element>
```

Figure 5. 27: XML schema for customer database Part-II

```
        <xsd:element name="LineItem">
                <xsd:complexType>
                        <xsd:all>
                                <xsd:element name="Price" type="xsd:float" />
                                <xsd:element name="InvoiceKey" type="xsd:int" />
                                <xsd:element name="LineItemKey" type="xsd:int" />
                                <xsd:element name="PartKey" type="xsd:int" />
                                <xsd:element name="Quantity" type="xsd:int" />
                        </xsd:all>
                </xsd:complexType>
        </xsd:element>
</xsd:schema>
```

Figure 5.28: XML schema for customer database Part-III

5.2.1.2.3.2 IPEDO XML document generation

The database transformation tool generates the XML document from XML data document using logic explained in section 4.5.2. The structure of this XML is valid against the XML Schema generated in previous section. Resultant XML document is shown in *Figures 5.29, 5.30* and *5.31*.

```
<topstructure>
        <Part>
                <PartKey>1</PartKey>
                <PartName>football</PartName>
                <PartColor>white</PartColor>
                <PartSize>5</PartSize>
        </Part>
        <Part>
                <PartKey>2</PartKey>
                <PartName>baseball</PartName>
                <PartColor>grey</PartColor>
                <PartSize>2</PartSize>
        </Part>
```

Figure 5.29: XML document for IPEDO Part-I

```
<Part>
        <PartKey>3</PartKey>
        <PartName>speakers</PartName>
        <PartColor>black</PartColor>
        <PartSize>5</PartSize>
</Part>
<Part>
        <PartKey>4</PartKey>
        <PartName>computer case</PartName>
        <PartColor>white</PartColor>
        <PartSize>8</PartSize>
</Part>
<Invoice>
        <InvoiceKey>1</InvoiceKey>
        <CustomerKey>1</CustomerKey>
        <OrderDate>2001-01-01 00:00:00.000</OrderDate>
        <ShipDate>2001-01-10 00:00:00.000</ShipDate>
</Invoice>
<Invoice>
        <InvoiceKey>2</InvoiceKey>
        <CustomerKey>2</CustomerKey>
        <OrderDate>2001-02-01 00:00:00.000</OrderDate>
        <ShipDate>2001-02-12 00:00:00.000</ShipDate>
</Invoice>
<Invoice>
        <InvoiceKey>3</InvoiceKey>
        <CustomerKey>3</CustomerKey>
        <OrderDate>2001-05-10 00:00:00.000</OrderDate>
        <ShipDate>2001-05-15 00:00:00.000</ShipDate>
</Invoice>
<Invoice>
        <InvoiceKey>4</InvoiceKey>
        <CustomerKey>4</CustomerKey>
        <OrderDate>2001-06-20 00:00:00.000</OrderDate>
        <ShipDate>2001-06-25 00:00:00.000</ShipDate>
</Invoice>
<Customer>
        <CustomerKey>1</CustomerKey>
        <Name>jim</Name>
        <Address>1200 A drive</Address>
        <City>San Marcos</City>
        <State>TX</State>
        <PostalCode>78666</PostalCode>
</Customer>
<Customer>
        <CustomerKey>2</CustomerKey>
        <Name>tom</Name>
        <Address>1300 B drive</Address>
        <City>Austin</City>
        <State>TX</State>
```

Figure 5. 30: XML document for IPEDO Part-II

```
            <PostalCode>78752</PostalCode>
        </Customer>
            <Customer>
            <CustomerKey>3</CustomerKey>
            <Name>bob</Name>
            <Address>1400 C drive</Address>
            <City>Austin</City>
            <State>TX</State>
            <PostalCode>78575</PostalCode>
        </Customer>
        <Customer>
            <CustomerKey>4</CustomerKey>
            <Name>mat</Name>
            <Address>200 springs drive</Address>
            <City>San Marcos</City>
            <State>TX</State>
            <PostalCode>78666</PostalCode>
        </Customer>
        <LineItem>
            <LineItemKey>1</LineItemKey>
            <InvoiceKey>1</InvoiceKey>
            <PartKey>1</PartKey>
            <Quantity>2</Quantity>
            <Price>10.0</Price>
        </LineItem>
        <LineItem>
            <LineItemKey>2</LineItemKey>
            <InvoiceKey>2</InvoiceKey>
            <PartKey>2</PartKey>
            <Quantity>3</Quantity>
            <Price>20.0</Price>
        </LineItem>
        <LineItem>
            <LineItemKey>3</LineItemKey>
            <InvoiceKey>3</InvoiceKey>
            <PartKey>3</PartKey>
            <Quantity>10</Quantity>
            <Price>200.0</Price>
        </LineItem>
        <LineItem>
            <LineItemKey>4</LineItemKey>
            <InvoiceKey>4</InvoiceKey>
            <PartKey>4</PartKey>
            <Quantity>10</Quantity>
            <Price>300.0</Price>
        </LineItem>
</topstructure>
```

Figure 5. 31: XML document for IPEDO Part-III

5.2.2 Case Study: Object-oriented database to Heterogeneous databases

In this case study we will discuss the customer database transformation inserted into the Ozone database in the previous case study to IPEDO and SQL Server 2000. The first step is the forward transformation and then reverse transformation.

5.2.2.1 Forward Transformation

The first we will generate the XML structure document and then the XML data document from the customer database stored in Ozone.

5.2.2.1.1 Structure generation

Consider the Java class files shown in *Figures 5.16, 5.17, 5.18, 5.19, 5.20, 5.21, 5.22 and 5.23.* Using the tool we will parse these files and extract necessary information and generate the XML structure document as shown in *Figure 5.32.*

```
<DatabaseStructure name="CustomerDatabase">
        <ComponentList>
                <ComponentDescription name="Customer">
                        <AttributeList>
                                <Attribute name="Name" datatype="String" />
                                <Attribute name="State" datatype="String " />
                                <Attribute name="PostalCode" datatype="String" />
                                <Attribute name="City" datatype="String" />
                                <Attribute name="CustomerKey" datatype="int" />
                                <Attribute name="Address" datatype="String" />
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="LineItem">
                        <AttributeList>
                                <Attribute name="Price" datatype="float" />
                                <Attribute name="InvoiceKey" datatype="int" />
                                <Attribute name="LineItemKey" datatype="int" />
                                <Attribute name="PartKey" datatype="int" />
                                <Attribute name="Quantity" datatype="int" />
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="Part">
                        <AttributeList>
                                <Attribute name="PartName" datatype="String" />
                                <Attribute name="PartSize" datatype="String" />
                                <Attribute name="PartColor" datatype="String" />
                                <Attribute name="PartKey" datatype="int" />
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="Invoice">
                        <AttributeList>
                                <Attribute name="OrderDate" datatype="String" />
                                <Attribute name="InvoiceKey" datatype="int" />
                                <Attribute name="ShipDate" datatype="String" />
                                <Attribute name="CustomerKey" datatype="int" />
                        </AttributeList>
                </ComponentDescription>
        </ComponentList>
        <RelationshipList>
                <Relation from="Invoice" to="Customer" cardinality="MANYTOONE" />
                <Relation from="LineItem" to="Invoice" cardinality="MANYTOONE" />
                <Relation from="LineItem" to="Part" cardinality="MANYTOONE" />
        </RelationshipList>
</DatabaseStructure>
```

Figure 5.32: XML structure document representing Ozone Customer database structure

5.2.2.1.2 Data generation

From the XML document shown in *Figure 5.25,* object names are extracted and used to extract object data from Ozone, which is transformed to the XML data, document, which is the same as shown in *Figure 5.24.*

5.2.2.2 Reverse Transformation

In this section we will dicuss the database transformation from XML documents to SQL Server 2000 and IPEDO.

5.2.2.2.1 XML documents to SQL Server 2000

The first step is the structure generation, and the second step is the data generation. First we will generate the XML document with SQL CREATE statements and then XML document with INSERT statements.

5.2.2.2.1.1 Structure generation

Using XML structure documents shown in *Figure5.32,* the tool generates XML document with SQL CREATE statements embedded within tags as shown in *Figure 5.33.*

```
<SQLCREATE>
        <SQLStatement>CREATE TABLE Part (PartName varchar(50), PartSize  varchar(10), PartColor
        varchar(10),PartKey int);
         </SQLStatement>
        <SQLStatement>CREATE TABLE Invoice (OrderDate datetime, InvoiceKey int,ShipDate datetime,
        CustomerKey int);
         </SQLStatement>
        <SQLStatement>CREATE TABLE Customer ( Name varchar(30) , State  varchar(2) , PostalCode
        varchar(10) , City varchar(30) , CustomerKey int, Address varchar(30) );
        </SQLStatement>
        <SQLStatement>CREATE TABLE LineItem (Price float(24) , InvoiceKey int , LineItemKey int , PartKey
        int , Quantity int );
        </SQLStatement>
</SQLCREATE>
```

Figure 5. 33: XML document with SQL CREATE statement for SQL Server 2000

5.2.2.2.1.2 Data generation

The tool generates a XML document with SQL INSERT statements, which is the same as shown in *Figure 5.15,* from the XML data document shown in *Figure5.24.*

5.2.2.2.2 XML documents to IPEDO

Here we will first generate XML schema and then XML the document to store in IPEDO.

5.2.2.2.2.1 Structure generation

Using the XML structure document shown in *Figure 5.32,* the tool generates the XML schema shown in *Figure 5.34.*

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
	<xsd:element name="topstructure">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element ref="Part" maxOccurs="unbounded" />
				<xsd:element ref="Invoice" maxOccurs="unbounded" />
				<xsd:element ref="Customer" maxOccurs="unbounded" />
				<xsd:element ref="LineItem" maxOccurs="unbounded" />
			</xsd:sequence>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="Part">
		<xsd:complexType>
			<xsd:all>
				<xsd:element name="PartName" type="xsd:string" />
				<xsd:element name="PartSize" type="xsd:string" />
				<xsd:element name="PartColor" type="xsd:string" />
				<xsd:element name="PartKey" type="xsd:int" />
			</xsd:all>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="Invoice">
		<xsd:complexType>
			<xsd:all>
				<xsd:element name="OrderDate" type="xsd:string" />
				<xsd:element name="InvoiceKey" type="xsd:int" />
				<xsd:element name="ShipDate" type="xsd:string" />
				<xsd:element name="CustomerKey" type="xsd:int" />
			</xsd:all>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="Customer">
		<xsd:complexType>
			<xsd:all>
				<xsd:element name="Name" type="xsd:string" />
				<xsd:element name="State" type="xsd:string" />
				<xsd:element name="PostalCode" type="xsd:string" />
				<xsd:element name="City" type="xsd:string" />
				<xsd:element name="CustomerKey" type="xsd:int" />
				<xsd:element name="Address" type="xsd:string" />
			</xsd:all>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="LineItem">
		<xsd:complexType>
			<xsd:all>
				<xsd:element name="Price" type="xsd:float" />
				<xsd:element name="InvoiceKey" type="xsd:int" />
				<xsd:element name="LineItemKey" type="xsd:int" />
				<xsd:element name="PartKey" type="xsd:int" />
				<xsd:element name="Quantity" type="xsd:int" />
			</xsd:all>
		</xsd:complexType>
	</xsd:element>
</xsd:schema>
```

Figure 5.34: XML Schema generated for IPEDO from XML structure document

5.2.2.2.2.2 Data generation

Using the XML data document shown in *Figure 5.15,* the tool generates the XML document shown in *Figures 5.29, 5.30* and *5.31.*

5.2.3 Case Study: Hierarchical database to Heterogeneous databases

In this case study we will discuss transformation of the customer database transformation from IPEDO to OZONE and SQL Server 2000.

5.2.3.1 Forward Transformation

First we will transform XML schema to the XML structure document and then transform the XML document to the XML data document.

5.2.3.1.1 Structure generation

Consider the XML schema for the customer database shown in *Figures 5.26, 5.27* and *5.28.* The XML XPY representation for the customer database is shown in *Figure 5.35.* Using the tool XML structure document is generated as shown in *Figure 5.36* from the Customer database schema.

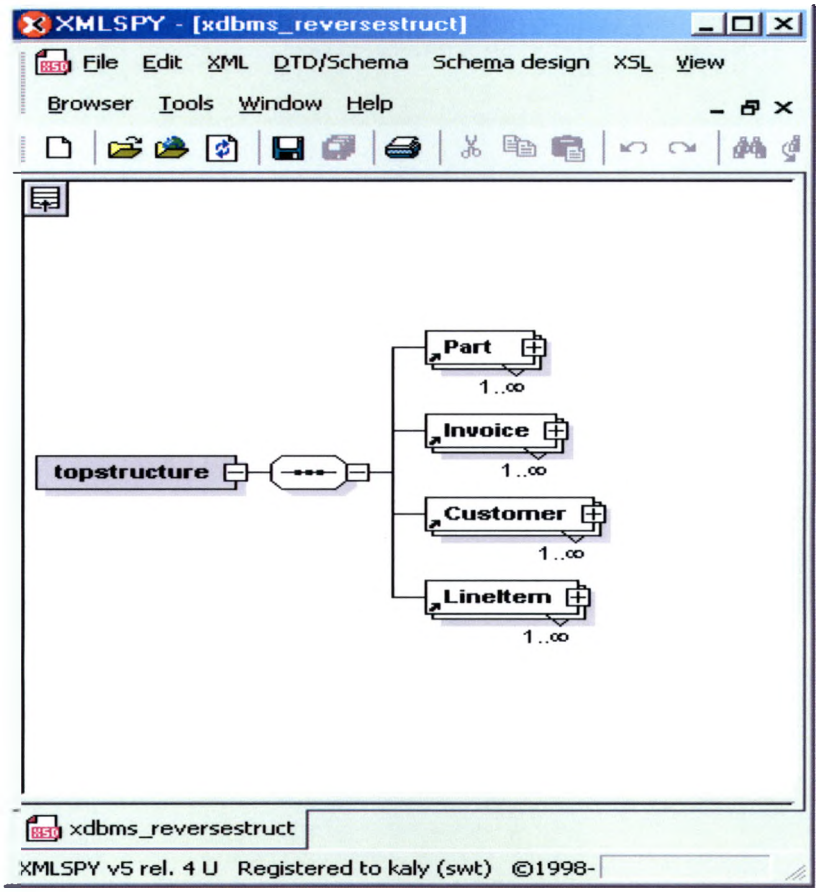

Figure 5.35: XML SPY representation for Customer Schema

```
<DatabaseStructure name="Customer">
        <ComponentList>
                <ComponentDescription name="Schema_Customer">
                        <PrimaryKey>
                                <Key name="CustomerKey" />
                        </PrimaryKey>
                        <AttributeList>
                                <Attribute name="Name" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="State" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="PostalCode" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="City" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="CustomerKey" datatype="xsd:int" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="Address" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="Schema_LineItem">
                        <PrimaryKey>
                                <Key name="LineItemKey" />
                        </PrimaryKey>
                        <ForeignKey>
                                <Key name="InvoiceKey" referTable="Schema_Invoice" referTableColumn="InvoiceKey" />
                                <Key name="PartKey" referTable="Schema_Part" referTableColumn="PartKey" />
                        </ForeignKey>
                        <AttributeList>
                                <Attribute name="Price" datatype="xsd:float" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="InvoiceKey" datatype="xsd:int" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="LineItemKey" datatype="xsd:int" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="PartKey" datatype="xsd:int" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="Quantity" datatype="xsd:int" isNull="YES" maxOccurs="1" minOccurs="1" />
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="Schema_Part">
                        <PrimaryKey>
                                <Key name="PartKey" />
                        </PrimaryKey>
                        <AttributeList>
                                <Attribute name="PartName" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="PartSize" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="PartColor" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="PartKey" datatype="xsd:int" isNull="YES" maxOccurs="1" minOccurs="1" />
                        </AttributeList>
                </ComponentDescription>
                <ComponentDescription name="Schema_Invoice">
                        <PrimaryKey>
                                <Key name="InvoiceKey" />
                        </PrimaryKey>
                        <ForeignKey>
                                <Key name="CustomerKey" referTable="Schema_Customer" referTableColumn="CustomerKey" />
                        </ForeignKey>
                        <AttributeList>
                                <Attribute name="OrderDate" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="InvoiceKey" datatype="xsd:int" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="ShipDate" datatype="xsd:string" isNull="YES" maxOccurs="1" minOccurs="1" />
                                <Attribute name="CustomerKey" datatype="xsd:int" isNull="YES" maxOccurs="1" minOccurs="1" />
                        </AttributeList>
                </ComponentDescription>
        </ComponentList>
</DatabaseStructure>
```

Figure 5. 36: XML structure document representing XML Schema

5.2.2.1.2 Data generation

Consider the XML document for Customer database data shown in *Figures 5.29, 5.30* and *5.31.* The tool generates an XML data document shown in *Figures 5.37,5.38* and *5.39* from the Customer database data.

```
<DatabaseData name="topstructure">
        <ComponentList>
                <ComponentData name="Schema_Part">
                        <AttributeDataList>
                                <ObjectData>
                                        <Data name="PartKey" value="1" />
                                        <Data name="PartName" value="football" />
                                        <Data name="PartColor" value="white" />
                                        <Data name="PartSize" value="5" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="PartKey" value="2" />
                                        <Data name="PartName" value="baseball" />
                                        <Data name="PartColor" value="grey" />
                                        <Data name="PartSize" value="2" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="PartKey" value="3" />
                                        <Data name="PartName" value="speakers" />
                                        <Data name="PartColor" value="black" />
                                        <Data name="PartSize" value="5" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="PartKey" value="4" />
                                        <Data name="PartName" value="computer case" />
                                        <Data name="PartColor" value="white" />
                                        <Data name="PartSize" value="8" />
                                </ObjectData>
                        </AttributeDataList>
                </ComponentData>
                <ComponentData name=" Schema_Invoice">
                        <AttributeDataList>
                                <ObjectData>
                                        <Data name="InvoiceKey" value="1" />
                                        <Data name="CustomerKey" value="1" />
                                        <Data name="OrderDate" value="2001-01-01 00:00:00.000" />
                                        <Data name="ShipDate" value="2001-01-10 00:00:00.000" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="InvoiceKey" value="2" />
                                        <Data name="CustomerKey" value="2" />
                                        <Data name="OrderDate" value="2001-02-01 00:00:00.000" />
                                        <Data name="ShipDate" value="2001-02-12 00:00:00.000" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="InvoiceKey" value="3" />
                                        <Data name="CustomerKey" value="3" />
                                        <Data name="OrderDate" value="2001-05-10 00:00:00.000" />
                                        <Data name="ShipDate" value="2001-05-15 00:00:00.000" />
                                </ObjectData>
                                <ObjectData>
                                        <Data name="InvoiceKey" value="4" />
                                        <Data name="CustomerKey" value="4" />
                                        <Data name="OrderDate" value="2001-06-20 00:00:00.000" />
                                        <Data name="ShipDate" value="2001-06-25 00:00:00.000" />
                                </ObjectData>
                        </AttributeDataList>
                </ComponentData>
```

Figure 5. 37: XML data document for IPEDO XML document Part-I

```
<ComponentData name=" Schema_Customer">
        <AttributeDataList>
                <ObjectData>
                        <Data name="CustomerKey" value="1" />
                        <Data name="Name" value="jim" />
                        <Data name="Address" value="1200 A drive" />
                        <Data name="City" value="San Marcos" />
                        <Data name="State" value="TX" />
                        <Data name="PostalCode" value="78666" />
                </ObjectData>
                <ObjectData>
                        <Data name="CustomerKey" value="2" />
                        <Data name="Name" value="tom" />
                        <Data name="Address" value="1300 B drive" />
                        <Data name="City" value="Austin" />
                        <Data name="State" value="TX" />
                        <Data name="PostalCode" value="78752" />
                </ObjectData>
                <ObjectData>
                        <Data name="CustomerKey" value="3" />
                        <Data name="Name" value="bob" />
                        <Data name="Address" value="1400 C drive" />
                        <Data name="City" value="Austin" />
                        <Data name="State" value="TX" />
                        <Data name="PostalCode" value="78575" />
                </ObjectData>
                <ObjectData>
                        <Data name="CustomerKey" value="4" />
                        <Data name="Name" value="mat" />
                        <Data name="Address" value="200 springs drive" />
                        <Data name="City" value="San Marcos" />
                        <Data name="State" value="TX" />
                        <Data name="PostalCode" value="78666" />
                </ObjectData>
        </AttributeDataList>
</ComponentData>
<ComponentData name=" Schema_LineItem">
        <AttributeDataList>
                <ObjectData>
                        <Data name="LineItemKey" value="1" />
                        <Data name="InvoiceKey" value="1" />
                        <Data name="PartKey" value="1" />
                        <Data name="Quantity" value="2" />
                        <Data name="Price" value="10.0" />
                </ObjectData>
                <ObjectData>
                        <Data name="LineItemKey" value="2" />
                        <Data name="InvoiceKey" value="2" />
                        <Data name="PartKey" value="2" />
                        <Data name="Quantity" value="3" />
                        <Data name="Price" value="20.0" />
                </ObjectData>
                <ObjectData>
                        <Data name="LineItemKey" value="3" />
                        <Data name="InvoiceKey" value="3" />
                        <Data name="PartKey" value="3" />
                        <Data name="Quantity" value="10" />
                        <Data name="Price" value="200.0" />
                </ObjectData>
```

Figure 5. 38: XML data document for IPEDO XML document Part-II

```
                    <ObjectData>
                        <Data name="LineItemKey" value="4" />
                        <Data name="InvoiceKey" value="4" />
                        <Data name="PartKey" value="4" />
                        <Data name="Quantity" value="10" />
                        <Data name="Price" value="300.0" />
                    </ObjectData>
                </AttributeDataList>
            </ComponentData>
        </ComponentList>
</DatabaseData>
```

Figure 5. 39: XML data document for IPEDO XML document Part-III

5.2.3.2 Reverse Transformation

We will transform the customer database structure and data represented by the XML structure document and the XML data document to SQL Server 2000 and Ozone.

5.2.3.2.1 XML documents to SQL Server 2000

The first step is generating the structure and the second step is generating the data.

5.2.3.2.1.1 Structure generation

From the XML structure document shown in *Figure 5.36,* the tool generates an XML document with the SQL CREATE statement as shown in *Figure 5.40*

```
<SQLCREATE>
    <SQLStatement>CREATE TABLE Schema_Customer ( Name varchar(50) , State varchar(50) ,
    PostalCode varchar(50) , City varchar(50) , CustomerKey int primary key , Address varchar(50) );
    </SQLStatement>
    <SQLStatement>CREATE TABLE Schema_LineItem ( Price float(50) , InvoiceKey int ,
    LineItemKey int primary key , PartKey int , Quantity int );
    </SQLStatement>
    <SQLStatement>CREATE TABLE Schema_Part ( PartName varchar(50) , PartSize varchar(50) ,
    PartColor varchar(50) , PartKey int primary key );
    </SQLStatement>
    <SQLStatement>CREATE TABLE Schema_Invoice ( OrderDate varchar(50) , InvoiceKey int
     primary key , ShipDate varchar(50) , CustomerKey int );
    </SQLStatement>
</SQLCREATE>
```

Figure 5.40: SQL CREATE statements for IPEDO XML Schema customer database

The database transformation tool generates anXML document with SQL ALTER statements shown in *Figure 5.41* from the information embedded in foreign key tags of the XML structure document.

```
<ForeignKey>
      <SQLSTATEMENT>ALTER TABLE Schema_LineItem ADD FOREIGN
      KEY(InvoiceKey) REFERENCES Schema_Invoice
      </SQLSTATEMENT>
      <SQLSTATEMENT>ALTER TABLE Schema_LineItem ADD FOREIGN
      KEY(PartKey) REFERENCES Schema_Part
      </SQLSTATEMENT>
      <SQLSTATEMENT>ALTER TABLE Schema_Invoice ADD FOREIGN
      KEY(CustomerKey) REFERENCES Schema_Customer
      </SQLSTATEMENT>
</ForeignKey>
```

Figure 5.41: XML document with SQL ALTER statements

5.2.3.2.1.2 Data generation

The tool generates an XML document with SQL INSERT statements, which is shown in *Figures 5.42* and *5.43,* from the XML data document shown in *Figures 5.37, 5.38* and *5.39.*

```
<rdbmsinsertstatements>
      <sqlstatement>insert into Schema_Part ( PartKey, PartName, PartColor,
      PartSize) values ('1','football','white','5'); </sqlstatement>
      <sqlstatement>insert into Schema_Part ( PartKey, PartName, PartColor,
      PartSize) values ('2','baseball','grey','2');
      </sqlstatement>
      <sqlstatement>insert into Schema_Part ( PartKey, PartName,
      PartColor, PartSize) values('3','speakers','black','5');
      </sqlstatement>
      <sqlstatement>insert into Schema_Part ( PartKey, PartName, PartColor,
      PartSize) values ('4','computer case','white','8');
      </sqlstatement>
      <sqlstatement>insert into Schema_Invoice ( InvoiceKey, CustomerKey,
      OrderDate, ShipDate) values ('1','1','2001-01-01 00:00:00.000','2001-
      01-10 00:00:00.000');
      </sqlstatement>
      <sqlstatement>insert into Schema_Invoice ( InvoiceKey, CustomerKey,
      OrderDate, ShipDate) values ('2','2','2001-02-01 00:00:00.000','2001-
      02-12 00:00:00.000');
      </sqlstatement>
      <sqlstatement>insert into Schema_Invoice ( InvoiceKey, CustomerKey,
      OrderDate, ShipDate) values ('3','3','2001-05-10 00:00:00.000','2001-
      05-15 00:00:00.000');
      </sqlstatement>
      <sqlstatement>insert into Schema_Invoice ( InvoiceKey, CustomerKey,
      OrderDate, ShipDate) values ('4','4','2001-06-20 00:00:00.000','2001-
      06-25 00:00:00.000');
      </sqlstatement>
      <sqlstatement>insert into Schema_Customer ( CustomerKey, Name,
      Address, City, State, PostalCode) values ('1','jim','1200 A
      drive','San Marcos','TX','78666');
      </sqlstatement>
      <sqlstatement>insert into Schema_Customer ( CustomerKey, Name,
      Address, City, State, PostalCode) values ('2','tom','1300 B
      drive','Austin','TX','78752');
      </sqlstatement>
      <sqlstatement>insert into Schema_Customer ( CustomerKey, Name,
      Address, City, State, PostalCode) values ('3','bob','1400 C
      drive','Austin','TX','78575');
      </sqlstatement>
      <sqlstatement>insert into Schema_Customer ( CustomerKey, Name,
      Address, City, State, PostalCode) values ('4','mat','200 springs
      drive','San Marcos','TX','78666');
      </sqlstatement>
      <sqlstatement>insert into Schema_LineItem ( LineItemKey, InvoiceKey,
      PartKey, Quantity, Price) values ('1','1','1','2','10.0');
      </sqlstatement>
      <sqlstatement>insert into Schema_LineItem ( LineItemKey, InvoiceKey,
      PartKey, Quantity, Price) values ('2','2','2','3','20.0');
      </sqlstatement>
      <sqlstatement>insert into Schema_LineItem ( LineItemKey, InvoiceKey,
      PartKey, Quantity, Price) values ('3','3','3','10','200.0');
      </sqlstatement>
```

Figure 5.42: XML document with SQL INSERT statements Part-I

```
        <sqlstatement>insert into Schema_LineItem ( LineItemKey, InvoiceKey,
        PartKey, Quantity, Price) values ('4','4','4','10','300.0');
        </sqlstatement>
</rdbmsinsertstatements>
```

Figure 5. 43: XML document with SQL INSERT statements Part-II

5.2.3.2.2 XML documents to Ozone

The first step is generating the structure and the second step is generating the data.

5.2.3.2.2.1 Structure generation

From the XML structure document shown in *Figure 5.36,* the tool generates Java class files, which are shown in *Figures 5.44* thru *Figure 5.51.*

```
import org.ozoneDB.OzoneRemote;

public interface Schema_Customer extends OzoneRemote {
        public void setName(String Name);
        public String getName();
        public void setState(String State);
        public String getState();
        public void setPostalCode(String PostalCode);
        public String getPostalCode();
        public void setCity(String City);
        public String getCity();
        public void setCustomerKey(int CustomerKey);
        public int getCustomerKey();
        public void setAddress(String Address);
        public String getAddress();

}
```

Figure 5. 44: Schema_Customer Interface for Schema_Customer element

```
import org.ozoneDB.OzoneRemote;

public interface Schema_Invoice extends OzoneRemote {
        public void setOrderDate(String OrderDate);
        public String getOrderDate();
        public void setInvoiceKey(int InvoiceKey);
        public int getInvoiceKey();
        public void setShipDate(String ShipDate);
        public String getShipDate();
        public void setCustomerKey(int CustomerKey);
        public int getCustomerKey();

}
```

Figure 5.45: Schema_Invoice interface for Schema_Invoice element

```
import org.ozoneDB.OzoneObject;
import java.util.*;

public class Schema_CustomerImpl extends OzoneObject implements Schema_Customer {

        String Name;
        public void setName(String Name) { this.Name=Name; }

        public String getName(){ return Name; }
        String State;
        public void setState(String State) { this.State=State; }

        public String getState(){ return State; }
        String PostalCode;
        public void setPostalCode(String PostalCode) { this.PostalCode=PostalCode; }

        public String getPostalCode(){ return PostalCode; }
        String City;
        public void setCity(String City) { this.City=City; }

        public String getCity(){ return City; }
        int CustomerKey;
        public void setCustomerKey(int CustomerKey) { this.CustomerKey=CustomerKey; }

        public int getCustomerKey(){ return CustomerKey; }
        String Address;
        public void setAddress(String Address) { this.Address=Address; }

        public String getAddress(){ return Address; }
}
```

Figure 5. 46: Schema_Customer for Schema_Customer element

```
import org.ozoneDB.OzoneRemote;

public interface Schema_LineItem extends OzoneRemote {
        public void setPrice(float Price);
        public float getPrice();
        public void setInvoiceKey(int InvoiceKey);
        public int getInvoiceKey();
        public void setLineItemKey(int LineItemKey);
        public int getLineItemKey();
        public void setPartKey(int PartKey);
        public int getPartKey();
        public void setQuantity(int Quantity);
        public int getQuantity();

}
```

Figure 5.47: Schema_LineItem interface for Schema_LineItem element

```
import org.ozoneDB.OzoneObject;
import java.util.*;

public class Schema_InvoiceImpl extends OzoneObject implements Schema_ Invoice {

        Schema_CustomerImpl customer;
        String OrderDate;
        public void setOrderDate(String OrderDate) { this.OrderDate=OrderDate; }

        public String getOrderDate(){ return OrderDate; }
        int InvoiceKey;
        public void setInvoiceKey(int InvoiceKey) { this.InvoiceKey=InvoiceKey; }

        public int getInvoiceKey(){ return InvoiceKey; }
        String ShipDate;
        public void setShipDate(String ShipDate) { this.ShipDate=ShipDate; }

        public String getShipDate(){ return ShipDate; }
        int CustomerKey;
        public void setCustomerKey(int CustomerKey) { this.CustomerKey=CustomerKey; }

        public int getCustomerKey(){ return CustomerKey; }
}
```

Figure 5.48: Schema_Invoice class for Schema_Invoice element

```
import org.ozoneDB.OzoneRemote;

public interface Schema_Part extends OzoneRemote {
        public void setPartName(String PartName);
        public String getPartName();
        public void setPartSize(String PartSize);
        public String getPartSize();
        public void setPartColor(String PartColor);
        public String getPartColor();
        public void setPartKey(int PartKey);
        public int getPartKey();

}
```

Figure 5. 49: Schema_Part Interface for Schema_Part element

```
import org.ozoneDB.OzoneObject;
import java.util.*;

public class Schema_PartImpl extends OzoneObject implements Schema_Part {


        String PartName;
        public void setPartName(String PartName) { this.PartName=PartName; }

        public String getPartName(){ return PartName; }
        String PartSize;
        public void setPartSize(String PartSize) { this.PartSize=PartSize; }

        public String getPartSize(){ return PartSize; }
        String PartColor;
        public void setPartColor(String PartColor) { this.PartColor=PartColor; }

        public String getPartColor(){ return PartColor; }
        int PartKey;
        public void setPartKey(int PartKey) { this.PartKey=PartKey; }

        public int getPartKey(){ return PartKey; }
}
```

Figure 5.50: Schema_PartImpl class for Schema_Part element

```
import org.ozoneDB.OzoneObject;
import java.util.*;

public class Schema_LineItemImpl extends OzoneObject implements Schema_LineItem {

        Schema_InvoiceImpl invoice;
        Schema_PartImpl part;

        float Price;
        public void setPrice(float Price) { this.Price=Price; }

        public float getPrice(){ return Price; }
        int InvoiceKey;
        public void setInvoiceKey(int InvoiceKey) { this.InvoiceKey=InvoiceKey; }

        public int getInvoiceKey(){ return InvoiceKey; }
        int LineItemKey;
        public void setLineItemKey(int LineItemKey) { this.LineItemKey=LineItemKey; }

        public int getLineItemKey(){ return LineItemKey; }
        int PartKey;
        public void setPartKey(int PartKey) { this.PartKey=PartKey; }

        public int getPartKey(){ return PartKey; }
        int Quantity;
        public void setQuantity(int Quantity) { this.Quantity=Quantity; }

        public int getQuantity(){ return Quantity; }
}
```

Figure 5.51: Schema_LineItemImpl class for Schema_LineItem element

5.2.3.2.2.2 Object-oriented database data generation

From the XML data document, the database transformation tool generates a Java class file with the method that calls for object creation and insertion into the Ozone database using logic explained in section 4.5.2. The generated driver program to insert objects is shown in *Figure5.52.*

```
import java.util.*;
import org.ozoneDB.*;
public class OODBMSDataDriver {
        public static ExternalDatabase      db;
        public static void main( String[] args ) throws Exception {
                db = ExternalDatabase.openDatabase( "ozonedb.remote://localhost:3333" );
                System.out.println( "Connected..." );
                db.reloadClasses();
                OODBMSDataDriver od=new OODBMSDataDriver();
                od.start();
                db.close();
        }
        public void start(){
                createSchema_Part("objPart1","football","5","white",1);
                createSchema_Part("objPart3","baseball","2","grey",2);
                createSchema_Part("objPart5","speakers","5","black",3);
                createSchema_Part("objPart7","computer case","8","white",4);
                createSchema_Invoice("objInvoice1","2001-01-01 00:00:00.000",1,"2001-01-10 00:00:00.000",1);
                createSchema_Invoice("objInvoice3","2001-02-01 00:00:00.000",2,"2001-02-12 00:00:00.000",2);
                createSchema_Invoice("objInvoice5","2001-05-10 00:00:00.000",3,"2001-05-15 00:00:00.000",3);
                createSchema_Invoice("objInvoice7","2001-06-20 00:00:00.000",4,"2001-06-25 00:00:00.000",4);
                createSchema_Customer("objCustomer1","jim","TX","78666","San Marcos",1,"1200 A drive");
                createSchema_Customer("objCustomer3","tom","TX","78752","Austin",2,"1300 B drive");
                createSchema_Customer("objCustomer5","bob","TX","78575","Austin",3,"1400 C drive");
                createSchema_Customer("objCustomer7","mat","TX","78666","San Marcos",4,"200 springs drive");
                createSchema_LineItem("objLineItem1", (float)10.0,1,1,1,2);
                createSchema_LineItem("objLineItem3", (float)20.0,2,2,2,3);
                createSchema_LineItem("objLineItem5", (float)200.0,3,3,3,10);
                createSchema_LineItem("objLineItem7", (float)300.0,4,4,4,10);
        }

        public void createSchema_Customer(String objName,String var5,String var7,String var9,String var11,int var13,String var15)
        {
                Schema_Customer schema_customerObj = (Schema_Customer )(db.createObject( Schema_CustomerImpl.class.getName(), 0, objName ));
                schema_customerObj.setName(var5);
                schema_customerObj.setState(var7);
                schema_customerObj.setPostalCode(var9);
                schema_customerObj.setCity(var11);
                schema_customerObj.setCustomerKey(var13);
                schema_customerObj.setAddress(var15);

        }
        public void createSchema_LineItem(String objName, float var5,int var7,int var9,int var11,int var13)
        {
                Schema_LineItem schema_lineitemObj = (Schema_LineItem )(db.createObject( Schema_LineItemImpl.class.getName(), 0, objName ));
                schema_lineitemObj.setPrice(var5);
                schema_lineitemObj.setInvoiceKey(var7);
                schema_lineitemObj.setLineItemKey(var9);
                schema_lineitemObj.setPartKey(var11);
                schema_lineitemObj.setQuantity(var13);

        }
        public void createSchema_Part(String objName,String var5,String var7,String var9,int var11)
        {
                Schema_Part schema_partObj = (Schema_Part )(db.createObject( Schema_PartImpl.class.getName(), 0, objName ));
                schema_partObj.setPartName(var5);
                schema_partObj.setPartSize(var7);
                schema_partObj.setPartColor(var9);
                schema_partObj.setPartKey(var11);

        }
        public void createSchema_Invoice(String objName,String var5,int var7,String var9,int var11)
        {
                Schema_Invoice schema_invoiceObj = (Schema_Invoice )(db.createObject( Schema_InvoiceImpl.class.getName(), 0, objName ));
                schema_invoiceObj.setOrderDate(var5);
                schema_invoiceObj.setInvoiceKey(var7);
                schema_invoiceObj.setShipDate(var9);
                schema_invoiceObj.setCustomerKey(var11);

        }
}
```

Figure 5. 52: Java driver class

CHAPTER V

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this research we proposed a generic framework for database transformation between heterogeneous databases using XML as intermediate format. We learned that each data model has its own theoretical base for representing components, attributes of components, data type conventions, size of attributes, relationships between components. Based on these criteria of data models, we were able to successfully evolve the structure DTD and data DTD. Structure DTD and data DTD define generic tags for representing heterogeneous database structure and data independent of its theoretical base. Along with these DTDs we defined several mapping techniques from heterogeneous databases to the hierarchical structure of XML documents specified by the structure DTD and the data DTD and vice versa. In this research, we successfully achieved the following objectives:

- Design of structure DTD and data DTD for representing database structure and data
- Design of data type mapping DTD to support data type mapping between heterogeneous databases data type conventions.
- Database transformation from hierarchical database (XML documents) to relational databases and vice-versa.
- Database transformation from hierarchical database (XML documents) to object-oriented databases and vice-versa.
- Database transformation from relational databases to object-oriented databases and vice-versa.

Some of the problems we faced in the design of framework for database transformation process are as follows:

Naming problems: As each data model has its own naming rules for specifying names of its components and attributes, conflict arises when we transform a database from a relational database to a hierarchical database and object-oriented database. A relational database allows tables and columns to be made up from two words; for example, consider the column name "Category Name." When this column name is transformed to the equivalent variable name in an object-oriented database or the element name in a hierarchical database it is considered illegal. So we formed a single word by combining both words before transforming.

Inheritance: XML schemas and programming languages provide an easy way to extend existing structures through inheritance, but because of lack of support for inheritance in relational databases, we resolved the inheritance by adding the columns of the parent table to the child table.

Illegal characters: XML does not allow direct use of characters such as &, <, > in the data, while tables and programming languages allow it. So while transforming data from relational and object-oriented databases to XML, illegal characters are replaced with appropriate characters.

Size of attributes: When creating table structure in a relational database, one should define the column size, but in case of a hierarchical database or an object-oriented database, we do not have to specify size. So, when we create a table structure in a relational database for database being transformed from a hierarchal or object-oriented

database, we specified default values for the size of columns depending on their data type.

6.2 Future work

This research is able to address most of the basic issues related to database transformation between heterogeneous databases using XML as an intermediate format. Some of the possible future work:

Design of wrappers on top of the XML documents. This wrapper will use the XML documents as the temporary database and provide basic database services.

DTD Extensions: The structure DTD and data DTD can be extended to support post-relational database's and object-relational database's theoretical base.

GUI enhancements: The visual GUI part of the database transformation tool can be modified to support database design changes through the logical diagrams for the database to be transformed.

Data type support: Support for data types can be extended. At present, this research supports basic data types such as int, float and text. It can be extended to support other data types such as: multimedia data types which include video, audio and image files, GIS data types such as vector data type and raster data type; and programming language data types such as Vectors and Hash tables.

BIBLIOGRAPHY

[ArMiSh '01]
I. Budak Arpinar, John Miller, Amit P. Sheth. *An Efficient Data Extraction and Storage Utility for XML Documents.* Technical Report, LSDIS Lab, Computer Science Department, University of Georgia, 2001

[Baru '99]
Chaitanya Baru. *XViews: XML views of relational schemas.* Technical Report, San Diego Supercomputer Center, University of California San Diego, October 7, 1999

[BaKrKrMu '00]
Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, Ravi Murthy. *Oracle8i - The XML Enabled Data Management System.* 16th International conference on Data Engineering, Feburary 28th – March 03, 2000

[BeCa '01]
Elisa Bertino, Barbara Catania. *Integrating XML and Databases.* IEEE Internet computing, Vol. 5, No. 4, pages 84-88, August 2001.
http://www.computer.org/internet/ic2001/w4084abs.htm

[Bierman '00]
G.M. Bierman. *Using XML as an Object Interchange Format.* Technical Report, Department of Computer Science, University of Warwick, May 17, 2000

[BoBoBu '00]
R. Bourret, C. Bornh¨ovd, A. Buchmann. *A Generic Load/Extract Utility for Data Transfer between XML Documents and Relational Databases.* The Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000), P-134, June 2000.
http://www.computer.org/Proceedings/wecwis/0610/06100134abs.htm

[BoFrRoSi '02]
Phil Bohannon, Juliana Freire, Prasan Roy, Jerome Simeon. *From XML Schema to Relations: A Cost-Based Approach to XML Storage.* Technical Report, Bell Laboratories, 2001.
http://citeseer.nj.nec.com/465172.html

[Claypool '02]
Kajal Tilak Claypool. *Managing Schema Change in an Heterogeneous Environment.* Ph.D. diss., Computer Science Department, Worcester Polytechnic Institute 2002.
http://www.wpi.edu/Pubs/ETD/Available/etd-0617102-213436/unrestricted/kclaypool.pdf

[FlKo '99]
Daniela Florescu and Donald Kossmann. *Storing and Querying XML Data using an RDBMS.* IEEE Data Engineering Bulletin, 22(3):27–34, 1999.

[FlLeMe '98]
Daniela Florescu Alon Levy Alberto Mendelzon, *Database Techniques for the World-Wide Web: A Survey.* SIGMOD Record, 27(3):59--74 (1998).

[FoPaBl '01]
Joseph Fong, Francis Pang, Chris Bloor, *Converting Relational Database into XML Document.* The 12th International Workshop on Database and Expert Systems Applications, Munich Germany,September 03 - 07, 2001.
http://www.computer.org/proceedings/dexa/1230/12300061abs.htm?SMSESSION=NO

[Hohenstein '00]
Uwe Hohenstein. *Supporting Data Migration between Relational and Object-Oriented Databases Using a Federation Approach.* 2000 International Database Engineering and Applications Symposium (IDEAS'00), Yokohama, Japan, September 18 - 20, 2000.
http://www.computer.org/proceedings/ideas/0789/07890371abs.htm

[JaScZü '96]
Jens Jahnke, Wilhelm Schäfer, Albert Zündorf. *A Design Environment for Migrating Relational to Object Oriented Database Systems.* 1996 International Conference on Software Maintenance (ICSM '96), Monterey, CA, November 04 - 08, 1996

[Jr '01]
JGraph open source project. *JGraphpad*, 2001.
http://jgraph.sourceforge.net/index.html

[KaRePr '01]
Elisabeth Kapsammer, Werner Retschitzegger, Birgit Proell. *Mapping Database Content to Xml Pages: A Metadata-Based Approach.* 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI), Orlando, July 22-25, 2001 [ISBN 980-07-7545-5].

[KlMe '00]
Meike Klettke, Holger Meyer. *XML and ObjectRelational Database Systems — Enhancing Structural Mappings Based On Statistics.* In: Informal Proc.WebDBWorkshop, pp 151–170, 2000

[KaKaRe '01]
Gerti Kappel, Elisabeth Kapsammer, Werner Retschitzegger. *Architectural Issues for Integrating XML and Relational Database Systems – The X-Ray Approach.* XML

Technologies and Software Engineering (XSE2001) Toronto, Canada, May 12-19, 2001, IEEE Computer Society Press, [ISBN 0-7695-1050-7]

[LeMaChCh '02]
Dongwon Lee, Murali Mani, Frank Chiu, Wesley 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), McLean, VA, USA, November 2002.

[MaLe '02]
Murali Mani, Dongwon Lee.*XML to Relational Conversion using Theory of Regular Tree Grammars.* VLDB Conference, 28th Preceedings, 2002.

[O'Brien '00]
Mark John O'Brien. *Creating data exchange standards with XML: a waste?.* First International Conference on Web Information Systems Engineering (WISE'00)-Volume 2, Hong Kong, China, June 19 - 20, 2000.

[PeHaMa '99]
C. Petrou, S. Hadjiefthymiades, D. Martakos. *An XML-based, 3-tier Scheme for Integrating Heterogeneous Information Sources to the WWW.* 10th International Workshop on Database & Expert Systems Applications, Florence, Italy, September 01 - 03, 1999.

[PsMi '99]
Giuseppe Psaila, Politecnico di Milano, *ERX: A Data Model for Collections of XML Documents.*1999 Workshop on Knowledge and Data Engineering Exchange, Chicago, Illinois, November 07 - 07, 1999.

[Renner '01]
Andreas Renner, *XML Data and Object Databases: The Perfect Couple.*17th International Conference on Data Engineering, Heidelberg, Germany, April 02 - 06, 2001

[RuPa '02]
Kanda Runapongsa and Jignesh M. Patel. *Storing and Querying XML Data in Object-Relational DBMS.* EDBT Workshops 2002

[Schöning '01]
Dr. Harald Schöning, *Tamino – a DBMS Designed for XML.* 17th International Conference on Data Engineering, Heidelberg, Germany, April 02 - 06, 2001

[ShShBaCaLiPiRe '00]

Jayavel Shanmugasundaram, Eugene Shekita, Rimon Barr, Michael Carey, Bruce Lindsay, Hamid Pirahesh, Berthold Reinwald. *Efficiently Publishing Relational Data as XML Documents.* In VLDB 2000.

[Trinidad '00]
Gerardo Trinidad. *XML and Databases for Internet Applications.* Proceedings of the Philippines Computing Science Congress, November 29 - December 1, 2000

[w3c '00]
World Wide Web Consortium. *Extensible Markup Language (XML) 1.0. (Second Edition),* Oct 2000.
http://www.w3.org/TR/1998/REC-xml-19980210.html

[w3c '97]
World Wide Web Consortium Working Draft. *Document Object Model (DOM),* 09 December 1997.
http://www.w3.org/TR/WD-DOM/level-one-core

[Megginson '2002]
David Megginson *The Simple API for XML (SAX),* Jan 2002.
http://www.saxproject.org/

[VaVa '01]
Iraklis Varlamis, Michalis Vazirgiannis. *Bridging XML-Schema and relational databases. A system for generating and manipulating relational databases using valid XML documents.* Proceedings of the 2001 ACM Symposium on Document engineering, Atlanta, Georgia, 2001.

[VoPiRe '01]
Dan Vodislav, Jean-Pierre, and Sirot Chantal Reynaud. *Semantic Integration of XML Heterogeneous Data Sources.* 2001 International Database Engineering & Applications Symposium (IDEAS '01), Grenoble, France, July 16 - 18, 2001

APPENDICES

APPENDIX A1: Source Code for Forward Transformation

```
package com.migrationtool.xmlgenerator;
import com.migrationtool.*;
import com.jgraph.*;
import java.util.*,
import java.io.*;
import java.sql.*;
import com.migrationtool.configfiles.*;
import com.migrationtool.rdbms.*;
import org.w3c.dom.*;
import com.migrationtool.domloader.*;

/**
 * Generates xml struct from classes, xml schemas and tables
 * called by JGraphpad for conversion
 */
public class XMLGenerator {

        //Hashtable holding relational database key information
        Hashtable primaryKeyHash=new Hashtable();
        Hashtable foriegnKeyHash=new Hashtable();
        Hashtable foriegnKeyXMLHash=new Hashtable();
        Hashtable primaryKeyXMLHash=new Hashtable();
        //Hashtable for Object-oriented database class information
        Hashtable classObjectsHash=new Hashtable();
        Hashtable oodbmsHash=new Hashtable();

        //Declaring object for getting file location information
        FileInformation fileInfo=new FileInformation(),

        String targetPrefix="lib:";
        String output;
        String relationship="<RelationshipList>";

        //XML Schema key information
        Hashtable primaryKeyToElementNameHash=new Hashtable();
        Hashtable foriegnKeyToElementNameHash=new Hashtable();
        Hashtable foriegnKeyToPrimaryKeyHash=new Hashtable(),
        Hashtable  elementsKeyAndKeyRefHash=new Hashtable();
        Hashtable referHash=new Hashtable();

        //XML Schema element information
        Hashtable elementsHash=new Hashtable();

        //Hashtable for holding inheritance information
        Hashtable inheritanceHash=new Hashtable();

        //Constructor calls the function to load Hashtable with file location information
```

```
public XMLGenerator(){
        loadFileInfoHash();
}

//Loads config file into hashtable in class FileInformation
public void loadFileInfoHash(){
        fileInfo.loadFileXML("config.xml");
}

//Generated XML structure document for Object-oriented database classes
public String generateOODBMSXMLStruct(Hashtable oodbmsHash){
        this.oodbmsHash=oodbmsHash;

        String output="<DatabaseData name=\"OODBMS\">\n\t<ComponentList>";
        Enumeration enum=oodbmsHash.keys();
        while(enum.hasMoreElements()){
                String className=(String)enum.nextElement();
                output+="<ComponentDescription name=\""+className+"\">";
                output+="\n\t\t\t<AttributeList>";
                Hashtable classInfoHash=(Hashtable)  oodbmsHash.get(className);
                if(classInfoHash.containsKey("VARIABLES")){
                        Vector variablesVect=(Vector)classInfoHash.get("VARIABLES");
                        if(variablesVect.size()==0) continue;
                        System.out.println(variablesVect);
                        for(int i=0;i<variablesVect.size();i++){
                                String variable=(String)variablesVect.elementAt(i);
                                Vector tempVect=parse(variable,":");
                                String datatype=(String)tempVect.elementAt(0);
                                output+="<Attribute name=\""+datatype+"\"" ;

                        }
                }
                output+="\n\t\t\t</AttributeList>";
                output+="\n\t\t</ComponentDescription>";
        }
        output+="\n\t</ComponentList>";
        output+="\n</DatabaseData>";
        Hashtable fileHash=fileInfo.getFileInformation("oodbms","struct");
        String fileName=(String)fileHash.get("filename");
        String filePath=(String)fileHash.get("filelocation");
        writeToFile(output,fileName,filePath);
        return output;
}

//Loads the object names from object-oriented database
public Hashtable loadOODBMSObjectNames(){
        Hashtable fileHash=fileInfo.getFileInformation("oodbms","transformer");
        String fileName=(String)fileHash.get("filename");
        String filePath=(String)fileHash.get("filelocation");
        Document document = null;
        document = DOMUtil.parse(filePath+fileName);
        NodeList list = document.getElementsByTagName("classobjects");
        for (int i = 0; i < list.getLength(); i++){
                Node element = list.item(i);
                Hashtable       attributesHash=getAttributes(element);
```

```
                        String name=(String)attributesHash.get("name");
                        String implementingClass=(String)attributesHash.get("classimplements");
                        Hashtable classHash=new Hashtable();
                        classHash.put("CLASSIMPL",implementingClass);
                        NodeList childNodes=element.getChildNodes();
                        for(int k=0;k<childNodes.getLength();k++){
                                Node childNode=childNodes.item(k);
                                String childName=childNode.getNodeName();
                                if(childName.equalsIgnoreCase("objects")){
                                        NodeList objectNodes=childNode.getChildNodes();
                                        Hashtable objectHash=new Hashtable();
                                        for(int ob=0;ob<objectNodes.getLength();ob++){
                                                Node objectNode=objectNodes.item(ob);
                                                String objectName=objectNode.getNodeName();
                                                if(objectName.equalsIgnoreCase("objectnames")){

        objectHash.put((ob+"").trim(),getAttributes(objectNode));
                                                }
                                        }
                                        classHash.put("OBJECTHASH",objectHash);
                                        classObjectsHash.put(name,classHash);
                                }
                        }
                }
                return classObjectsHash;
        }

        //Generates the driver program to insert objects into Object-oriented database
        public void generateOODBMSDriver(Hashtable oodbmsHash){
                this.oodbmsHash=oodbmsHash;
                loadOODBMSObjectNames();
                String output="";
                String classBody="import java.util.*;\n";
                classBody+="import com.migrationtool.xmlgenerator.*;\n";
                classBody+="import com.migrationtool.oodbms.ozone.testfiles.*;\n";
                classBody+="import org.ozoneDB.*;\n";
                classBody+="public class OODBMSDriver extends OzoneObject{ \n";
                classBody+="public static ExternalDatabase db;\n";
                classBody+=" public static void main( String[] args ) throws Exception {\n";
                classBody+=" db = ExternalDatabase.openDatabase( \"ozonedb:remote://localhost:3333\"
);\n";
                classBody+="  System.out.println( \"Connected ...\" );\n";
                classBody+=" db.reloadClasses();\n";
                classBody+="OODBMSDriver od=new OODBMSDriver();\n";
                classBody+="od.start();\n",
                classBody+="db.close();\n}\n";
                classBody+="XMLGenerator xg=new XMLGenerator();";
                classBody+="Hashtable classObjectsHash=new Hashtable();\n";
                classBody+="public void start(){\n";
                classBody+="classObjectsHash =xg.loadOODBMSObjectNames();\n";
                classBody+="String output=\"\";\n";
                classBody+="Hashtable objectListHash=null;\n";
                classBody+="Enumeration enumObjectHash=null;\n";
                Enumeration enum=classObjectsHash.keys();
                Vector localClassVect=new Vector();
```

```
                while(enum.hasMoreElements()){
                        String className=(String)enum.nextElement();
                        Hashtable classHash=(Hashtable)classObjectsHash.get(className);
                        String classImpl=(String)classHash.get("CLASSIMPL");
                        if(!oodbmsHash.containsKey(classImpl)){
                                continue;
                        }
                        Hashtable objectListHash=(Hashtable)classHash.get("OBJECTHASH");
                        Enumeration enumObjectHash=objectListHash.keys();
                        String
        localClassBody="objectListHash=(Hashtable)classObjectsHash.get(\""+className+"\");\n
enumObjectHash=objectListHash.keys();\n";
                        output+="public void getObjects"+className+"(String objName,String
output){",
                        output+="try{\n";
                        Vector driverMethodVect=new Vector();
                        boolean firstBool=true;
                        boolean firstClassBool=true;
                        while(enumObjectHash.hasMoreElements()){
                                Hashtable
objectHash=(Hashtable)objectListHash.get(enumObjectHash.nextElement());
                                if(firstClassBool){
                                        localClassBody+="
while(enumObjectHash.hasMoreElements()){ \nHashtable
objectHash=(Hashtable)objectListHash.get(enumObjectHash.nextElement());\n",
        localClassBody+="getObjects"+className+"((String)objectHash.get(\"name\"),output);}\n";
                                        firstClassBool=false;
                                        localClassVect.addElement(localClassBody);
                                }
                                Hashtable classInfoHash=(Hashtable)  oodbmsHash.get(classImpl);
                                String objectName=(String)objectHash.get("name");
                                if(classInfoHash.containsKey("METHODVARIABLES")){
                                        Vector
variablesVect=(Vector)classInfoHash.get("METHODVARIABLES");
                                        for(int i=0;i<variablesVect.size();i++){
                                                String variable=(String)variablesVect.elementAt(i);
                                                Vector tempVect=parse(variable,":");
                                                String datatype=(String)tempVect.elementAt(0);
                                                String methodName=(String)tempVect.elementAt(1);
                                                String
variableName=(String)tempVect.elementAt(2),
                                                String tempClass=className+"
"+className.toLowerCase()+" = "+"("+className+")(db.objectForName( objName ));\n";
                                                tempClass+="output+=\"<Data
name=\\\""+variableName+"\\\"
value=\\\"\"+"+className.toLowerCase()+"."+methodName+"()+\"\\\"/>\";";
                                                driverMethodVect.addElement(tempClass);
                                        }
                                }
                                if(firstBool){
                                        for(int m=0;m<driverMethodVect.size();m++){
                                                output+=(String)driverMethodVect.elementAt(m);
                                        }
                                        output+="}catch(Exception ex){System.out.println(ex),}\n}";
                                        firstBool=false;
```

```
                        }
                }
        }
        for(int m=0;m<localClassVect.size();m++){
                classBody+=(String)localClassVect.elementAt(m);
        }
        classBody+="\n}";
        Hashtable fileHash=fileInfo.getFileInformation("oodbms","driver");
        String fileName=(String)fileHash.get("filename");
        String filePath=(String)fileHash.get("filelocation");
        writeToFile(classBody+"\n"+output+"\n}",fileName,filePath);

}

//Gets all the method names for a class
public Vector getInterfaceMethodList(String className){
        Vector methodListVect=new Vector();
        Hashtable classInfoHash=(Hashtable) oodbmsHash.get(className);
        Vector getMethodsVect=(Vector)classInfoHash.get("GETMETHODS");
        Vector setMethodsVect=(Vector)classInfoHash.get("SETMETHODS");
        for(int i=0;i<setMethodsVect.size();i++){
                Hashtable methodInfoHash=(Hashtable)setMethodsVect.elementAt(i);
                String methodName=(String)methodInfoHash.get("METHODNAME");
                String datatype=(String)methodInfoHash.get("METHODDATATYPE");
                methodListVect.addElement(methodName+":"+datatype+"+"),
        }
        for(int i=0;i<getMethodsVect.size();i++){
                Hashtable methodInfoHash=(Hashtable)getMethodsVect.elementAt(i);
                String methodName=(String)methodInfoHash.get("METHODNAME");
                String datatype=(String)methodInfoHash.get("RETURNTYPE");
                methodListVect.addElement(datatype+":"+methodName);
        }
        return methodListVect;
}

//Gets inheritance information for classes
public Vector getInheritanceList(String className){
        Hashtable classInfoHash=(Hashtable)  oodbmsHash.get(className);
        Vector   inheritVect=new Vector();
        if(classInfoHash.containsKey("CLASS")){
                Hashtable classHash=(Hashtable)  classInfoHash.get("CLASS");
                System.out.println(className+"--------> "+classHash);
                inheritVect=(Vector)classHash.get("IMPLEMENTS");
        }
        return inheritVect;
}

//Generates XML data document for Relational database
public String generateERXMLData(Hashtable tableHash ){
        Enumeration tableEnum=tableHash.keys();
        RDBMSData rd=new RDBMSData();
        Connection con=rd.connectDB("SQLSERVER","kalyan","kalyan");
        String output="<DatabaseData name=\""+"pubs"+"\">\n\t<ComponentList>";
        while(tableEnum.hasMoreElements()){
                String tableName=(String)tableEnum.nextElement();
```

```
                String query="select * from \""+tableName+"\"";
                output=rd.extractData(query,tableName,output);
            }
            output+="\n\t</ComponentList>";
            output+="\n</DatabaseData>";
            Hashtable fileHash=fileInfo.getFileInformation("rdbms","data");
            String fileName=(String)fileHash.get("filename");
            String filePath=(String)fileHash.get("filelocation");
            writeToFile(output,fileName,filePath);
            System.out.println("-------------------DONE-------");
            return output;
        }

        //generates the relationship tags representing relationship between tables
        public void setERRelationshipHash(Hashtable primaryKeyHash,Hashtable foriegnKeyHash){
            Enumeration enumForiegn=foriegnKeyHash.keys();
            while(enumForiegn.hasMoreElements()){
                String tableName=(String)enumForiegn.nextElement();
                Vector foriegnTablesVect=(Vector)foriegnKeyHash.get(tableName);
                for(int i=0;i<foriegnTablesVect.size();i++){
                    String referTableName=(String)foriegnTablesVect.elementAt(i);
                    if(primaryKeyHash.containsKey(referTableName)){
                        Vector
referPrimaryKeyVect=(Vector)primaryKeyHash.get(referTableName);
                        Vector
primaryKeyVect=(Vector)primaryKeyHash.get(tableName);
                        for(int v=0,v<referPrimaryKeyVect.size();v++){

        if(primaryKeyVect.indexOf(referPrimaryKeyVect.elementAt(v)) != (-1)){
                                relationship+="<Relation
from=\""+tableName+"\" to=\""+referTableName+"\" cardinality=\"OneToOne\"/>\n";
                            }
                            else{
                                relationship+="<Relation
from=\""+tableName+"\" to=\""+referTableName+"\" cardinality=\"ManyToOne\"/>\n";

                            }
                        }
                    }

                }
            }
            relationship+="</RelationshipList>";
        }

        //Generates the XML structure representation for relational database structure
        public String generateERXMLStruct(Hashtable tableHash){
            String output="<DatabaseStructure>\n\t<ComponentList>";
            Enumeration tableEnum=tableHash.keys();
            while(tableEnum.hasMoreElements()){
                String tableName=(String)tableEnum.nextElement();
                output+="\n\t\t<ComponentDescription name= \""+tableName+"\">";
                Hashtable tableInformationHash=(Hashtable)tableHash.get(tableName);
                Vector
primaryKeyVect=(Vector)tableInformationHash.get("PRIMARYKEYSET");
```

```
                    Vector keySetVect=(Vector)tableInformationHash.get("KEYSET");
                    if(primaryKeyVect != null){
                        if(primaryKeyVect.size() != 0){
                            output+="\n\t\t\t<PrimaryKey>";
                            String primaryKey=(String)primaryKeyVect.elementAt(0);
                            output+="\n\t\t\t\t<Key name=\""+primaryKey+"\"/>";
                            output+="\n\t\t\t</PrimaryKey>";
                        }
                    }
                    if(keySetVect != null){
                        if(keySetVect.size() != 0){
                            output+="\n\t\t\t<ForeignKey>";
                            for(int k=0;k<keySetVect.size();k++){
                                Hashtable
tableKeyHash=(Hashtable)keySetVect.elementAt(k);
                                String
foreignKey=(String)tableKeyHash.get("FKCOLUMNNAME");
                                String
referTable=(String)tableKeyHash.get("PKTABLENAME");
                                output+="\n\t\t\t\t<Key name=\""+foreignKey+"\"
referTable=\""+referTable+"\""+"/>";
                            }
                            output+="\n\t\t\t</ForeignKey>";
                        }
                    }
                    Hashtable columnHash=(Hashtable) tableInformationHash.get("COLUMNS");
                    Enumeration columnEnum=columnHash.keys();
                    output+="\n\t\t\t<AttributeList>";
                    while(columnEnum.hasMoreElements()){
                        String columnName=(String) columnEnum.nextElement();
                        Hashtable
columnInformationHash=(Hashtable)columnHash.get(columnName);
                        String
datatype=(String)columnInformationHash.get("COLUMNTYPE");
                        String
isAuto=(String)columnInformationHash.get("COLUMNAUTO");
                        String
allowsNull=(String)columnInformationHash.get("ALLOWSNULL");
                        String
columnSize=(String)columnInformationHash.get("COLUMNSIZE");
                        output+="\n\t\t\t\t<Attribute name=\""+columnName+"\"
datatype=\""+datatype+" \" isNull=\""+allowsNull+" \" isAutoIncrement=\""+isAuto+"\"
columnSize=\""+columnSize+"\"/>",
                    }
                    output+="\n\t\t\t</AttributeList>";
                    output+="\n\t\t</ComponentDescription>";
                }
                output+="\n\t</ComponentList>";
                output+=relationship;
                output+="\n</DatabaseStructure>";
                Hashtable fileHash=fileInfo.getFileInformation("rdbms","struct");
                String fileName=(String)fileHash.get("filename");
                String filePath=(String)fileHash.get("filelocation");
                writeToFile(output,fileName,filePath);
                return output;
```

```
        }

        //Sets the hash tables with XML Schema element and related information
        public void setXMLSchemaHashtables(Hashtable elementsHash,Hashtable
elementsKeyAndKeyRefHash,Hashtable referHash, Hashtable inheritanceHash){
                this.elementsHash=elementsHash;
                this.elementsKeyAndKeyRefHash=elementsKeyAndKeyRefHash;
                this.referHash=referHash;
                this.inheritanceHash=inheritanceHash;

        }
        //Sets the key information for the root element of XML structure document
        public void setXMLSchemaTopStructureHashtables(Hashtable
primaryKeyToElementNameHash,Hashtable foriegnKeyToElementNameHash,Hashtable
foriegnKeyToPrimaryKeyHash){

                this.primaryKeyToElementNameHash=primaryKeyToElementNameHash;
                this.foriegnKeyToElementNameHash=foriegnKeyToElementNameHash;
                this.foriegnKeyToPrimaryKeyHash=foriegnKeyToPrimaryKeyHash,

        }

        //Extracts the key information from key tag, keyref tag and unique key tag encountered while
processing XML Schema element
        public String seperateKeys(String elementName, String key){
                String output="";
                if(!elementsKeyAndKeyRefHash.containsKey(elementName)){
                        return output;
                }
                Hashtable
keyAndKeyRefHash=(Hashtable)elementsKeyAndKeyRefHash.get(elementName);
                Vector fieldsVect=new Vector();
                String componentPath="";
                if(keyAndKeyRefHash.containsKey(key)){
                        Vector pKeyVect=(Vector) keyAndKeyRefHash.get(key);
                        for(int pk=0;pk<pKeyVect.size();pk++){
                                Hashtable keys=(Hashtable) pKeyVect.elementAt(pk);
                                Enumeration enumKeys=keys.keys();
                                while(enumKeys.hasMoreElements()){
                                        String keyElement=(String)enumKeys.nextElement();
                                        Hashtable pathFieldHash=(Hashtable) keys.get(keyElement);
                                        Hashtable
selectorAttributesHash=(Hashtable)pathFieldHash.get("SELECTOR"),
                                        Hashtable
attributesHash=(Hashtable)pathFieldHash.get("NODE");
                                        Hashtable fieldHash=(Hashtable)
pathFieldHash.get("FIELD");
                                        String selectorPath=(String)pathFieldHash.get("XPATH");
                                        if(key.equalsIgnoreCase("KEY") ||
key.equalsIgnoreCase("UNIQUE")){
                                                Enumeration enumField=fieldHash.keys();
                                                String
primaryKey=(String)enumField.nextElement();
                                                output+="\n\t\t\t\t<Key
name=\""+primaryKey+"\"/>";
```

```
                                        }
                                        if(key.equalsIgnoreCase("KEYREF")){
                                                Enumeration enumField=fieldHash.keys();
                                                String foriegnKey=(String)enumField.nextElement();
                                                String referName=(String)attributesHash.get("refer");
                                                String referTable=(String)referHash.get(referName);
                                                output+="\n\t\t\t\t<Key name=\""+foriegnKey+"\"
referTable=\""+referTable+"\""+"/>";
                                        }
                                }
                        }
                }
                return output;
        }

        //Generated Key tags for XML Schema in the XML structure document.
        public void generateForiegnPrimaryKeys(){
                Hashtable primaryKeyHash=new Hashtable(),
                Hashtable
topHash=(Hashtable)elementsKeyAndKeyRefHash.get("Schema_topstructure");
                Vector keyRefVect=(Vector)topHash.get("KEYREF");
                Vector keyVect=(Vector)topHash.get("KEY");
                for(int i=0;i<keyVect.size();i++){
                        Hashtable keyHash=(Hashtable)keyVect.elementAt(i);
                        Enumeration enumForiegn=keyHash.keys();
                        String primary=(String)enumForiegn.nextElement();
                        String tableName=(String)primaryKeyToElementNameHash.get(primary);
                        Hashtable inHash=(Hashtable)keyHash.get(primary);
                        Hashtable fieldHash=(Hashtable)inHash.get("FIELD");
                        Enumeration enumField=fieldHash.keys();
                        String field=(String)enumField.nextElement();
                        primaryKeyHash.put(tableName,field);
                        String output="<Key name=\""+field+"\"/>";
                        Vector tempVect=new Vector();
                        if(primaryKeyXMLHash.containsKey("Schema_"+tableName)){
                                tempVect=(Vector)primaryKeyXMLHash.get("Schema_"+tableName);
                        }
                        tempVect.addElement(output);
                        primaryKeyXMLHash.put("Schema_"+tableName,tempVect);

                }
                for(int i=0;i<keyRefVect.size();i++){

                        Hashtable keyRefHash=(Hashtable)keyRefVect.elementAt(i);
                        Enumeration enumForiegn=keyRefHash.keys();
                        String foriegn=(String)enumForiegn.nextElement();
                        String tableName=(String)foriegnKeyToElementNameHash.get(foriegn);
                        String referKeyName=(String)foriegnKeyToPrimaryKeyHash.get(foriegn);
                        String
referTableName=(String)primaryKeyToElementNameHash.get(referKeyName);
                        String referTableKeyName=(String)primaryKeyHash.get(referTableName);
                        Hashtable inHash=(Hashtable)keyRefHash.get(foriegn);
                        Hashtable fieldHash=(Hashtable)inHash.get("FIELD");
                        Enumeration enumField=fieldHash.keys();
                        String field=(String)enumField.nextElement();
```

```
                        String output="<Key name=\""+field+"\"
referTable=\""+"Schema_"+referTableName+"\" referTableColumn=\""+referTableKeyName+"\" />";
                        Vector tempVect=new Vector();
                        if(foriegnKeyXMLHash.containsKey("Schema_"+tableName)){
                                tempVect=(Vector)foriegnKeyXMLHash.get("Schema_"+tableName);
                        }
                        tempVect.addElement(output);
                        foriegnKeyXMLHash.put("Schema_"+tableName,tempVect),
                }
        }

        //Generates the complete XML structure document for XML Schema
        public String generateXMLSchemaStruct(){
                boolean isTop=false;
                if(elementsHash.containsKey("Schema_topstructure")){
                        generateForiegnPrimaryKeys();
                        isTop=true;
                }
                Enumeration enumElement=elementsHash.keys();
                output="<DatabaseStructure name=\""+targetPrefix+"\">\n\t<ComponentList>";
                while(enumElement.hasMoreElements()){
                        String elementName=(String)enumElement.nextElement();
                        if(elementName.equalsIgnoreCase("Schema_topstructure")){
                                continue;
                        }
                        output+="\n\t\t<ComponentDescription name= \""+elementName+"\">";
                        if(inheritanceHash.containsKey(elementName)){
                                output+="\n\t\t\t<Inheritance>";
                                Hashtable inheritHash=(Hashtable)inheritanceHash.get(elementName);
                                String
inheritanceType=(String)inheritHash.get("INHERITANCETYPE");
                                String parent=(String)inheritHash.get("PARENTNAME");
                                output+="\n\t\t\t\t<Parent name=\"Schema_"+parent+"\"/>";
                                output+="\n\t\t\t</Inheritance>";
                        }
                        if(isTop){
                                output+="\n\t\t\t<PrimaryKey>";
                                if(primaryKeyXMLHash.containsKey(elementName)){
                                        Vector
primaryKeyVect=(Vector)primaryKeyXMLHash.get(elementName);
                                        for(int i=0;i<primaryKeyVect.size();i++){
                                                output+=(String)primaryKeyVect.elementAt(i);
                                        }
                                        output+="\n\t\t\t</PrimaryKey>";
                                }
                                if(foriegnKeyXMLHash.containsKey(elementName)){
                                        output+="\n\t\t\t<ForeignKey>";
                                        Vector
foriegnKeyVect=(Vector)foriegnKeyXMLHash.get(elementName);
                                        for(int i=0;i<foriegnKeyVect.size();i++){
                                                output+=(String)foriegnKeyVect.elementAt(i);
                                        }
                                        output+="\n\t\t\t</ForeignKey>";
                                }
                        }
```

```
                        if(elementsKeyAndKeyRefHash.containsKey(elementName) && !isTop){
                                output+="\n\t\t\t<PrimaryKey>";
                                output+=seperateKeys(elementName,"KEY");
                                output+="\n\t\t\t</PrimaryKey>";
                                output+="\n\t\t\t<UniqueKey>";
                                output+=seperateKeys(elementName,"UNIQUE");
                                output+="\n\t\t\t</UniqueKey>";
                                output+="\n\t\t\t<ForeignKey>";
                                output+=seperateKeys(elementName,"KEYREF");
                                output+="\n\t\t\t</ForeignKey>";
                        }
                        output+="\n\t\t\t<AttributeList>";
                        if(!isTop){
                                output+="\n\t\t\t\t<Attribute name=\""+elementName+"ID"+"\"
isAuto=\"YES\" datatype=\""+"xsd:string"+" \" isNull=\""+"NO"+" \" maxOccurs=\""+"=1"+" \"
minOccurs=\""+"=1"+"\"/>";
                        }
                        Vector columnsVect=(Vector)elementsHash.get(elementName);
                        System.out.println(elementName);
                        System.out.println(columnsVect);
                        for(int i=0;i<columnsVect.size();i++){
                                Hashtable columnHash=(Hashtable)columnsVect.elementAt(i);
                                Enumeration enumColumn=columnHash.keys();
                                String attributeName="";
                                String attributeDatatype="xsd:string";
                                String maxOccurs="1";
                                String minOccurs="1";
                                String isNull="YES",
                                while(enumColumn.hasMoreElements()){
                                        String columnKeys=(String)enumColumn.nextElement();
                                        if(columnKeys.equalsIgnoreCase("IDREF") ){
                                                Hashtable
columnDescription=(Hashtable)columnHash.get(columnKeys);
                                                if(columnDescription.containsKey("name"))

        attributeName=(String)columnDescription.get("name")+columnKeys;
                                                if(columnDescription.containsKey("ref"))

        attributeName=(String)columnDescription.get("ref")+columnKeys;
                                                if(columnDescription.containsKey("maxOccurs"))

        maxOccurs=(String)columnDescription.get("maxOccurs");
                                                if(columnDescription.containsKey("minOccurs"))

        minOccurs=(String)columnDescription.get("minOccurs");
                                                if(columnDescription.containsKey("use")){

        isNull=(String)columnDescription.get("use");
                                                        if(isNull.equalsIgnoreCase("required")){
                                                                isNull="NO";
                                                        }
                                                }
                                        }
                                        else {
```

```
                                                String
columnDescription=(String)columnHash.get(columnKeys);

                                                if(columnKeys.equalsIgnoreCase("name"))
                                                        attributeName=columnDescription;
                                                if(columnKeys.equalsIgnoreCase("ref"))
                                                        attributeName=columnDescription;
                                                if(columnKeys.equalsIgnoreCase("value"))
                                                        attributeName=columnDescription;
                                                if(columnKeys.equalsIgnoreCase("maxOccurs"))
                                                        maxOccurs=columnDescription;
                                                if(columnKeys.equalsIgnoreCase("minOccurs"))
                                                        minOccurs=columnDescription;
                                                if(columnKeys.equalsIgnoreCase("use")){
                                                        isNull=columnDescription;
                                                        if(isNull.equalsIgnoreCase("required")){
                                                                isNull="NO";
                                                        }
                                                }
                                                if(columnKeys.equalsIgnoreCase("type"))
                                                        attributeDatatype=columnDescription;
                                        }
                                }
                                output+="\n\t\t\t\t<Attribute name=\""+attributeName+"\"
datatype=\""+attributeDatatype+" \" isNull=\""+isNull+" \" maxOccurs=\""+maxOccurs+" \"
minOccurs=\""+minOccurs+"\"/>";
                        }
                        output+="\n\t\t\t</AttributeList>";
                        output+="\n\t\t</ComponentDescription>";
                }
                output+="\n\t</ComponentList>";
                output+="\n</DatabaseStructure>";
                System.out.println(output);
                Hashtable fileHash=fileInfo.getFileInformation("xdbms_schema","struct");
                String fileName=(String)fileHash.get("filename");
                String filePath=(String)fileHash.get("filelocation");
                writeToFile(output,fileName,filePath);
                return output;
        }

        //Writes the XML structure and data document to the location specified.
        public void writeToFile(String output,String fileName,String filePath){
                try{
                        File f=new File(filePath+fileName);
                        FileWriter fw=new FileWriter(f);
                        fw.write(output);
                        fw.close();
                }catch(IOException ex){}
        }

        //For a supplied tag extract its attributes
        public Hashtable getAttributes(Node node){
                String strValue = "";
                String strName = "";
                Hashtable attributesHash=new Hashtable();
```

```
                NamedNodeMap attributes=node.getAttributes();
                for(int i=0;i<attributes.getLength();i++){
                        Node n=attributes.item(i);
                        strName = n.getNodeName().trim();
                        strValue = n.getNodeValue().trim();
                        attributesHash.put(strName,strValue);
                }
                return attributesHash;
        }

        //Parses a string with the delimiter
        public Vector parse(String str,String par){
                Vector parseVect=new Vector();
                StringTokenizer st=new StringTokenizer(str,par);
                while(st.hasMoreElements()){
                        String token=(String)st.nextToken();
                        parseVect.addElement(token);
                }
                return parseVect;
        }
}
```

APPENDIX A2: Source Code for Reverse Transformation

```
package com.migrationtool.reversetransformation;

import java.io.*;
import java.util.*;
import java.text.NumberFormat;
import com.migrationtool.datatypes.*;
import org.w3c.dom.Document;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import com.migrationtool.configfiles.*;
import com.migrationtool.domloader.*;

//Generates database structure information specific to database

public class ReverseStructureGenerator{
        private Document doc;
        DatatypeMapping datatypeMapping=new DatatypeMapping();
        RestrictionsRDBMSDatatypeMapping restrictionsRDBMSDatatypeMapping=new
RestrictionsRDBMSDatatypeMapping();
        String prefix="";
        String conversionType;
        //Variables for holding file location informations
        String filePath;
        String fileName;
        String supportFileName;
        String supportFileLocation;

        //Holds Inheritance tag information
        Hashtable inheritHash=new Hashtable();

        //Holds attributes of ComponentDescription tag
        Hashtable componentAttributesHash=new Hashtable();

        //Holds Foreign key tag information
        Hashtable foreignKeyHash=new Hashtable();
        Hashtable rdbmsRestrictionHash=new Hashtable();
        Hashtable oodbmsHash=new Hashtable();

        //Holds information related to primary keys and relationship tags specific to each database
        Hashtable relationshipHash=new Hashtable();
        Hashtable relationPrimaryKeyHash=new Hashtable();
        Hashtable keyElementHash=new Hashtable();
        Hashtable keyForKeyRefElementHash=new Hashtable();
        Hashtable keyRefElementHash=new Hashtable();
        Hashtable primaryKeysHash=new Hashtable();
        Hashtable foreignKeysHash=new Hashtable();
        Hashtable uniqueKeysHash=new Hashtable();

        String output="";
        //Used for auto generation of keys
```

```
        int count=0;

        //Sets the required file location information
        public void setInformation(String databaseType,String fileName,String filePath,String
prefix,Document doc){
                conversionType=databaseType;
                this.fileName=fileName;
                this.filePath=filePath;
                this.prefix=prefix,
                this.doc=doc;
        }
        //Sets supporting files location information
        public void setSupportFileInformation(String fileName,String filePath){
                supportFileName=fileName,
                supportFileLocation=filePath;
        }

        //Extracts inheritance tag specific information such as parent tag attributes and child tag attributes
        public void startInheritance(){
                NodeList list = doc.getElementsByTagName("ComponentDescription");
                for (int i = 0; i < list.getLength(); i++){
                        Node element = list.item(i);
                        Hashtable       attributesHash=getAttributes(element);
                        String name=(String)attributesHash.get("name");
                        NodeList childNodes=element.getChildNodes();
                        for(int k=0;k<childNodes.getLength();k++){
                                Node childNode=childNodes.item(k);
                                String childName=childNode.getNodeName();
                                if(childName.equalsIgnoreCase("AttributeList")){
                                        NodeList childAttributeNodes=childNode.getChildNodes(),
                                        Vector parentVect=new Vector();
                                        for(int m=0;m<childAttributeNodes.getLength();m++){
                                                Node
childAttributeNode=childAttributeNodes.item(m);
                                                String
childAttributeName=childAttributeNode.getNodeName();

        if(childAttributeName.equalsIgnoreCase("Attribute")){
                                                        Hashtable
attHash=getAttributes(childAttributeNode);
                                                        parentVect.addElement(attHash);
                                                }
                                        }

        componentAttributesHash.put(attributesHash.get("name"),parentVect);
                                }
                                if(childName.equalsIgnoreCase("Inheritance")){
                                        NodeList childInheritNodes=childNode.getChildNodes();
                                        Vector parentVect=new Vector();
                                        for(int m=0;m<childInheritNodes.getLength();m++){
                                                Node childInheritNode=childInheritNodes.item(m);
                                                String
childInheritName=childInheritNode.getNodeName();
                                                if(childInheritName.equalsIgnoreCase("Parent")){
```

```
                                                    Hashtable
attHash=getAttributes(childInheritNode);

        parentVect.addElement(attHash.get("name")),
                                                }
                                        }

        inheritHash.put(((String)attributesHash.get("name")).trim(),parentVect);
                                }

                }
        }
}

        //Loads the Object-oriented database hashtable with related information for variables and
datatypes
        public void oodbms(String name,String variable,String datatype){

                if(oodbmsHash.containsKey(name.trim())){
                        Hashtable tempHash=(Hashtable)oodbmsHash.get(name.trim());
                        tempHash.put(variable.trim(),datatype);
                        oodbmsHash.put(name,tempHash),
                }
                else{
                        Hashtable tempHash=new Hashtable();
                        tempHash.put(variable.trim(),datatype);
                        oodbmsHash.put(name.trim(),tempHash);

                }
        }

        //Generates data-type information related to each class
        public void generateOODBMSDatatypeInformation(){
                Enumeration enum=oodbmsHash.keys(),
                String output="";
                while(enum.hasMoreElements()){
                        String className=(String)enum.nextElement();
                        output+=className+"#";
                        Hashtable tempHash=(Hashtable)oodbmsHash.get(className),
                        Enumeration enumTemp=tempHash.keys();
                        while(enumTemp.hasMoreElements()){
                                String keyTemp=(String)enumTemp.nextElement();
                                String datatype=(String)tempHash.get(keyTemp);
                                output+=keyTemp+"-"+datatype+"|";
                        }
                        output+="\n";
                }
                writeToFile(supportFileName,supportFileLocation,output),
        }

        //Resolves inheritance for databases
        public void addInheritance(Node child, String name){
                if(inheritHash.containsKey(name)){
                        Vector parentVect=(Vector) inheritHash.get(name);
                        for(int m=0;m<parentVect.size();m++){
```

```
                                String parent=(String)parentVect.elementAt(m);
                                Vector
parentAttributesVect=(Vector)componentAttributesHash.get(parent);
                                for(int at=0;at<parentAttributesVect.size();at++){
                                        Hashtable
attributesHash=(Hashtable)parentAttributesVect.elementAt(at);
                                        String val=(String)attributesHash.get("name");
                                        String databaseType=conversionType.toLowerCase();
                                        String datatype=(String)attributesHash.get("datatype");
                                        Hashtable
datatypeHash=datatypeMapping.getDatatype(fromDBMS,datatype,databaseType);
                                        Vector
datatypeVect=(Vector)datatypeHash.get(databaseType);
                                        String datatypeRequested=(String)datatypeVect.elementAt(0),
                                        output+=val +" "+datatypeRequested+",";
                                }
                        }
                }
        }

        //Generates XML document containing Foreign key information
        public void generateForeignKeyXML(){
                String foreignKeyXML="",
                Enumeration enumForeign=foreignKeyHash.keys();
                foreignKeyXML+="<ForeignKey>\n";
                while(enumForeign.hasMoreElements()){
                        String table_name=(String)enumForeign.nextElement();
                        Vector tempVect=(Vector)foreignKeyHash.get(table_name);
                        for(int i=0;i<tempVect.size();i++){
                                Hashtable attributesHash=(Hashtable)tempVect.elementAt(i);
                                String keyName=(String)attributesHash.get("name");
                                String referTableName=(String)attributesHash.get("referTable");
                                String sqlStatement="ALTER TABLE "+table_name+" ADD
FOREIGN KEY("+keyName+")"+" REFERENCES "+referTableName+ " ";

        foreignKeyXML+="<SQLSTATEMENT>"+sqlStatement+"</SQLSTATEMENT>",
                        }
                }
                foreignKeyXML+="</ForeignKey>";
                System.out.println(foreignKeyXML);
                writeToFile(supportFileName,supportFileLocation,foreignKeyXML);
        }

        //Generates primary keys for XML Schema elements
        public void generateKeyElements(){
                Enumeration enumHash=primaryKeysHash.keys();
                while(enumHash.hasMoreElements()){
                        String compName=(String)enumHash.nextElement();
                        String key="<xsd:key
name=\""+compName+primaryKeysHash.get(compName)+"PK\" >\n";
                        key+="<xsd:selector xpath=\""+compName+"\"/>\n";
                        key+="<xsd:field xpath=\""+primaryKeysHash.get(compName)+"\"/>\n";
                        key+="</xsd:key>\n\n";
```

```
        keyForKeyRefElementHash.put(compName,compName+primaryKeysHash.get(compName)+"PK
"),
                        keyElementHash.put(compName,key);
                }
        }

        //Generates foreign keys for XML Schema elements
        public void generateKeyRefElements(){
                Enumeration enumHash=foreignKeyHash.keys();
                while(enumHash.hasMoreElements()){
                        String compName=(String)enumHash.nextElement();
                        Vector tempVect=(Vector)foreignKeyHash.get(compName);
                        Vector keyRefElementVect=new Vector();
                        for(int i=0;i<tempVect.size();i++){
                                Hashtable attributeHash=(Hashtable)tempVect.elementAt(i);
                                String referKeyTable=(String)attributeHash.get("referTable");
                                String keyref="<xsd:keyref
name=\""+compName+attributeHash.get("name")+"FK\"
refer=\""+keyForKeyRefElementHash.get(referKeyTable)+"\">\n";
                                keyref+="<xsd:selector xpath=\""+compName+"\"/>\n",
                                keyref+="<xsd:field xpath=\""+attributeHash.get("name")+"\"/>\n";
                                keyref+="</xsd:keyref>\n\n";
                                keyRefElementVect.addElement(keyref);
                        }
                        keyRefElementHash.put(compName,keyRefElementVect);
                }
        }

        //Gets primary key and foriegn key for XML Schema elements
        public String getKeyAndKeyRef(){
                String keyandKeyRef="";
                Enumeration enumKey=keyElementHash.keys();
                while(enumKey.hasMoreElements()){
                        keyandKeyRef+=(String)keyElementHash.get(enumKey.nextElement());
                }
                Enumeration enumKeyRef=keyRefElementHash.keys();
                while(enumKeyRef.hasMoreElements()){
                        Vector
tempVect=(Vector)keyRefElementHash.get(enumKeyRef.nextElement());
                        for(int i=0;i<tempVect.size();i++)
                                keyandKeyRef+=(String)tempVect.elementAt(i),
                }
                return keyandKeyRef;
        }

        //Starts the generation of Reverse XML Structure documents specific to each database
        public String startProcessing(String fromDBMS){
                this.fromDBMS=fromDBMS;
                boolean isInheritance=false;
                startInheritance();
                String topstructure="";
                NodeList list = doc.getElementsByTagName("ComponentDescription");
                if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")) {
                        topstructure+="<?xml version=\"1.0\"?>\n";
```

```
                topstructure+="<xsd:schema
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\">\n";
                topstructure+="<xsd:element name=\"topstructure\">\n";
                topstructure+="\t<xsd:complexType>\n\t\t<xsd:sequence>\n";
            }
            if(conversionType.equalsIgnoreCase("RDBMS")){
                output="<SQLCREATE>";

        rdbmsRestrictionHash=restrictionsRDBMSDatatypeMapping.getRDBMSRestrictionHash();
            }
            if(conversionType.equalsIgnoreCase("OODBMS"))
                output="<OODBMSCLASS>";
            for (int i = 0; i < list.getLength(); i++){
                Node element = list.item(i);
                Hashtable       attributesHash=getAttributes(element);
                String name=(String)attributesHash.get("name");
                if(conversionType.equalsIgnoreCase("RDBMS")){
                    name=(String)attributesHash.get("name");
                    Vector tableVect=parse(name," ");
                    if(tableVect.size() >1){
                        name="\""+name+"\"";
                    }
                    if(!name.equalsIgnoreCase("Schema_topstructure"))
                        output+="\t<SQLStatement>"+" CREATE TABLE "+name+
" ( ",
                }
                if(conversionType.equalsIgnoreCase("OODBMS")){
                    if(!name.equalsIgnoreCase("Schema_topstructure")){
                        output+=" <classstatement><classdeclaration>public
class</classdeclaration> <classname name=\""+attributesHash.get("name")+ "\"/><sep>{</sep> ";
                    }
                }
                if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")){
                    String compName=(String)attributesHash.get("name");
                    Vector parseSpaceVect=parse(compName," "),
                    if(parseSpaceVect.size() >1){

        compName=(String)parseSpaceVect.elementAt(0)+"_"+(String)parseSpaceVect.elementAt(1);
                    }
                    topstructure+="\t\t\t<xsd:element ref=\""+compName+"\"
maxOccurs=\"unbounded\" />\n";
                    output+="\t<xsd:element name=\""+compName+"\">";
                    if(inheritHash.containsKey(attributesHash.get("name"))){
                        Vector inheritVect=(Vector)inheritHash.get(name);
                        String parent=(String)inheritVect.elementAt(0);
                        parseSpaceVect=parse(parent," ");
                        if(parseSpaceVect.size() >1){

        parent=(String)parseSpaceVect.elementAt(0)+"_"+(String)parseSpaceVect.elementAt(1);
                        }
                        output+="\n\t\t<xsd:extension base=\""+parent+"\">\n";
                        isInheritance=true;
                    }
                    output+="\n\t\t\t<xsd:complexType>\n\t\t\t\t<xsd:all>\n";
                    name=compName;
```

```
            }
            if(conversionType.equalsIgnoreCase("XDBMS_DTD")){
                    output+="\n\n<!ELEMENT "+attributesHash.get("name")+ " > \n\n";
                    output+="<!ATTLIST "+attributesHash.get("name")+ "  \n\n";
            }
            NodeList childNodes=element.getChildNodes();
            //          if(conversionType.equalsIgnoreCase("RDBMS"))
            //                      addInheritance(element,name);

            for(int k=0;k<childNodes.getLength();k++){
                    Node childNode=childNodes.item(k);
                    String childName=childNode.getNodeName();
                    if(!childName.equalsIgnoreCase("#text")){
                            if(!name.equalsIgnoreCase("Schema_topstructure"))
                                    processChild(childNode,name);
                    }
            }
            if(conversionType.equalsIgnoreCase("RDBMS")){
                    if(!name.equalsIgnoreCase("Schema_topstructure"))
                    {                       int index=output.lastIndexOf(","),
                                    if(index > 0)
                                            output=output.substring(0,index);
                                    output+=");"+"\n\t</SQLStatement>";
                                    output+="\n",
                    }
            }
            if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")){
                    output+="\n\t\t\t\t</xsd:all>\n\t\t\t</xsd:complexType>\n",
                    if(isInheritance){
                            output+="\n\t\t\t</xsd:extension>\n";
                            isInheritance=false;
                    }
                    output+="\t\t</xsd:element>\n";
            }
            if(conversionType.equalsIgnoreCase("OODBMS")){
                    if(!name.equalsIgnoreCase("Schema_topstructure")){
                            output+="\n <sep>}</sep> ";
                            output+="</classstatement>\n";
                    }
            }
            if(conversionType.equalsIgnoreCase("XDBMS_DTD")){
                    fileName="xmldtd.xml";
                    int index=output.lastIndexOf(",");
                    output=output.substring(0,index);
                    output+=" > \n";
            }
    }
    if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")){
            topstructure+="\t\t</xsd:sequence>\n</xsd:complexType>\n";
            generateKeyElements();
            generateKeyRefElements();
            topstructure+=getKeyAndKeyRef();
            topstructure+="</xsd:element>\n";
            topstructure+=output;
            topstructure+="</xsd:schema>\n";
```

```
                writeToFile(fileName,filePath,topstructure);
            }
            if(conversionType.equalsIgnoreCase("RDBMS")){
                output+="</SQLCREATE>";
                writeToFile(fileName,filePath,output);
                generateForeignKeyXML();
            }
            if(conversionType.equalsIgnoreCase("OODBMS")){
                output+="</OODBMSCLASS>";
                writeToFile(fileName,filePath,output);
                generateOODBMSDatatypeInformation();
                classFiles=generateOzoneOODBMSStruct();
            }
            return output,
        }
        //Gets class informaion
        String classFiles;
        public String getClassFiles(){
            return classFiles;
        }

        //Processes the child nodes of XML structure document
        public void processChild(Node child,String name){
            String tab="\t";
            String topElement=child.getNodeName();
            NodeList childNodes=child.getChildNodes();
            Vector foreignKeyVect=new Vector();
            for(int k=0;k<childNodes.getLength();k++){
                Node element=childNodes.item(k);
                if(!element.getNodeName().equals("#text")){
                    String elemName=(String)element.getNodeName();
                    Hashtable attributesHash=new Hashtable();
                    if(!element.hasChildNodes()){
                        if(elemName.equalsIgnoreCase("Key")){
                            attributesHash=getAttributes(element);
                            if(topElement.equals("PrimaryKey")){

        primaryKeysHash.put(name,attributesHash.get("name"));
                            }
                            if(topElement.equals("ForeignKey")){

        foreignKeysHash.put(attributesHash.get("name"),attributesHash.get("referTable"));
                                if(foreignKeyHash.containsKey(name)){
                                    Vector
tempVect=(Vector)foreignKeyHash.get(name);

        tempVect.addElement(attributesHash);

        foreignKeyHash.put(name,tempVect);
                                }else{
                                    Vector tempVect=new Vector();

        tempVect.addElement(attributesHash);

        foreignKeyHash.put(name,tempVect);
```

```
                                                }
                                        }
                                        if(topElement.equals("UniqueKey")){

        uniqueKeysHash.put(attributesHash.get("name"),"");
                                        }
                                }
                        }
                }
                for(int k=0;k<childNodes.getLength();k++){
                        Node element=childNodes.item(k);
                        String elemName=(String)element.getNodeName();
                        Hashtable attributesHash=new Hashtable();
                        if(elemName.equalsIgnoreCase("Attribute")){
                                attributesHash=getAttributes(element);
                                String val=(String)attributesHash.get("name");
                                String databaseType=conversionType.toLowerCase();
                                String datatype=(String)attributesHash.get("datatype");
                                String compName=datatype;
                                Vector parseSpaceVect=parse(compName," ");
                                if(parseSpaceVect.size() >1){
                                        compName=(String)parseSpaceVect.elementAt(0);
                                }
                                datatype=compName;
                                String columnSize=(String)attributesHash.get("columnSize");
                                if(datatype.indexOf(":") != (-1)){
                                        Vector tempVect=parse(datatype,":");
                                        datatype=(String)tempVect.elementAt(1);
                                }
                                if(fromDBMS.indexOf("_") != (-1)){
                                        Vector tempVect=parse(fromDBMS,"_");
                                        fromDBMS=(String)tempVect.elementAt(0);

                                }
                                Hashtable
datatypeHash=datatypeMapping.getDatatype(fromDBMS,datatype,databaseType);
                                if(conversionType.equalsIgnoreCase("RDBMS")){
                                        Vector
datatypeVect=(Vector)datatypeHash.get(databaseType);
                                        String datatypeRequested=(String)datatypeVect.elementAt(0);
                                        if(rdbmsRestrictionHash.containsKey(datatype) ||
!fromDBMS.equalsIgnoreCase("RDBMS")){
                                                if(rdbmsRestrictionHash.containsKey(datatype))
                                                        output+=val +" "+datatypeRequested+" ";
                                                else
                                                        output+=val +" "+datatypeRequested+"(50)
";
                                        }
                                        else{
                                                output+=val +"
"+datatypeRequested+"("+columnSize+")";
                                        }
                                }
                                if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")){
```

```
                                        Vector
datatypeVect=(Vector)datatypeHash.get(databaseType);
                                        String datatypeRequested=(String)datatypeVect.elementAt(0);
                                        output+="\t\t\t\t<xsd:element name=\""+val +"\"
type=\"xsd:"+datatypeRequested+"\" />\n";
                                }
                                if(conversionType.equalsIgnoreCase("XDBMS_DTD")){
                                        Vector
datatypeVect=(Vector)datatypeHash.get(databaseType);
                                        String datatypeRequested=(String)datatypeVect.elementAt(0);
                                        output+=""+val +attributesHash.get("datatype")+"
#REQUIRED";
                                }
                                if(conversionType.equalsIgnoreCase("OODBMS")){
                                        System.out.println(databaseType);
                                        Vector
datatypeVect=(Vector)datatypeHash.get(databaseType);
                                        String datatypeRequested=(String)datatypeVect.elementAt(0);
                                        oodbms(name,val,datatypeRequested);
                                        output+="\n\n<variableinformation>"+"<datatype
name=\""+datatypeRequested+" \"/>"+"<variable name=\""+val+" \" />";
                                        String firstLetter=val.substring(0,1);
                                        String secondPart=val.substring(1);
                                        output+="\n<setmethod
name=\"set"+firstLetter.toUpperCase()+secondPart+" \"><declaration> public
void</declaration><methoddata data=\"("+ datatypeRequested+" "+val+")\n"+"{\n
this."+val+"="+val+";\n} \"/></setmethod>",
                                        output+="\n\n<getmethod name=\" public
"+datatypeRequested +" get"+firstLetter.toUpperCase()+secondPart+"()\" data=\"{\n return "
+val+";"+"\n} \" /> </variableinformation>";
                                }
                                String key="@$@#@#@";
                                if(primaryKeysHash.containsKey(name))
                                        key=(String)primaryKeysHash.get(name);
                                if(key.equalsIgnoreCase(val.trim())){


                                        if(conversionType.equalsIgnoreCase("RDBMS")){

                                                output+=" primary key ";
                                        }
                                        if(conversionType.equalsIgnoreCase("OODBMS")){
                                        }

        if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")){
                                                Vector tempVect= new Vector();
                                                if(relationPrimaryKeyHash.containsKey(name)){

        tempVect=(Vector)relationPrimaryKeyHash.get(name);
                                                }
                                                tempVect.addElement(val);
                                                relationPrimaryKeyHash.put(name,tempVect);
                                        }
                                }
                                if(foreignKeysHash.containsKey(val)){
```

```
                                        // if(conversionType.equalsIgnoreCase("RDBMS"))
                                        //                                      output+=" foreign key
references "+foreignKeysHash.get(val)+"  ";
                                        if(conversionType.equalsIgnoreCase("OODBMS")){
                                        }

        if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")){
                                        }
                                }
                                if(uniqueKeysHash.containsKey(val)){
                                        output+=" unique key   ";
                                }
                                if(conversionType.equalsIgnoreCase("RDBMS"))
                                        output+=" , ";
                                if(conversionType.equalsIgnoreCase("OODBMS")){
                                }
                                if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")){
                                }
                                if(conversionType.equalsIgnoreCase("XDBMS_DTD")){
                                        output+=" , \n";
                                }
                        }
                }
        }
        //Generates the Java classes specific to Ozone database
        public String generateOzoneOODBMSStruct(){
                String classFiles="";
                FileInformation fileInfo=new FileInformation();
                fileInfo.loadFileXML("config.xml");
                Hashtable fileHash=fileInfo.getFileInformation("oodbms","reversestruct");
                String fileStructName=(String)fileHash.get("filename");
                String fileStructPath=(String)fileHash.get("filelocation");
                Document document = null;
                document = DOMUtil.parse(fileStructPath+fileStructName);
                NodeList list = document.getElementsByTagName("classstatement");

                for(int i=0;i<list.getLength();i++){
                        Node element=list.item(i);
                        NodeList childNodes=element.getChildNodes();
                        Vector methodVect=new Vector(),
                        String className="";
                        String classfile="";
                        classfile+="import org.ozoneDB.OzoneObject;\n";
                        classfile+="import java.util.*,\n";
                        String interfacefile="";
                        interfacefile+="import org.ozoneDB.OzoneRemote;\n";

                        Hashtable variableHash=new Hashtable();
                        for(int k=0;k<childNodes.getLength();k++){
                                Node childNode=childNodes.item(k);
                                String  childName=childNode.getNodeName();
                                if(childName.equalsIgnoreCase("classname")){
                                        Hashtable tempHash=getAttributes(childNode);
                                        className=(String)tempHash.get("name");
```

```
                                        interfacefile+="\npublic interface "+className+" extends
OzoneRemote {\n";
                                        classfile+="\npublic class "+className+"Impl extends
OzoneObject implements "+className+" {\n";
                                }
                                if(childName.equalsIgnoreCase("variableinformation")){
                                        NodeList variableChildNodes=childNode.getChildNodes();
                                        String datatype="";
                                        String methodName="";
                                        String variable="";
                                        String varName="";
                                        for(int var=0,var<variableChildNodes.getLength();var++)
                                        {
                                                Node variableNode=variableChildNodes.item(var);
                                                String varinfo=variableNode.getNodeName();
                                                if(varinfo.equalsIgnoreCase("datatype")){
                                                        Hashtable
datatypeHash=getAttributes(variableNode);
                                                        datatype=(String)datatypeHash.get("name");
                                                }
                                                if(varinfo.equalsIgnoreCase("variable")){
                                                        Hashtable
variableTempHash=getAttributes(variableNode);

        variable=(String)variableTempHash.get("name");
                                                        classfile+="\n"+ datatype+"
"+variable+";\n";

                                                }
                                                if(varinfo.equals("getmethod")){
                                                        Hashtable
setHash=getAttributes(variableNode);

        classfile+="\n"+(String)setHash.get("name")+(String)setHash.get("data");

        interfacefile+=(String)setHash.get("name")+";\n";

                                                }
                                                if(varinfo.equalsIgnoreCase("setmethod")){
                                                        Hashtable
methodTempHash=getAttributes(variableNode),

        methodName=(String)methodTempHash.get("name");
                                                        classfile+="public void "+methodName;
                                                        interfacefile+="public void
"+methodName+"("+datatype+" "+variable+");\n";

                                                        NodeList
setChildNodes=variableNode.getChildNodes();
                                                        for(int
set=0;set<setChildNodes.getLength();set++){
                                                                Node
setChild=setChildNodes.item(set);
                                                                String
setChildName=setChild.getNodeName();
```

```
        if(setChildName.equals("methoddata")){
                                                                    Hashtable
setHash=getAttributes(setChild),

        classfile+=(String)setHash.get("data")+"\n";

                                                                }

                                                            }

                                                        }
                                                    }
                                                }
                                    }
                                    interfacefile+="\n}";
                                    classfile+="\n}";
                                    classFiles+="\t\t\t\t\t\t\t\t\t";
                                    classFiles+=interfacefile+"\n";
                                    classFiles+=classfile+"\n";
                                    fileHash=fileInfo.getFileInformation("oodbms","reversestructgenerated");
                                    String filePath=(String)fileHash.get("filelocation");
                                    writeToFile(className+"Impl.java",filePath,classfile);
                                    writeToFile(className+".java",filePath,interfacefile);
                        }
                        return classFiles;
        }

        //Writes the generated database files to location supplied
        public void writeToFile(String fileName,String filePath,String output){

                        try{
                                File f=new File(filePath+fileName);
                                FileWriter fw=new FileWriter(f);
                                fw.write(output);
                                fw.close();
                        }catch(IOException ex){}
                        System.out.println(fileName+" "+filePath);
                        System.out.println("--------DONE----------");

        }
        //Extracts attributes of tag
        public Hashtable getAttributes(Node node){
                        String strValue = "";
                        String strName = "";
                        Hashtable attributesHash=new Hashtable();
                        NamedNodeMap attributes=node.getAttributes();
                        for(int i=0;i<attributes.getLength();i++){
                                Node n=attributes.item(i);
                                strName = n.getNodeName().trim();
                                strValue = n.getNodeValue().trim();
                                attributesHash.put(strName,strValue);
                        }
                        return attributesHash;
        }
```

```
        //Parse string with the supplied delimiter
        public Vector parse(String str,String par){
                Vector parseVect=new Vector();
                StringTokenizer st=new StringTokenizer(str,par);
                while(st.hasMoreElements()){
                        String token=(String)st.nextToken();
                        parseVect.addElement(token);
                }
                return parseVect;
        }
}
```

Revese Data Generator

```
package com.migrationtool.reversetransformation;

import java.io.*;
import java.util.*;
import java.text.NumberFormat,
import org.w3c.dom.Document;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import com.migrationtool.configfiles.*;
import com.migrationtool.domloader.*;
import com.migrationtool.reversetransformation.rdbms.*;

//Generates XML documents for IPEDO, Driver program with object creation method
 //calls for object-oriented database and SQL INSERT statements for Relational database

 public class ReverseDataGenerator{
        private Document doc;
        String prefix="";
        String conversionType;
        String output="";
        String fileName="";
        String filePath="";
        String supportFileName;
        String supportFileLocation;
        String oodbmsObjectCreationFunctionCalls="";
        String classBody="import java.util.*;\n";
        String fromDBMS;
        Hashtable objectNamesHash=new Hashtable();
        Hashtable classMethodCallVariableOrderingHash=new Hashtable();
        String classObjectCreation="";
        Hashtable methodDeclarationVariableOrderingHash=new Hashtable();
        Hashtable classHash=new Hashtable();
```

```
        Hashtable variableDatatypeHash=new Hashtable();

        public void setSupportFileInformation(String fileName,String filePath){
                supportFileName=fileName;
                supportFileLocation=filePath;
        }

        public void setInformation(String databaseType,String fileName,String filePath,String
prefix,Document doc){

                conversionType=databaseType;
                this.fileName=fileName;
                this.filePath=filePath;
                this.prefix=prefix;
                this.doc=doc;
        }

        //Starts the processing of information stored in XML data document
        public String startProcessing(String fromDBMS){
                this.fromDBMS=fromDBMS;
                if(conversionType.equalsIgnoreCase("OODBMS")){
                        classBody+="import org.ozoneDB.*;\n";
                        classBody+="public class OODBMSDataDriver { \n";
                        classBody+="public static ExternalDatabase          db;\n";
                        classBody+=" public static void main( String[] args ) throws Exception {\n";
                        classBody+=" db = ExternalDatabase.openDatabase(
\"ozonedb:remote://localhost:3333\" );\n";
                        classBody+="  System.out.println( \"Connected ...\" );\n",
                        classBody+=" db.reloadClasses();\n";
                        classBody+="OODBMSDataDriver od=new OODBMSDataDriver();\n";
                        loadOODBMSStruct();
                }
                if(conversionType.equalsIgnoreCase("RDBMS")){
                        output="<rdbmsinsertstatements>\n";
                }
                NodeList list = doc.getElementsByTagName("AttributeDataList");
                if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")){
                        output="<topstructure>\n";
                }
                for (int i = 0; i < list.getLength(); i++){
                        Node element = list.item(i);
                        NodeList childNodes=element.getChildNodes();
                        for(int k=0;k<childNodes.getLength();k++){
                                Node childNode=childNodes.item(k);
                                String childName=childNode.getNodeName(),
                                Node parentNode=element.getParentNode();
                                Hashtable       attributesHash=getAttributes(parentNode);
                                String parentName=(String)attributesHash.get("name");
                                String objName="obj"+parentName+k;
                                if(!childName.equalsIgnoreCase("#text")){
                                        if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")
|| conversionType.equalsIgnoreCase("XDBMS_DTD")){
                                                output+="\t<"+parentName+">\n";
                                                System.out.println("Writing data.."+parentName);
```

```
                                        }
                                        if(conversionType.equalsIgnoreCase("RDBMS")){

if(fromDBMS.equalsIgnoreCase("XDBMS_SCHEMA")){

parentName="Schema_"+parentName.trim();
                                                }

if(!parentName.equalsIgnoreCase("Schema_topstructure"))
                                                        output+="<sqlstatement> insert into
"+parentName.trim()+ " ";
                                        }
                                        if(conversionType.equalsIgnoreCase("OODBMS")){

if(fromDBMS.equalsIgnoreCase("XDBMS_SCHEMA")){

parentName="Schema_"+parentName.trim();
                                                }

if(!parentName.equalsIgnoreCase("Schema_topstructure")){
                                                        System.out.println("Writing
data.."+parentName);
                                                        String   objectClassNameXML="\n\t<class
name=\""+parentName+"\">";
                                                        String objectName="<objectName
value=\""+objName+"\"/>";
                                                        if(!
objectNamesHash.containsKey(objectClassNameXML.trim())){

objectNamesHash.put(objectClassNameXML.trim(),objectName);
                                                        }
                                                        else {
                                                                String
objectNamesXML=(String)objectNamesHash.get(objectClassNameXML.trim());

objectNamesXML+=objectName.trim();

objectNamesHash.put(objectClassNameXML.trim(),objectNamesXML);
                                                        }

oodbmsObjectCreationFunctionCalls+="create"+parentName+"(\""+objName+"\"";
                                                }
                                        }
                                        if(!parentName.equalsIgnoreCase("Schema_topstructure"))
                                                processChild(childNode,objName,parentName);

                                        if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA")
|| conversionType.equalsIgnoreCase("XDBMS_DTD"))
                                                output+="\t</"+parentName+">\n";

                                        if(conversionType.equalsIgnoreCase("OODBMS")){

if(!parentName.equalsIgnoreCase("Schema_topstructure"))

oodbmsObjectCreationFunctionCalls+=");\n";
```

```
                                        }
                                        if(conversionType.equalsIgnoreCase("RDBMS")){

if(!parentName.equalsIgnoreCase("Schema_topstructure")){
                                                int index=output.lastIndexOf(",");
                                                output=output.substring(0,index);
                                                output+=");</sqlstatement>";
                                                output+="\n";
                                        }
                                }
                        }

                }
                if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA") ||
conversionType.equalsIgnoreCase("XDBMS_DTD")){
                        output+="</topstructure>\n";
                        writeToFile(output,fileName,filePath);
                        System.out.println("-------DONE XDBMS DATA CONVERSION-------");

                }

                if(conversionType.equalsIgnoreCase("RDBMS")){
                        output+="</rdbmsinsertstatements>\n",
                        writeToFile(output,fileName,filePath);
                }

                if(conversionType.equalsIgnoreCase("OODBMS")){
                        output+=classBody;
                        output+="od.start();\n";
                        output+="db.close();\n}\n";
                        output+="public void start(){\n";
                        output+=oodbmsObjectCreationFunctionCalls;
                        output+="}\n";
                        output+=classObjectCreation;
                        output+="\n}\n";
                        writeToFile(output,fileName,filePath);
                        String objectNamesXML="<objectnames>\n";
                        Enumeration objectEnum=objectNamesHash.keys();
                        while(objectEnum.hasMoreElements()){
                                String key=(String)objectEnum.nextElement();
                                String val=(String)objectNamesHash.get(key);
                                objectNamesXML+=key+"\n"+val+"\n</class>";
                        }
                        objectNamesXML+="\n</objectnames>";
                        writeToFile(objectNamesXML,"objectnames.xml",filePath);
                        System.out.println("-------DONE OODBMS DATA CONVERSION-------");

                }
                return output;
        }

        //Processes the child nodes of XML data document
        public void processChild(Node child,String objName,String parentName){
```

```
                Vector rdbmsOrderColVect=new Vector();
                Vector rdbmsOrderDataVect=new Vector();
                NodeList childNodes=child.getChildNodes();
                Hashtable  methodCallVariableOrderingHash=new Hashtable();
                for(int i=0;i<childNodes.getLength();i++){
                        Node dataNode=childNodes.item(i);
                        if(!dataNode.getNodeName().equalsIgnoreCase("#text")){
                                Hashtable attributesHash=getAttributes(dataNode);
                                String name=((String)attributesHash.get("name")).trim();
                                String value=((String)attributesHash.get("value")).trim();
                                if(conversionType.equalsIgnoreCase("OODBMS")){
                                        methodCallVariableOrderingHash.put(name,value);
                                }

                                if(conversionType.equalsIgnoreCase("XDBMS_SCHEMA") ||
conversionType.equalsIgnoreCase("XDBMS_DTD")){
                                        output+="\t\t<"+name+">"+value.trim()+"</"+name+">\n";
                                }
                                if(conversionType.equalsIgnoreCase("RDBMS")){
                                        rdbmsOrderColVect.addElement(name);
                                        rdbmsOrderDataVect.addElement(value);
                                }
                        }
                }
                if(conversionType.equalsIgnoreCase("OODBMS")){
                        Vector
methodOrderVect=(Vector)methodDeclarationVariableOrderingHash.get(parentName);
                        for(int met=0;met<methodOrderVect.size();met++){
                                String variableName=(String)methodOrderVect.elementAt(met);
                                String
data=(String)methodCallVariableOrderingHash.get(variableName);
                                String datatype=getDatatype(variableName,parentName);
                                if(datatype.equalsIgnoreCase("String"))
                                        oodbmsObjectCreationFunctionCalls+=",\""+data+"\"";
                                else{
                                        if(datatype.equalsIgnoreCase("int")){
                                                if((data == null || data.equalsIgnoreCase("null")))

oodbmsObjectCreationFunctionCalls+=","+0;
                                                else
oodbmsObjectCreationFunctionCalls+=","+data;
                                        }
                                        else{
                                                if(datatype.equalsIgnoreCase("float"))
                                                        oodbmsObjectCreationFunctionCalls+=",
(float)"+data;
                                                else
oodbmsObjectCreationFunctionCalls+=","+data;
                                        }
                                }
                        }
                }
                if(conversionType.equalsIgnoreCase("RDBMS")){
                        output+=" (";
```

```
                    for(int i=0;i<rdbmsOrderColVect.size();i++){
                        output+=" "+rdbmsOrderColVect.elementAt(i)+",";
                    }
                    int index=output.lastIndexOf(",");
                    output=output.substring(0,index);
                    output+=")";
                    output+=" values (";
                    for(int i=0;i<rdbmsOrderDataVect.size();i++){
                        output+="'"+rdbmsOrderDataVect.elementAt(i)+"',";
                    }
                }
            }

            //Loads the class information for generated objects
            public void loadOODBMSStruct(){

                FileInformation fileInfo=new FileInformation();
                fileInfo.loadFileXML("config.xml");
                Hashtable fileHash=fileInfo.getFileInformation("oodbms","reversestruct");
                String fileStructName=(String)fileHash.get("filename");
                String fileStructPath=(String)fileHash.get("filelocation");
                String classBody="";
                Document document = null;
                document = DOMUtil.parse(fileStructPath+fileStructName);
                NodeList list = document.getElementsByTagName("classstatement");
                for(int i=0,i<list.getLength();i++){
                    Node element=list.item(i);
                    NodeList childNodes=element.getChildNodes();
                    String className="";
                    Vector methodVect=new Vector();
                    Hashtable variableHash=new Hashtable();
                    Vector methodDeclarationVariableOrderingVect=new Vector();
                    for(int k=0;k<childNodes.getLength();k++){
                        Node childNode=childNodes.item(k);
                        String childName=childNode.getNodeName();
                        if(childName.equalsIgnoreCase("classname")){
                            Hashtable tempHash=getAttributes(childNode);
                            className=(String)tempHash.get("name");
                            classBody+="\npublic void create"+className+"(String
objName";
                        }
                        if((k+1)==childNodes.getLength()){
                            classBody+=")\n";
                        }
                        if(childName.equalsIgnoreCase("variableinformation")){
                            NodeList variableChildNodes=childNode.getChildNodes();
                            String datatype="";
                            String methodName="";
                            String variable="";
                            String varName="";
                            for(int var=0;var<variableChildNodes.getLength();var++){
                                Node variableNode=variableChildNodes.item(var);
                                String varinfo=variableNode.getNodeName();
                                if(varinfo.equalsIgnoreCase("datatype")){
```

```
                                                    Hashtable
datatypeHash=getAttributes(variableNode);

datatype=(String)datatypeHash.get("name");
                                                }
                                    if(varinfo.equalsIgnoreCase("variable")){
                                                    Hashtable
variableTempHash=getAttributes(variableNode);

variable=(String)variableTempHash.get("name");
                                                    variableHash.put(variable,datatype);
                                                    varName="var"+k;
                                                    classBody+=","+datatype+" "+varName;

methodDeclarationVariableOrderingVect.addElement(variable);

                                                }
                                                if(varinfo.equalsIgnoreCase("setmethod")){
                                                    Hashtable
methodTempHash=getAttributes(variableNode);

methodName=(String)methodTempHash.get("name");

methodVect.addElement(methodName+"("+varName+");\n");
                                                }
                                            }
                                        }
                                    }

methodDeclarationVariableOrderingHash.put(className,methodDeclarationVariableOrderingVect);
                                    classBody+="{\n",
                                    classBody+=className+" "+className.toLowerCase()+"Obj =
("+className+" )(db.createObject( "+className+"Impl.class.getName(), 0, objName ));\n";
                                    for(int mn=0;mn<methodVect.size();mn++){

classBody+=className.toLowerCase()+"Obj."+(String)methodVect.elementAt(mn);
                                    }
                                    classBody+="\n}";
                                    classHash.put(className,variableHash),
                            }
                            classObjectCreation=classBody;
                    }

                    //Writes the files to location specified
                    public void writeToFile(String output,String fileName,String filePath){
                            try{
                                    File f=new File(filePath+fileName);
                                    FileWriter fw=new FileWriter(f);
                                    fw.write(output);
                                    fw.close();
                            }catch(IOException ex){}
                    }

                    //Extracts attributes of a tag
                    public Hashtable getAttributes(Node node){
```

```
                String strValue = "";
                String strName = "";
                Hashtable attributesHash=new Hashtable();
                NamedNodeMap attributes=node.getAttributes();
                for(int i=0;i<attributes.getLength();i++){
                        Node n=attributes.item(i);
                        strName = n.getNodeName().trim();
                        strValue = n.getNodeValue().trim();
                        attributesHash.put(strName,strValue);
                }
                return attributesHash;
        }

        //Returns data information for a variable of given class
        public String getDatatype(String variableName,String className){
                Hashtable tempHash=(Hashtable)classHash.get(className),
                String datatype=(String)tempHash.get(variableName);
                return datatype;
        }

        //Parse string with the given delimiter
        public Vector parse(String line,String delimiter){
                Vector parseVect=new Vector();
                StringTokenizer st=new StringTokenizer(line,delimiter);
                while(st.hasMoreElements()){
                        parseVect.addElement(st.nextToken());
                }

                return parseVect;
        }
}
```

APPENDIX A3: Source Code for Data type Transformer

```
package com.migrationtool.datatypes;

import java.io.*;
import java.util.*;
import java.text.NumberFormat;
import com.migrationtool.domloader.*;
import org.w3c.dom.Document;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.*;

//Loads the XML document having data type mapping information between heterogeneous databases and
//Provides methods to access the document nodes.

public class DatatypeMapping {

   Document doc = null;
   String thisDatabaseType="rdbms";
   static String xsdFile="F:/j2sdk1.4.1_01/bin/com/migrationtool/datatypes/datatypemapping.xml";

   //Constructor loads the file
   public DatatypeMapping(){
      doc = DOMUtil.parse(xsdFile);
   }

   //Returns the data type information for database information provided

   public Hashtable getDatatype(String fromDBMS,String datatype,String databaseType){
      thisDatabaseType=fromDBMS;
      Hashtable databaseDatatypeHash=new Hashtable();
      if(thisDatabaseType.equalsIgnoreCase(databaseType)){
         Vector tempVect=new Vector();
         tempVect.addElement(datatype);
         databaseDatatypeHash.put(databaseType,tempVect);
         return databaseDatatypeHash;
      }
      Element rootElement=doc.getDocumentElement();
      NodeList databaseNode=rootElement.getChildNodes();
      for(int i=0;i<databaseNode.getLength();i++){
         Node na=databaseNode.item(i);
         String topChildName=na.getNodeName();
         if(topChildName.equals(thisDatabaseType)){
            NodeList childNodes=na.getChildNodes();
            for(int k=0;k<childNodes.getLength();k++){
               Node childNode=childNodes.item(k);
               String childName=childNode.getNodeName();
               if(childName.equals(datatype)){
                  NodeList node=childNode.getChildNodes();
                  for(int m=0;m<node.getLength();m++){
                     Node databaseTypeRequested=node.item(m);
                     String nodeName=databaseTypeRequested.getNodeName();
```

```
                    Vector datatypeVect=new Vector();
                    if(!nodeName.equals("#text")){
                      NodeList datatypeNodes=databaseTypeRequested.getChildNodes();
                      for(int n=0;n<datatypeNodes.getLength();n++){
                        Node datatypeNode=datatypeNodes.item(n);
                        String datatypeName=datatypeNode.getNodeName();
                        if(!datatypeName.equals("#text")){
                          Hashtable attributesHash=getAttributes(datatypeNode);
                          String value=(String)attributesHash.get("name");
                          datatypeVect.addElement(value);
                        }
                      }
                      databaseDatatypeHash.put(nodeName,datatypeVect);
                    }

                  }
                }
              }

            }
          }
          return databaseDatatypeHash;
        }

        //Process attributes of the tag.
        public Hashtable getAttributes(Node node){
          String strValue = "";
          String strName = "";
          Hashtable attributesHash=new Hashtable();
          NamedNodeMap attributes=node.getAttributes();
          for(int i=0;i<attributes.getLength();i++){
            Node n=attributes.item(i);
            strName = n.getNodeName().trim();
            strValue = n.getNodeValue().trim();
            attributesHash.put(strName,strValue);
          }
          return attributesHash;
        }
        //Parses string with supplied delimiter
        public Vector parse(String line,String delimiter){
          Vector parseVect=new Vector();
          StringTokenizer st=new StringTokenizer(line,delimiter);
          while(st.hasMoreElements()){
            parseVect.addElement(st.nextToken());
          }
          System.out.println(parseVect);

          return parseVect;
        }
      }
```

VITA

Kalyan Pydipati was born in Raipur, India, on June 7th, 1978, the son of Pydipati Srinivasrao and Pydipati Nagamani. After finishing his high school from Gondia, INDIA, in 1995 he received his Bachelor's degree in Computer Engineering from Kavikulguru Institute of Technology and Science (University of Nagpur), Ramtek, INDIA. He then completed his Master's in Computer Science from Southwest Texas State University, San Marcos, Texas.

Permanent Address: 106 Bobcat drive, #C3, San Marcos, Texas – 78666, USA

This thesis was typed by Kalyan Pydipati.