UPDATING RELATIONAL DATABASES THROUGH XML VIEWS[1]

Vanessa de Paula Braganholo Cesar Roberto Mariano Vittori
Carlos A. Heuser

UFRGS
Instituto de Informática
Porto Alegre – RS – Brazil
[vanessa, heuser@inf.ufrgs.br]

UTN
Facultad Regional Santa Fé
Santa Fé – Argentina
cvittori@frsf.utn.edu.ar

Abstract: *This paper discusses the problem of developing computer applications that internally handle data in XML format but store data in a relational database. The problem of building XML instances from relational databases is well known and several proposals are presented in literature. However, most of these proposals do not solve the problem of updating the database through an XML view. In the paper, a language called R2X (Relations to XML) that allows the specification of updatable XML views over relational databases is presented.*

1. Introduction

It is well known that XML is becoming a standard for the interchange of data on the WEB. However, corporate data still resides in relational databases. With the adoption of XML, applications that need to process XML instances are being developed. Examples of such applications are those that use XML in data interchange with other applications or for presentation to end-users. Such applications need to retrieve data from a relational database and construct XML instances from the retrieved data. If the application needs to modify data, update operations must be issued directly against the database.

In literature several approaches for building XML views from a relational database has been proposed. Many of them [2, 3, 4, 6, 9] focus on the problem of building read-only views, thus leaving the problem of updating the database to the programmer. However, some of them provide the construction of updatable views of relational databases. Examples of such proposals are Oracle XML-SQL Utility [1], IBM DB2 XML Extender [5], Microsoft SQL Server 2000 [8] and an extension to the W3C XQuery language to provide updates proposed in [7].

In this paper, we present *R2X* (*relations to XML*), a language for the specification of updatable views against relational databases. This proposal differs from others in the following points:

- In designing R2X we have tried to give the programmer freedom in constructing the XML views. The programmer may combine data from several tables in the result of a query. In the selection criteria, standard SQL predicates are used providing the full expression power of this language. There is no limit on the levels of nesting of XML elements in the query result. The only restriction we have imposed is that the result must be updatable, i.e. there must be a 1:1 mapping between XML elements and objects in the database. Without this restriction updates on

[1] This work was partially sponsored by CNPq, IBM and Solectron

the XML view cannot be mapped unambiguously to updates against the database.

In the Oracle XML-SQL Utility [1] updatable views are restricted to a single table (although this table may contain a nested table). For databases in the first normal form this results in flat views, without nesting of elements.

- In R2X the user application specifies the update to be performed simply by returning the updated XML view. The R2X processor is in charge of identifying the changes that occurred on the XML instance and mapping them to operations on the relational database.

  In [7], IBM DB2 XML Extender [5] and in Oracle XML-SQL Utility [1], the individual update operations (insert, update and delete) must be explicitly submitted to the update processor. The IBM DB2 XML Extender is further restricted to process insert operations only.

  Microsoft SQL Server 2000 [8] does not require the user to explicitly identify the operations that are performed on the XML instance. However, the user must submit to the update processor only those elements of an XML view that are being updated.

This paper is structured as follows. Section 2 describes the overall architecture of the R2X processor. Section 3 presents the R2X language and discusses how views are constructed from the relational database. Section 4 shows how views may be updated and explains how updates on views are mapped to SQL update operations against the relational database. Finally, section 5 contains the concluding remarks.

2. The architecture of the R2X processor

The R2X processor is a software that acts as a middleware between a *user application* and a *relational DBMS*. The R2X processor builds XML views from the relational database and passes them to the user application. The user application may modify these XML views. The R2X processor updates the database according to the modifications that were made to the XML view.

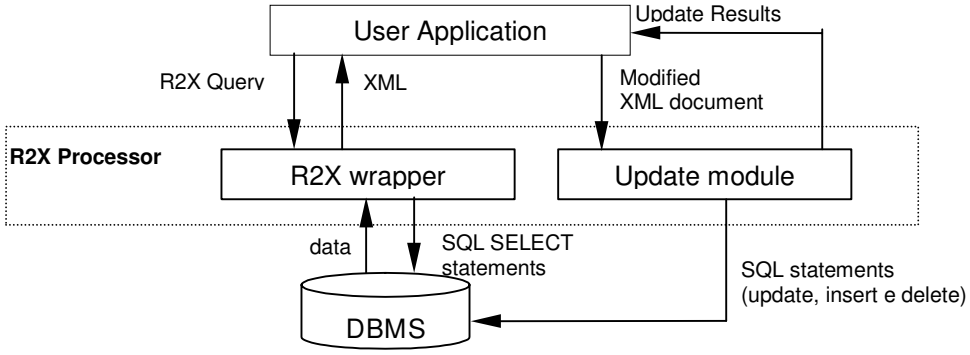

Figure 1: The R2X processor

The overall architecture is shown in Figure 1. A user application communicates with the R2X processor as follows:

1. The user application issues a query to the R2X processor. This query is written in the *R2X language*, a language that allows the specification of XML views over relational databases.
2. The R2X query is processed by a module called *R2X wrapper*. The wrapper constructs the specified XML instance, and passes this XML instance and its DTD to the user application.
3. The user application modifies the XML instance and sends it back to the R2X processor.
4. The modified XML instance is processed by the *update module*. The update module generates SQL statements by comparing the original and modified XML instances and applies the modifications to the relational database.

3. The R2X language

This section presents the R2X language. R2X is a language aimed at the specification of updatable XML views over a relational database. The language is presented through examples.

In the examples of R2X queries that follow, the database shown in Figure 2 is used. The database comprises departments, professors, and courses. The primary keys of each table are underlined.

Professors

ProfCode	ProfName	DeptCode
1	Ann	1
2	John	1
3	Bill	2

Departments

DeptCode	DeptName	Acr
1	Computer Science	CS
2	Philosophy	PH
3	Mathematics	

Courses

CourseCode	DeptCode	CourseName
1	1	Database Systems
2	2	Philosophy I
3	1	Compilers
4	3	Calculus

ProfCourses

ProfCode	DeptCode
1	1
1	3
2	1
3	2

Figure 2: Database used in examples

We suppose that the R2X processor has access to the database schema containing at least the following information: for each table, its name, the column names, column domains and the primary key; for each foreign key constraint, its name, the referred table, the referring table and the referring columns.

In the example database of Figure 2, we consider the following constraints:

```
TABLE Courses:
CONSTRAINT CoursesDept foreign key(DeptCode) references Departments
TABLE Professors:
CONSTRAINT ProfDept foreign key(DeptCode) references Departments
TABLE ProfCourses:
CONSTRAINT ProfCoursesProf foreign key(ProfCode) references Professors
CONSTRAINT ProfCoursesCourse foreign key(CourseCode) references Courses
```

A R2X query is basically an outmost *constructor clause* that contains other nested constructor clauses. The R2X query is represented as an XML instance. The DTD of the R2X language can be found at `www.inf.ufrgs.br/~vanessa/r2x.dtd`. Each constructor clause is represented in the R2X query as an XML element. Figure 3 presents an example of a R2X query and Figure 4 presents the result of the execution of this query against the example database of Figure 2.

There are three types of constructors:

LIST constructor Each LIST constructor refers to a table of the database. The LIST constructor constructs an XML element that contains a list of elements, one for each tuple of the base table. The name of the referred table is given by the `source` attribute. In the example of Figure 3, the outmost LIST constructor refers to the Departments table.

SEQUENCE constructor The SEQUENCE constructor specifies the XML elements that are constructed one for each tuple of the table referred by the embedding LIST constructor. The SEQUENCE constructor specifies a composite XML element. This composite element contains one child element for each constructor contained in the SEQUENCE constructor.

ATOM constructor The ATOM constructor specifies an atomic XML element that contains the value of a column in the database. The name of the referred column is given by the `source` attribute.

The outmost element of a R2X query is always a LIST constructor. A LIST constructor contains a single SEQUENCE constructor. A SEQUENCE constructor may contain ATOM and nested LIST constructors.

The tag name of a constructed XML element is specified by the `tagName` attribute of the

corresponding R2X constructor. In the example query, the tag name of the element specified by the outmost LIST constructor is `Departments`.

```
<R2X>
  <LIST source="Departments" tagName="Departments" alias="d"
        filter="d.acr IS NOT NULL">
    <SEQUENCE tagName="Department">
      <ATOM source="DeptCode"/>
      <ATOM source="DeptName"/>
      <LIST source="Courses" join="CoursesDept" tagName="Courses" alias="c"
         filter="c.CourseCode IN (SELECT distinct CourseCode FROM ProfCourses)">
        <SEQUENCE tagName="Course">
          <ATOM source="CourseCode"/>
          <ATOM source="CourseName"/>
        </SEQUENCE>
      </LIST>
    </SEQUENCE>
  </LIST>
</R2X>
```

Figure 3: An example of a R2X query

```
<Departments>
  <Department id="1">
    <DeptCode>1</DeptCode>
    <DeptName>Computer Science</DeptName>
    <Courses>
      <Course id="2">
        <CourseCode>1</CourseCode>
        <CourseName>Database Systems</CourseName>
      </Course>
      <Course id="3">
        <CourseCode>3</CourseCode>
        <CourseName>Compilers</CourseName>
      </Course>
    </Courses>
  </Department>
  <Department id="4">
    <DeptCode>2</DeptCode>
    <DeptName>Philosophy</DeptName>
    <Courses>
      <Course id="5">
        <CourseCode>2</CourseCode>
        <CourseName>Philosophy I</CourseName>
      </Course>
    </Courses>
  </Department>
</Departments>
```

Figure 4: Resulting XML instance from query in Figure 3

Associated to each LIST constructor, a filter clause may be specified. This clause is a predicate that specifies the tuples of the table that will generate elements in the output. The filter clause is represented in the R2X query by the `filter` attribute. The syntax used is the same of a search condition in the SQL SELECT command. In the filter clause, the table is denoted by an alias, specified by the `alias` attribute. Thus, in the example query, the filter clause of the outmost LIST constructor (`d.acr IS NOT NULL`) specifies that only those tuples of the Departments table that contain an acronym will be constructed in the output.

In the result of the example query, the `Departments` element is a list of `Department` elements, one for each tuple of the Departments table. Each `Department` element is constructed by the SEQUENCE constructor embedded in the outmost LIST constructor. In the example, the SEQUENCE constructor specifies that each `Department` element will contain three child elements, `DeptCode`, `DeptName` and `Courses`. The `DeptCode` and `DeptName` elements are constructed by ATOM constructors.

The `Courses` element is constructed by the nested LIST constructor. A nested LIST constructor must refer to a table that is related by a foreign key constraint to the table referred by the parent LIST constructor. In the example, the nested LIST constructor refers to the Courses table and the parent LIST constructor refers to the Departments table. The Courses table is related to the

Departments table by the CoursesDept foreign key constraint. The foreign key constraint used to relate both tables is specified by the `join` attribute of the nested LIST constructor (a `join` attribute may appear only in nested LIST constructors).

```
<R2X>
  <LIST source="Departments" tagName="Departments" alias="d">
    <SEQUENCE tagName="Department">
      <ATOM source="DeptCode"/>
      <ATOM source="DeptName"/>
      <LIST source="Courses" join="CoursesDept" tagName="Courses" alias="c">
        <SEQUENCE tagName="Course">
          <ATOM source="CourseCode"/>
          <ATOM source="CourseName"/>
        </SEQUENCE>
      </LIST>
      <LIST source="Professors" join="ProfDept" tagName="Professors"
                alias="p" filter="p.ProfCode = 3">
        <SEQUENCE tagName="Prof">
          <ATOM source="ProfCode"/>
          <ATOM source="ProfName"/>
        </SEQUENCE>
      </LIST>
    </SEQUENCE>
  </LIST>
</R2X>
```

Figure 5: R2X query with two nested tables at the same level

In the same way as the outmost LIST constructor, a nested LIST constructor outputs an element for each tuple of the referred table. In the nested LIST constructor there is an additional restriction: to be selected a tuple must be related to the tuple corresponding to the parent element. In this way, the `Courses` element embedded in a `Department` element will contain only elements corresponding to tuples related to the current Department. This selection of tuples may be further restricted by the filter clause.

Apart from limits imposed by a specific implementation of the R2X processor, lists may be arbitrarily nested and a sequence may contain several lists. One example is the query in Figure 5. This query builds a list of departments. For each department a SEQUENCE element is constructed containing the department identification, the department name, the list of courses belonging to the department and the list of professors belonging to the department.

In order to guarantee that the XML instance produced by a R2X query will be updatable, it is necessary that each element correspond uniquely to one object (a table, a line or a column value) in the database. Therefore, a restriction on the query formulation must be observed. The primary key of each table must appear in the sequence element that represents a tuple of the table. Further, for each nested list the columns referred in the foreign key constraint used to build the nested list must appear in the output XML instance.

4. Update Operations

This section explains the operations that can be performed on the elements of an XML instance by a user application. Each type of operation is explained through an example. There are four types of operations:

INSERT An insert operation consists of inserting a sub-tree in the XML instance.
DELETE A delete operation removes a sub-tree of the XML instance.
UPDATE An update operation modifies an atomic value.
MOVE A move operation excludes a sub-tree consisting of a sequence element and its child elements and inserts it in a different location at the same nesting level.

Not all elements of an XML instance may be updated. We classify the elements of the result of a R2X query in two types:

updatable elements An element is updatable if there is a 1:1 mapping between the element and an

object (table, tuple or column) in the database.

non-updatable elements An element is non-updatable if there is an n:1 mapping between the element and an object in the database. In other terms, an element in non-updatable if it represents a database object that appears more than once in the XML instance.

In the way R2X was designed, non-updatable elements will appear only in a special case. We will first consider update operations on updatable elements and leave the discussion of non-updatable elements to the end of this section.

In the examples that follow each operation is exemplified. They are all based on the XML instance of Figure 4.

```
<Departments>
  ...
  <Department>
    <DeptCode>4</DeptCode>
    <DeptName>Medicine</DeptName>
    <Courses>
      <Course>
        <CourseCode>5</CourseCode>
        <CourseName>Anatomy</CourseName>
      </Course>
    </Courses>
  </Department>
</Departments>
```

Figure 6: Modified XML instance: complex element Department inserted

The instance shown in Figure 6 is the result of an insertion operation in the example instance of Figure 4. The considered operation inserts the elements shown in boldface in Figure 6. Thus, this operation inserts the Department named Medicine, as well as one course of this department: Anatomy. This insertion must be reflected in the database by inserting a tuple in the Departments table and another one in the Courses table.

The update module of the R2X processor (Figure 1) compares the original instance and the modified instance and generates the following SQL statements:

```
INSERT INTO Departments (DeptCode, DeptName) VALUES (4, 'Medicine')
INSERT INTO Courses (CourseCode, DeptCode, CourseName) VALUES (5, 4, 'Anatomy')
```

Notice that the value of the `Courses.DeptCode` column is inferred from the fact that a `Course` element is nested in a `Department` element. This derives from the fact that the relationship between these entities is represented in XML by nesting elements.

```
<Departments>
  <Department id="1">
    <DeptCode>1</DeptCode>
    <DeptName>Computer Science</DeptName>
    <Courses>
      <Course id="2">
        <CourseCode>1</CourseCode>
        <CourseName>Database Systems</CourseName>
      </Course>
      <Course id="3">
        <CourseCode>3</CourseCode>
        <CourseName>Compilers</CourseName>
      </Course>
    </Courses>
  </Department>
</Departments>
```

Figure 7: Modified XML instance: element `Department id="4"` deleted

```
<Departments>
  ...
      <Course id="2">
        <CourseCode>1</CourseCode>
        <CourseName>DB Systems and Applications</CourseName>
      </Course>
  ...
```

Figure 8: Modified XML instance: value of element `CourseName` updated

```
<Departments>
  <Department id="1">
    <DeptCode>1</DeptCode>
    <DeptName>Computer Science</DeptName>
    <Courses>
      <Course id="2">
        <CourseCode>1</CourseCode>
        <CourseName>Database Systems</CourseName>
      </Course>
    </Courses>
  </Department>
  <Department id="4">
    <DeptCode>2</DeptCode>
    <DeptName>Philosophy</DeptName>
    <Courses>
      <Course id="5">
        <CourseCode>2</CourseCode>
        <CourseName>Philosophy I</CourseName>
      </Course>
      <Course id="3">
        <CourseCode>3</CourseCode>
        <CourseName>Compilers</CourseName>
      </Course>
    </Courses>
  </Department>
</Departments>
```

Figure 9: Modified XML instance: Course Compilers moved

Figure 7 shows the modified instance that results of a DELETE operation over the example instance of Figure 4. In this case, the Department with attribute `id="4"` (Philosophy) was removed.

The result of the removal of an XML sub-tree on the database is that the tuples corresponding to the removed elements will be deleted from the database. In the example, the `Department` element contained a `Course` element, referring to the course Philosophy I. Thus, not only the tuple corresponding to the removed department will be deleted, but also the course tuple. The generated SQL statements are the following:

```
DELETE FROM Courses WHERE CourseCode=2
DELETE FROM Departments WHERE DeptCode=2
```

The UPDATE operation is defined only on atomic elements. Figure 8 shows the result of the UPDATE operation on the example instance of Figure 4. The content of the `CourseName` element of the `Course` with `id="2"` has been modified. In this case, the generated SQL statement is:

```
UPDATE Courses SET CourseName="DB Systems and Applications" WHERE CourseCode=1
```

The result of a MOVE operation is illustrated in Figure 9. In this example, the Course named Compilers former belonging to the Computer Science department was moved to the Philosophy department. The `id` attribute of the sequence element being moved must remain the same.

This operation has the following semantics. In our approach, a child sequence element represents a tuple related by a foreign key constraint to the tuple represented by the parent sequence element. Therefore, moving a child sequence element from one parent to another, is interpreted in terms of the database as the update of a foreign key. In the example, the MOVE operation is mapped to an update of the value of the foreign key that relates a tuple in table Courses (corresponding to the child element) with a tuple in table Departments (corresponding to the parent element). The generated UPDATE operation on the database is

```
UPDATE Courses SET DeptCode=2 WHERE CourseCode=3
```

The operations above are allowed on updatable elements. As mentioned before, there is a special case where non-updatable elements appear. This happens when a database object is represented several times in the XML instance. In a R2X query, this will occur only when the nesting of a sequence inside another, results from the traversal of a foreign key in the n:1 direction. The relationships established by the nesting of elements in an XML instance build a *tree* (each element has a single parent). The relationships established by the foreign keys in a database may build a

graph (a tuple may be "child" of several other tuples). In order to obtain a 1:1 mapping between the database objects (nodes of the graph) and the XML elements (nodes of the tree), the graph must be traversed in the 1:n direction.

```
<R2X>
  <LIST source="Courses" tagName="Courses" alias="c">
    <SEQUENCE tagName="Course">
      <ATOM source="CourseCode"/>
      <ATOM source="CourseName"/>
      <LIST source="Departments" join="CoursesDept"
            tagName="Departments" alias="d">
        <SEQUENCE tagName="Department">
          <ATOM source="DeptCode"/>
          <ATOM source="DeptName"/>
        </SEQUENCE>
      </LIST>
    </SEQUENCE>
  </LIST>
</R2X>
```

Figure 10: R2X query that results an XML Instance with non-updatable elements

In Figure 10 a R2X query that constructs an XML instance that contains non-updatable elements is shown. The result of this query is shown in Figure 11. This query contains two nested LIST constructors, one for Courses (outmost) and another one for Departments (embedded). The embedded LIST constructor holds the foreign key of the CoursesDept constraint, and the outmost LIST constructor holds the primary key of this constraint. Thus, in the example, the foreign key constraint is traversed in the n:1 direction (from the foreign to the primary key). The result is that one department may be represented more than once in the XML instance. Considering the contents of the example database, elements `Department id="2"` and `Department id="6"` represent the same object in the database. These elements are called non-updatable.

The tuple represented by a non-updatable element may not be updated through this element. As the tuple may be represented by several XML elements an update through one of those elements is ambiguous. The consequence is that the UPDATE operation may not be performed on non-updatable elements.

```
<Courses>
  <Course id="1">
    <CourseCode>1</CourseCode>
    <CourseName>Database Systems</CourseName>
    <Departments>
      <Department id="2">
        <DeptCode>1</DeptCode>
        <DeptName>Computer Science</DeptName>
      </Department>
  ...
  <Course id="5">
    <CourseCode>3</CourseCode>
    <CourseName>Compilers</CourseName>
    <Departments>
      <Department id="6">
        <DeptCode>1</DeptCode>
        <DeptName>Computer Science</DeptName>
      </Department>
    </Departments>
  </Course>
  <Course id="7">
    <CourseCode>4</CourseCode>
    <CourseName>Calculus</CourseName>
    <Departments>
      <Department id="8">
        <DeptCode>3</DeptCode>
        <DeptName>Mathematics</DeptName>
      </Department>
  ...
</Courses>
```

Figure 11: XML instance with non-updatable elements

However, in the special case of a non-updatable *sequence* element that is contained in an updatable

element, some update operations are allowed. In this case, restricted versions of the INSERT, DELETE and MOVE operations are allowed. What the allowed operations have in common is that they do not update the tuple represented by the non-updatable element, but update the foreign that refers to this tuple and are contained in the tuple represented by the updatable parent element. The operation semantics for the non-updatable elements are the following:

```
<Courses>
  <Course id="1">
    <CourseCode>1</CourseCode>
    <CourseName>Database Systems</CourseName>
    <Departments>
    </Departments>
  </Course>
  ...
</Courses>
```

Figure 12: Result of a DELETE operation on non-updatable elements

DELETE The deletion of a non-updatable element specifies that the key that implements the foreign key constraint must be set to NULL. Figure 12 shows the result of a DELETE operation on the XML instance shown in Figure 11. The department of the Database Systems Course (`Course id="1"`) was removed. The SQL statement produced by this operation is:

```
UPDATE Courses SET DeptCode=NULL WHERE CourseCode=1
```

MOVE When a non-updatable element is moved, two operations are performed on the database. First, the key that implements the foreign key constraint of the parent of the removed element is set to null. Additionally, the key of the parent of the inserted element is updated. Figure 13 shows an example of that. This operation was executed on the XML instance of Figure 12. The Mathematics department (`Department id="8"`) was moved from the `Course id="7"` to `Course id="1"`. The generated SQL statements are:

```
UPDATE Courses SET DeptCode=NULL WHERE CourseCode=4
UPDATE Courses SET DeptCode=3 WHERE CourseCode=1
```

INSERT The insertion of a non-updatable element means an update in the key that implements the foreign key constraint. Figure 14 shows the result of the insertion of a non-updatable element in the XML instance of Figure 12 . It inserts one Department in the Database System Course (`Course id="1"`). The semantics of this INSERT operation is to connect this course with the Computer Science department. The generated SQL statement is the following:

```
UPDATE Courses SET DeptCode=1 WHERE CourseCode=1
```

Note that only the foreign key is updated. This operation does not insert a Department tuple in the database. Therefore, the inserted element must only contain the primary key of the database object it is representing, as shown in Figure 14

```
<Courses>
  <Course id="1">
    <CourseCode>1</CourseCode>
    <CourseName>Database Systems</CourseName>
    <Departments>
     <Department id="8">
        <DeptCode>3</DeptCode>
        <DeptName>Mathematics</DeptName>
      </Department>
    </Departments>
  </Course>
  ...
  <Course id="7">
    <CourseCode>4</CourseCode>
    <CourseName>Calculus</CourseName>
    <Departments>
    </Departments>
  </Course>
</Courses>
```

Figure 13: Result of a MOVE operation on non-updatable elements

```
<Courses>
  <Course id="1">
    <CourseCode>1</CourseCode>
    <CourseName>Database Systems</CourseName>
    <Departments>
      <Department>
        <DeptCode>1</DeptCode>
      </Department>
    </Departments>
  </Course>
  ...
</Courses>
```

Figure 14: Result of an INSERT operation on non-updatable elements

The operations explained in this section may be combined in several ways. When the user application submits the modified XML instance to the R2X processor, the Update Module identifies each operation and reflects them in the database. So, the result of the modifications made by the user application in the XML instance can be a *list* of SQL statements.

An example is an update in which the XML instance in Figure 13 results from the XML instance in Figure 11. In this case, the update module will identify the combination of a DELETE and a MOVE operation, exactly those described in the above examples. These two operations will be executed against the database as a single transaction. In this way, it's possible to the user application to modify the XML instance performing several operations, and then submitting the modified XML instance to the R2X processor.

5. Concluding remarks and future work

R2X is a language for specifying updatable XML views over relational databases. Using R2X, view construction and update processing are specified *declaratively*. To construct a view, the user specifies the structure of the resulting XML instance, as well as the sources of the XML elements in the relational database. To update the database through the view, the user simply returns the modified view to the R2X processor. The R2X processor identifies the changes that occurred on the view and maps them to update operations against the relational database.

The R2X language was developed in the context of a project to build wrappers to access legacy databases. In this project, applications interface with end-users through WEB browsers. The applications handle internally data in XML format and use the R2X processor to access and update a relational database.

References

[1] Oracle xml sql utility for java. Oracle Corporation, 1999. http://technet.oracle.com/tech/xml/oracle_xsu/.

[2] C. Baru. Xviews: Xml views of relational schemas. In *Proceedings of the Workshop on Databases and Expert System Applications*, Florence, Italy, September 1999. http://citeseer.nj.nec.com/252703.html

[3] M. Carey et al. Xperanto: Publishing object-relational data as xml. In *Third International Workshop on the Web and Databases*, Dallas, Texas, May 2000. http://www.cs.wisc.edu/ jai/papers/XperantoOverview.pdf.

[4] Mary Fernández, Wang-Chiew Tan, and Dan Suciu. Silkroute: Trading between relations and xml. In *Proceedings of the Ninth International World Wide Web Conference*, 2000.

[5] IBM. *XML Extender Administration and Programming*, 2000.

[6] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Efficiently publishing relational data as xml documents. *VLDB Conference*, September 2000. http://www.cs.wisc.edu/ jai/pubs.html.

[7] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating xml. In *Proceedings of SIGMOD 2001*, May 2001.

[8] Joshua Trupin. Sql server 2000: New xml features streamline web-centric app development. Msdn Magazine.

[9] C. Vittori, C. Dorneles, and C. Heuser. Creating xml documents from relational data sources. In *Proceedings of EC-Web 2001*, Munich, Germany, September 2001.