Constraint Preserving Transformation from Relational Schema to XML Schema

Chengfei Liu
Centre for Internet Computing and E-Commerce
Faculty of Information and Communication Technologies
Swinburne University of Technology
Melbourne, VIC 3122, Australia
cliu@swin.edu.au

Millist W. Vincent, Jixue Liu
Advanced Computing Research Centre
School of Computer and Information Science
University of South Australia
Adelaide, SA 5095, Australia
{millist.vincent, jixue.liu}@unisa.edu.au

Abstract. XML has become the standard for publishing and exchanging data on the Web. However, most business data is managed and will remain to be managed by relational database management systems. As such, there is an increasing need to efficiently and accurately publish relational data as XML documents for Internet-based applications. One way to publish relational data is to provide virtual XML documents for relational data via an XML schema which is transformed from the underlying relational database schema such that users can access the relational database through the XML schema. In this paper, we discuss issues in transforming a relational database schema into the corresponding XML schema. We aim to preserve all integrity constraints defined in a relational database schema, to achieve high level of nesting and to avoid introducing data redundancy in the transformed XML schema. In the paper, we first propose a basic transformation algorithm which introduces no data redundancy, then we improve the algorithm by exploring further nesting of the transformed XML schema.

Keywords: Schema Transformation, XML, XML Schema, Relational Databases

1. Introduction

XML [4, 1] has become the standard format for publishing and exchanging data on the Web. However, most business data is still stored and maintained in relational database management systems (RDBMSs). In fact, RDBMSs will remain dominant in managing business data in the foreseeable future because of their powerful data management services. Given that relational databases are proprietary and only accessible within enterprises while XML documents are designed for accessing and interchanging over the Internet, there is an increasing need to

efficiently and accurately publish relational data as XML documents for Internet-based applications.

One approach to publish relational data is to create XML views of the underlying relational data. Through the XML views, users may access the relational databases as though they were accessing XML documents. Once XML views are created over a relational database, queries in an XML query language like XML-QL [6] or XQuery [3] can be issued against these XML views for the purpose of accessing relational databases. SilkRoute [8] is one of the systems taking this approach. In SilkRoute, XML views of a relational database are defined using a relational to XML transformation language called RXL, and then XML queries are issued against these views. The XML queries and views are combined together by a query composer and the combined RXL queries are then translated into the corresponding SQL queries. XPERANTO [14, 5, 13] takes a similar approach. One problem in the SilkRoute and XPERANTO approaches is that users cannot see the integrity constraints buried in the relational schema from the XML views defined. It is important for users to be aware of the constraints in the XML schema against which they are going to issue queries.

Another approach [12] to publish relational data is to provide virtual XML documents for relational data via an XML schema which is transformed from the underlying relational database schema such that users can access the relational database through the XML schema. In this approach, there is a need to generate an integrated XML schema from the underlying relational database schema, which is the topic of this paper. It is also highly desirable that the generated XML schema preserves all integrity constraints that are defined in the underlying relational database schema. We aim to achieve this, which makes a significant distinction compared with the view approach taken by SilkRoute and XPERANTO.

Currently, there are two options recommended by the W3C for defining an XML schema. One is the Document Type Definition (DTD) [4, 1] and the other is the XML Schema [7]. We choose XML Schema because DTD has a number of limitations.

XML Schema offers great flexibility in modeling documents. Therefore, there exist many ways to map a relational database schema into a schema in XML Schema. For examples, In DB2XML [15], an algorithm is used to map relations to XML elements in almost one-to-one manner. Based on a flat translation similar to DB2XML, NeT [11] derives nested structures from flat relations by repeatedly applying the *nest* operator on tuples of each relation. XViews [2] constructs a graph based on primary key/foreign key relationship and generates candidate views by

choosing the node with either maximum in-degree or zero in-degree as the root element.

In this paper, we discuss issues in transforming a relational database schema into the corresponding schema in XML Schema. We aim to achieve the level of nesting of the transformed XML schema as high as XViews and NeT. In addition, we aim to guarantee that the transformed XML schema preserves all the integrity constraints defined in the relational database schema and is highly normalized with no redundancy introduced.

The rest of the paper is organized as follows. In Section 2, we give a brief introduction to XML Schema, especially the features which will be used in the schema transformation. In Section 3, we present the mapping rules of a basic transformation algorithm which converts a relational schema together with integrity constraints to the corresponding schema in XML Schema without introducing redundancy. The improvement of the basic algorithm is discussed in Section 4 with more nested structure explored. Section 5 discusses the related work and Section 6 concludes the paper.

2. XML Schema

XML Schema [7] is the W3C XML language for describing and constraining the content of XML documents. Compared with DTD, it offers many appealing features.

- XML Schema provides very powerful data typing. A rich set of built-in data types are provided. Based on that, users are allowed to derive their own simple types by restriction and complex types by both restriction and extension. In DTD, only a very limited number of built-in types are provided, most for defining attributes only. User cannot define their own types, not to mention complex types.

- XML Schema provides comprehensive support for representing integrity constraints such as id/idref, key/keyref, unique, fine grained cardinalities, etc. while DTD only provides limited support such as id/idref. The cardinality constraints provided by DTD is mainly based on Kleine closure.

- Apart from the sequence and selection compositors for grouping elements, XML Schema also supports other compositors such as set.

- XML Schema has the same syntax as XML. This allows the schema itself be processed by the same tools that read the XML documents it describes. In contrast, DTD is in a non-XML syntax.

- Namespaces are well supported in XML Schema but not in DTD.

While DTD is still used for very simple applications, XML Schema is becoming a dominant XML schema language.

The following example illustrates the main features of XML Schema. The URI "http://www.w3.org/2001/XMLSchema" identifies the namespace *xsd* where the XML Schema vocabulary recommended by W3C is defined. The URI "http://www.swin.edu.au/CompanyML" identifies the target namespace for the schema to be defined. For each schema, only one root element is allowed to be declared. In the example, the root element called *Company_XML* is declared under which there are four subelements: *Employee, Dept, Project* and *WorksOn.*

```
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.swin.edu.au/CompanyML"
  xmlns="http://www.swin.edu.au/CompanyML"
  elementFormDefault="qualified">
 <xsd:element name="Company_XML">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Employee" minOccurs="0"
        maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:attribute name="eno" type="xsd:id" use="required"/>
      <xsd:attribute name="dno" type="xsd:idref" use="required"/>
      <xsd:attribute name="supEno" type="xsd:idref" use="optional"/>
      <xsd:attribute name="name" type="xsd:string" use="required/>
      <xsd:attribute name="city" type="xsd:string" use="optional"/>
      <xsd:attribute name="salary" type="xsd:int" use="optional"/>
     </xsd:complexType>
    </xsd:element>
    <xsd:element name="Dept" minOccurs="0"
        maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="DeptLoc" minOccurs="0"
           maxOccurs="unbounded">
        <xsd:complexType>
```

```
      <xsd:attribute name="city" type="xsd:string" use="required"/>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="dno" type="xsd:id" use="required"/>
   <xsd:attribute name="mgrEno" type="xsd:idref" use="optional"/>
   <xsd:element name="dname" type="xsd:string" use="required"/>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="Project" minOccurs="0"
     maxOccurs="unbounded">
  <xsd:complexType>
   <xsd:attribute name="pno" type="xsd:id" use="required"/>
   <xsd:attribute name="dno" type="xsd:idref" use="required"/>
   <xsd:attribute name="pname" type="xsd:string" use="required"/>
   <xsd:attribute name="city" type="xsd:string" use="optional"/>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="WorksOn" minOccurs="0"
     maxOccurs="unbounded">
  <xsd:complexType>
   <xsd:attribute name="eno" type="xsd:idref" use="required"/>
   <xsd:attribute name="pno" type="xsd:idref" use="required"/>
   <xsd:attribute name="hours" type="xsd:int" use="optional"/>
  </xsd:complexType>
 </xsd:element>
 </xsd:sequence>
 </xsd:complexType>
</xsd:element>
<xsd:key name="PK_DeptLoc">
 <xsd:selector xpath="//DeptLoc"/>
 <xsd:field xpath="../@dno"/>
 <xsd:field xpath="@city"/>
</xsd:key>
<xsd:unique name="UNIQUE_mgrEno">
 <xsd:selector xpath="//Dept"/>
 <xsd:field xpath="@mgrEno"/>
</xsd:unique>
<xsd:key name="PK_WorksOn">
 <xsd:selector xpath="//WorksOn"/>
 <xsd:field xpath="@eno"/>
 <xsd:field xpath="@pno"/>
</xsd:key>
```

```
</xsd:schema>
```

Each of the four elements under *Company_XML* allows 0 or multiple occurrences, which are specified by the *minOccurs* and the *maxOccurs* cardinality constraints of the element. Specific number can be indicated here for *minOccurs* and *maxOccurs* if required. The default values for both *minOccurs* and *maxOccurs* is 1. The cardinality constraint for attributes is specified using the *use* attribute of the *xsd:attribute* element. As each attribute can take at most one value at a time, the *required* and *optional* values are used for compulsory and optional attributes, respectively.

The *Employee* element has three compulsory attributes *eno*, *dno* and *name*, and three optional attributes *supEno*, *city* and *salary*. *eno* serves as the identity of the instances of *Employee*, *dno* is intended to reference an instance of element *Dept*, and *supEno* is intended to reference an instance of element *Employee*. The *Dept* element has two compulsory attributes *dno* and and *dname*, one optional attribute *mgrEno*, and one element *DeptLoc*. *dno* serves as the identity of the instances of *Dept* while *mgrEno* is intended to reference an instance of *Employee*. *mgrEno* is unique for *Dept* elements. The *DeptLoc* element has one compulsory attribute *city*, this attribute together with the attribute *dno* of its parent element serve as the identity of the instances of *DeptLoc* by using a *key* element definition. XML Schema supports two mechanisms to represent identity and reference: one is *id/idref* which is also supported in DTD, the other is *key/keyref* which is not supported by DTD. *id* and *idref* only apply to a single element/attribute while *key* and *keyref* can apply to multiple elements/attributes. The *Project* element has three compulsory attributes *pno*, *dno* and *pname*, and one optional attribute *city*. *pno* serves as the identity of the instances of *Project* while *dno* is intended to reference an instance of *Dept*. The *WorksOn* element has two compulsory attributes *eno* and *pno*, and one optional attribute *hours*. *eno* and *pno* together serve as the identity of the instances of *WorksOn* using *key* element definition. Individually, *eno* and *pno* is intended to reference an instance of *Employee* and *Project*, respectively.

3. Schema Transformation

In a relational database schema, different types of integrity constraints may be defined. In SQL, the system supported integrity constraints include primary keys (PKs), foreign keys (FKs), null/not-null, and unique. It is important to map all these constraints to the target XML

schema. Also we aim to achieve a high level of nesting and to avoid introducing redundancy in the target schema.

As previously discussed, XML Schema supports two mechanisms to represent identity and reference: *id/idref* and *key/keyref*. There are differences in using these two mechanisms. The former supports the dereference function in path expressions in most XML query languages including XQuery. This is important for navigational representation of queries. However, it only applies to a single element/attributes. It also has a problem in precisely representing a reference. No restriction is given to protect an *idref* element/attribute from referencing an unexpected element. The latter may apply to multiple elements/attributes but cannot support the dereference function. For schema translation, we leave the choice of these two mechanisms to users. For multi-attribute primary/foreign keys, however, only *key/keyref* can be used. For this purpose, we will differentiate the single attribute primary/foreign keys from multi-attribute primary/foreign keys while transforming the relational database schema to XML schema. We also classify a relation into four categories based on different types of primary keys.

DEFINITION 3.1. **regular relation**
A regular relation is a relation where the primary key contains no foreign keys.

DEFINITION 3.2. **component relation**
A component relation is a relation where the primary key contains one foreign key. This foreign key references another relation which we call the parent relation of the component relation. The other part of the primary key serves as a local identifier under the parent relation.

The component relation is used to represent a component or a multi-valued attribute of its parent relation.

DEFINITION 3.3. **supplementary relation**
A supplementary relation is a relation where the whole primary key is also a foreign key which references another relation.

The supplementary relation is used to supplement another relation or to represent a subclass for transforming a generalization hierarchy from a conceptual schema.

DEFINITION 3.4. **association relation**
An association relation is a relation where the primary key contains more than one foreign key, each of which references a participant relation of the association.

3.1. Basic Mapping Rules and Algorithm

Based on the above discussion and definitions, we first give a set of mapping rules, then an algorithm based on this set of rules.

RULE 1. *For a relational database schema* Sch, *a root element named* Sch_XML *is created in the corresponding XML schema as follows.*

```
<?xml version="1.0"?>
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="targetNamespaceURI"
   xmlns="targetNamespaceURI"
   elementFormDefault="qualified">
  <xsd:element name="Sch_XML">
    <xsd:complexType>
      <xsd:sequence>
        <!-- transformed relation schema of Sch -->
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

RULE 2. *For each regular or association relation R, the following element with the same name as the relation schema is created and then put under the root element.*

```
<xsd:element name="R" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <!-- the attributes of R -->
  </xsd:complexType>
</xsd:element>
```

RULE 3. *For each component relation R_1, let its parent relation be R_2. Then an element similar to Rule 2 and with the same name as the component relation is created and then placed as a child element of R_2.*

RULE 4. *For each supplementary relation R_1, let the relation which R_1 references be R_2. Then the following element with the same name as the supplementary relation schema is created and then placed as a child element of R_2. Notice, there is a difference between the transformed element of a component relation and the transformed element of a supplementary relation on* maxOccurs.

```
<xsd:element name="R1" minOccurs="0" maxOccurs="1">
  <xsd:complexType>
    <!-- the attributes of R1 -->
  </xsd:complexType>
</xsd:element>
```

RULE 5. *For each multiple attribute primary key PK of a regular or an association relation R, suppose the key attributes are $PKA_1, \cdots, PKA_n$, an attribute of the element for R is created for each $PKA_i (1 \leq i \leq n)$ with the corresponding data type. After that a* key *element is defined with a* selector *to select the element for R and several* fields *to identify $PKA_1, \cdots, PKA_n$. The name of the element PK should be unique within the namespace.*

```
<xsd:element name="R" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:attribute name="PKA1" type="xsd:PKA1_type" use="required"/>
    ... ...
    <xsd:attribute name="PKAn" type="xsd:PKAn_type" use="required"/>
  </xsd:complexType>
</xsd:element>
<xsd:key name="PK">
  <xsd:selector xpath="//R"/>
  <xsd:field xpath="@PKA1"/>
  ... ...
  <xsd:field xpath="@PKAn"/>
</xsd:key>
```

RULE 6. *For each single attribute primary key with the name PKA of regular relation R, two options can be taken. The first option is to use the* xsd:key *element as in Rule 5. The second option is to use the* xsd:id *type for creating an attribute of the element for R as follows.*

```
<xsd:element name="R" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:attribute name="PKA" type="xsd:id" use="required"/>
  </xsd:complexType>
</xsd:element>
```

RULE 7. *For each multiple attribute primary key PK of a component relation R_1, let its parent relation be R_2, and the key attributes are $GKA_1, \cdots, GKA_m, LKA_1, \cdots, LKA_n$ where $GKA_1, \cdots, GKA_m$ is a foreign key referencing the parent relation R_2. Then an attribute of*

the element for R_1 is created for each $LKA_i(1 \leq i \leq n)$ with the corresponding data type. After that a key *element is defined with a* selector *to select the element for R_1 under the element R_2, several* fields *to identify $GKA_1, \cdots, GKA_m$ belonging to its parent element and several* fields *to identify $LKA_1, \cdots, LKA_n$. The name of the element PK should be unique within the namespace.*

```
<xsd:element name="R1" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:attribute name="LKA1" type="xsd:LKA1_type" use="required"/>
    ... ...
    <xsd:attribute name="LKAn" type="xsd:LKAn_type" use="required"/>
  </xsd:complexType>
</xsd:element>
<xsd:key name="PK">
  <xsd:selector xpath="//R1"/>
  <xsd:field xpath="../@GKA1"/>
  ... ...
  <xsd:field xpath="../@GKAm"/>
  <xsd:field xpath="@LKA1"/>
  ... ...
  <xsd:field xpath="@LKAn"/>
</xsd:key>
```

RULE 8. *Ignore the mapping for the primary key of each supplementary relation.*

RULE 9. *For each multiple attribute foreign key FK of a relation R, except one which is contained in the primary key of a component or supplementary relation, suppose FK references PK of the referenced relation, and the foreign key attributes are $FKA_1, \cdots, FKA_n$, an attribute of the element for R is created for each $FKA_i(1 \leq i \leq n)$ with the corresponding data type if FKA_i is not part of any primary key. Then a* keyref *element is defined with a* selector *to select the element for R and several* fields *to identify $FKA_1, \cdots, FKA_n$. The name of the element FK should be unique within the namespace and* refer *of the element is the name of the key element of the primary key which it references.*

```
<xsd:element name="R" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:attribute name="FKA1" type="xsd:FKA1_type"/>
    ... ...
```

```
    <xsd:attribute name="FKAn" type="xsd:FKAn_type"/>
  </xsd:complexType>
</xsd:element>
<xsd:keyref name="FK" refer="PK">
  <xsd:selector xpath="//R"/>
  <xsd:field xpath="@FKA1"/>
  ... ...
  <xsd:field xpath="@FKAn"/>
</xsd:keyref>
```

RULE 10. *For each single attribute foreign key FKA of a relation R, except one which is contained in the primary key of a component or supplementary relation, two options can be taken. The first option is to use the* xsd:keyref *element as in Rule 9. The second option is to use the* xsd:idref *type for creating an attribute of the element for R as follows.*

```
<xsd:element name="R" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:attribute name="FKA" type="xsd:idref"/>
  </xsd:complexType>
</xsd:element>
```

RULE 11. *For each non-key attribute of a relation R, an attribute with corresponding name and data type is created for the element of R.*

RULE 12. *For each attribute with a not-null constraint, add* use = "required" *to the attribute declaration. For all other attributes without a* use *attribute, add* use = "required" *to the attribute declaration.*

RULE 13. *For each* unique *constraint defined on attributes $UA_1, \cdots, UA_n$ of a relation R, a* unique *element is defined with a* selector *to select the element for R and several* fields *to identify $UA_1, \cdots, UA_n$. The name of the* unique *element should be unique within the namespace.*

```
<xsd:unique name="UniqueName">
  <xsd:selector xpath="//R"/>
  <xsd:field xpath="@UA1"/>
  ... ...
  <xsd:field xpath="@UAn"/>
</xsd:unique>
```

Based on the above mapping rules, it is easy to have the following transformation algorithm.

ALGORITHM 1. **Basic Schema Transformation**
Input: A relational database schema *Sch* with constraints and an option to use id/idref or key/keyref.
Output: A corresponding XML schema *Sch_XML* which preserves the constraints and is redundancy free.
Step 1: create root element *Sch_XML* for the relational database schema *Sch* by applying Rule 1.
Step 2: Get next relation schema R, return *Sch_XML* until there is no relation schema left.
Step 3: If R is for a regular or an association relation create an element by applying Rule 2.
Step 4: If R is for a component relation create an element by applying Rule 3.
Step 5: If R is for a supplementary relation create an element by applying Rule 4.
Step 6: If R is for a regular relation and the primary key of the relation contains a single attribute, map the primary key by applying Rule 5 for key/keyref option or by applying Rule 6 for id/idref option.
Step 7: If R is for a regular/association relation and the primary key of the relation contains multiple attributes, map the primary key by applying Rule 5.
Step 8: If R is for a component relation, map the primary key by applying Rule 7.
Step 9: If R is for a supplementary relation, map the primary key by applying Rule 8.
Step 10: For each foreign key in R, if it contains a single attribute, map the foreign key by applying Rule 9 for key/keyref option or applying Rule 10 for id/idref option; otherwise map the foreign key by applying Rule 9.
Step 11: For each non-key attribute, use Rule 11 to map it.
Step 12: For each not-null constraint, use Rule 12 to map it.
Step 13: For each unique constraint, use Rule 13 to map it.
Step 14: Goto Step 2.

3.2. AN EXAMPLE

We use the following relational database schema *Company* to illustrate the above algorithm. In the schema, primary keys are <u>underlined</u> while foreign keys are in *italic* font. The /U after an attribute or a set of attributes stands for a *unique* constraint on the attribute or the set of attributes while the /N after an attribute stands for a *not-null* constraint on the attribute.

Employee(<u>eno</u>, name/N, city, salary, *dno*/N, *supEno*)
Dept(<u>dno</u>, dname/N, *mgrEno*/U)
DeptLoc(<u>*dno*, city</u>)
Project(<u>pno</u>, pname/N, city, *dno*/N)
WorksOn(<u>*eno*, *pno*</u>, hours)

If the above schema is given as an input to the basic schema transformation algorithm, the schema in XML Schema *Company_XML* shown in Section 2 will be generated. All the constraints defined on the relational schema *Company* are preserved in the XML schema *Company_XML*. The id/idref option is used in the transformation.

3.3. DISCUSSION

As XML allows nested structure, redundancy may be brought in when transforming a flat relation structure to a nested XML structure. For example, if we put element Dept under element Project, the same department will be repeated in all projects in the department. However, if we put elements Dept and Project at the same level or put the element Project under the element Dept, there is no data redundancy introduced.

Rule 1 to Rule 13 used in the basic algorithm are relatively straightforward for mapping a relational database schema to the corresponding XML schema. One property of the basic algorithm is redundancy free preservation, i.e., Rule 1 to Rule 13 do not introduce any data redundancy provided the relational schema is redundancy free.

THEOREM 3.1. *If the relational database schema* Sch *is redundancy free, the XML schema* Sch_XML *generated by the basic transformation algorithm is also redundancy free.*

This theorem is easy to prove. For a regular or an association relation R, an element with the same name R is created under the root element, so the relation R in *Sch* is isomorphically transformed to an element in *Sch_XML*. For a component relation R, a sub-element with the same name R is created under its parent R_p. Because of the foreign key constraint, we have the functional dependency $PK_R \rightarrow PK_{R_p}$, i.e., there is a many to one relationship from R to R_p, therefore it is impossible that a tuple of R is placed more than one time under different element of R_p. Similar to a component relation, there is no redundancy introduced for a supplementary relation. $\square$

4. Exploring Nested Structures

As we can see, the basic transformation algorithm introduced above fails to explore all possible nested structures. For example, the *Project* element can be moved to be under the *Dept* element if every project belongs to a department. Nesting is important in XML schema because it allows navigation of path expressions to be processed efficiently; otherwise, we have to use either *idref* or *keyref*. If we use *idref*, we may use system supported dereference function to get the referenced elements. In XML, the dereference function is expensive because *id* and *idref* types are value based. If we use *keyref*, we have to put an explicit *join* condition in an XML query to get the referenced elements. Therefore, we need to explore all possible nested structure by further investigating the referential integrity constraints in the relational schema. For this purpose, we introduce a reference graph. In the reference graph, we also include the *unique* and *not-null* constraints defined together with a foreign key constraint.

DEFINITION 4.1. *Given a relational database schema $Sch = \{R_1, \cdots, R_n\}$, a* reference graph *of the schema Sch is defined as a labeled directed graph $RG = (V, E, L)$ where V is a finite set of nodes $v_1, \cdots, v_n$ representing relation schema $R_1, \cdots, R_n$ in Sch, respectively; E is a finite set of arcs, if there is a foreign key defined in R_i which references R_j, an arc $e = <v_i, v_j> \in E$; L is a set of labels for edges by applying a labeling function from E to the set of foreign keys denoted by the foreign key attributes together with unique/not-null constraints.*
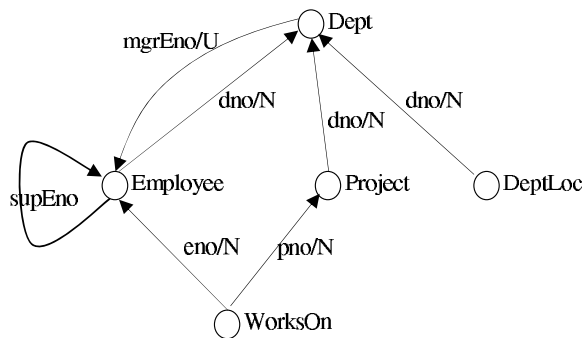

Figure 1. A Reference Graph

The reference graph of the relational schema *Company* is shown as in Figure 1. In the graph, the element of node *DeptLoc* has been put under the element of node *Dept* by Rule 3. ¿From the graph, we may have the following improvements if certain conditions are satisfied.
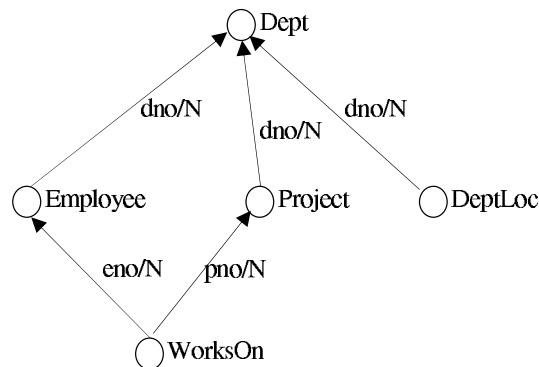
Figure 2. The Modified Reference Graph

(1) The element of node *Project* could be put under the element of node *Dept* if the foreign key *dno* is defined as *not-null*. This is because that node *Project* only references node *Dept* and a many to one relationship from *Project* to *Dept* can be derived from the foreign key constraint. In addition, the *not-null* foreign key means every project has to belong one department. As a result, one project can be put under one department and cannot be put twice under different departments in the XML document.

(2) A loop exists between *Employee* and *Dept.* In general, what we can get from this is a many to many relationship between *Employee* and *Dept.* However, the foreign key *mgrEno* of Dept reflects a one to one relationship from *Dept* to *Employee.* This semantics can be captured by checking the *unique* constraint defined for the foreign key *mgrno.* If there is such a unique constraint defined, the foreign key *mgrEno* of Dept really suggests a one to one relationship from *Dept* to *Employee.* For the purpose of nesting, we delete the arc from Dept to Employee labelled *mgrno* from the reference graph. The real relationship from *Employee* to *Dept* is many to one. As such, the element of the node *Employee* can also be put under the element of the node *Dept* if the foreign key *dno* is defined to *not-null.* The foreign key *supEno* represents a many to one reflexive relationship. It has been best represented as a foreign key in the element for *Employee*, so we can delete this kind of arc as well. The resulting reference graph is shown in Figure 2.

(3) The node *WorksOn* references two nodes *Employee* and *Project.* The element of *WorksOn* can be put under either *Employee* and *Project* if the corresponding foreign key is *not-null.* However, which node to choose to put under all depends on which path will be used often in queries.

Obviously the basic algorithm can be improved to allow more nested structures. To achieve this, we generate a reference graph for a relational database schema and simplify it by checking whether some loops can be removed. Then we explore maximum nesting by the following theorems.

THEOREM 4.1. *In a reference graph $RG(V, E, L)$, let $v_1, v_2 \in V$ denote relations R_1 and R_2, respectively. If the out-degree of v_1 is 1 and $< v_1, v_2 >\in E$ and* not-null *is associated with the label of $< v_1, v_2 >$ and there is no loop between v_1 and v_2, then we can move the element for R_1 to be under the element for R_2 without introducing data redundancy.*

The proof of this theorem has already explained by the relationships between *Project* and *Dept*, and between *Dept* and *Employee* in Figure 1. The fact that the only arc from v_1 to v_2 and no loop between the two nodes represents a many to one relationship from R_1 to R_2, while the *not-null* foreign key gives a many to exact one relationship from R_1 to R_2. Therefore, for each instance of R_1, it is put only once under exactly one instance of R_2, no redundancy will be introduced. □

Similarly, we have the following.

THEOREM 4.2. *In a reference graph $RG(V, E, L)$, let $v_0, v_1, \cdots, v_k \in V$ denote relations $R_0, R_1, \cdots, R_k$, respectively. If $< v_0, v_1 >, \cdots, < v_0, v_k >\in E$ and* not-null *is associated with the label of at least one of these arcs, say, $< v_0, v_l >$ and there is no loop between v_0 and any of $v_1, \cdots, v_k$, then we can move the element for R_0 to be under the element for R_l without introducing data redundancy.*

¿From Theorem 4.1 and Theorem 4.2, we have the following rules.

RULE 14. *If there is only one many to one relationship from relation R_1 to another relation R_2 and the foreign key of R_1 to R_2 is defined as* not-null, *then we can move the element for R_1 to be under the element for R_2 as a child element.*

RULE 15. *If there are more than one many to one relationship from relation R_0 to other relations $R_1, \cdots, R_k$, then we can move the element for R_0 to be under the element for R_i ($1 \le i \le k$) as a child element provided the foreign key of R_0 to R_k is defined as* not-null.

By many to one relationship from relation R_1 to R_2, we mean that there exists at least one arc from node v_1 for R_1 to node v_2 for R_2, and there is no loop between v_1 and v_2 in the reference graph.

If we apply Rule 14 to the transformed XML schema *Company_XML*, the elements for *Project* and *Employee* will be moved to be under *Dept*, consequently, the attribute *dno* with *idref* type will be removed from both *Project* and *Employee* elements. Furthermore, if we apply Rule 15 and choose to put *WorksOn* be under *Employee*, the element for *WorksOn* will be moved to be under the element for *Employee*, consequently, the attribute *eno* with *idref* type will be removed from the *WorksOn* element. The primary key for *WorksOn* will also be changed with *eno* refers to the *eno* of its parent element *Employee*. The improved XML schema is given below.

```xml
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.swin.edu.au/CompanyML"
  xmlns="http://www.swin.edu.au/CompanyML"
  elementFormDefault="qualified">
 <xsd:element name="Company_XML">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Dept" minOccurs="0"
        maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="DeptLoc" minOccurs="0"
           maxOccurs="unbounded">
        <xsd:complexType>
         <xsd:attribute name="city" type="xsd:string" use="required"/>
        </xsd:complexType>
       </xsd:element>
       <xsd:element name="Project" minOccurs="0"
           maxOccurs="unbounded">
        <xsd:complexType>
         <xsd:attribute name="pno" type="xsd:id" use="required"/>
         <xsd:attribute name="pname" type="xsd:string" use="required"/>
         <xsd:attribute name="city" type="xsd:string" use="optional"/>
        </xsd:complexType>
       </xsd:element>
       <xsd:element name="Employee" minOccurs="0"
           maxOccurs="unbounded">
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element name="WorksOn" minOccurs="0"
```

```
                    maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="pno" type="xsd:idref" use="required"/>
            <xsd:attribute name="hours" type="xsd:int" use="optional"/>
          </xsd:complexType>
         </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="eno" type="xsd:id" use="required"/>
        <xsd:attribute name="supEno" type="xsd:idref" use="optional"/>
        <xsd:attribute name="name" type="xsd:string" use="required/>
        <xsd:attribute name="city" type="xsd:string" use="optional"/>
        <xsd:attribute name="salary" type="xsd:int" use="optional"/>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
     <xsd:attribute name="dno" type="xsd:id" use="required"/>
     <xsd:attribute name="mgrEno" type="xsd:idref" use="optional"/>
     <xsd:element name="dname" type="xsd:string" use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
<xsd:key name="PK_DeptLoc">
 <xsd:selector xpath="//DeptLoc"/>
 <xsd:field xpath="../@dno"/>
 <xsd:field xpath="@city"/>
</xsd:key>
<xsd:unique name="UNIQUE_mgrEno">
 <xsd:selector xpath="//Dept"/>
 <xsd:field xpath="@mgrEno"/>
</xsd:unique>
<xsd:key name="PK_WorksOn">
 <xsd:selector xpath="//WorksOn"/>
 <xsd:field xpath="../@eno"/>
 <xsd:field xpath="@pno"/>
</xsd:key>
</xsd:schema>
```

¿From the above improved XML schema *Company_XML*, we can see that all nested structures have been explored.

Theorem 4.2 also allows that R_i is the same as R_j for $1 \le i < j \le k$. For example, the relation Supervision(*supervisorEno, superviseeEno*)
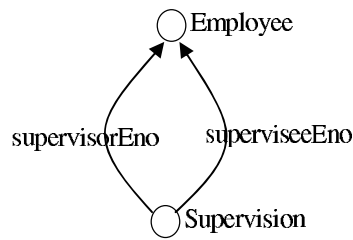
Figure 3. A Reference Graph for m:n Reflexive Relationship

stands for a many to many reflexive relationship between employees, i.e., an employee may supervise many supervisees and an employee may be supervised by many supervisors. Its reference graph is shown in Figure 3. Obviously, the element for *Supervision* can be moved to be under the element for *Employee*. Either *supervisorEno* or *supervisorEno* may be chosen as *idref* attribute under the element for *Supervision*. The XML schema for the relation schema *Supervision* is given below. Here we choose to use the arc with the label *supervisorEno*.

```
<xsd:element name="Employee" minOccurs="0"
    maxOccurs="unbounded">
 <xsd:complexType>
  <xsd:sequence>
   ... ...
   <xsd:element name="Supervision" minOccurs="0"
       maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:attribute name="supervisorEno" type="xsd:idref"
         use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="eno" type="xsd:id" use="required"/>
 </xsd:complexType>
</xsd:element>
<xsd:key name="PK_Supervision"/>
  <xsd:selector xpath="//Supervision"/>
  <xsd:field xpath="../@eno"/>
  <xsd:field xpath="@supervisorEno"/>
</xsd:key>
```

5. Related Work

SilkRoute [8] and XPERANTO [14, 5, 13] choose to publish relational data by creating XML views of the underlying relational data. The advantage of this approach is the data independence achieved through the created views. However, users cannot see the integrity constraints buried in the underlying relational schema from those XML views. This may cause difficulty to write XML queries precisely. The approach taken in this paper can solve this problem by preserving integrity constraints defined in the relational schema in the transformed XML schema.

An early work in transforming relational schema to XML schema is DB2XML [15]. DB2XML uses a simple algorithm to map flat relational model to flat XML model in almost one-to-one manner. DTD is used for the target XML schema.

Based on a flat translation similar to DB2XML, NeT [11] derives nested structures from flat relations by repeatedly applying the *nest* operator on tuples of each relation. A problem in this approach is that the derivation process is solely based on values with no consideration of the semantics of the schema. As such, the resulting nested structures may not be useful at all. NeT also choose DTD for target schema, therefore, does not consider the transformation of integrity constraints.

XViews [2] constructs a graph based on primary key/foreign key relationship and generates candidate views by choosing the node with either maximum in-degree or zero in-degree as the root element. The candidate XML views generated maybe highly nested. DTD is also chosen for target XML schema. Similarly, this approach does not consider the preservation of integrity constraints. It also suffers considerable level of data redundancy.

Compared with DB2XML, NeT and XViews, we use XML Schema as the schema language for target schema. This allows us to take integrity constraints into account and preserves them in the transformed XML schema. Similar to NeT and XViews, we explore high level of nested structures as well. However, our derivation approach captures semantics that are buried in the relational schema and maps them accurately to the target XML schema. In NeT and XViews, semantical information such as integrity constraints are not used to guide the derivation of nested structures. As such, redundancy is introduced in XViews and unexpected nested structures may be obtained in NeT.

For integrating XML and relational databases, Kappel et al. [10] give a comprehensive comparison of the concepts and corresponding mapping patterns between XML and relational databases. In [10] and their X-Ray approach [9], three basic kinds of mappings $ET\_R_{direct/indirect}$

(an XML element to a relation), $ET\_A_{direct/indirect}$ (an XML element to an attribute of a relation) and $A\_A_{direct/indirect}$ (an XML attribute to an attribute of a relation) have been proposed and reasonable mappings of these three basic mapping patterns from DTD to relational schema have been discussed.

6. Conclusion

This paper addressed the issues in mapping relational database schema to XML schema. To generate a high quality XML schema from a relational schema, we believe that a schema transformation algorithm should provide the following features:

— preserving integrity constraints of the underlying relational database schema.

— avoiding introducing data redundancy.

— exploring all possible nested structures.

The schema transformation algorithm presented in this paper provides all three features.

We believe that the proposed algorithm is effective and practical. In the future, we will investigate how an XML schema can be generated from a view of a relational database.

Acknowledgements

References

1. Abiteboul, S., P. Buneman, and D. Suciu: 2000, *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers.
2. Baru, C.: 1999, 'XViews: XML Views of Relational Schemas'. In: *Proceedings of DEXA Workshop*. pp. 700–705.
3. Boag, S., D. C. amd M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu: 2002, 'XQuery 1.0: An XML Query Language'. W3C Working Draft, http://www.w3.org/TR/2002/WD-xquery-20020430/.
4. Bray, T., J. Paoli, C. Sperberg-McQueen, and E. Maler: 2000, 'Extensible Markup Language (XML) 1.0 (Second Edition)'. W3C Recommendation, http://www.w3.org/TR/REC-xml.

5. Carey, M., J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian: 2000, 'XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents'. In: *Proceedings of VLDB*. pp. 646–648.

6. Deutsch, A., M. Fernandez, D. Florescu, A. Levy, and D. Suciu: 1998, 'XML-QL: A Query Language for XML'. Submission to W3C, http://www.w3.org/TR/NOTE-xml-ql/.

7. Fallside, D.: 2001, 'XML Schema Part 0: Primer'. W3C Recommendation, http://www.w3.org/TR/xmlschema-0/.

8. Fernandez, M., Y. Kadiyska, D. Suciu, A. Morishima, and W. Tan: 2002, 'SilkRoute: A Framework for Publishing Relational Data in XML'. *ACM Trans. Database Syst.* **27**(4), 438–493.

9. Kappel, E. Kapsammer, S. Rausch-Schott, and W. Retschitzegger: 2000, 'X-Ray - Towards Integrating XML and Relational Database Systems'. In: *Proceedings of the 19th ER Int. Conf.*

10. Kappel, G., E. Kapsammer, and W. Retschitzegger: 2004, 'Integrating XML and Relational Database Systems'. *World Wide Web* **7**(4), 343–384.

11. Lee, D., M. Mani, F. Chiu, and W. Chu: 2001, 'Nesting-Based Relational-to-XML Schema Translation'. In: *Proceedings of the WebDB*. pp. 61–66.

12. Liu, C., M. Vincent, J. Liu, and M. Guo: 2003, 'A Virtual XML Database Engine for Relational Databases'. In: *Proceedings of XSYM*. pp. 37–51.

13. Shanmugasundaram, J., J. Kiernan, E. Shekita, C. Fan, and J. Funderburk: 2001, 'Querying XML Views of Relational Data'. In: *Proceedings of VLDB*. pp. 261–270.

14. Shanmugasundaram, J., E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald: 2000, 'Efficiently Publishing Relational Data as XML Documents'. In: *Proceedings of VLDB*. pp. 65–76.

15. Turau, V.: 2001, 'Making Legacy Data Accessible for XML Applications'. http://www.informatik.fh-wiesbaden.de/ turau/DB2XML/.