Two-Way Mapping between Object-Oriented Databases and XML

Taher Naser
School of Informatics, Bradford University, Bradford, West Yorkshire, United Kingdom

Reda Alhajj
Department of Computer Science, University of Calgary, Calgary, Alberta, Canada;
Department of Computer Science, Global University, Beirut, Lebanon

Mick J. Ridley
School of Informatics, Bradford University, Bradford, West Yorkshire, United Kingdom

Keywords: algorithms, data migration, data re-engineering, object-oriented databases, XML

Received: October 12, 2008

This paper presents a novel approach for mapping an existing object-oriented database into XML and vice versa. The major motivation to carry out this study is the fact that it is necessary to facilitate platform independent exchange of the content of object oriented databases and the need to store XML in a structured database. There are more common features between the object-oriented model and XML and thus the the two-way mapping from object-oriented databases into XML (and vice versa) should be less problematic. To achieve the mapping, what we call the object graph is derived based on characteristics of the schema to be mapped. For object-oriented schema, the object graph simply summarizes and includes all nesting and inheritance links, which are the basics of the object-oriented model. Then, the inheritance is simulated in terms of nesting to get a simulated object graph. This way, everything in a simulated object graph is directly representable in XML format. Finally, we handle the mapping of the actual data from the object-oriented database into corresponding XML document(s). On the other hand, the common features between the object-oriented model and XML make it is more attractive to map from XML into object-oriented database; such mapping preserves database specifics. To achieve the mapping, the object graph is derived based on characteristics of the XML schema; it simply summarizes and includes all complex and simple elements and the links, which are the basics of the XML schema. Then, the links are simulated in terms of nesting to get a simulated object graph. This way, everything in a simulated object graph is directly representable in object-oriented database. Finally, we handle the mapping of the actual data from XML document(s) into the corresponding object-oriented database.

Povzetek: Prispevek predstavlja izvirno dvostransko preslikavo med objektnimi podatkovnimi bazami in XML.

1 Introduction

XML is emerging as the standard format for data exchange between different partners. Since most of the data nowadays reside in structured databases including relational and object-oriented databases, it is important to automate the process of generating XML documents containing information from existing databases. Of course, one would like to preserve as much information as possible during the transformation process. The object-oriented database [12, 13] to XML conversion involves mapping the classes and attributesŠ names into XML elements and attributesŠ names, creating XML hierarchies, and processing values in an application specific manner. This paper addresses the mapping of the contents of an existing object-oriented database into XML; the reverse process is also supported to allow storing XML data in object-oriented database. The major motivation to carry out this study is the fact that there is a need for platform independent format for exchanging data; XML is accepted as one standard for achieving such task. We initiated this study based on our previous research related to object-oriented databases and database re-engineering as illustrated, respectively, in [3, 4, 5, 6, 7, 8] and [9, 10].

The mapping from object-oriented data into XML has not received considerable attention. On the other hand, there exist several tools that enable the composition of XML documents from relational data, such as IBM DB2 XML Extender, SilkRoute, and XPERANTO. XML Extender [15] serves as a repository for XML documents as well as their Document type declarations (DTDs), and also generate XML documents from existing data stored in relational database; it is used to define the mapping of DTD to relational tables and columns. XSLT and Xpath syntax are used to specify the transformation and the location path. SilkRoute [17] is described as a general, dynamic, and efficient tool for viewing and querying relational data in XML. XPERANTO [14] is a middleware solution for publishing XML; object-relational data can be published as XML documents. It can be used by developers who prefer to work in a “pure XML" environment. However, the mapping from the relational schema to the XML schema is specified by human experts. Therefore, when a large relational schema and corresponding data need to be translated into XML documents, a significant investment of human effort is required to initially design the target schema. Finally, the work described in [20] requires knowing the catalog contents in order to extract the relational database schema. The conversion of Relational-to-ER-to-XML has been proposed in [18]. This reconstructs the semantic model, in the form of ER model, from the logical schema capturing user’s knowledge, and then converts it to the XML document. However, many-to-many (M:N) and nary relationships are not considered properly. Finally, DB2XML [26] is a tool for transforming data from relational databases into XML documents; DTDs are generated by describing the characteristics of the data for making the documents self contained and usable as a data exchange format.

The conversion of Relational-to-ER-to-XML is described in [18]. VXE-R [21] is an engine for transforming a relational schema into equivalent XML schema. As the mapping from XML to object-oriented databases is concerned, the work described in [16] generates an object-oriented database schema from DTDs, stores it into the object-oriented database and processes XML queries; it mainly concentrate on representing the semi-structural part of XML data by inheritance. However, in this paper we differentiate between inheritance and nesting, which is a more natural approach for handling object-oriented databases. The work presented in [19] focuses on the ability to wrap an XML schema definition in an object-oriented virtual database mediator system to help solving the integration problems between XML documents and other applications that are not using XML. Toth and Valenta [25] investigated possibilities of reusing already known techniques from object and object-oriented processing in XML-native database systems.

This paper addresses the two-way mapping of the contents of an existing XML and object-oriented database. The major motivation to carry out this study is the fact that there are more common features between XML and object-oriented databases; thus it is more attractive to store XML schema and Data, and more data is preserved. This is actually the backward engineering [24]; the forward engineering part extracts XML from object-oriented database [23]. The forward engineering process takes a given object-oriented database as input and produces a corresponding XML schema and XML document(s). The first step in the process is to derive a summary of the object-oriented schema. This has been realized as object graph which includes inheritance and nesting links present in the object-oriented schema. Then, the object graph is transformed into XML schema and the object-oriented data is mapped into corresponding XML document(s). The process is capable of producing both nested and flat XML schemas. However, as the transformation is from object-oriented databases, producing the nested schema is preferred and more emphasized. The backward engineering process, on the other hand, takes a given XML schema as input and produces a corresponding object-oriented schema. The first step in the process is to derive a summary of the XML schema. This has been realized as object graph, which includes inheritance and nesting links derived from the XML schema. Then, the object graph is mapped into object-oriented schema. The process is capable of taking as input both nested and flat XML schemas. However, as the mapping is into object-oriented schema, nested XML schema is preferred and more emphasized.

The rest of the paper is organized as follows. Described in Section 2 is the information related to the object-oriented schema and the XML schema; the object graph is also defined. Section 3 presents the algorithm that derives the XML schema from the object graph. Section 4 describes the backward engineering process. Section 5 includes a summary and the conclusions.

2 The necessary background and terminology

2.1 Object-Oriented Database Characteristics

In this section, we investigate characteristics of the given object-oriented database and as a result derive the object graph. We start by presenting the basic terminology and definitions required to understand the analysis.

2.1.1 The Basic Terminology and Definitions

We are mainly interested in class characteristics as present in Definition 2.1 and illustrated in Example 2.1, given next.

Definition 2.1 (Class).
A class is defined to be a tuple, $(C_p(c), C_b(c), L_{attributes}(c), L_{behavior}(c), L_{instances}(c), OIDG)$, *where* c *is class identifier,* $C_p(c)$ *is a list*[1] *of direct superclasses of class* c, $C_b(c)$ *is a set*[2] *of direct subclasses of class* c, $L_{attributes}(c)$ *is the set of additional attributes locally defined in class* c, $L_{behavior}(c)$ *is the set of additional methods added to the definition of class* c, $L_{instances}(c)$ *is the set of object identifiers of objects added locally to class* c, *and* $OIDG$ *is object identifier generator that holds*

[1] A list notation is used for the superclasses because their order is important for conflict resolution due to polymorphism and overriding. Conflicts are resolved according to certain predefined rules discussed in [4].

[2] Conflict resolution is not applicable here because only objects are concerned, hence the set notation is utilized.

the identifier to be granted to the next object to be added to $L_{instances}(c)$. •

Every attribute in a class has a domain. Inheritance makes it possible for a class to utilize the attributes and methods defined for its superclasses, without violating polymorphism and overriding rules discussed in [4]. The set of objects of a class includes objects in its subclasses. All of this is formalized in the following definitions.

Definition 2.2 (Domain). *Let c_1, c_2, ..., and c_n be primitive and user defined classes, where primitive classes include reals, integers, strings, etc. The following are possible domains,*

1. *$(a_1{:}c_1, a_2{:}c_2, ..., a_n{:}c_n)$ is a tuple domain; a possible value is a tuple with the constituting values being object identifiers selected from classes c_1, c_2, ..., and c_n, respectively.*

2. *c_i, $1{\leq}i{\leq}n$, is a domain; a possible value is an object identifier from class c_i.*

3. *$\{d\}$ is a domain, where d may be any of the two domains defined in 1 or 2; a possible value is a set of values from domain d.*

4. *$[d]$ is a domain, where d may be any of the two domains defined in 1 or 2; a possible value is a list of values from domain d.* •

Definition 2.3 (Attributes). *Given a class c; the set of attributes that determine the state of each object in $L_{instances}(c)$ is denoted by $W_{attributes}(c)$ and defined recursively in terms of the attributes defined for objects of the classes in $C_p(c)$.*
$W_{attributes}(c) = L_{attributes}(c) \bigcup_{i=1}^{n} W_{attributes}(c_{p_i})$. •

Definition 2.4 (Behavior). *Given a class c and let $C_p(c)$=[c_{p_1}, c_{p_2}, ..., c_{p_n}] be the list of its direct superclasses. The whole behavior for class c, denoted by $W_{behavior}(c)$, is recursively defined to include the whole behavior of the classes in $C_p(c)$.*
$W_{behavior}(c)$=$L_{behavior}(c)$ $\bigcup_{i=1}^{n} W_{behavior}(c_{pi})$. •

Definition 2.5 (Extent). *Given a class c and let $C_b(c)$={c_{b_1}, c_{b_2}, ..., c_{b_m}} be the set of its direct subclasses. All objects that understand at least the behavior in $W_{behavior}(c)$, constitute the extent of class c, denoted by $W_{instances}(c)$. This set is recursively defined in terms of the extents of the classes in $C_b(c)$.*
$W_{instances}(c) = L_{instances}(c) \bigcup_{i=1}^{m} W_{instances}(c_{b_i})$. •

Example 2.1 (Classes).
Next are characteristics of the classes in the object-oriented schema:

Person:
$C_p(Person)$*=[]* $C_b(Person)$*={Student, Staff, Secretary}*
$L_{attributes}$*(Person)={SSN:integer, name:string, age:integer, sex:character, spouse:Person, nation:Country}*
$L_{behavior}(Person)$*={SSN(), SSN(i), name(), name(t), age(), age(i), sex(), sex(t), spouse(), spouse(p), nation(), nation(c)}*

Country:
$C_p(Country)$*=[]* $C_b(Country)$*={}*
$L_{attributes}(Country)$*={Name:string, area:integer, population:integer}*
$L_{behavior}(Country)$*={Name(), Name(t), area(), area(i), population(), population(i)}*

Student:
$C_p(Student)$*=[Person]*
$C_b(Student)$*={ResearchAssistant}*
$L_{attributes}(Student)$*={StudentID:integer, gpa:real, student_in:Department,Takes:{(course:Course, grade:string)}}*
$L_{behavior}(Student)$*={StudentID(), StudentID(i), gpa(), gpa(i), student_in(), student_in(d), Takes(), Takes(t)}*

Staff:
$C_p(Staff)$*=[Person]*
$C_b(Staff)$*={ResearchAssistant}*
$L_{attributes}(Staff)$*={StaffID:integer, salary:integer, works_in:Department}*
$L_{behavior}(Staff)$*={StaffID(), StaffID(i), salary(), salary(i), works_in(), works_in(d)}*

ResearchAssistant:
$C_p(ResearchAssistant)$*=[Student, Staff]*
$C_b(ResearchAssistant)$*={}*
$L_{attributes}(ResearchAssistant) = \{\}$
$L_{behavior}(ResearchAssistant) = \{\}$

Course:
$C_p(Course)$*=[]* $C_b(Course)$*={}*
$L_{attributes}(Course)$*={Code:integer, title:string, credits:integer, Prerequisite:{Course}}*
$L_{behavior}(Course)$*={Code(), Code(i), title(), title(t), credits(), credits(i), Prerequisite(), Prerequisite(c)}*

Department:
$C_p(Department)$*=[]* $C_b(Department)$*={}*
$L_{attributes}(Department)$*={Name:string, head:Staff}*
$L_{behavior}(Department)$*={Name(), Name(t), head(), head(t)}*

Secretary:
$C_p(Secretary) = [Person]$ $C_b(Secretary) = \{\}$
$L_{attributes}(Secretary)$*={words/ minute:integer, works_in:Department}*
$L_{behavior}(Secretary)$*={words/ minute(), words/mimute(i), works_in(), works_in(d)}* •

It is clear from Example 2.1 that the behavior of a class contains two methods for every attribute. These methods are automatically generated by the system when the attribute is defined. For instance, the two methods $SSN()$, $SSN(i)$ are included in $L_{behavior}(Person)$ because attribute SSN belongs to $L_{attributes}(Person)$. While the first method retrieves the value of attribute SSN from the receiving object, the second method $SSN(i)$ sets the value of attribute SSN within the receiving object to the value of the argument i.

2.1.2 The Necessary Object-Oriented Schema Information

Related to the object-oriented schema, the analysis is based on the domain information summarized in the following table.

$ObjectAttributes(class\ name,\ attribute\ name,\ domain)$

Class Name	Attribute Name	Domain
Person	ssn	integer
Person	name	string
Person	age	integer
Person	sex	integer
Country	name	string
Country	area	integer
Country	population	integer
Student	studentID	integer
Student	gpa	real
Staff	satffID	integer
Staff	salary	integer
Course	code	integer
Course	title	string
Course	credits	integer
Department	name	string
Secretery	word_minute	integer
T1	grade	string

(a)

Class Name	Attribute Name	Domain
Person	spouse	Person
Person	nation	Country
Student	student_in	Department
Student	Takes	T1
Staff	work_in	Department
Course	prerequisite	Course
Department	head	Staff
Secretery	work_in	Department
T1	course	Course

(b)

Table 1: *ObjectAttributes*: (a) a list of all attributes with primitive domains (b) a list of all attributes with non-primitive domains

The *ObjectAttributes* table includes information about all attributes in the object-oriented schema. For each attribute, it is required to know its name, class and domain. Attributes with primitive domains and attributes with non-primitive domains are placed in separate occurrences of the *ObjectAttributes* table, namely *ObjectAttributes*(a) and *ObjectAttributes*(b), respectively. Table 1 includes the *ObjectAttributes* information related to the object-oriented schema introduced in Example 2.1. Each domain of the tuple type is assigned a short name that consists of the letter $'T'$ suffixed with a consecutive non-decreasing number, starting with 1. For instance, as shown in the fourth row in Table 1(b), the short name T_1 has been assigned to the domain of the attribute $Takes$ from $L_{instances}(Student)$. This way, it becomes trivial to identify attributes that appear within a tuple domain as illustrated in the last row of each of *ObjectAttributes*(a) and *ObjectAttributes*(b) in Table 1.

2.2 XML Schema Characteristics

XML schema is a language for describing the structure and constraining the content of XML documents. So, it can be described as a set of rules to which an XML document must conform in order to be considered *well-formed* and *valid* document.

In our work, we will use XML Schema complex type elements, primitive type elements and sequence indicator. XML Schema allows us to define the cardinality of an element (*i.e.*, the number of its possible occurrences) with some precision. This cardinality constraint can be explicated by associating the two XML built-in attributes *minOccurs* and *maxOccurs*, with subelements under the "complexType" element. We can specify both *minOccurs* (the minimum number of occurrences) and *maxOccurs* (the maximum number of occurrences). It is possible to set *maxOccurs* to unbounded, which means that there can be as many occurrences of the character element as the author wishes. Both attributes have a default value of one. If both *minOccurs* and *maxOccurs* are omitted, the subelement must appear exactly once.

Definition 2.6 (ComplexType). *A complextype element is defined as a tuple, ($C_{complextypes}(ct)$, $C_{primitivetypes}(ct)$, $C_{keys}(ct)$, $C_{keyrefs}(ct)$), where ct is the complextype identifier, $C_{complextypes}(ct)$ is the complextype elements (subelements) of complextype ct, $C_{primitivetypes}(ct)$ is the set of primitive type elements of ct, $C_{keys}(ct)$ is the set of keys defined for ct, $C_{keyrefs}(ct)$ is the set of key references defined for ct.*

To demonstrate the complex type concept introduced in Definition 2.6, consider Example 2.2 which starts by a description of some related entities followed by the corresponding XML schema definition.

Example 2.2 (XML Schema). *Consider the following set of related entities. Each entity has a set of attributes and a primary key; the domain of each attribute is also specified. Drawing the corresponding entity-relationship diagram is straightforward by considering the given summary.*

Person Complex Type*: Key = SSN*
attributes = {ssn :integer; name:string; age :integer; sex :character; spouse :Person; nation :Country}

Country Complex Type*: KEY= Name*
attributes = {name:string; area :integer; population :integer}

Student Complex Type*: Key=StudentID*
attributes = {StudentID:integer; gpa :real; student in :Department; Takes:{(course :Course; grade :string)}}

Staff Complex Type*: Key=StaffID*
attributes = {StaffID:integer; salary :integer; works_in :Department}

ResearchAssistant Complex Type*: links to both student and staff; hence gets the key of either one.*

attributes ={student:Student,staff:Staff}
Course Complex Type*:Key=code*
attributes = {Code :integer; title :string; credits :integer; Prerequisite :{Course}}
Department Complex Type*: Key=name*
attributes = {name:string; head :Staff}
Secretary Complex Type*: links to person; hence gets the primary key of person.*
attributes ={person:Person, wordsperminute :integer; works in :Department}

The next XML Schema segment depicts part of the nested XML Schema. It describes the PERSON and COUNTRY complextype elements. It shows the COUNTRY element as a child element of the PERSON complex type element. PERSON complextype includes an empty PERSON_Object element, where PERSON_Object element is defined as a complex type that includes all attributes of PERSON element. The "sequence" indicator is used as sequential semantic for the set of subelements. It is a flat XML Schema, where the subelement COUNTRY in the complex type element PERSON is defined as a string element type. The two parts of XML schema are connected by “key” and “keyref” constraints.

```
<xsd:complexType name=" PERSON">
   <xsd:sequence>
     <xsd:element name=" PERSON_Object"
      type="PERSON_Tuple" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name=" PERSON _Object">
   <xsd:sequence>
     <xsd:element name="SSN" type="xsd:int" />
     <xsd:element name="NAME" type="xsd:string"/>
     <xsd:element name="AGE" type="xsd:int" />
     <xsd:element name="SEX" type="xsd:string"/>
     <xsd:element name="SPOUSE" type="PERSON"/>
     <xsd:element name="NATION" type="xsd:String"/>
   </xsd:sequence>
</xsd:complexType> <xsd:complexType
  name=" COUNTRY_Class">
    <xsd:sequence>
      <xsd:element name=" COUNTRY _Tuple"
       type= "COUNTRY_Tuple"
       maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType> <xsd:complexType
  name="COUNTRY_Object">
   <xsd:sequence>
     <xsd:element name="NAME" type="xsd:string" />
     <xsd:element name="AREA" type="xsd:int" />
     <xsd:element name="POPULATION type="xsd:int"/>
   </xsd:sequence>
</xsd:complexType>
<!-- Define Primary Keys and Keyrefs -- >
<xs:key name="PERSON_PrimaryKey">
  <xs:selector xpath="db:PERSON/db:PERSON_Object"/>
  <xs:field xpath="db:SSN"/>
</xs:key>

<xsd:key name="COUNTRY_PrimaryKey">
 <xsd:selector
  path="db:COUNTRY_Class/db:COUNTRY_Object"/>
 <xs:field xpath="db:Name"/>
</xs:key>
```

```
<xs:keyref name="PERSON.nation"
 refer="db:COUNTRY_PrimaryKey">
 <xs:selector
  xpath="db:PERSON/db:PERSON_Object"/>
 <xs:field xpath="nation"/>
</xs:keyref>
```

A corresponding nested XML schema may be constructed by the same way; but “key” and “keyref” are replaced by the actual nested instead. In other words, a Person element will have a subelement “nation”, which includes details of the Country element representing the nationality of the Person element. Here it is worth mentioning that although a nested XML schema reflects better the natural structure and linkage between elements, a corresponding XML document would occupy much more space and will be more difficult to handle in order to maintain database consistency in case of a dynamic database with frequent updates.

2.3 The Object Graph

In this section, we use the information present in the *ObjectAttributes*(b) table and the inheritance information as defined in Section 2.1.1, to draw what we call the *Object Graph* (*OG*) that includes all possible relationships between the classes present in the given object-oriented schema. Nodes in *OG* are classes and representatives of tuple type domains. Two nodes are connected by a link to show the inheritance or a nesting relationship between them. Nodes and links are represented by small rectangles and directed arrows, respectively. Inheritance and nesting links are assigned the scores 0 and 1, respectively. A link is assigned the score 2 if it is connecting a node that represents a tuple domain and the class in which it is referenced. To illustrate this, refer to attribute $Takes$ in $L_{attributes}(Student)$ in Example 2.1 and to the corresponding link connecting the two nodes T_1 and $Student$ in Figure 1. More formal details related to OG are included in Definition 2.7, given next.

Definition 2.7 (Object Graph**).**
Every object-oriented schema has a corresponding OG graph (V, E) *such that,*

1. *for every class* c *in the object-oriented schema there is a corresponding node* c *in* V*,*

2. *for all classes* c_1 *and* c_2*, such that* $c_2 \in C_p(c_1)$*, an edge* $(c_1, c_2, 0)$ *is added to* E

3. *for every class* c
   for every attribute $a \in L_{attributes}(c)$*, such that* a *has a non-primitive domain,*
   if domain of a *involves a class, say* c'*, then an edge* $(c, c', 1)$ *is added to* E
   else if domain of a *involves a tuple* T_i*,* $(i{\geq}1)$ *then*
   a node T_i *is added to* V *and an edge* $(T_i, c, 2)$

```
is added to $E$
        for every class $c''$ that appears as a domain in
tuple $T_i$, an edge $(T_i, c'', 1)$
        is added to $E$                                    •
```

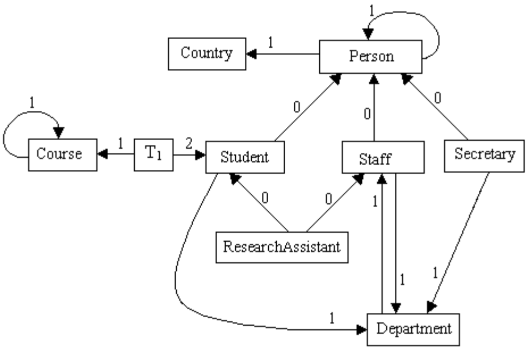

Figure 1: The Object Graph of the object-oriented schema in Example 2.1

As definition 2.7 is concerned, every node T_i in V, such that there exists an edge $(T_i, c'', 2)$ in E, corresponds in the relational graph to a node that represents a relationship that involves more that two relations, or a relationship with attributes. This will become more clear later in Section **??**, when the equivalence of the two graphs RG and OG is investigated. Shown in Figure 1 is the OG graph derived from the information present in Table 1(b) and the inheritance information in the C_p lists in Example 2.1.

3 Forward engineering: transforming object graph into XML schema

In this section, we first present the algorithm for transforming the object graph to flat XML schema(OG2FXML); then we present the algorithm for transforming the object graph to nested XML schema (OG2NXML).

3.1 Object Graph to Flat XML Schema Transformation

The OG2FXML in pseudo-code is depicted in Algorithm 3.1.

Algorithm 3.1 OG2FXML (object graph to Flat XML Conversion)

Input: The Object Graph

Output: The corresponding flat XML schema

Step:

1. Transform each node in the object graph (we call it class hereafter) into a "complexType" in the XML schema.
2. Map each attribute in a class transformed in Step (1) into a subelement within the corresponding "complexType".
3. Create a root element as the object-oriented database schema name and insert each class identified in Step (1) as a subelement with the corresponding "complexType".
4. Define the primary key for each class identified in Step (1) by using "key" element.
5. Map in the object graph each link between classes identified in Step (1) by using "keyref" element.

EndAlgorithm 3.1

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

- Each class E in the object graph is translated into an XML "complexType" of the same name E in the XML schema. In each "complexType" E, there is only one empty element, which includes several subelements. For example, COUNTRY is translated into a "complexType" named COUNTRY_Class. The empty element is called COUNTRY_Object.

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

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

- Each attribute A_i in E is mapped into a subelement of the corresponding "complexType" E. For example, COUNTRY is mapped into a "complexType" named COUNTRY_Object, inside which there are several subelements such as Name, Area and Population. They are attributes of the COUNTRY class. The XML schema for COUNTRY is:

```
<xs:complexType name="COUNTRY_Object">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" />
    <xs:element name="Area" type="xs:int" />
    <xs:element name="Population" 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: Name, Area and Population. They must appear in instance documents in the same order as they are declared here. The XML schema also provides another constructor called “all", which allows elements to appear in any order, and each element must appear once or not at all.

– Each class in the object graph is mapped into the XML schema. We first need to create a root element that represents the entire given object-oriented database. We create the root element as a “complexType" in XML schema and give it the same name as the object-oriented database schema. It then inserts each class as a subelement of the root element. An example which contains the eight classes PERSON, COUNTRY, STUDENT, STAFF, RESEARCH_ASSISTANT, COURSE, DEPARTMENT, and SECRETARY is now presented. We give the root element the name UNIVERSITY:

```
<xs:element name="UNIVERSITY">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="PERSON_Class"
       type="db:PERSON_Class" />
      <xs:element name="COUNTRY_Class"
       type="db:COUNTRY_Class" />
       <xs:element name="STUDENT_Class"
        type="db:TUDENT_Class" />
       <xs:element name="STAFF_Class"
        type="db:STAFF_Class" />
       <xs:element name="RESEARCH_ASSISTANT_Class"
        type="db:RESEARCH_ASSISTANT_Class" />
       <xs:element name="COURSE_Class"
        type="db:COURSE_Class" />
       <xs:element name="DEPARTMENT_Class"
        type="db:DEPARTMENT_Class" />
       <xs:element name="SECRETARY_Class"
        type="db:SECRETARY_Class" />
      </xs:sequence>
  </xs:complexType>
<!-- definition of keys and keyrefs -->
    . . . . . .
</xs:element>
```

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

```
<xs:key name="COUNTRY_PrimaryKey">
  <xs:selector
   xpath="db:COUNTRY_Class/db:COUNTRY_Object"/>
    <xs:field xpath="db:Name" />
</xs:key>
<xs:key name="PERSON_PrimaryKey">
  <xs:selector
   xpath="db:PERSON_Class/db:PERSON_Object"/>
  <xs:field xpath="db:nation" />
</xs:key>
<xs:keyref name="PERSON.nation"
 refer="db:COUNTRY_PrimaryKey">
  <xs:selector
   xpath="db:PERSON_Class/db:PERSON_Object"/>
  <xs:field xpath="nation" />
</xs:keyref>
```

In this example, nation is like a foreign key in PERSON, so we use “keyref" to specify the foreign key relationship between COUNTRY and PERSON. Compared to DTD, the XML schema provides a more flexible and powerful mechanism through “key" and “keyref", which share the same syntax as “unique" also make referential constraints possible in XML documents.

In general, OG2FXML is a straightforward and effective transformation algorithm, but it is only applicable when generating a flat XML structure from an object graph of an object-oriented database. As the name implies, OG2FXML cannot handle the nested features provided by XML. We remedy this problem in the OG2NXML algorithm which will be presented in the following section.

3.2 Object Graph to Nested XML Schema Transformation

In XML schema, we can use nested complex type elements to define the relationship between two elements. One advantage of the nested XML structure is to store all related information in one fragment of an XML document. This reduces the time for data retrieval when users query on the XML document. Algorithm 3.2 (OG2NXML) does the transformation from the object graph to a nested XML structure.

The OG2NXML depends on the nesting sequence specified in the object graph and generates an output of nested XML schema. The OG2NXML in pseudo-code is depicted in Algorithm 3.2.

Algorithm 3.2 OG2NXML (object graph to Nested XML Conversion)
Input: The object graph
Output: The corresponding nested XML schema
Step:

```
    For classes connected by a link labeled with 1 in the
    object graph, we nest the element that correspond to
    the class at the head of the arrow inside the element
    that correspond to the class at the tail of the arrow.

    For classes connected by a link labeled with 0 do

        Extend the element that correspond to the sub-
        class to include the content of the element that
        correspond to the superclass.
```

EndAlgorithm 3.2

To illustrate the nesting process, consider the UNIVERSITY database; it is taken as input by OG2NXML which generates as output the XML schema in a nested structure. The element of COUNTRY is nested under the element of PERSON. The nested element then included separately inside the elements of STAFF and STUDENT because PERSON is a superclass of each of the two latter classes. This way, inheritance is resolved by extending the content of the subclass to include the attributes defined in the superclasses. Handling the inheritance relationship in this way is more natural because it is not supported in XML.

3.2.1 Generating XML Documents

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

Algorithm 3.3 GenXMLDoc (Generating XML Document)
Input: XML schema and object-oriented database
Output: The corresponding XML Document
Step:

```
Create XML document and set its namespace declaration
Create a root element of the XML document with the
    same name as the root name of the XML schema
For each class $R$ in the object-oriented database do
    If R is selected and does not contain any nested
        classes
        Create $R$_Class element for $R$
        Let queryString = “select * from $R$"
        ResultSet = execute(queryString)
        For each object $T$ in ResultSet do
            Create R_Object element for object $T$
            Create an element for each attribute in $R$
                and insert it into $R$_Object element
    else if $R$ is selected and contains a nested class $R_c$
        then
        Create $R$_Class element for $R$ and $R_c$_Class
            for $R_c$
        Let queryString = “select selectedAttrs from $R$,
            $R_c$"
        ResultSet = execute(queryString)
        For each object $T$ in ResultSet do
            Create $R$_Object element for the tuple of
                $R$, and $R_c$_Object element for the object of $R_c$
            Create an element for each selected attribute in $R$ and insert it to $R$_Object
                element, and do same for $R_c$
```

EndAlgorithm 3.3

Algorithm 3.3 can generate flat XML documents as well as nested XML documents, depending on the processed XML schema. In Algorithm 3.3, a query is executed to obtain all objects that satisfy the constraints so one element is created to store data of each object in the result set.

4 Backward engineering: from XML schema to object-oriented database

4.1 XML Schema Information

Related to the nested XML schema, the analysis is based on the domain information summarized in the following table:

XMLAttributesNE(complextype, element name, domain)

To understand better the content and purpose of this table, *XMLAttributesNE* shown in Table 2 includes information about all elements and attributes in the XML schema given in Example 2.2. For each element, it is required to know its complextype name, element name, and the domain. Elements with primitive domains and elements with non-primitive domains are placed in separate occurrences of the XMLattributes table, namely *XMLElementsNE*(a) and *XMLElementsNE(b)*, respectively.

Concerning the information in Table 2(b), user involvement is required to suggest which element is inherited - representing a superclass in the object-oriented Database -, the element that can represent the nested non primitive domain attributes and the element that can represent a tuple type domain attribute. In Table 2(b), an “Inheritance Flag” is assigned to each element. The score 0 is given for the candidate superclass elements (inherited element), score 1 is given for the nested non- primitive domain elements and score 2 is assigned for the tuple domain elements. Subclasses are not included in the attributes of the class because they could be inspired by considering the superclasses list of subclasses. For instance, the “nation” element is given the value 1 for the “Inheritance Flag” because it is a nested non-primitive domain. The inheritance flag in row 3 is given the value 1 because student_in of type DEPARTMENT is a nested type, while it is given the value 0 in row 5 because PERSON is candidate superclass for STUDENT (inheritance). Also, row seven is give the value 0 as ResearchAssistant is a subclass of STUDENT and STAFF; that means STUDENT and STAFF are superclasses for ResearchAssistant. This way, it becomes trivial to identify superclasses, subclasses and non-primitive domain attributes using *XMLElementsNE*(a) and *XMLElementsNE*(b) as given in Table 2.

Related to the flat XML schema, the analysis is based on the domain information summarized in the following table: *XMLAttributesFL (complextype, element name, domain, expected domain ,inheritanceflag, keys information, key ref information)* The need for the information depicted in *XMLAttributesFL* is better understood by considering the XML schema in Example 2.2; the corresponding *XMLAttributesFL* shown in Table 3 includes information about all elements and attributes in the flat XML schema. This information is described in 3 tables named “a”, “b", and “c". For each element, it is required to know its complextype name, element name, domain, expected domain name and the inheritance status (inherited or not). Also,

Complex Type Name	Element Name	Domain
Person	ssn	integer
Person	name	string
Person	age	integer
Person	sex	integer
Country	name	string
Country	area	integer
Country	population	integer
Student	studentID	integer
Student	gpa	real
Staff	satffID	integer
Staff	salary	integer
Course	code	integer
Course	title	string
Course	credits	integer
Department	name	string
Secretery	word_minute	integer
T1	grade	string

(a)

Complex Type Name	Element Name	Domain	Inheritance Flag
Person	spouse	Person	1
Person	nation	Country	1
Student	student_in	Department	1
Student	Takes	T1	2
Student	person	Person	0
Staff	work_in	Department	1
ResearchAssistant	student	Student	0
ResearchAssistant	staff	Staff	0
Course	prerequisite	Course	1
Secretery	work_in	Department	1
T1	course	course	1

(b)

Table 2: *XMLAttributesNE:* (a) a list of all elements attributes with primitive domain (b) a list of all elements attributes with non-primitive domains

Complex Type Name	Element Name	Domain	Expected Domain	Inheritance Flag
Person	ssn	integer	integer	9
Person	name	string	string	9
Person	age	integer	integer	9
Person	sex	integer	integer	9
Person	spouse	string	Person	1
Person	nation	string	Country	1
Country	name	string	string	9
Country	area	integer	integer	9
Country	population	integer	integer	9
Student	studentID	integer	integer	9
Student	gpa	real	real	9
Student	student_in	string	Department	1
Student	Takes	T1	T1	2
Student	person	integer	Person	0
Staff	satffID	integer	integer	9
Staff	salary	integer	integer	9
Staff	work_in	integer	Department	1
ResearchAssistant	student	integer	Student	0
ResearchAssistant	staff	integer	Staff	0
Course	code	integer	integer	9
Course	title	string	string	9
Course	credits	integer	integer	9
Course	prerequisite	string	Course	1
Department	name	string	string	9
Secretery	word_minute	integer	integer	9
Secretary	work_in	string	Department	1
T1	Course	string	Course	1
T1	grade	string	string	9

(a)

Key Name	Complex Type Name	Element Name
Person_pk	Person	ssn
Country_pk	Country	name
Student_pk	Student	studentid
Staff_pk	Staff	staffid
Course_pk	Course	code
Department_pk	Department	name

(b)

Key Reference Name	Ref. Complex type	Ref. element	Refer to Element
Person.nation	Person	nation	country_pk
student.Student_in	Student	Student_in	Department_pk
Secretery.work_in	Secretery	work_in	Department_pk

(c)

Table 3: *XMLAttributesFL:* (a) a list of all elements attributes with primitive and non primitive domains (b) a list of all keys for the complex type elements (c) a list of key references of the complex type elements

it is required to know the complex type elements "keys" and “keyrefs”. Information about the elements is placed in *XMLElementsFL* Table 3(a), Keys information is placed in *XMLKeys* Table 3(b), while key reference information is replaced in *XMLKeyRef* Table 3(c) .

In Table 3(a), user involvement is required to suggest which element is inherited (a candidate superclass in the object-oriented database), the element that can represent the nested non-primitive domain attributes and the element that can represents the tuple type domain attributes. A value is assigned to “Inheritance Flag" for each element. The score 0 is given for the inherited element (candidate superclass elements), score 1 is given for the nested non-primitive domain elements, score 2 is assigned for the tuple domain elements, and score 9 is assigned to the primitive domain elements. User involvement and the information available in *XMLKeys* Table 3(b) and in *XMLKeyRef* Table 3(c) can define the expected non-primitive domain for flat XML primitive domain elements. For instance, the “nation" element is given the value 1 for the “Inheritance Flag" and an expected domain COUNTRY. This is because by analyzing and connecting the information in *XMLKeys* Table 3(b) and in *XMLKeyRef* Table 3(c), it can be shown that there is a reference link between the nation element and the country_pk in COUNTRY complex type. Row 1 is given the score 9 because it is a primitive domain element. In STUDENT complex type element, “person" element of primitive type element is given the score 0 because the user involvement can decide that this is an inherited element, and thus it could be mapped as a superclass for STUDENT. As a result, itt becomes easy to construct Table 2 from information in Table 2. Explicitly, primitive domains can be mapped into *XMLElementsNE* Table 2(a), and expected non-primitive domains can be mapped into *XMLElementsNE* Table 1(b). This way, it becomes trivial to identify superclasses, subclasses and non-primitive domain attributes using *XMLElementsNE* (a) and (b) in Table 2.

To sum up, the information needed to map into the object-oriented database is summarized in table *XMLElementsNE*. This table is derived directly derivable from a nested XML schema. However, for flat XML schema, the process involves like a preprocessing step to derive the information in table *XMLElementsFL*, which is used to construct *XMLElementsNE*. This open the door for a new relational to object-oriented database conversion by converting a relational database directly into a flat XML schema and then map the latter into object-oriented schema.

As Example 2.2 is concerned, shown in Figure 2.1 is the object graph derived from the information present in Table 2(b) and the inheritance information provided by an expert based on the content of Table 2.

4.2 Transferring Object Graph into Object-Oriented Schema

In this section, we present an algorithm for transforming the object graph to object-oriented Schema (OG2OODB).

Algorithm 4.1 OG2OODB (OG to object-oriented Schema conversion)
Input: The Object Graph
Output: The corresponding object-oriented Schema

1. Transfer each node in the object graph (we call it complextype hereafter) into a class in the object-oriented schema. Exclude nodes like T_i, $(i \geq 1)$.

2. Map each subelement of primitive type in table *XMLElementsNE*(a) into a primitive attributes in the corresponding class. Exclude subelements like T_i, $(i \geq 1)$.

3. Map each subelement of non primitive domain with score 1 defined in table *XMLElementsNE*(b) into the non primitive attributes in the corresponding class. Exclude subelements like T_i, $(i \geq 1)$.

4. Map each subelement of non-primitive domain with score 2 defined in table *XMLElementsNE*(b) as a tuple non-primitive attributes in the corresponding class. Add to this tuple non-primitive attributes all elements of complex type name equivalent to its domain.

5. Add each subelement of non-primitive domain with score 0 defined in table *XMLElementsNE*(b) into the superclasses list of the corresponding class.

EndAlgorithm 4.1

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

Each complextype E in the object graph is translated into a class of the same name E in the object-oriented schema. In each “complexType” E, there is only one empty element, which includes several subelements. Those primitive and non-primitive subelements and their domains are mapped into class attributes with the same domains. Also, superclasses of the class are added to its superclasses list. Information related to three example classes is given next; only attributes, superclasses and subclasses are shown; functions are excluded for space limitation and because they are trivial. Each attribute satisfies encapsulation by have two corresponding functions, one to set its value and one to return its value.

PERSON class can be depicted as

$PERSON_{attributes}$ = {ssn :integer; name:string; age :integer; sex :character; spouse :Person; nation Country}
$PERSON_{superclasses}$ = [ ]
$PERSON_{subclasses}$ ={Student, Staff, Secretary}

COUNTRY class can be depicted as

$COUNTRY_{attributes}$ = {name:string; area :integer; population :integer}
$COUNTRY_{superclasses}$ = [ ]
$COUNTRY_{subclasses}$ = [ ]

STUDENT class can be depicted as

$STUDENT_{attributes}$ = {StudentID:integer; gpa :real; student in :Department; Takes:{(course :Course; grade

:string)}}
$STUDENT_{superclasses}$ = [PERSON]
$STUDENT_{subclasses}$ =ResearchAssistant

5 Summary and conclusions

In this paper, we considered the mapping between object-oriented database and XML. This turns into forward and backward mappings. For the forward mapping from object-oriented into XML, we first analyze the object-oriented database to construct the object graph, which is equivalent to the class hierarchy with all inheritance and nesting links indicated. Different scores are assigned to links in both graphs in order to differentiate inheritance from nesting links. Then we developed two algorithms to produce for the object graph a corresponding flat or nested XML schema. Here, it is worth noting that the inheritance is handled differently by the two algorithms. While the former resolves the inheritance using key and keyref, the latter expands the subclass element to include the content of the superclass element; the latter is a more natural way, but the former is easier to deal with if we need to transform the XML into the relational model, which does not support nesting. Finally, we handle the mapping of the object-oriented data into XML document(s). For the backward mapping from XML into object-oriented database, we first analyze flat and nested XML Schema to construct the object graph, which is equivalent to the class hierarchy with all inheritance and nesting links indicated. Nested XML schema complex types are directly mapped to the proposed candidate classes. User involvement is required to differentiate between the nested complex types and the inheritance ones that will be mapped as superclasses. Flat XML schema depends on key and keyref data to resolve the inheritance and the nesting. User involvement is required to decide for those complex types that do not have enough information in key and keyref data. Flat XML data is mapped into the same tables used for the nested XML data, so one algorithm was sufficient to handle both types of nested and flat XML schemas.

References

[1] U. Ahmad, et al., "An Integrated Approach for Extraction of Objects from XML and Transformation to Heterogeneous Object Oriented Databases," *Proc. of ICEIS,* pp.445-449, 2003.

[2] R. Alhajj, F. Polat and C. Yilmaz, "Views as First-Class Citizens in Object-Oriented Databases," *VLDB Journal,* Vol.14, No.2, pp.155-169, 2005.

[3] R. Alhajj and A. Elnagar, "Incremental Materialization of Object-Oriented Views," *Data & Knowledge Engineering,* Vol.29, No.2, pp.121-145, Nov. 1998.

[4] R. Alhajj and F. Polat, "Reusability and Schema Evolution in an Object-Oriented Query Model," *Proc. of the ASME European Conference on Systems Design and Applications,* France, pp.21-29, Jul. 1996.

[5] R. Alhajj and F. Polat, "Proper Handling of Query Results Towards Maximizing Reusability in Object-Oriented Databases," *Information Sciences: An International Journal,* 107/1-4, pp.247-272, Jun. 1998.

[6] R. Alhajj and M.E. Arkun, "A Formal Data Model and Object Algebra for Object-Oriented Databases," *Applied Mathematics and Computer Science,* Vol. 2, No. 1, pp. 49-63, 1992.

[7] R. Alhajj and F. Polat, "Closure Maintenance in an Object-Oriented Query Model," *Proc. of the ACM International Conference on Information and Knowledge Management,* Maryland, pp.72-79, Nov. 1994.

[8] R. Alhajj and M.E. Arkun, "A Query Model for Object-Oriented Database Systems," *Proc. of the* 9^{th} *IEEE International Conference on Data Engineering,* Vienna, pp.163-172, Apr. 1993.

[9] R. Alhajj and F. Polat, "Database Reverse Engineering," *Proc. of the* 14^{th} *International Symposium on Computer and Information Sciences,* Kusadasi, Oct. 1999.

[10] R. Alhajj, "Documenting Legacy Relational Databases," *Proc. of the International Workshop on Reverse Engineering of Information Systems, in conjunction with the International Conference on Conceptual Modeling,* Lecture Notes in Computer Science, Springer-Verlag, Paris, pp.161-172, Nov. 1999.

[11] M. Andersson, "Extracting an Entity-Relationship Schema from a Relational Database through Reverse Engineering," *Proc. of the* 13^{th} *International Conference on Entity-Relationship Approach,* Manchester, pp.403-419, Dec. 1994.

[12] R.G.G. Cattel, et al., *The Object Database Standard: ODMG-93, Morgan Kaufmann,* 1994.

[13] R.G.G. Cattel, *Object Data Management: Object-Oriented and Extended Relational Database Systems, Addison-Wesley,* 1994.

[14] M. Carey, et al, "XPERATO: Publishing Object-Relational Data as XML," *Proc. of the International Workshop on Web and Databases,* May 2000.

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

[16] T.-S. Chung, S. Park, S.-Y. Han, and H.-J. Kim. "Extracting Object-Oriented Database Schemas from XML DTDs Using Inheritance," *Proc. of the International Conference on Electronic Commerce and Web Technologies,* pp.49-59, 2001.

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

[18] J. Fong, F. Pang, and C. Bloor, “Converting Relational Database into XML Document,” *Proc. of the International Workshop on Electronic Business Hubs,* pp61-65, Sep. 2001.

[19] T. Johansson and R. Heggbrenna. “Importing XML Schema into an Object-Oriented Database Mediator System,” In Uppsala Master’s Theses in Computing Science no. 260 Examensarbete DV3 20 p, 2004-01-12, ISSN 1100-1836, 2003.

[20] D. Lee, et al, “Nesting based Relational-to-XML Schema Translation,” *Proc. of the International Workshop on Web and Databases,* May 2001.

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

[22] A. Lo, R. Alhajj and K. Barker, “VIREX: Visual Relational to XML Conversion Tool,” *Visual Languages and Computing,* Vol.17, No.1, pp.25-45, 2006.

[23] T. Naser, K. Kianmehr, R. Alhajj and M. J. Ridley, “Transforming Object-Oriented Database into XML,” *Proc. of IEEE IRI,* pp.600-605, Aug. 2007.

[24] T. Naser, R. Alhajj and M. J. Ridley, “Reengineering XML into Object-Oriented Database,” *Proc. of IEEE IRI,* Jul. 2008.

[25] D. Toth and M. Valenta, “Using Object and Object-Oriented Technologies for XML-native Database Systems,” *Proc. of the Dateso Annual International Workshop on DAtabases, TExts, Specifications and Objects,* 2006.

[26] V. Turau, “Making Legacy Data Accessible for XML applications," 1999, http://www.informatik.fh-wiesbaden.de/ turau/ps/legacy.pdf.

[27] C. Wang, A. Lo, R. Alhajj, and K. Barker, “Converting Legacy Relational Database into XML Database through Reverse Engineering,” *Proc. of ICEIS,* 2004.

[28] “Extensible Markup Language (XML) 1.0 (Fourth Edition).” W3C Recommendation 16 August 2006, edited in place 29 Sept 2006