COCALEREX: USER-ORIENTED APPROACH FOR REENGINEERING RELATIONAL DATABASES INTO XML

CHUNYAN WANG, ANTHONY LO, REDA ALHAJJ, KEN BARKER
Advanced Database Systems and Applications Lab
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{wangch, chiul, alhajj, barker}@cpsc.ucalgary.ca

ABSTRACT

In this paper, we present COCALEREX (converting relational to XML). It handles catalog-based by analyzing the metadata and legacy relational databases by first applying the reverse engineering approach described in [2] to extract the ER (Extended Entity Relationship) model from legacy relational databases. This reverse engineering approach is employed also to extract from catalog-based databases meta-data not available in the catalog. The ER is converted to XML schema with many-to-many and nary relationships considered. COCALEREX has a user-friendly interface that displays the result of each phase of the conversion process. Experimental results are encouraging, demonstrating the applicability and effectiveness of the proposed approach.

Keywords: reengineering, XML schema, relational database, conversion, user interface.

1. INTRODUCTION

XML has emerged as the standard format for publishing and exchanging data over the Internet. Since most data is currently stored and maintained in relational database management systems (RDBMS), which are still dominant, it is important to automate the process of generating XML documents containing information from existing databases. The Relational-to-XML transformation involves mapping relational tables and attributes into XML elements and attributes, creating XML hierarchies, and processing values in an application specific manner. As described in the literature, researchers mostly considered transforming relational databases that have rich corresponding catalogs. Although a large number of the existing relational databases are classified as legacy, the transformation of legacy relational databases to XML has received little attention. Legacy database systems are characterized by old-fashioned architecture, lack of the related documentation (missing or vague catalogs) and non-uniformity resulting from numerous mostly unorganized extensions.

Realizing the importance of transforming legacy databases into XML documents, we have developed a system, named **COCALEREX** (**Co**nversion of **Ca**talog-based and **Le**gacy **Re**lational databases to **X**ML), which successfully handles the transformation process for both legacy and catalog-based RDBMS. Our approach highly benefits from our previous findings on reverse engineering of legacy databases as detailed in [2], which leads to successful understanding of the design of an existing relational database. From our experience, we realized that some commercial RDBMS do not support the functionality to retrieve primary and foreign keys information from their metadata. For instance, even the latest version of MySql does not totally support such functionality. COCALEREX can extract all the required metadata either from the catalog or by analyzing database content.

As the work described in this paper is mainly a reengineering process, two basic steps are identified for transforming relational databases into XML, namely reverse engineering and forward engineering. The major target of the first step is reconstructing the ER model from the given relational database, and this process requires knowing the metadata. For legacy relational databases, reverse engineering is employed to deduce information about functional dependencies, keys and inclusion dependencies. For catalog-based databases, COCALEREX first connects to the utilized RDBMS by using JDBC/ODBC to obtain all the required available metadata information; and reverse engineering is employed here to extract information missing from the catalog, if any. Second, the obtained ER model is transformed into XML schema in the forward engineering step. Our approach smoothly handles all types of relationships allowed in the ER model, including many-to-many and nary relationships.

COCALEREX has been developed base on the framework described in [3]. Users may use COCALEREX for viewing the underlying relational data as either flat or nested XML structure. For the latter case, users can specify the nesting sequence. Further, users can directly view the result of each phase during the process.

COCALEREX consists of three main components. The first extracts the ER model for legacy databases; it is named EELRR- **E**xtracting **E**R Model from **L**egacy **R**elational Database by **R**everse Engineering Module. The second component is for extracting the ER model from cog-based databases; it is named EECR- **E**xtracting **E**R Model from **C**atalog-based **R**elational Database Module. The third is dedicated for transforming the obtained ER model into XML structure; it is named ER2X- **ER** model to **XML** Module. Experimental results are encouraging, demonstrating the applicability and effectiveness of the proposed approach.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Section 3 describes the reverse engineering process. Section 4 discusses the forward engineering process. Section 5 is summary and conclusions.

2. RELATED WORK

There exist several applications that enable the composition of XML documents from relational data, such as IBM DB2 XML Extender [9], SilkRoute [12], and XPERANTO [7]. XML Extender serves as a repository for XML documents as well as their Document Type Definitions (DTDs), and also generates XML documents from existing data stored in relational databases. It is used to define the mapping of relational tables and columns to DTD. XSLT and XPath syntax are used to specify the transformation and the location path. SilkRoute is described as a general, dynamic, and efficient tool for viewing and querying relational data in XML. It serves as middle-ware between a relational database and an application accessing that data over the Internet. In SilkRoute, XML views of relational databases are defined using a relational to XML transformation language called RXL, and then XML-QL queries are issued against views. The query composer combines the queries and views together, and the combined RXL queries are then translated into the corresponding SQL queries. In order to use SilkRoute system, it is necessary to learn the new language RXL. XPERANTO is middle-ware solution for publishing XML; object-relational data can be published as XML documents. It can be used by users who prefer to work with a "pure XML" environment. However, the transformation from relational schema to XML schema is specified by human experts.

The work described in [15] requires knowing the catalog contents of the given relational database in order to extract the relational schema. The conversion of Relational-to-ER-to-XML has been proposed in [13]. This reconstructs the semantic model, in the form of ER model, from the logical schema model capturing user's knowledge, and then transforms the ER model to the XML document. However, many-to-many (M:N) and nary ($n \geq 3$) relationships are not considered properly. Finally, VXE-R [5] is an engine for transforming a relational schema into equivalent XML schema. Then XML queries are issued directly against the XML schema. VXE-R is only used for a certain type of catalog-based relational databases; it does not work for legacy databases. Also, VXE-R does not provide a visualize interface to users.

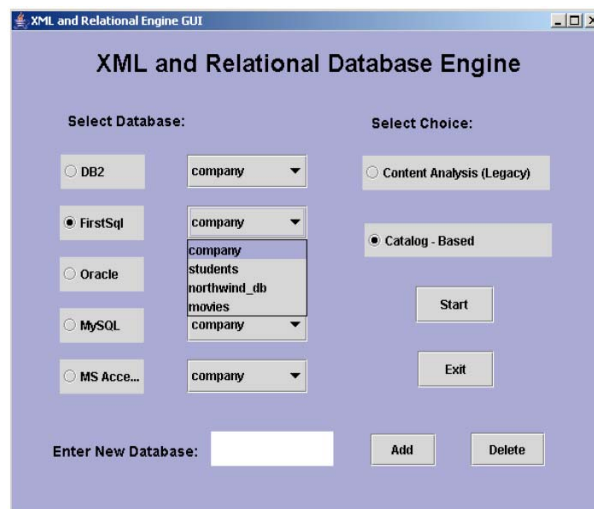

Figure 1: Main GUI of COCALEREX System

3. Reverse Engineering

COCALEREX is capable of handling both legacy and catalog-based databases. So users can use the main GUI shown in Figure 1 to specify the category of the database to be converted into XML. For the former category, the system calls the set of functions built inside EELRR to extract all the possible meta-information from the legacy database. The extracted information is required for constructing the ER model. For the latter category, the system connects to the RDBMS by using JDBC/ODBC, and calls the set of functions built inside EECR to get the catalog meta-information necessary for constructing the ER model. The meta-information includes candidate and foreign keys.

3.1. Extracting the ER Model from Legacy Relational Databases

EELRR is the most complex component of COCALEREX. Its main purpose is to extract all the necessary meta-information, including the foreign and candidate/primary keys) from the given legacy database; then it constructs the ER model. For this task, we implemented the reverse engineering approach proposed we already proposed in [2]. According to the flowchart shown in Figure 2, the ER model extraction process can be divided into six main steps; for more

details, the reader is referred to [2], which includes details of all the algorithms utilized for the reverse engineering process.

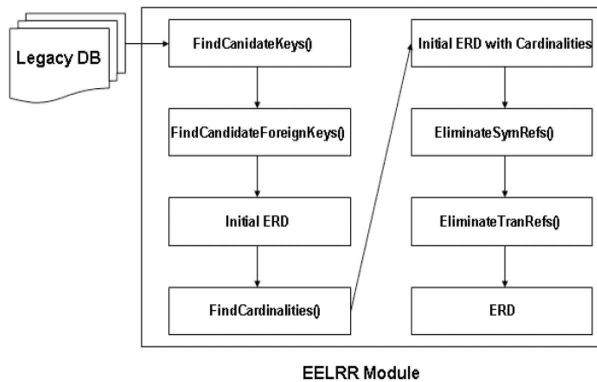

Figure 2: The Flow Chart of EELRR Module

1. For each table in the relational database: the powerset of its set of attributes is the input to the "Find Candidate Keys" algorithm, which finds all possible candidate keys in the database.
2. Run the "Find Candidate Foreign Keys" algorithm on the result obtained from Step 1 to find all attributes in the Foreign Keys, which are simply representatives of the Candidate Keys.
3. Use the result from Step 2 to construct the initial ERD.
4. Run the "Find Cardinalities" algorithm to determine the cardinalities of the relationships in the initial ERD.
5. Remove the extra information (if exist) from the initial ERD.
   (a) Run the "Eliminate Symmetric References" algorithm to eliminate all symmetric references [2] from the initial ERD.
   (b) Run the "Eliminate Transitive References" algorithm to eliminate transitive references [2] from the initial ERD.
   (c) As a result of executing steps (a) and (b), the optimized ERD is generated, and then displayed using the GUI.
6. Run the "Identify Relationships" algorithm to identify all many-to-many and nary relationships in the database, if any.

The process outlined above is detailed more in our work described in [3]. To illustrate this process, shown in Figure 3 is an example ERD extracted by EELRR for the example Company database described in [3].

3.2. Extracting ER Model from Catalog-based Relational Database

As a requirement of EECR, COCALEREX connects to the given catalog-based relational database by using JDBC/ODBC. The required foreign and primary keys information are obtained from the given RDBMS that support getPrimaryKeys() and getImportedKeys() functions; otherwise, the system invokes EELRR to extract the not-supported information. The EECR component of the reverse engineering process has been tested for different RDBMS, including DB2, FirstSql, MySql and Oracle.

4. FORWARD ENGINEERING

Existing approaches to deal with the conversion of relational databases into XML have mainly concentrated on one-to-one (1:1) and one-to-many (1:M) Relationships; while many-to-many (M:N) and nary (n>2) relationships have not received enough attention. We argue that it is essential to equally consider all types of relationships for the two cases of having a flat and a nested XML schema. So, we consider all types of relationships in our conversion process as detailed in the rest of this paper.

COCALEREX can properly handle 1:1, 1:M, M:N and nary relationships in the process of converting a given relational database into XML. It provides some functions that allow users to partially convert a selected portion of a given relational database into XML schema and produces the corresponding virtual XML document. For example, users who want to view from Figure 3 only the three relations: EMPLOYEE, DEPARTMENT, and DEPT_LOCATIONS converted into XML can do so by selecting them from the ERD displayed on the screen; and then their desired XML schema and document(s) will be displayed on the screen after clicking on the "Convert to XML" button shown in the bottom of Figure 3.

The responsibility of the ER2X component of COCALEREX is transforming the ER model of the given database into the corresponding XML schema. We decided to convert into XML schema and not DTD because the former is a more comprehensive and rigorous method for defining the content model of an XML document. The schema itself is an XML document, and so can be processed by the same tools that read the XML documents it describes. The XML schema supports rich built-in types and allows building complex types based on the supported built-in types. It also supports *key*, *keyref* and *unique constraints*, which are important for transforming a relational schema into XML schema.

We also consider mapping all different types of relational schema constraints, including: primary keys (PKs), foreign keys (FKs), null/not-null, unique, etc, to the XML schema. Basically, the null/not-null constraint can be easily represented by properly setting "minOccurs" of the XML element transformed from the relational attribute. The unique constraint can also be represented in a straightforward manner by the unique mechanism in the XML schema.

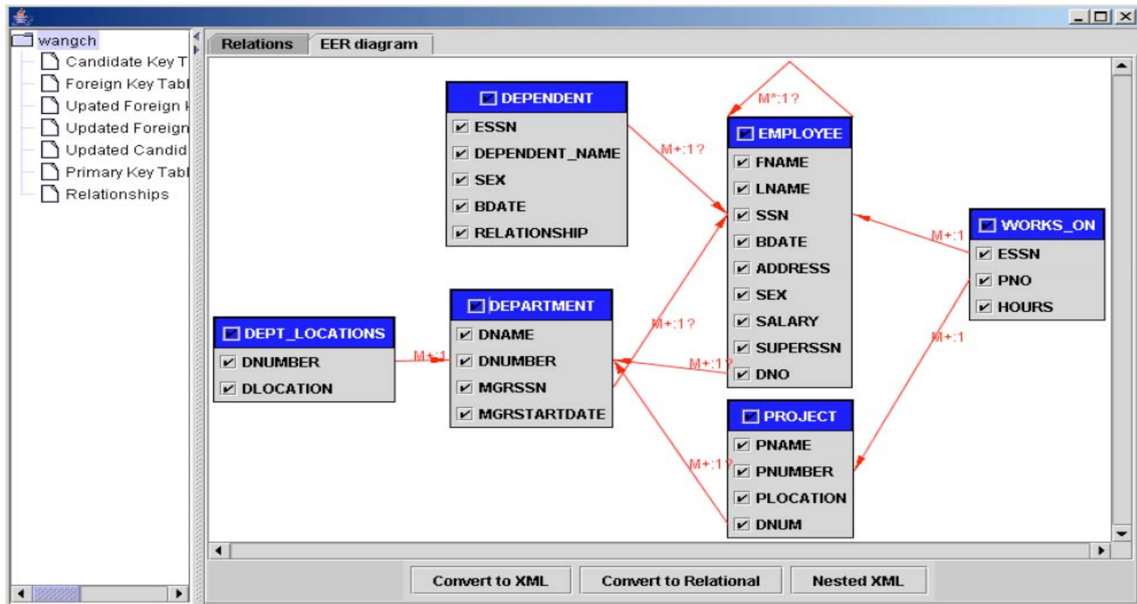

Figure 3: ERD Generated from Legacy Database

The ER2X component of COCALEREX by default generates a flat structure of the XML schema. However, users may specify a nested structure in a way to improve the performance of querying the corresponding XML documents. In the rest of this section, we first introduce the conversion into flat XML schema, and then we discuss the conversion into nested XML schema.

4.1. ER Model to Flat XML Schema

In this section, we present the proposed process for translating a conceptual schema (ER model) into a flat XML schema. The main steps of the process are given in Algorithm 4.1.

Algorithm 4.1 (ER Model to Flat XML Schema Conversion)

Input: The ER model

Output: The corresponding flat XML schema

Step:

1. Transform each entity type, M:N or nary relationship (we call them objects hereafter) from the ER model into a complex-type in the XML schema.
2. Map each attribute in an object transformed in Step (1) into a subelement within the corresponding complex-type.
3. Create a root element with the same name as the relational database schema name; and insert each of the three types of objects identified in the ER model as in Step (1) as a subelement with the corresponding complex-type.
4. Define the primary key for each of the three types of objects identified in Step (1) by using the "key" element.
5. Map in the ER model under consideration, each link between the three types of objects identified in Step (1) by using the "keyref" element.

EndAlgorithm

To understand the steps of Algorithm 4.1, we present next more details of the process with supporting examples.

Each object *E* in the ER model is translated in the XML schema into an XML complex-type of the same name *E*. In each complex-type *E*, there is only one empty element, which includes several subelements. This is demonstrated next where the PROJECT is translated into a complex-type named "PROJECT_Relation"; the empty element is called "PROJECT_Tuple".

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

The cardinality constraint in the ER model can be explicated by associating two XML built-in attributes, also called indicators, namely "minOccurs" and

“maxOccurs” with subelements under the XML complex-Type. The default value for both the “maxOccurs” and the “minOccurs” is 1. In case specified, the value for “minOccurs” should be either 0 or 1, and the value for “maxOccurs” should be greater than or equal to 1. If both “minOccurs” and “maxOccurs” are omitted, then the subelement must appear exactly once.

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

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

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

Each object in the ERD is mapped into the XML schema. We first need to create a root element that represents the entire given relational database. We create the root element as a complex-type in the XML schema, and give it the same name as the relational database schema, and then insert each object as a subelement of the root element. Next is an example which contains the six objects DEPARTMENT, DEPENDENT, DEPT_LOCATIONS, EMPLOYEE, PROJECT, WORKS_ON. We give the root element the name COMPANY:

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

The elements “key” and “keyref” are used to enforce the uniqueness and referential constraints. They are among the great features introduced in the XML schema. Also, we can use “key” and “keyref” to specify the uniqueness scope and multiple attributes in creating composite keys. Here is an example:

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

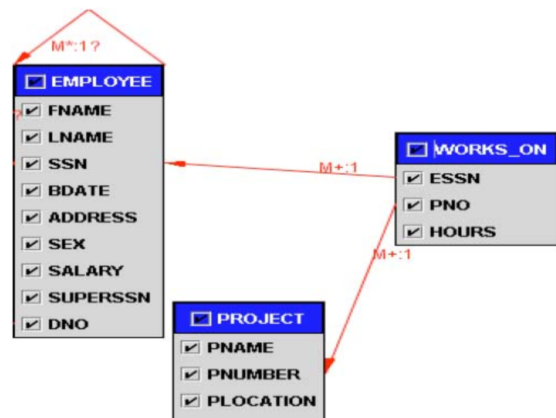

Figure 4: Many-to-Many relationship

In this example, we first specify the primary key for each object in the ER model. From the ForeignKeys table, we have PNUMBER as the primary key of PROJECT; ESSN and PNO together form a composite primary key for WORKS_ON. PNO is a foreign key in WORKS_ON, so we use “keyref” to specify the foreign key relationship between PROJECT and WORKS_ON.

Compared to DTD, the XML schema provides a more flexible and powerful mechanism through “key” and “keyref”, which share the same syntax as “unique” and also make referential constraints possible in XML documents.

An example M:N relationship is shown in Figure 4, where WORKS_ON is a binary relationship type. EMPLOYEE:WORKS_ON has cardinality ratio 1:M, which means that each employee record can be related to more than one record in WORKS_ON; the same is valid for the PROJECT:WORKS_ON relationship.

4.2. Transforming ER Model into Nested XML Schema

COCALEREX is capable of producing nested XML schema by employing Algorithm 4.2, given next in this section. The system may derive a nested XML schema if nesting is specified by the user as a choice, without a particular nesting request. Also, there is an interface for the user to specify the required nesting of different objects from the ER model. This facility gives power to users who are familiar with the most commonly raised types of queries, and who may specify a nested structure to speed up the processing of such queries.

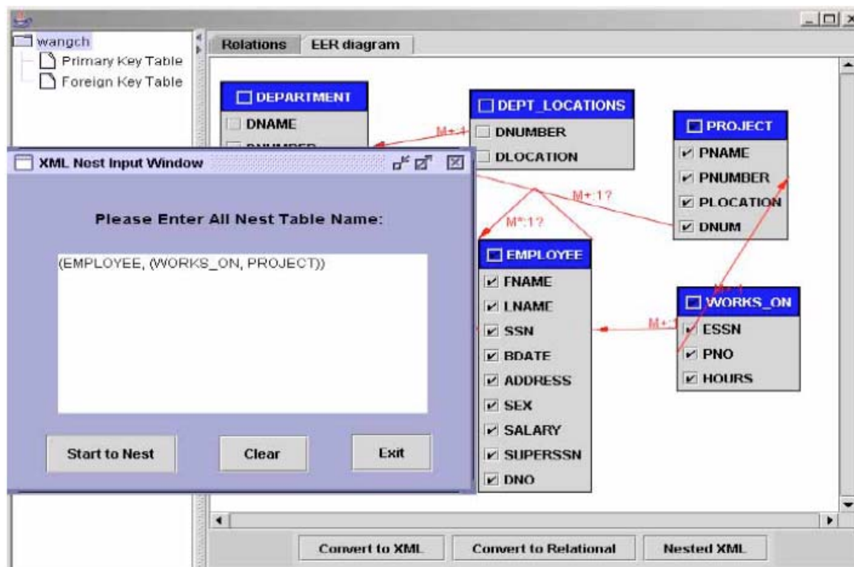

Figure 5: Nesting Input GUI

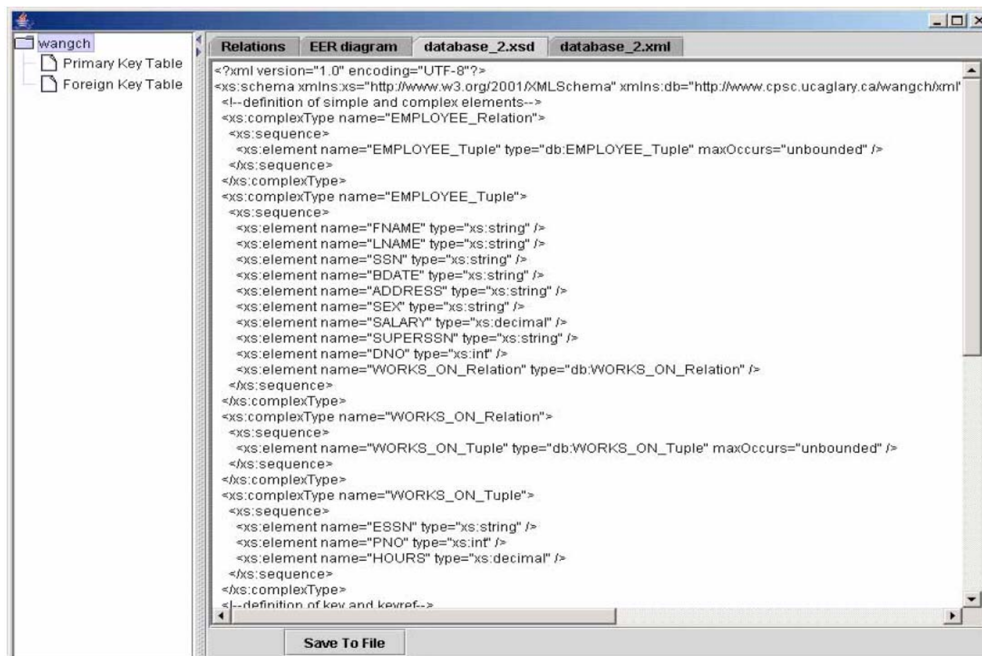

Figure 6: EMPLOYEE, WORKS ON and PROJECT Nested XML Schema Output

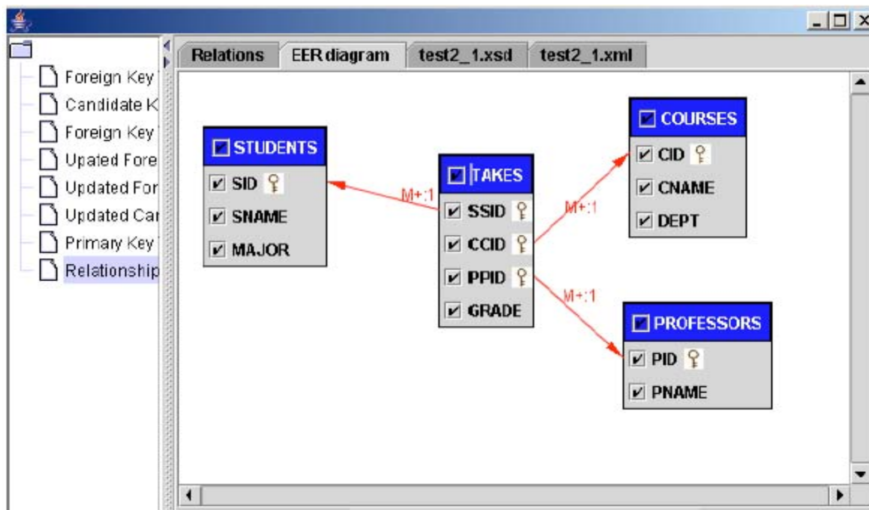

Figure 7: A Sample nary relationship

Relations | EER diagram | test2_1.xsd | test2_1.xml | test2_7.xsd | test2_7.xml

```
<xs:complexType name="PROFESSORS_Tuple">
 <xs:sequence>
  <xs:element name="PID" type="xs:string" />
  <xs:element name="PNAME" type="xs:string" />
  <xs:element name="PROFESSORS_nAryRelationship" type="db:PROFESSORS_nAryRelatic
 </xs:sequence>
</xs:complexType>
<xs:complexType name="STUDENTS_nAryRelationship">
 <xs:sequence>
  <xs:element name="STUDENTS_nAryRelationship_Tuple" maxOccurs="unbounded">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="CID" type="xs:string" maxOccurs="unbounded">
      <xs:keyref name="COURSES.CID" refer="COURSES_PrimaryKey">
       <xs:selector xpath="STUDENTS_Relation/STUDENTS_Tuple/COURSES_LIST" />
       <xs:field xpath="CID" />
      </xs:keyref>
     </xs:element>
     <xs:element name="PID" type="xs:string" maxOccurs="unbounded">
      <xs:keyref name="PROFESSORS.PID" refer="PROFESSORS_PrimaryKey">
       <xs:selector xpath="STUDENTS_Relation/STUDENTS_Tuple/PROFESSORS_LIST" />
       <xs:field xpath="PID" />
      </xs:keyref>
     </xs:element>
```

Save To File

Figure 8: Nested XML schema output for the example nary relationship shown in Figure 7

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

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

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

1. The nesting specified using $(E_1;E_2;E_3;E_4;E_5)$ means that each of E_5, E_4, E_3, and E_2 is nested inside E_1. This is one level tree rooted at $E1$ and each node E_i $(2 \leq i \leq 5)$ is at level one.
2. The nesting specified using $(E_1; (E_2; (E_3; (E_4;E_5))))$ means that E_5 is nested inside E_4, E_4 is nested

inside E_3, E_3 is nested inside E_2, and E_2 is nested inside E_1. This is a chain, i.e., 4 levels tree rooted at E_1 and each node E_i ($2 \le i \le 5$) is at level $i - 1$.

3. The nesting specified using $(E_1;(E_2;E_3);E_4;(E_5;(E_6;E_7));E_8)$, is interpreted as follows: E_3 is nested inside E_2, $E7$ is nested inside E_6 and E_6 is nested inside E_5; then E_2, E_4, E_5 and E_8 are all nested inside E_1.

To illustrate the nesting process, consider the COMPANY database where users may most of the time write queries to retrieve information about employees and their projects. For this case, users can specify the nesting sequence in the Nested Input GUI as: (EMPLOYEE, (WORKS_ON, PROJECT)).

The ER2X module takes such sequence as input, and generates as output the XML schema in nested structure. The element PROJECT is nested under the element WORKS_ON; then the nested element moves under the element EMPLOYEE to build two levels of nested XML structure.

Algorithm 4.2 (ER Model to Nested XML Schema Conversion)

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

// if the nesting sequence is specified as tuple then the
// system decides on the nesting sequence as outlined in
// the else part of the algorithm

Output: The nested XML schema

Steps:

If the user specified a non-empty nesting tuple) then

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

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

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

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

    If E_2 contains only foreign keys then

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

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

        If E_i contains some instances not participating in the relationship then

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

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

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

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

Else (the user has not specified the nesting sequence and hence the system will decide on the nesting)

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

  For objects connected by M:N or nary relationships do

    Repeatedly nest the other object(s) involved in the relationship inside object E_i which is involved in the relationship and has the next smallest number of instances not participating in the relationship. (Attributes of each such relationship are added inside the deepest nested object.)

    For each object E_j nested inside E_i do

      If E_j contains some instances not participating in the relationship then

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

EndAlgorithm

In the example shown in Figure 4, WORKS_ON has two foreign keys: ESSN refers to EMPLOYEE and PNO refers to PROJECT; and also WORKS_ON has an attribute HOURS. COCALEREX allows users to specify the objects they want to nest by using the interface shown in Figure 5, where a user may enter the nesting sequence as (EMPLOYEE, (WORKS_ON, PROJECT)). After clicking on the "Start to Nest" button, only EMPLOYEE, WORKS_ON and PROJECT are selected. When the user clicks on "Convert to XML" button, the XML schema output generated by COCALEREX is displayed on the screen as shown in Figure 6.

In the nested XML schema example above, because WORKS_ON.ESSN is the foreign key of EMPLOYEE, and WORKS_ON.PNO is the foreign key of PROJECT. They are both removed by Algorithm 4.2, so only WORKS_ON.HOURS is left. If WORKS_ON did not contain the attribute HOURS, then it would have been removed, which is another case. In the XML document generated, all records in PROJECT which relate to WORKS_ON are nested under a particular WORKS_ON record, and all records in WORKS_ON which relate to a particular EMPLOYEE are nested under that EMPLOYEE record. This will form two levels of nested XML structure.

Shown in Figure 8 is the nested XML schema for the example nary relationship TAKES, which connects three objects: STUDENTS, COURSES, and PROFESSORS as shown in Figure 7. There are three foreign keys in TAKES: SID, CID and PID, in addition to the attribute GRADE. Assume users often query for information related to students taking courses and their professors. So, they may choose to nest both COURSES and PROFESSORS under TAKES, then nest TAKES under STUDENTS. Part of the output two levels nested XML schema Shown in Figure 8.

Compared to the flat XML structure, the nested XML structure in fact improves the response of certain queries because there is no need to use the keyref to perform additional scans to find information in other parts of the referenced XML document. However, the nested XML structure has data redundancy because some records will repeat several times. Here, it is important to emphasize that the main purpose for facilitating nested structures is to allow the users to construct XML document(s) most suitable for efficient information retrieval. So, it is more appropriate to let the users make the nesting decision.

4.3. Generating XML Documents

After the XML schema is obtained, COCALEREX can generate the required XML document(s) from the considered relational database. Algorithm 4.3 checks top-down through the list of selected objects and generates an element for each object.

Algorithm 4.3 (Generating XML Documents based on a produced XML Schema)

Input: The XML schema and the Relational database

Output: The corresponding XML Document(s)

Steps:

Create XML document and set its namespace declaration

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

For each relation R in the relational database do

If R is selected and does not contain any nested relations then

Create R_Relation element for R

Let queryString = “select * from R”

ResultSet = execute(queryString)

For each tuple T in ResultSet

Create R_Tuple element for tuple T

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

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

Create R_Relation element for R and R_c_Relation for R_c

Let queryString = “select selectedAttrs from R, R_c”

ResultSet = execute(queryString)

For each tuple T in ResultSet do

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

Create an element for each selected attribute in R and insert it into the R_Tuple element; and do the same for R_c

EndAlgorithm

Algorithm 4.3 can generate flat XML document(s) as well as nested XML document(s), depending on the processed XML schema. In Algorithm 4.3, a query is executed to obtain all tuples that satisfy the constraints, one element is created to store data of each tuple in the result set. The following shows fragment of the XML document output for the COMPANY database.

```
<?xml version="1.0" encoding="UTF-8"?> <test
xmlns="http://www.cpsc.ucaglary.ca/wangch/xml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"xsi:schemaLocation="http://www.cpsc.ucaglary
.ca/wangch/xml test_2.xsd">
  <db:EMPLOYEE_Relation
xmlns:db="http://www.cpsc.ucaglary.ca/wangch/xml">
    <db:EMPLOYEE_Tuple>
      <db:FNAME>James</db:FNAME>
      <db:LNAME>Borg</db:LNAME>
      <db:SSN>888665555</db:SSN>
      <db:BDATE>1927-11-10</db:BDATE>
      <db:ADDRESS>450 Stone, Houston,
        TX</db:ADDRESS>
      <db:SEX>M</db:SEX>
      <db:SALARY>55000.00</db:SALARY>
      <db:SUPERSSN />
      <db:DNO>1</db:DNO>
      <db:WORKS_ON_Relation>
        <db:WORKS_ON_Tuple>
          <db:HOURS />
          <db:PROJECT_Relation>
            <db:PROJECT_Tuple>
              <db:PNAME>Reorganization</db:PNAME>
              <db:PNUMBER>20</db:PNUMBER>
            <db:PLOCATION>Houston</db:PLOCATION>
              <db:DNUM>1</db:DNUM>
            </db:PROJECT_Tuple>
          </db:PROJECT_Relation>
        </db:WORKS_ON_Tuple>
      </db:WORKS_ON_Relation>
    </db:EMPLOYEE_Tuple>
    . . . . . .
  <db:EMPLOYEE_Relation
```

5. SUMMARY AND CONCLUSIONS

In this paper, we introduced the architecture and main components COCALEREX, which has been developed

as XML database engine. We presented some of the functionalities provided by COCALEREX; users can select the category of relational database they want to use as either legacy or catalog-based. COCALEREX extracts all useful information from the given relational database to construct the ER model; and then transforms the ER model into the corresponding XML schema and virtual document. COCALEREX can properly handle binary and nary relationships. It is capable of generating flat and nested XML structure based on users' desire. The result of each phase in the process is displayed on the GUI; this provides a friendly visualization to users so that they can clearly view the results in each phase. COCALEREX can also be used as a tool for designers to redesign or update their existing database systems. This way designers' workload is considerably reduced. We argue that COCALEREX is a useful tool for users to construct an ER model from a given relational database and view the corresponding XML schema and documents(s) in a user-friendly way.

REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener, "The Lorel Query Language for Semistructured Data," *International Journal on Digital Libraries*, Vol.1, No.1, pp.68-88, April 1997.

[2] R. Alhajj, "Extracting the Extended Entity-Relationship Model from a legacy Relational Database," *Information Systems*, Vol.28, No.6, pp.597-618, 2003.

[3] C. Wang, A. Lo, R. Alhajj and K. Barker, "Converting Legacy Relational Database into XML Database through Reverse Engineering," *Proceedings of the International Conference on Enterprise Information Systems*, Porto, Portugal, Apr. 2004.

[4] Lo, R. Alhajj and K. Barker, "Flexible User Interface for Converting Relational Data into XML," *Proceedings of the International Conference on Flexible Query Answering Systems*, Springer-Verlag, Lyon, France, June 2004.

[5] C. Liu, M. W. Vincent, J. Liu, and M. Guo, "A Virtual XML Database Engine for Relational Databases," Springer-Verlag, 2003.

[6] Bonifati and D. Lee, "Technical Survey of XML Schema and Query Languages," *Technical report, UCLA Computer Science Department*, June 2001.

[7] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Schanmugasundaram, E. Shekita and S. Subramanian, "XPERATO: Publishing Object-Relational Data as XML," *Proceedings of the International Workshop on Web and Databases*, May 2000.

[8] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca, "XML-GL: a graphical language for querying and restructuring XML documents," *Computer Networks*, 31(11–16), pp.1171-1187, 1999.

[9] J. Cheng and J. Xu, "IBM DB2 XML Extender," *IBM Silcom Valley*, February, 2000.

[10] T.T. Chinenyanga and N. Kushmerik, "Expressive retrieval from XML documents," *Proceedings of ACM International Conference on Research and development in Information Retrieval*, New York, pp.163-171, 2001.

[11] M. Erwig, "Xing: A Visual XML Query Language," *Journal of Visual Languages and Computing*, Vol.14, No.1, pp.5-45, 2003.

[12] M. F. Fernandez, W. C. Tan and D. Suciu, "SilkRoute: Trading between Relational and XML," *Proceedings of the International Conference on World Wide Web*, May 2000.

[13] J. Fong, F. Pang and C. Bloor, "Converting Relational Database into XML Document," *Proceedings of the International Workshop on Electronic Business Hubs*, pp.61-65, Sep. 2001.

[14] G. Kappel, E. Kapsammer, S. Rausch-Schott and W. Retschitzegger, "X-Ray - Towards Integrating XML and Relational Database Systems," *Proceedings of the International Conference on Conceptual Modeling*, pp. 339-353, Salt Lake City, UT, Oct. 2000.

[15] D. Lee, et al, "Nesting based Relational-to-XML Schema Translation," *Proceedings of the International Workshop on Web and Databases*, May 2001.

[16] D. Lee, et al, "NeT and CoT: Translating Relational Schemas to XML Schemas using Semantic Constraints," *Proceedings of ACM CIKM*, McLean, Virginia, Nov. 2002.

[17] M. Mani, et al, "Taxonomy of XML Schema Language using Formal Language Theory," *In Extreme Markup Languages*, Montreal, Canada, August, 2001.

[18] J. Shanmugasundaram, et al, "Efficiently Publishing Relational Data as XML Documents," *VLDB Journal*, Vol.10, pp.133-154, 2001.