Universidade Federal do Rio Grande do Sul
Instituto de Informática
Programa de Pós–Graduação em Computação

Updating Relational Databases through XML Views

by
Vanessa de Paula Braganholo

Thesis Proposal

Professor Carlos A. Heuser
Advisor

Porto Alegre, September 2002

Contents

List of Abreviations

¬1NF	Non-First Normal Form
1NF	First Normal Form
4NF	Fourth Normal Form
BCNF	Boyce-Codd Normal Form
DBMS	Database Management System
DTD	Document Type Definition
EFD	Extended Functional Dependency
NF	Normal Form
NF^2	Non First Normal Form
NFD	Nested Functional Dependency
NF-N3	Normal Form for Sets of Not Necessarily Normalized Relations
NF-SS	Normal Form for Semistructured Data
NNF	Nested Normal Form
PNF	Partitioned Normal Form
R2X	Relations to XML
uR2X	underlying Relations to XML
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XNF	XML Normal Form

List of Figures

List of Tables

Abstract

This work contains a thesis proposal in the area of XML views over relational databases. This thesis proposes a solution for the problem of updating relational databases through XML views. The proposed solution is based on a query language called R2X and on a concept of normalization for XML views.

The R2X language constructs XML views from relational databases, while normalization removes the ambiguity in an XML view, thus solving the update problem. In order to normalize an XML view, we benefit from the relation between XML and non-first normal form relations (NF^2). Our proposal is to work on a normal form for NF^2 relations and map it to XML.

The implementation model of R2X is based on NF^2 relations. More specifically, an R2X query generates flat views that are transformed into NF^2 relations through *nest* operations. The resulting NF^2 view is than mapped to XML.

A normal form for NF^2 relations will be used to eliminate the ambiguity in the NF^2 views. This normal form will be used in R2X to guarantee that it generates only updatable views.

Keywords: XML views, update through views, non-first normal form relations, normal forms for non-first normal form relations

Chapter 1

Introduction

Since its introduction, XML quickly became the universal format for publishing and exchanging data on the Web. Despite that, most of corporate data still resides in relational databases. This is due to the maturity of indexing, query processing and storing techniques of the relational management systems.

In recent years, several types of applications appeared, where data stored in relational databases need to be translated to XML. Examples are Web applications that separate content from presentation. Applications of this type represent content internally in XML and present data to the user in HTML. Another example are applications that use XML to interchange data.

In order to implement such types of applications, tools for generating XML instances from relational data sources have been proposed in the literature. In this work, these tools are called *wrappers* because they can be thought as an intermediate layer between a user application and a relational database. These tools can be classified in two types.

Tools that map the relational model to XML and use XML query languages to build the views: This type of solution is suitable for users who are familiar with the XML model and its query languages. However there is a big semantic gap between queries in an XML language and the corresponding query in SQL. Due to this fact the construction of an XML wrapper in this approach is not a simple task [FER 2000].

Tools that use SQL to construct the views and map the relational view to XML: Such approach is suitable for users who are familiar with the relational model and the SQL language. In this approach the XML wrapper has a single task: the construction of XML instances from the result of SQL queries.

Most of the existing proposals present limitations such as constraints on the nesting of XML elements and the need of procedural specification of the mapping from SQL to XML.

Another problem consists in updating the relational database through XML views. From the application point of view, it is natural that the changes made in the XML view be automatically mapped to the underlying relational database. However, few of these tools supply this kind of support [TAT 2001, ORA 2001, CHE 2000, WAH 2000]. Additionally, tools that support updates generally present restrictions regarding the way changes are submitted, the kind of view that can be updated (usually only flat views) and also the kind of update operations that can be performed (some are restricted to only insertions).

This work fits in this context, since it proposes a mechanism to update relational databases through XML views. The mechanism is based on a language that constructs *updatable*

XML views over relational databases and on a concept of normalization for XML views. The language is called R2X (***R**elations **to** **XML***). It was designed over a more generic language called uR2X (***u**nderlying **R**elations **to** **XML***). Both R2X and uR2X follow the first approach, that is, they use SQL to build the views and map the relational views to XML.

uR2X was designed as a base language for building XML wrappers over relational databases. The main design goal of uR2X was to provide a powerful query language that has an efficient implementation. uR2X is a powerful language because it allows the construction of XML views with arbitrary structure and nesting. It has an efficient implementation because it relies on the query optimization methods of the relational database. However, uR2X is not a user-friendly language because the user must deal with the details of the mapping from XML to SQL.

Using the uR2X wrapper we proposed a wrapper architecture for the R2X language. R2X is more user-friendly than uR2X, but less powerful. As the main design goal of R2X is the generation of updatable XML views, it is necessary to guarantee this feature. In order to accomplish this, we propose to normalize the XML view to remove ambiguity.

The proposal for the XML view normalization will take advantage of the relation between non-first normal form relations (NF^2) and XML. NF^2 relations have been used to represent XML instances [ABI 84, SIL 2002a], since both models have common features: multivalued attributes and hierarchical structure. Consequently, we can work in a normal form for NF^2 relations and map it to XML.

With this in mind, we decided to use NF^2 relations in the uR2X and R2X design. The implementation model of both languages was defined using nested relations. With R2X being based in NF^2 relations, it is possible to use (or adapt) one of the proposals for normal forms for NF^2 relations presented in literature [MAK 77, ROT 88, LIN 89, MOK 96] in order to remove ambiguity in the views generated by R2X.

The versions of both languages presented in this work are an evolution of the XML/SQL language [VIT 2001] and of a previous version of R2X [BRA 2001a]. The present versions rely on the syntax of XML Schema [XML 2001], whereas the previous versions had a specific syntax.

The remainder of this work is organized as follows. Chapter 2 reviews the areas related to this work. Chapter 3 presents uR2X and R2X languages, focusing in the NF^2 implementation model and in the proposed update mechanism. Also, the relation of XML and nested relations is established. The thesis contributions and the tasks to be done in order to get the desired results are presented in chapter 4.

Chapter 2

Related Work

The main goal of the thesis is to define a mechanism to update relational databases through XML views. In order to accomplish this, we propose a query language called R2X and an implementation model based in NF^2 relations. In this scenario, three main related areas can be identified: non-first normal form relations, XML wrappers for relational databases and database update through views. These areas are briefly described in this chapter.

This chapter is structured as follows. Section 2.1 discusses nested relations. More specifically, it presents the *nest* operation, which is used in the uR2X and R2X implementation model. Additionally, section 2.2 presents normal forms for non-first normal form relations. Normal forms remove ambiguity of NF^2 relations. The tools that generate XML views from relational databases (XML wrappers for relational databases) are shown in section 2.3. Finally, section 2.4 presents some considerations in database update through views.

Before going further, we present a sample database (Figure 2.1) that will be used in the examples through this work. This database comprises departments, equipment, employees and dependents. Each department is described by a department code, a name and an acronym. For each department, there is several associated equipment. Each equipment is described by an equipment code, a description and the department code to which it belongs. There are several employees in each department, and employees are described by an employee code, a name, a birth date and the department code for which they work. Also, an employee has several dependents. Each dependent has a dependent code, a name, a birth date and the code of the employee that he depends on. The primary keys of each table are underlined in figure 2.1.

This sample database will be used in the next section to exemplify the *nest* operation, and also in chapter 3 to exemplify uR2X and R2X queries.

2.1 Non-First Normal Form relations

A nested relation (*non-first normal form relation*) [MAK 77, ABI 84, JAE 82] is a relation whose tuple components are atomic values or repeating group instances [OZS 87]. Another definition of nested relation is given in [ELM 2000]. It defines a non-first normal form relation as a relation that allows composite and multivalued attributes, thus leading to complex tuples with hierarchical structure. This hierarchical structure resembles the XML model and has been used to represent it [ABI 84, SIL 2002a].

Department

<u>DeptCode</u>	DeptName	Acr
1	Sales	S
2	Human Resources	HR
3	Buys	

Employee

<u>EmpCode</u>	EmpName	BirthDate	DeptCode
1	Bill	01/01/1976	1
2	Ann	05/10/1970	1
3	Peter	12/08/1964	3
4	Marc	25/11/1969	2
5	John	12/09/1970	1
6	Rachel	25/03/1979	2

Equipment

<u>EquipCode</u>	EquipDescription	DetpCode
1	Fax	1
2	Telephone	2
3	Pentium 950 MHZ	2
4	Fax	3
5	Copier Machine	3

Dependent

<u>DepCode</u>	DepName	BirthDate	EmpCode
1	Mary	10/02/1997	3
2	Paul	25/05/1988	3
3	Catherine	23/09/1988	2
4	Marc	01/12/1999	5

Foreign Key Constraints:

On table Employee:
CONSTRAINT EmpDept foreign key(DeptCode) references Department
On table Dependent:
CONSTRAINT DepEmp foreign key(EmpCode) references Employee
On table Equipment:
CONSTRAINT EquipDept foreign key(DeptCode) references Department

Figure 2.1: Database used in examples

Departments

DeptCode	DeptName	Employees			
		EmpCode	EmpName	Dependents	
				DepCode	DepName
1	Sales	1	Bill		
		2	Ann	3	Catherine
		5	John	4	Marc
2	Human Resources	4	Marc		
		6	Rachel		
3	Buys	3	Peter	1	Mary
				2	Paul

Figure 2.2: Non-First Normal Form Relation - Departments

A relational database is a set of flat relations (first normal form relations) whereas an XML instance can be considered an NF^2 view. Several researchers [ARI 83, JAE 82, FIS 85, ROT 88] address the problem of obtaining non-first normal form relations from normalized ones. They do this by defining *composition* and *decomposition* operations [ARI 83], or *nest* and *unnest* operations. The nest operator partitions relations and creates nested relations for each partition formed [ROT 87a]. More specifically, the nest operator groups together tuples of a relation r which agree on all the attributes that are not in a given set of attributes, say $\{B_1, B_2, \ldots, B_n\}$. It forms a single tuple, for every such group, which has a new attribute, say A, in place of $\{B_1, B_2, \ldots, B_n\}$, whose value is the set of all $\{B_1, B_2, \ldots, B_n\}$ values of the tuples being grouped together [COL 89]. The syntax is $\nu_{A=(B_1,B_2,\ldots,B_n)}(r)$ (syntax extracted from [ROT 87a]).

Figure 2.2 shows an example of a non-first normal form relation named Departments (the graphical notation adopted in this figure is the one used in [ROT 87a]). This relation shows, for each department, the employees associated with it. For each employee, the employee code, the name and a list of associated dependents are shown. Each dependent has a code and a name. Note that not every employee has dependents.

The association of Employees and Departments is made through nesting. In this case, we say that the Employees relation is embedded in the Departments relation and the Dependents relation is embedded in the Employees relation.

DeptCode	DeptName	EmpCode	EmpName	DepCode	DepName
1	Sales	1	Bill		
1	Sales	2	Ann	3	Catherine
1	Sales	5	John	4	Marc
2	Human Resources	4	Marc		
2	Human Resources	6	Rachel		
3	Buys	3	Peter	1	Mary
3	Buys	3	Peter	2	Paul

Figure 2.3: Flat Relation

This relation can be obtained by applying a sequence of *nest* operators in the relation of Figure 2.3. The traditional algebra query that generates the flat relation of Figure 2.3 is the following.

$$\pi_{(DeptCode, DeptName, EmpCode, EmpName, DepCode, DepName)}$$
$$(Department * (Employee \sqsupset\!\bowtie_{(Employee.CodEmp=Dependent.CodEmp)} Dependent))$$

Over this relation, a nest operation (ν) is applied to create the nested relation Dependents.

$$\nu_{Dependents=(DepCode,DepName)}$$
$$(\pi_{(DeptCode, DeptName, EmpCode, EmpName, DepCode, DepName)}$$
$$(Department * (Employee \sqsupset\!\bowtie_{(Employee.CodEmp=Dependent.CodEmp)} Dependent))$$

Further, a nest operation is applied to create the nested relation Employees.

$$\nu_{Employees=(EmpCode,EmpName)}$$
$$(\nu_{Dependents=(DepCode,DepName)}$$
$$(\pi_{(DeptCode, DeptName, EmpCode, EmpName, DepCode, DepName)}$$
$$(Department *$$
$$(Employee \sqsupset\!\bowtie_{(Employee.CodEmp=Dependent.CodEmp)} Dependent)))$$

This query generates the relation shown in Figure 2.2. The nested structure of this relation resembles the XML hierarchical structure.

2.2 Normal Forms for Non-First Normal Form Relations

Makinouchi [MAK 77] was the first one to say that a nested relation can be normalized in order to avoid update anomalies, just as in Codd's third normal form [COD 71, COD 74]. However, it is not necessary that this relation be in the first normal form. According to [MAK 77], there are two alternatives to remove redundancy from a relation: to decompose the relation considering the 1NF, 2NF, 3NF and so on, or to nest the relation, putting it in Normal Form.

After this definition, other proposals for normal forms for nested relations appeared. The main ones are *Nested Normal Form* (NNF) [MOK 96], *Partitioned Normal Form* (PNF) [ROT 88, HUL 90] and *Normal Form for Sets of Not-Necessarily Normalized Relations* (NF-N3) [LIN 89].

Partitioned Normal Form seams to be the most well accepted [MOK 96]. In fact, the proposals for Nested Normal Form [OZS 87, ROT 87, OZS 89, EMB 92, MOK 96] adopt

PNF as a starting point. In this way, a nested relation has to be in PNF to be in Nested Normal Form. Also, [ABI 84, ABI 86] uses PNF in Verso Model. The Verso Model is a database model that uses non first normal form relations to represent data [ABI 84]. Despite being prior to the definition of PNF, the Verso Model works in a way that only PNF relations are accepted.

However, PNF is strictly a basic level normalization goal. Although PNF relations have less redundancy than their non-PNF counterparts, some nesting schemes will be less redundant than others, even if all relations are in PNF [ROT 88]. This indicates that PNF does not remove all data redundancy.

The work in [MOK 96] proposes a new definition to Nested Normal Form. In this work, a formal definition to data redundancy is presented. The definition is based in functional and multivalued dependencies, and is used to prove that NNF detects potential redundancy. It also presents examples that prove that the previous definitions of Nested Normal Form [OZS 87, OZS 89, ROT 87] did not fully detected redundancy in data. The notion of redundancy in [MOK 96] is based on the idea that an atomic value *v* in a nested or flat relation is redundant if we can erase *v* and then from what remains and from a single FD or MVD that holds, determine what *v* must have been.

An important contribution in [MOK 96] is the proof that 4NF and BCNF are special cases of NNF when the dependencies are limited to FDs.

NF-N3 enforces the normalization of the whole database (this is the reason for the name Normal Form for *Sets* of Not-Necessarily Normalized Relations). In this way, it avoids global redundancy among relations. Unlike other definitions of Normal Forms, NF-N3 requires the NF^2 relation being normalized to have a primary key.

2.2.1 Functional and Multivalued Dependencies

While NF, PNF and NF-N3 uses extended functional and multivalued dependencies in their definitions of Normal Forms, NNF uses regular functional and multivalued dependencies. NF, PNF and NF-N3 extend the definitions of functional dependency to work with sets. In this way, they are capable of expressing the following functional dependency for the relation of Figure 2.2:

$$DeptCode\ DeptName \rightarrow Employees$$
$$EmpCode\ EmpName \rightarrow Dependents$$

In this case, *Employees* and *Dependents* are sets, or nested relations. The only difference in the definitions of functional dependencies is the graphical notation adopted in each proposal (NF uses ’$\Rightarrow$’, PNF uses ’$\rightarrow$’ and NF-N3 uses ’$\Longrightarrow$’).

NNF, however, uses regular functional and multivalued dependencies. This is possible because sets are not allowed in functional and multivalued dependencies. As an example, in the case of Figure 2.2 they use the following set of multivalued dependencies:

$$\{DeptCode\ DeptName\} \twoheadrightarrow \{EmpCode\ EmpName\ DepCode\ DepName\}$$
$$\{DeptCode\ DeptNameEmpCodeEmpName\} \twoheadrightarrow \{DepCode\ DepName\}$$
$$\{EmpCode\ EmpName\} \twoheadrightarrow \{DepCode\ DepName\}$$

NFF uses *scheme trees* to represent NF^2 relations. These scheme trees are used to obtain the set of multivalued dependencies that holds for a given NF^2 relation. In the case of Figure 2.2, the scheme tree is shown in Figure 2.4. Nodes of a scheme tree are one or more atomic attributes, and each edge represents an MVD.

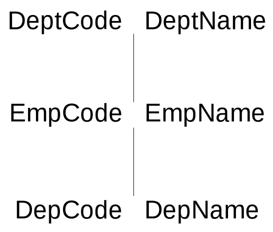


Figure 2.4: Scheme tree that represents the NF^2 relation of Figure 2.2

Besides these functional and multivalued dependencies used in Normal Forms definitions, another definition of functional dependency is presented in [HAR 99]. It is called *Nested Functional Dependency*. The definition of Nested Functional Dependency allows the use of *paths* in a functional dependency. As an example, the following nested functional dependency can be expressed for the relation of Figure 2.2.

$Departments : [DeptCode \rightarrow Employees : EmpCode]$

Additional details on Normal Forms for NF^2 relations can be found in [BRA 2001].

2.3 XML Wrappers to relational databases

As mentioned before, in literature there are two alternatives to the construction of XML wrappers over relational databases.

One alternative is to use an XML query language, such as XML-QL [DEU 98], XQL [ISH 98], or XQuery [BOA 2001]. In this approach the relational database model must be mapped to XML. The queries submitted in an XML query language are mapped to SQL. Examples of such approach are SilkRoute [FER 2000], Xperanto [CAR 2000] and [TAT 2001].

Another alternative consists of using SQL queries and mapping the resulting relational views into the XML model. Such alternative is suitable for users who are familiar with the relational model and the SQL language.

Several tools that follow this alternative have been proposed. In most of them the structure of the resulting XML instance resembles the flat structure of the underlying relational views. No nesting of collections is allowed. Approaches of this type are DB2XML [TUR 2001], Oracle XML SQL Utility [ORA 2001, HIG 2001], as well as extensions of the APIs SAX and DOM for databases [LAD 2001].

Tools that allow the construction of XML instances with nesting of collections are the ODBC2XML tool [INT 2001], IBM DB2 XML Extender [CHE 2000, IBM 2000], SQLXML (XML for SQL Server 2000) [CON 2001] and DB/XML Transform [DAT 2001]. The limitation of ODBC2XML is that the specification of the result is procedural and, for complex documents, involves the execution of iterative SQL queries. This can compromise the efficiency of the execution of the query, because the query optimizer of the relational DBMS is not used. In the IBM DB2 XML Extender tool, each query consists of a single SELECT SQL command. This limits the expressive power of the language when dealing with several 1:n relationships involving the same entity (see example in Section 3.3.2). SQLXML and IBM DB2 XML Extender are proprietary solutions that work only with their respective databases (Microsoft SQL Server and IBM DB2). ODBC2XML works with any relational database that allows ODBC/JDBC connections.

This alternative of using SQL queries and mapping the resulting relational views to the

Table 2.1: Comparison of approaches for building XML views from relational databases

	Comparison criteria											
	1	2	3	4	5	6	7	8	9	10	11	12
DB2XML	SQL	×	✓	✓	D	✓	-	×	-	×	×	✓
DB/XML Transform	-	✓	×	✓	-	✓	-	×	-	×	×	✓
IBM DB2 XML Extender	SQL	✓	×	×	P	✓	-	×	-	×	✓	×
ODBC2XML	ODBC2XML template	✓	✓	×	P	✓	-	×	-	×	×	✓
Oracle XSQL	SQL	×	×	×	P	✓	-	×	-	×	✓	×
SAX and DOM Extensions	SQL	×	×	×	P	×	-	✓	XQL	×	×	✓
SilkRoute	RXL	✓	✓	×	P	✓	RXL	✓	XML-QL	×	×	✓
SQL Server	SQL + for XML or XML Schema	✓	✓	×	D	✓	-	✓	XPath	×	✓	×
[TAT 2001]	XQuery	✓	×	×	D	✓	-	×	-	×	✓	✓
Xperanto	XQuery	✓	×	×	D	✓	XQGM	✓	XQuery	✓	×	×

XML model is the one followed in uR2X, as the language is intended to serve as a base to the construction of wrappers and not as a language for application developers.

Table 2.1 shows a brief comparison of the proposals for generating XML views from relational databases presented in this section. The comparison criteria were numbered in the table. They are listed above.

1 - Query language used in view definition: Xperanto has a default view expressed in XML Schema [CAR 2000]. More elaborated views are defined over the default view using XQuery [FAN 2002].

In Oracle XSQL, there is a default mapping from SQL to XML. In this case, the mapping is rather simple and flat. More elaborated views can be generated using XSLT transformations [ORA 2001].

In DB2XML [TUR 99, TUR 2001], the views are constructed using SQL queries. It is possible to choose the root element name by specifying it between square brackets ("[]") in the SQL query.

SQL Server 2000 [CON 2001] allows XML views to be generated using SQL plus the for XML command, or using annotated XML Schema files. In these files, the user specifies how to map XML to the underlying relations.

In ODBC2XML [INT 2001], XML views are constructed using an XML template. The template specifies the SQL queries (through processing instructions) and the structure of the resulting XML document.

In SAX and DOM extensions [LAD 2001], only one type of SQL query is allowed. The API has a default SQL statement that selects all rows from a given table (SELECT * FROM TABLE). Different SQL statements can be obtained by overriding a specific method in the API and changing its implementation to return a different SQL query.

The approach proposed in [TAT 2001] supposes that XML documents are stored in a relational database according to an inline mapping that uses the document DTD to create database tables. The XML views are then built using XQuery. A query expressed in XQuery is mapped to SQL to retrieve the desired data (although the paper does not show how this mapping is done).

2 - Nesting of collections: In Oracle XSQL, SQL queries produce flat XML views. However, it's possible to apply XSLT transformations in these views to obtain *non-flat* views [ORA 2001].

DB2XML also produces flat views, but nested views can be obtained by using XSLT transformations. The same applies to SAX and DOM Extensions.

3 - Allows the use of several SQL statements: Complex views sometimes require several SQL statements. This can happen when the power of expression of SQL does not suffice to generate the view, or when the view requires the traversal of several 1:n relationships involving the same table. Consequently, tools that do no allow the use of several SQL statements have limited power of expression.

SQL Server allows using several SQL statements. In this case, the user can specify an XML template that batch together several XML producing commands (XPATH and/or for XML) to create complex documents.

Other tools that allow the use of several SQL statements are DB2XML, ODBC2XML and SilkRoute.

4 - Generates view schema: From the studied proposals, only two generates the schema of the XML view: DB2XML and DB/XML Transform. DB2XML works with DTDs, while DB/XML Transform works with both DTDs and XML Schema.

5 - View specification: In the comparison table, for each tool, the view specification is marked as P (procedural) or D (declarative).

In DB/XML Transform, the view specification is made through a graphical interface where the user builds an XML tree and maps it to the database. This proposal does not offer a specific query language in order to build the XML views. So, it can be classified neither as procedural nor as declarative.

In IBM DB2 XML Extender, the specification of an XML view is made through a mapping file named DAD (*Document Access Definition*). This map file is required and has to be explicitly written by the user or DBA.

The extension of SAX and DOM APIs works as a layer between the database and an XML application. They do not construct the view, but provide access to the database tables as if they were XML documents. However, the author proposes another extension that uses XSLT to obtain real XML documents. For this reason, this tool was classified as having a procedural view specification.

6 - Allows the choice of element's names: In all proposals it is possible to choose the elements name, except for the SAX and DOM extensions. In this case, the elements names are the same as the table and column names. DB2XML allow to choose just the name of root element [TUR 2001]. In this case, the remaining names are the same as the corresponding column names. This can be changed by specifying an XSLT transformation to be applied during the view generation.

7 - Intermediate language: Proposals that allow queries over the XML views generally do query composition, and uses an intermediate language to express the composed query. The query in the intermediate query language is then translated to SQL. Only tools classified in the first alternative perform query composition. However, despite being classified as a tool that follows the first alternative, the approach presented in [TAT 2001] does not allow the views to be queried, and so, does not use query composition.

8 - Allows queries over the XML views: Some proposals allow the generated XML views to be queried. They are Xperanto, SilkRoute, SAX and DOM APIs extensions and SQL Server.

9 - Query language used to query the XML view: As a new version of the XPeranto proposal, XTables [FAN 2002] uses the W3C standard XQuery to query the XML views (the previous version used XML-QL).

10 - Views on top of views: Xperanto allows the specification of views on top of views, that is, to use a view to define another view.

11 - Update Support: Tools that allow update through the XML views are Oracle XML-SQL Utility, IBM DB2 XML Extender, Microsoft SQL Server 2000 and [TAT 2001].

Oracle XML-SQL Utility allows the insertion of flat XML documents in the database through a built-in mapping. Generic XML documents need to be translated to a flat structure in order to be inserted in the database.

IBM DB2 XML Extender allows the update of XML documents with nested structure. An XML document may be mapped to several tables. The mapping between the XML document and the database has to be specified by the database administrator. Only INSERT operations are allowed through this interface. UPDATE and DELETE operations must be issued directly against the database using standard SQL statements.

Microsoft SQL Server 2000 allows the construction of complex XML documents from the data in the database. Its update approach uses *updatagrams*. Updatagrams work by presenting a before and after view of data that is marked-up using XML.

The approach presented in [TAT 2001] proposes an extension to the W3C standard XQuery language to provide updates. With this language extension, it is possible to execute insert, update and delete operations on an XML view. SQL statements may be generated to map these operations to the underlying relational database.

12 - Supports several databases: Some of the analyzed proposals are proprietary solutions that work only with a given database. This is the case of Oracle XML-SQL Utility (that works only with the Oracle database), IBM DB2 XML Extender (DB2) and SQLXML (SQL Server 2000). Xperanto is closely related to IBM DB2, since it uses an intermediate representation to the user queries called XQGM (XML Query Graph Model). Queries represented in XQGM are easily translated to QGM (Query Graph Model) used by DB2 [CAR 2000]. This choice of internal representation makes this proposal difficult to be adapted to other DBMSs.

The remaining proposals accept ODBC/JDBC connections. In this way, they work with any relational database that supplies ODBC/JDBC drivers.

2.4 Updating relational databases through views

Not every view that can be defined is updatable. The problem of identifying what views are updatable and what views are not is a classical problem in the database research area. Several researchers studied this problem, and established some features that a view must have in order to be updatable.

Any view can support read operations. However, since only base relations are actually stored, only base relations can be physically updated. To make an update via a view, it must be possible to propagate the updates down to the underlying base relations.

If the view is simple, then this propagation is straightforward. If the view is a one-to-one mapping of tuples in some base relation but some columns of the base are missing from the view, then update and delete present no problem, but insert requires that the unspecified (*invisible*) fields of the new tuples in the base relation be filled in with the *undefined* value. This may or may not be allowed by the integrity constraints on the base relation.

Beyond these very simple rules, propagation of updates from views to base relations becomes complicated, dangerous, and sometimes impossible. Views derived from join are not necessarily third normal form relations, and hence may have unpleasant update properties [CHA 75].

In literature, mechanisms to update relational databases through views are called *update translators*. [CHA 75, KEL 85, DAT 2000], among others, established rules that must be followed by update translators in order to correctly translate view update requests into database updates. Some of these rules are briefly presented here.

The most important rule concerns the main problem in view updates: ambiguity. Given a view update, there must be a unique operation which can be applied to the base relation and which will result in exactly the specified changes to the user's view [CHA 75, FUR 85]. More specifically, there should be no ambiguity.

The scope of the update translation is also defined by a rule. A view update must affect only information visible within the rectangle of the view [CHA 75]. No tuples outside the view must be affected by view updates, that is, there should be no side effects [KEL 85, FUR 85].

These are the two basic rules in view updates. However, it is necessary to analyze two more points: view updatability and update operations[DAT 2000].

View updatability is a semantic issue, not a syntactic one. In order to be considered updatable, a view must not be analyzed in terms of how it was syntactically defined. On the contrary, it must be analyzed in terms of how it was semantically defined. As an example, the following two definitions are semantically identical.

```
CREATE VIEW V (PROFCODE, PROFNAME, AGE, CITY)
AS SELECT PROFCODE, PROFNAME, AGE, CITY
   FROM PROFESSOR WHERE AGE > 25 OR CITY = 'Miami';

CREATE VIEW V (PROFCODE, PROFNAME, AGE, CITY)
AS (SELECT PROFCODE, PROFNAME, AGE, CITY
   FROM PROFESSOR WHERE AGE > 25)
   UNION
   (SELECT PROFCODE, PROFNAME, AGE, CITY
   FROM PROFESSOR WHERE CITY = 'Miami');
```

Since they have the same meaning, the two views must be considered both updatable, or both non-updatable. In fact, the SQL standard, and most of today's SQL products, adopt the *ad hoc* position that the first is updatable and the second is not [DAT 2000].

Regarding the update operations, the following must be observed. It is desirable to regard UPDATE as shorthand for a DELETE-INSERT sequence. The shorthand requires some slight refinement in the case of projection views, in order to avoid data loss [KEL 85, DAT 2000].

All updates on views must be implemented by the same kind of updates on the underlying tables. That is, INSERTs map to INSERTs and DELETE maps to DELETEs. UPDATEs can be ignored thanks to the previous point. INSERT and DELETE should be inverses of each other [DAT 2000].

Each database tuple should be affected by at most one step of the translation for any single view update request. Specifically, a translation cannot replace an inserted tuple, or delete a replaced tuple, or replace a tuple twice in succession [KEL 85].

The rules cannot assume that the database is well designed (e.g. fully normalized). However, they might on occasion produce a slightly surprising result if the database is *not* well designed [DAT 2000].

Additionally, the update rules must take into consideration any triggered procedures, including in particular referential actions such as cascade DELETE [DAT 2000].

The allowed update operations are also accomplished by the rules in literature. The main point is that there should be no reason for permitting certain types of updates but not others (e.g. DELETEs but not INSERTs) on a given view [DAT 2000].

Finally, a consideration must be made regarding transactions. If an attempt to update an underlying table fails for some reason, the original update will fail also, i.e., updates on views are all or nothing, just as updates on base tables are [DAT 2000].

The next chapter presents a proposal for updating relational databases through XML views. This proposal was defined by observing the features of NF^2 relations, updates through views and XML wrappers construction presented in this chapter.

Chapter 3

Two layered approach to update relational databases through XML views

This chapter presents a proposal for updating relational databases through XML views. The proposed solution includes a query language to build the XML views, and a wrapper architecture that implements this language. The query language proposed here, called R2X (**R**elations **to** **X**ML), was designed on top of a more generic language called uR2X (**u**nderlying **R**elations **to** **X**ML). We start with the R2X wrapper architecture. The uR2X language is presented in section 3.3 and the R2X language is introduced in section 3.4.

3.1 R2X Wrapper Architecture

The R2X wrapper (Figure 3.1) is a middleware between a *User Application* and a *Database Management System*. The Database Management System is an arbitrary Relational Database System. Its schema is mapped to XML, as explained later.

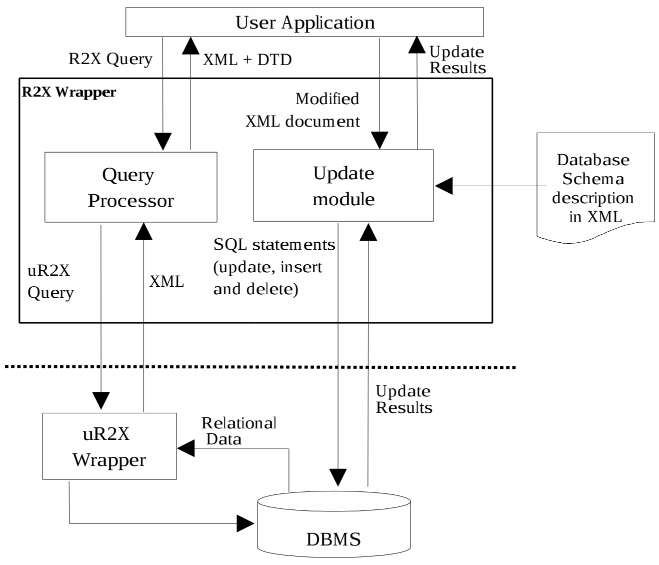


Figure 3.1: Two-layered Architecture

The User Application is any application that internally processes data in XML format and needs to access and update the underlying DBMS. To do this, the User Application submits an R2X query to the R2X wrapper. The query result (an XML view of the database) is then processed and modified by the user application and submitted back to the R2X wrapper. The R2X wrapper identifies the modifications performed by the application on the XML document and generates the corresponding INSERT, DELETE and UPDATE SQL statements. These statements are executed by the DBMS to reflect the modifications performed in the XML document. The R2X wrapper is composed of an *Update Module* and a *Query Processor*.

The Query Processor is composed of two modules: the *Translator* and the *ID and DTD Generator*, as shown in Figure 3.2. The Translator performs the mapping of an R2X query submitted by the user into the equivalent uR2X query. This uR2X query is processed by the uR2X Wrapper that outputs the query result in XML format. The ID and DTD Generator enrich the query result with information that will be needed by the Update Module in order to identify the modifications performed by the User Application over the query result.

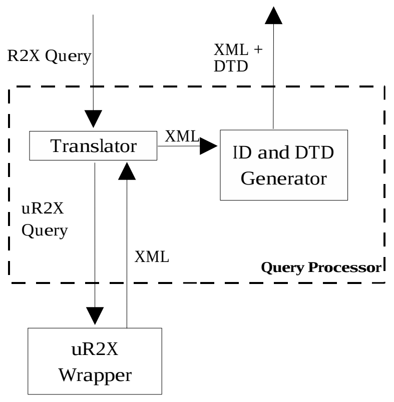


Figure 3.2: R2X Query Processor

In order to be capable of updating a relational database through XML Views, the R2X processor must know the underlying database schema. The database schema is described in XML. This XML description is also used when mapping R2X queries to uR2X queries.

For sake of simplicity, we use a very intuitive database description in XML. The XML description of the database of Figure 2.1 is shown in Appendix B. The database description holds information about data types, primary keys and foreign key constraints.

3.2 XML and NF^2

One of the design goals of uR2X is to allow the specification of XML instances with arbitrary nesting. In the database area the idea of nesting of data collections is not new and appeared in the non first normal form approaches (NF^2 or $\neg$1NF) [MAK 77, ABI 84, JAE 82, ELM 2000].

According to [ELM 2000], the nested relational model allows composite and multivalued attributes, thus leading to complex tuples with hierarchical structure. This hierarchical structure is similar to the XML structure. Thus, NF^2 tables may be represented as XML instances. The mapping is rather straightforward and is shown in section 3.2.2.

On the other side, some XML instances can be translated into an NF^2 table. An NF^2 table is a collection of tuples of the same type, whereas an XML element may be a collection of elements with different structure. However, for our purpose the NF^2 model suffices. We aim at the mapping to XML of collections of objects with the same structure as the ones contained in a relational database.

Section 3.2.1 shows the notation used in this work and section 3.2.2 shows the mapping of NF^2 relations to XML.

3.2.1 Notation

The notation adopted in this work is mainly based in [SIL 2002a], but it was simplified in order to prohibit a relation to have columns with distinct domains.

DEFINITION 1 *A **Relation Schema** RS is denoted by RS = (C_1:[d_1], C_2:[d_2], . . . , C_m:[d_m]), ($m \geq 1$), where C_i is called a column and d_i denotes one of the following: (i) and atomic value, represented by* atom*; (ii) a relation schema.*

As an example, consider the NF^2 table of Figure 3.3. This table contains data about departments and its employees. Each department is represented by a tuple. This tuple contains a nested table with the collection of employees of the department.

According to the above definition, the Relation Schema of Figure 3.3 is the following:

```
Departments =
      ( DeptCode: atom,
        DeptName:atom,
        Employees:
           [(EmpCode: atom,
             EmpName: atom)]
      )
```

DEFINITION 2 *Let RS = (C_1:[d_1], C_2:[d_2], . . . , C_m:[d_m]), ($m \geq 1$) be a relation schema. An **instance** R of RS denoted by R:RS is a set R = {<C_1:v_1^1, C_2:v_2^1, . . . , C_m:v_m^1>, . . . , <C_1:v_1^n, C_2:v_2^n, . . . , C_m:v_m^n>}, ($n \geq 0$), where v_k^j is one of the following: (i) an atomic value, if d_k is an* atom*; (ii) an instance of d_k, if d_k is a relation schema.*

With this definition, the relation shown in Figure 3.3 can be described as follows:

```
R = {<DeptCode: "1",
      DeptName: "Sales",
      Employees:
          {<EmpCode: "1",
            EmpName: "Bill">,
           <EmpCode: "2",
            EmpName: "Ann">,
           <EmpCode: "5",
            EmpName: "John">}
     >,
     <DeptCode: "2",
      DeptName: "Human Resources",
      Employees:
          {<EmpCode: "4",
            EmpName: "Marc">,
           <EmpCode: "6",
            EmpName: "Rachel">}
     >,
     <DeptCode: "3",
      DeptName: "Buys",
      Employees:
          {<EmpCode: "3",
            EmpName: "Peter">}
     >}
```

Departments

DeptCode	DeptName	Employees	
		EmpCode	EmpName
1	Sales	1	Bill
		2	Ann
		5	John
2	Human Resources	4	Marc
		6	Rachel
3	Buys	3	Peter

Figure 3.3: NF^2 table

3.2.2 Mapping Non First Normal Form Relations to XML

Nested relations have been used to represent semistructured data [SIL 2002a] and hierarchically organized data [ABI 84]. As the XML model is a tree [BRA 2000], mapping nested relations to XML is trivial.

The mapping is structured as follows:

DEFINITION 3 *Let RS be a relation schema, and R an instance of RS. RS's name is mapped to the root element in the XML document that corresponds to R.*

In this way, we assure that the XML document will have only one root. In the case of the example of Figure 3.3, the root element is `Departments`.

DEFINITION 4 *Let RS = (C_1:[d_1], C_2:[d_2], . . . , C_m:[d_m]) be a relation schema, and R = {<C_1:v_1^1, C_2:v_2^1, . . . , C_m:v_m^1>, . . . , <C_1:v_1^n, C_2:v_2^n, . . . , C_m:v_m^n>} an instance of R. For each v_k^j, there will be an XML element in the XML document that corresponds to R. The name of the XML element is C_k, and its content model is one of the following: (i) v_k^j, if d_k is an* atom*; (ii) a complex element corresponding to an instance of d_k, if d_k is a relation schema.*

According to this definition, the tuple <EmpCode: "1", EmpName: "Bill"> is represented in XML as follows:

```
<EmpCode>1</EmpCode>
<EmpName>Bill</EmpName>
```

DEFINITION 5 *Let RS be a relation schema, and R = {<C_1:v_1^1, C_2:v_2^1, . . . , C_m:v_m^1>, . . . , <C_1:v_1^n, C_2:v_2^n, . . . , C_m:v_m^n>} an instance of R. For each tuple in R, an XML element must be created. The name of this element is irrelevant, and can be set by the user.*

This is done in order to make it easier to identify each tuple in the XML document. In the previous example, an XML element must be inserted to enclose the Employee tuple. The element name was chosen to be Employee.

```
<Employee>
   <EmpCode>1</EmpCode>
   <EmpName>Bill</EmpName>
</Employee>
```

Considering the relation shown in Figure 3.3, the corresponding XML document is shown in Figure 3.4.

```
<Departments>
   <Department>
      <DeptCode>1</DeptCode>
      <DeptName>Sales</DeptName>
      <Employees>
        <Employee>
           <EmpCode>1</EmpCode>
           <EmpName>Bill</EmpName>
        </Employee>
        <Employee>
           <EmpCode>2</EmpCode>
           <EmpName>Ann</EmpName>
        </Employee>
        <Employee>
           <EmpCode>5</EmpCode>
           <EmpName>John</EmpName>
        </Employee>
      </Employees>
   </Department>
   <Department>
      <DeptCode>2</DeptCode>
      <DeptName>Human Resources</DeptName>
      <Employees>
         <Employee>
            <EmpCode>4</EmpCode>
            <EmpName>Marc</EmpName>
         </Employee>
         <Employee>
            <EmpCode>6</EmpCode>
            <EmpName>Rachel</EmpName>
         </Employee>
      </Employees>
   </Department>
   <Department>
      <DeptCode>3</DeptCode>
      <DeptName>Buys</DeptName>
      <Employees>
         <Employee>
            <EmpCode>3</EmpCode>
            <EmpName>Peter</EmpName>
         </Employee>
      </Employees>
   </Department>
</Departments>
```

Figure 3.4: XML document corresponding to the relation of Figure 3.3

3.3 uR2X

The uR2X language is an evolution of XML/SQL [VIT 2001]. XML/SQL has a specific syntax, whereas the syntax of uR2X relies on the XML Schema syntax [XML 2001].

An uR2X query is generated by a *User Application* (Figure 3.5) and submitted to an *uR2X Wrapper*. The wrapper transforms the uR2X query into SQL statements and submits them to a *relational DBMS*. The resulting tuples are then transformed to XML and sent back to the user application.

In the following sections, we will present uR2X through a series of examples. In these examples the sample database of Figure 2.1 is used.

3.3.1 An implementation model for uR2X

In this section the implementation model of uR2X is presented and justified. Usually one should present first a language and then its implementation. In our case we have chosen the reverse order, because the implementation aspects had strong influence on the language design.

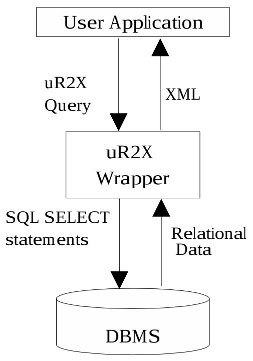


Figure 3.5: uR2X Wrapper

Considering the relation between NF^2 and XML, one approach to the mapping from relational databases to XML could be the use of a database engine capable of executing NF^2 operators. Such an engine should implement at least the operators of the classical relational algebra as well as the NF^2-specific *nest* operator [JAE 82, THO 86].

However, there are no DBMSs that implement efficiently the NF^2 operators. As we are aiming at efficiency of query execution, we prefer an implementation method that relies on relational database engines widely available.

The implementation method we propose comprises the following steps:

1. A query using classical (not NF^2) operators is executed over a relational DBMS obtaining the data needed to build the XML instance. The result of this query is called here *base view*. As this query is executed by the relational database engine the optimization algorithms implemented in this engine are used.

2. Over the flat base view, nest operations are applied to obtain the required nesting. These operations are executed by the uR2X wrapper. We suppose that the base view is ordered by the columns that will be used to group two tuples together. If the base view is ordered in this way, the wrapper simply scans the base view and builds the NF^2 view. Notice that even when there are several levels of nesting, this approach can be used. As an example, consider the query:

$$\nu_{Dependents=(Employees.DepCode,Employees.DepName)}$$
$$(\nu_{Employees=(EmpCode,EmpName,DepCode,DepName)}$$
$$(\pi_{(DeptCode,DeptName,EmpCode,EmpName,DepCode,DepName)}$$
$$(Department * Employee * Dependent)))$$

In this case, the two nest operations may be performed during a single table scan as long as the base view is adequately ordered.

3. The NF^2 view generated in step 2) is transformed into an XML instance. Actually, as the mapping from the NF^2 view to the XML instance is straightforward, the construction of this instance may occur directly in the previous step.

This approach still has a problem in the case that two nested tables are required at the same level of nesting in the NF^2 view. Consider the example of Figure 3.6. This table contains two nested tables (DeptEquipment and DeptEmployees) at the same nesting level.

Departments

DeptCode	DeptName	DeptEquipment		DeptEmployees	
		EquipCode	EquipName	EmpCode	EmpName
1	Sales	1	Fax	1	Bill
				2	Ann
				5	John
2	Human Resources	2	Telephone	4	Marc
		3	Pentium 950MHZ	6	Rachel
3	Buys	4	Fax	3	Peter
		5	Copier Machine		

Figure 3.6: NF^2 table with two nested tables at the same nesting level

Departments

DeptCode	DeptName	EquipCode	EquipName	EmpCode	EmpName
1	Sales	1	Fax	1	Bill
1	Sales	1	Fax	2	Ann
1	Sales	1	Fax	5	John
2	Human Resources	2	Telephone	4	Marc
2	Human Resources	2	Telephone	6	Rachel
2	Human Resources	3	Pentium 950MHZ	4	Marc
2	Human Resources	3	Pentium 950MHZ	6	Rachel
3	Buys	4	Fax	3	Peter
3	Buys	5	Copier Machine	3	Peter

Figure 3.7: Base view that holds data to construct the NF^2 view of Figure 3.6

In order to obtain this NF^2 table the following relational algebra query could be used.

$$\nu_{DeptEquipment=(EquipCode,EquipDescription)}$$
$$(\nu_{DeptEmployees=(EmpCode,EmpName)}$$
$$(\pi_{(DeptCode,DeptName,EquipCode,EquipDescription,EmpCode,EmpName)}$$
$$(Department * Equipment * Employee)))$$

The base view for this example (the part of the above expression that uses solely classical algebra operators) is presented in Figure 3.7. This view is not in fourth normal form. Due to this redundancy of data, in step 2) this example would require several scans over the base view or the storage of intermediate results.

Our objective is to use as much as possible the optimization methods of the relational database engine and implement the nest operator during a single table scan of the base view. For this reason, the implementation method of an uR2X query described previously is modified as follows:

1. Several base views are generated. Consider the NF^2 view of Figure 3.6. The tree that represents the nesting of tables in this figure can be graphically represented as shown in Figure 3.8. In our approach, for each table that appears as a leaf of the tree, there will be one classical algebra query. In the example, there will be two algebra queries, each one generating one base view. These queries are:

$$\pi_{DeptCode,DeptName,EquipCode,EquipDescription}$$
$$(Department * Equipment)$$

$$\pi_{DeptCode,DeptName,EmpCode,EmpName}$$
$$(Department * Employee)$$

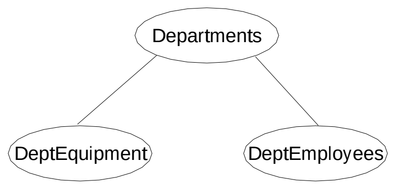


Figure 3.8: Tree that represents the structure of the nested tables in Figure 3.6

The base views resulting from these two queries are shown in Figure 3.9. The first base view has data about Departments and Equipment, and the second one shows Departments and Employees. Notice that each query has to retrieve data about the whole branch of the tree. Also, these queries must deliver the data in the order of the nest operator criteria, beginning with the outermost table. In the example, in the first base view the data is ordered by DeptCode and EquipCode, and in the second base view the data is ordered by DeptCode and EmpCode.

2. The wrapper scans each base view and combines the tuples in order to produce the NF^2 view and in consequence the XML instance. This operation acts as a modified *merge-join* in the involved tables [RAM 2001, SIL 2002]. It not only merges, but also applies nest operations on the base view to produce the NF^2 view. In the case of the example, this operation will merge the tuples that represent the same department, producing the NF^2 view of Figure 3.6.

   The implementation of this operation is as follows.

   The order of execution of the *nest* operator is not relevant. However, the scanning starts with the base view that holds information about departments and equipment (from now on called *first base view*), because equipment will appear before employees in the resulting XML view.

   The tuples of the base views involved in this operation are scanned sequentially. The first base view is scanned and the values of the first Department (DeptCode=1 and DeptName=Sales) are output. The tuples with "DeptCode=1 and DeptName=Sales" are scanned and the values of their columns are output producing the nested tuples corresponding to this department equipment. All the equipment belonging to the first department is output. However, before outputting an equipment, it is necessary to know if it belongs to the same department as the previous one. To do this, the next tuple is compared to the current tuple to check if the DeptCode and DeptName values (DeptCode and DeptName are the columns that were not specified in the *nest* operator) are the same. If yes, the next tuple is scanned, and its equipment output. When a new value of DeptCode+DeptName is found, the scanning starts on the second base view, producing the nested tuples corresponding to the department employees. When a new value of DeptCode+DeptName is found, the scanning in the first base view is resumed. This process continues until the two base views are completely scanned.

   More specifically, the type of join that is executed is a *full outer join*. As an example, consider the NF^2 view of Figure 3.6. Consider again that there is a new department in the database. This department has equipments but no employees were hired yet. In this case, the department would appear in the view, and the employee column would be empty.

DeptEquipment

DeptCode	DeptName	EquipCode	EquipDescription
1	Sales	1	Fax
2	Human Resources	2	Telephone
2	Human Resources	3	Pentium 950MHZ
3	Buys	4	Fax
3	Buys	5	Copier Machine

DeptEmployees

DeptCode	DeptName	EmpCode	EmpName
1	Sales	1	Bill
1	Sales	2	Ann
1	Sales	5	John
2	Human Resources	4	Marc
2	Human Resources	6	Rachel
3	Buys	3	Peter

Figure 3.9: Two base views that hold data to construct the NF^2 view of Figure 3.6

3.3.2 The uR2X syntax

This section presents the syntax of the uR2X language that implements the model shown in the previous section.

Basic syntax

An uR2X query is a valid XML document, according to the uR2X DTD presented in Appendix A.1. The root element of an uR2X query is the uR2X element.

Considering the implementation method presented in the previous section an uR2X query is composed of two clauses, the QUERY clause and the CONSTRUCT clause. The QUERY clause (represented by the query element) consists of a set of SQL statements.

Each SQL statement corresponds to a *base view*. Each SQL statement is enclosed within an sql element.

The CONSTRUCT clause (represented by the construct element) contains the specification of the XML document to be created from the base views. This part of the query specifies declaratively the join of the various base views, the nest operators that are applied to the base views as well as the mapping from the NF^2 table to the XML instance.

```
<Departments>
   <Department>
      <Dept>1</Dept>
      <Name>Sales</Name>
      <Acronym>S</Acronym>
   </Department>
   <Department>
      <Dept>2</Dept>
      <Name>Human Resources</Name>
      <Acronym>HR</Acronym>
   </Department>
   <Department>
      <Dept>3</Dept>
      <Name>Buys</Name>
      <Acronym></Acronym>
   </Department>
</Departments>
```

Figure 3.10: XML instance to be obtained from the first uR2X query example

In order to explain how an uR2X query is built, we begin with a simple example of a query over the example database. Suppose that the XML instance shown in Figure 3.10 is to be obtained from the example database. This instance is a list of Department elements. Each Department element corresponds to a department tuple. A Department element is composed of a code (Dept), a name (Name) and an acronym (Acronym).

As mentioned above an uR2X query is composed of two parts, the SQL queries and the CONSTRUCT clause. In the case of the example there are no nested collections. Thus a single base view is required. This base view is obtained by the SQL query that appears in the QUERY clause of the uR2X query of Figure 3.11. In this query, the SQL statement generates a base view with schema v1(DeptCode, DepName, Acr). The viewName attribute of the sql element identifies this base view in the uR2X query.

```
<uR2X>
  <query>
    <sql viewName="v1">SELECT d.DeptCode, d.DeptName, d.Acr
                       FROM Department d
    </sql>
  </query>
  <construct>
    <tableElement name="Departments" source="v1">
      <tupleElement name="Department">
        <atomElement name="Dept" source="DeptCode"/>
        <atomElement name="Name" source="DeptName"/>
        <atomElement name="Acronym" source="Acr" mandatory="yes"/>
      </tupleElement>
    </tableElement>
  </construct>
</uR2X>
```

Figure 3.11: uR2X query that results in an XML document with flat structure

In the CONSTRUCT clause, the resulting XML instance is specified using three uR2X *constructors*: TABLE, TUPLE and ATOM.

TABLE constructor: The TABLE constructor defines an element in the XML instance that corresponds to a table (possibly nested) in the NF^2 view. In an uR2X query, this constructor is represented by an element tableElement. Elements of this type are called *table elements*.

TUPLE constructor: The TUPLE constructor defines an element in the XML instance that corresponds to a tuple in the NF^2 view. In an uR2X query this constructor is represented by an element tupleElement. Elements of this type are called *tuple elements*.

ATOM constructor: The ATOM constructor defines an element in the XML instance that corresponds to a column value in the NF^2 view. In an uR2X query this constructor is represented by an element atomElement. Elements of this type are called *atomic elements*.

The construct clause has three purposes:

- specify the nest operators that build the NF^2 view from the base views;
- specify how the several base views used in an uR2X query are joined together and
- specify the mapping from the NF^2 view to the XML instance.

The name and source attributes of the constructors of an uR2X query specify the mapping from the NF^2 view to the XML instance. The data source specified by the source attribute depends on the type of constructor. Each TABLE constructor must specify the base view that contains the tuples that will be used to construct the *table element*. Each ATOM constructor must specify the name of the column that contains the values that will be used to construct the *atomic element*. The name attribute of an uR2X constructor specify the corresponding element names in the XML instance.

Nesting collections

The result of the previous example query (Figure 3.11) is flat, i.e., does not contain nested collections and is built from a single base view. Thus, neither nest operations nor join operations are required. The next example query (Figure 3.12) shows the specification of an XML instance that contains nested collections. In this case, an NF^2 view must be constructed. The XML instance to be obtained is shown in Figure 3.4. The nesting of collections in this query affects the following clauses:

- The CONSTRUCT clause contains nested TABLE constructors.
- Each TABLE constructor with the exception of the leaf-most has a nestBy attribute. The nestBy attribute specifies a nest operation. This attribute contains a list of column names. The nest operation builds a partition of the base view for each value of these columns.

The effect of the execution of this query is as follows. The base view is obtained from the SQL query specified in the QUERY clause. Over this base view, a nest operation is performed. The base view is sequentially scanned. The tuples that has the same value of DeptCode (the column name specified in the nestBy attribute) are grouped together, creating nested tuples containing the employees of the department. The mapping of this NF^2 view to XML is performed as explained in the previous example.

```
<uR2X>
  <query>
    <sql viewName="v2">SELECT d.DeptCode, d.DeptName, e.EmpCode, e.EmpName
                       FROM Department d, Employee e
                       WHERE d.DeptCode = e.DeptCode
                       ORDER BY d.DeptCode
    </sql>
  </query>
  <construct>
    <tableElement name="Departments" source="v2" nestBy="DeptCode">
      <tupleElement name="Department">
        <atomElement name="DeptCode" source="DeptCode"/>
        <atomElement name="DeptName" source="DeptName"/>
        <tableElement name="Employees" source="v2">
          <tupleElement name="Employee">
            <atomElement name="EmpCode" source="EmpCode"/>
            <atomElement name="EmpName" source="EmpName"/>
          </tupleElement>
        </tableElement>
      </tupleElement>
    </tableElement>
  </construct>
</uR2X>
```

Figure 3.12: Nesting of collections

As this query has a single base view, no *merge-join* operation is needed. To explain this operation, a further example is introduced.

```
<Departments>
  <Department>
    <DeptCode>1</DeptCode>
    <Name>Sales</Name>
    <DeptEquipment>
      <Equipment>Fax</Equipment>
    </DeptEquipment>
    <DeptEmployees>
      <Employee>
        <EmpCode>1</EmpCode>
        <Name>Bill</Name>
        <EmpDependents>
        </EmpDependents>
      </Employee>
      <Employee>
        <EmpCode>2</EmpCode>
        <Name>Ann</Name>
        <EmpDependents>
          <Dependent>Catherine</Dependent>
        </EmpDependents>
      </Employee>
      <Employee>
        <EmpCode>5</EmpCode>
        <Name>John</Name>
        <EmpDependents>
          <Dependent>Marc</Dependent>
        </EmpDependents>
      </Employee>
    </DeptEmployees>
  </Department>
  <Department>
    <DeptCode>2</DeptCode>
    <Name>Human Resources</Name>
    ...
</Departments>
```

Figure 3.13: XML instance with two collections at the same level (DeptEquipment and DeptEmployees)

Several base views

Figure 3.13 shows an XML instance obtained by the uR2X query of Figure 3.14. This XML instance contains two nested collections at the same level. DeptEquipment and DeptEmployees are nested into a Department tuple. In this case, two base views are required. Therefore, the QUERY clause contains two SQL statements that generate these base views.

These two base views must be joined in order to produce the NF^2 view. This occurs as follows. The base views are scanned sequentially. The modified *merge-join* (section 3.3.1) operation is applied. Tuples corresponding to the same department are joined together, building the required NF^2 view. The nest operators are executed simultaneously with the merge-join operation during the table scan of the several base views. Thus in the example query, the merge join of the two base views is executed at the same table scan as the nest of equipment into departments (on base view “v3”) and the nest of employees into departments and dependents into employees (on base view “v4”).

Relation to XML Schema

As mentioned before, the uR2X syntax was inspired in XML Schema [XML 2001]. Additional attributes were introduced to specify the information required by the uR2X wrapper that is not present in XML Schema. The attributes that were introduced specify the data source in the relational databases, as well as the nest operations. Further, instead of using a single element constructor to specify any type of element, we use different constructors,

```
<uR2X>
  <query>
    <sql viewName="v3">SELECT d.DeptCode, d.DeptName, eq.EquipDescription
                        FROM Equipment eq, Department d
                        WHERE d.DeptCode = eq.DeptCode
                        ORDER BY d.DeptCode
    </sql>
    <sql viewName="v4">SELECT d.DeptCode, e.EmpCode, e.EmpName,
                        dep.DepName
                        FROM Department d LEFT JOIN
                          (Employee e LEFT JOIN Dependent dep
                                ON e.EmpCode = dep.EmpCode)
                          ON d.DeptCode=e.DeptCode
                        ORDER BY d.DeptCode, e.EmpCode
    </sql>
  </query>
  <construct>
    <tableElement name="Departments" source="v3" nestBy="DeptCode">
      <tupleElement name="Department">
        <atomElement name="Name" source="DeptName"/>
        <tableElement name="DeptEquipment" source="v3">
          <atomElement name="Equipment" source="EquipDescription"/>
        </tableElement>
        <tableElement name="DeptEmployees" source="v4" nestBy="EmpCode">
          <tupleElement name="Employee">
            <atomElement name="EmpCode" source="EmpCode"/>
            <atomElement name="Name" source="EmpName"/>
            <tableElement name="EmpDependents" source="v4">
              <atomElement name="Dependent" source="DepName"/>
            </tableElement>
          </tupleElement>
        </tableElement>
      </tupleElement>
    </tableElement>
  </construct>
</uR2X>
```

Figure 3.14: uR2X query: two base views

atomElement, tupleElement or tableElement, depending on the type of element being declared. This syntax change is needed because each type of constructor requires specific attributes. In XML Schema, a simple element DeptCode is specified as:

```
<element name="DeptCode" type="integer"/>
```

In uR2X, this element is constructed by an ATOM constructor as follows:

```
<atomElement name="DeptCode" source="DeptCode"/>
```

In the same way, a complex element Department, composed of a department code (DeptCode) and a department name (Name), is specified in XML Schema as follows:

```
<element name="Department">
  <complexType>
    <sequence>
      <element name="DeptCode" type="integer"/>
      <element name="Name" type="string"/>
    </sequence>
  </complexType>
</element>
```

In uR2X, this complex element is specified by a TUPLE constructor as:

```
<tupleElement name="Department">
  <atomElement name="DeptCode" source="DeptCode"/>
  <atomElement name="Name" source="DeptName"/>
</tupleElement>
```

The complexType and sequence elements were eliminated in order to simplify the syntax, and the element constructor was changed to the corresponding one in uR2X.

Similarly, the TABLE constructor is also derived from a complexType in XML Schema.

Concluding example

In this section a further example of an uR2X query is presented. This example illustrates the fact that SQL clauses like aggregate functions, GROUP BY clauses, complex JOIN operations, and so on may be used. Further, the example shows that, due to lack of expressive power of SQL, in some cases, several base views are required.

The query shown in Figure 3.15 returns a single tuple containing the total number of faxes in the database as well as a list of Departments that has faxes. The result of this query is shown in Figure 3.16.

It is important to notice that in this case, two base views were used because it was impossible to obtain the information needed by the query using a single SQL statement. So, additional base views are necessary not only in the presence of several collections at the same nesting level, but also when it is not possible to obtain the data needed by the query through a single SQL statement.

3.4 R2X

As observed in the introduction, uR2X was designed with the intention of serving as a base layer for the implementation of more user-friendly relational to XML languages. In this section we present the R2X language, which is implemented using uR2X.

R2X provides a way of building *updatable* XML views of relational databases. Through the R2X wrapper a user application may declaratively specify XML views over a relational database and update the views directly. At the end of the transaction the user application passes the modified XML instance back to the R2X wrapper. The R2X wrapper maps the updates performed on the XML view to updates on the relational database.

```
<uR2X>
   <query>
      <sql viewName="v5"> SELECT EquipDescription, COUNT(EquipCode) as NumEquipment
                  FROM Equipment
                  WHERE EquipDescription='Fax'
                  GROUP BY EquipDescription
                  ORDER BY EquipDescription
      </sql>
      <sql viewName="v6"> SELECT EquipDescription, DeptName
                  FROM Equipment e, Department d
                  WHERE d.DeptCode=e.DeptCode
                  AND e.EquipDescription='Fax'
                  ORDER BY EquipDescription
      </sql>
   </query>
   <construct>
      <tableElement name="Equipment" source="v5" nestBy="EquipDescription">
         <tupleElement name="FaxDepartments">
            <atomElement name="NumEquipment" source="NumEquipment"/>
            <tableElement name="Departments" source="v6" >
               <atomElement name="DeptName" source="DeptName"/>
            </tableElement>
         </tupleElement>
      </tableElement>
   </construct>
</uR2X>
```

Figure 3.15: uR2X query

```
<Equipment>
   <FaxDepartments>
      <NumEquipment>2</NumEquipment>
      <Departments>
         <DeptName>Sales</DeptName>
         <DeptName>Buys</DeptName>
      </Departments>
   </FaxDepartments>
</Equipment>
```

Figure 3.16: Query result

3.4.1 R2X view definition

One of the design goals of uR2X was to provide a powerful query language. With the uR2X language XML views with arbitrary structure and nesting may be constructed from arbitrary SQL queries. Not every view that can be defined by an uR2X query is updatable, due to the *view ambiguity problem* [DAT 2000]. R2X restricts the constructed views to those which allow the mapping of view update operations to relational database update operations. Besides that, the constructed view is defined declaratively as opposed to uR2X where SQL statements are used.

Figure 3.17 presents an example of an R2X query and Figure 3.18 presents the result of the execution of this query against the example database of Figure 2.1. This query retrieves a list of Departments, and for each Department, the list of its Employees. The query retrieves, for each Department, the DeptCode and DeptName, and for each Employee, the EmpCode and EmpName.

The syntax of R2X is based on the syntax of uR2X. The R2X constructors have the same name as the constructors of uR2X. However, their semantics is different.

TABLE constructor: Each TABLE constructor refers to a table in the database. The TABLE constructor constructs an XML element that contains a list of elements, one for each tuple of the base table. It is represented in the R2X query by a tableElement.

```
<R2X>
   <tableElement name="Departments" source="Department" alias="d"
              filter="d.DeptCode=1 or d.DeptCode=2">
      <tupleElement name="Department">
         <atomElement name="DeptCode" source="DeptCode"/>
         <atomElement name="DeptName" source="DeptName"/>
         <tableElement name="Employees" source="Employee" alias="e" join="EmpDept">
            <tupleElement name="Employee">
               <atomElement name="EmpCode" source="EmpCode"/>
               <atomElement name="EmpName" source="EmpName"/>
            </tupleElement>
         </tableElement>
      </tupleElement>
   </tableElement>
</R2X>
```

Figure 3.17: R2X query: Departments and Employees

```
<Departments>
   <Department id="1">
      <DeptCode> 1 </DeptCode>
      <DeptName> Sales </DeptName>
      <Employees>
         <Employee id="2">
            <EmpCode> 1 </EmpCode>
            <EmpName> Bill </EmpName>
         </Employee>
         <Employee id="3">
            <EmpCode> 2 </EmpCode>
            <EmpName> Ann </EmpName>
         </Employee>
         <Employee id="4">
            <EmpCode> 5 </EmpCode>
            <EmpName> John </EmpName>
         </Employee>
      </Employees>
   </Department>
   <Department id="5">
      <DeptCode> 2 </DeptCode>
      <DeptName> Human Resources </DeptName>
      <Employees>
         <Employee id="6">
            <EmpCode> 4 </EmpCode>
            <EmpName> Marc </EmpName>
         </Employee>
         <Employee id="7">
            <EmpCode> 6 </EmpCode>
            <EmpName> Rachel </EmpName>
         </Employee>
      </Employees>
   </Department>
</Departments>
```

Figure 3.18: Result of R2X query

The name of the referred table is given by the source attribute. In the example of Figure 3.17, the outermost TABLE constructor refers to the Department table.

TUPLE constructor: The TUPLE constructor specifies the XML elements that are constructed one for each tuple of the table referred by the embedding TABLE constructor. The TUPLE constructor specifies a composite XML element. This composite element contains one child element for each constructor contained in the TUPLE constructor. It is represented in the query by a tupleElement.

ATOM constructor: The ATOM constructor specifies an atomic XML element that contains the value of a column in the database. The ATOM constructor is represented by an atomElement in the R2X query. The name of the referred column is given by the source attribute.

The outermost constructor of an R2X query is always a TABLE constructor. A TABLE constructor contains a single TUPLE constructor. A TUPLE constructor may contain ATOM and nested TABLE constructors (the R2X DTD is shown in appendix A.2).

Associated to each TABLE constructor, a filter clause may be specified. This clause is a predicate that specifies which tuples of the source table will generate elements in the output. The filter clause is represented in the R2X query by the filter attribute. 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 outermost TABLE constructor (d.DeptCode =1 or d.DeptCode=2) specifies that only those tuples of the Department with DeptCode equal to 1 or 2 will be selected by the query.

In the example query, the Employees element is constructed by the nested TABLE constructor. A nested TABLE constructor must refer to a table that is related by a foreign key constraint to the table referred by the parent TABLE constructor. In the example, the nested TABLE constructor refers to the Employee table and the parent TABLE constructor refers to the Department table. The Employee table is related to the Department table by the "EmpDept" foreign key constraint (Figure 2.1). The foreign key constraint used to relate both tables is specified by the join attribute of the nested TABLE constructor (join attributes appear only in nested TABLE constructors).

The three constructors of R2X (TABLE, TUPLE and ATOM) are similar to those used in uR2X queries. However, in order to guarantee that the XML instance produced by an R2X query will be updatable, it is necessary that each element corresponds 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 tuple elements inside a *table element* must contain the primary key of the table. Further, for each nested TABLE the columns referred in the foreign key used to build the nested table must appear in the output XML instance.

Observing these restrictions, one can say that in R2X, a TABLE constructor represents a database table, a TUPLE constructor is limited to represent a database tuple and an ATOM constructor always represents a database column. This can be clearer by analyzing the XML view of Figure 3.16. In this XML view, the element NumEquipment does not correspond to a database column. It is an aggregate. Also, the nested Departments does not represent a database table, neither does FaxDepartments. Consequently, they do not obey a further restriction: they are not related by a foreign key constraint. For all these reasons, this XML view can not be constructed by an R2X query.

By not allowing SQL statements, we guarantee important features such that tables will be joined only through foreign key columns, there will be no aggregate functions neither GROUP BY clauses in the queries. By assuring these features we give the first step in producing only updatable views.

3.4.2 An execution model for R2X

The execution model used in R2X is the same of uR2X. We use NF^2 views to represent the result of an R2X query. In this section, we explain how to obtain an algebra query from an R2X query.

The algebra query that corresponds to the R2X query of Figure 3.17 is the following.

$$\begin{aligned}
&\nu_{Employees=(EmpCode,EmpName)} \\
&\quad (\pi_{(d.DeptCode,d.DeptName,e.EmpCode,e.EmpName)} \\
&\quad\quad (\sigma_{(d.DeptCode=1 \text{ or } d.DeptCode=2)} \\
&\quad\quad\quad ((\rho\ d\ (Department)) \bowtie_{(d.DeptCode=e.DeptCode)} (\rho\ e\ (Employee)))))
\end{aligned}$$

This algebra query was obtained by processing the R2X query in the following way:

1. The source tables of the TABLE constructors were renamed (ρ) to the alias attribute of the corresponding TABLE constructor. In the example, the Department table was renamed to "d" and the Employee table was renamed to "e".

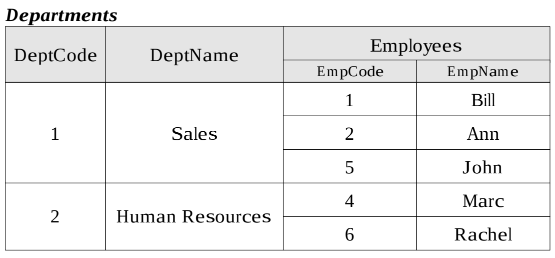
2. The source tables of the TABLE constructors were joined ($\bowtie$). The join criteria is written with the information obtained by the foreign key constraint specified in the join attribute. The R2X wrapper has access to the database schema (described in XML) and obtains the join criteria by consulting this description (appendix B).

3. The filter of each TABLE constructor is used in a selection operation (σ).

4. Over the result of this selection, a projection (π) is made. This projection involves each column name specified in the source attribute of the ATOM constructors of the query. Besides that, each column name is qualified with the alias of the corresponding TABLE constructor.

5. The nest operation (ν) is specified using the nested TABLE constructor. In the case of the example, the nest operation specifies a nested table Employees (name specified in the name attribute of the nested TABLE constructor) with columns EmpCode and EmpName (columns specified by the ATOM constructors of this TABLE constructor).

This query produces the NF^2 view shown in Figure 3.19.



Departments

DeptCode	DeptName	Employees	
		EmpCode	EmpName
1	Sales	1	Bill
		2	Ann
		5	John
2	Human Resources	4	Marc
		6	Rachel

Figure 3.19: NF^2 view resulting from the R2X query of Figure 3.17

The mapping to XML of the NF^2 view produced by an R2X query differs from the mapping used in uR2X in the following aspects. In the resulting XML instance the elements constructed by the TUPLE constructor have an id attribute. This attribute is automatically generated by the R2X wrapper (see section 3.5.1) and uniquely identifies each tuple in the constructed instance. This attribute is required for the update mechanism that will be

```
<R2X>
  <tableElement name="Departments" source="Department" alias="d">
    <tupleElement name="Department">
      <atomElement name="DeptCode" source="DeptCode"/>
      <atomElement name="DeptName" source="DeptName"/>
      <tableElement name="Equipment" source="Equipment" join="EquipDept" alias="eq">
        <tupleElement name="Equip">
          <atomElement name="EquipCode" source="EquipCode"/>
          <atomElement name="EquipDescription" source="EquipDescription"/>
        </tupleElement>
      </tableElement>
      <tableElement name="Employees" source="Employee" join="EmpDept" alias="e">
        <tupleElement name="Employee">
          <atomElement name="EmpCode" source="EmpCode"/>
          <atomElement name="EmpName" source="EmpName"/>
          <tableElement name="Dependents" source="Dependent" join="DepEmp" alias="dep">
            <tupleElement name="Dependent">
              <atomElement name="DepCode" source="DepCode"/>
              <atomElement name="DepName" source="DepName"/>
              <atomElement name="BirthDate" source="BirthDate"/>
            </tupleElement>
          </tableElement>
        </tupleElement>
      </tableElement>
    </tupleElement>
  </tableElement>
</R2X>
```

Figure 3.20: R2X query: Departments, Equipment and Employees

explained in section 3.4.3. The XML instance corresponding to the NF^2 view of Figure 3.19 is shown in Figure 3.18.

In R2X, it is also possible to have more elaborated nesting structures. Figure 3.20 shows an R2X query that has two TABLE constructors at the same nesting level. In this case, for each Department, the query shows a list of its Equipment and a list of its Employees. Besides that, for each Employee, a list of Dependents is shown.

The equivalent algebra query is as follows:

$$\begin{array}{l}
\nu_{Equipment=(EquipCode,EquipDescription)} \\
\quad (\nu_{Dependents=(Employees.DepCode,Employees.DepName,Employees.BirthDate)} \\
\quad\quad (\nu_{Employees=(EmpCode,EmpName,DepCode,DepName,BirthDate)} \\
\quad\quad\quad (\pi_{(d.DeptCode,d.DeptName,eq.EquipCode,eq.EquipDescription,} \\
\quad\quad\quad\quad {}_{e.EmpCode,e.EmpName,dep.DepCode,dep.DepName,dep.BirthDate)} \\
\quad\quad\quad\quad\quad ((((\rho\ d\ (Department)) \\
\quad\quad\quad\quad\quad\quad \bowtie_{(d.DeptCode=eq.DeptCode)}\ (\rho\ eq\ (Equipment))) \\
\quad\quad\quad\quad\quad\quad \bowtie_{(d.DeptCode=e.DeptCode)}\ (\rho\ e\ (Employee))) \\
\quad\quad\quad\quad\quad\quad \bowtie_{(dep.EmpCode=e.EmpCode)}\ (\rho\ dep\ (Dependent))))))
\end{array}$$

As mentioned in section 3.3.1, this kind of query is not efficient because it would require the storage of intermediate results or several scans of the base view. For this reason, in R2X we use several base views, just as in uR2X. We also use SQL in the implementation to make use of the DBMSs optimization algorithms. In fact, R2X is implemented over uR2X, that is, an R2X query is translated to an uR2X query, and then executed by the uR2X wrapper (see section 3.5 for further details on this translation).

Despite restricting the structure of the produced NF^2 view in order to allow updates, R2X is still a powerful query language. It has the same query power of expression of uR2X (but less structuring power). This is because the filter attribute of the TABLE constructor allows any search condition of SQL SELECT command (the syntax of a filter expression is the same as the SQL SELECT command). As an example, the outermost TABLE constructor of the R2X query of Figure 3.17 could specify a filter like d.DeptCode IN (SELECT distinct DeptCode FROM Equipment).

The XML instance that results from an R2X query contains two types of elements: *updatable elements* and *non-updatable elements*. In the next section the update operations that may be performed on the result of an R2X query will be explained and the difference between both types will be clear.

3.4.3 R2X update operations

The Update Module of the R2X Wrapper identifies the operations made by a user application in the XML instance by comparing the original instance and the modified one. The elements are identified by the id attribute. Each *tuple element* contains an id attribute that identifies it in the instance. As the *tuple element* corresponds to a database tuple, it is easy to determine what happened to a tuple after the modifications made by the user application. In this way, it's possible to know if a tuple was deleted, updated, moved or inserted.

The only concern that the user application must have when manipulating the XML view is that ids must not be reused. In fact, the user application does not need to change any id. It does not need to create the id when inserting a tuple in the view, because in this case, the id attribute is optional. The id is the identification of the tuples, serving as "key" in the comparison algorithm. The sequential order of its values is not relevant to the algorithm. The update algorithm is based on the fact that an id is unique in an XML instance, and once set to a certain tuple, it is not changed by the user application.

These ids are generated by the *ID and DTD Generator* module, explained in section 3.5.1.

An XML instance resulting from an R2X query can be manipulated an updated by a user application. There are four types of operations that can be performed:

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 the value of an atomic value.

MOVE: A move operation excludes a sub-tree consisting of a *tuple element* and its child elements and inserts it in a different location at the same nesting level.

The instance shown in Figure 3.21 is the result of an insertion operation in the example instance of Figure 3.18. The considered operation inserts the elements shown in bold face in Figure 3.21. Thus, this operation inserts the Department named Marketing, as well as two employees of this department: Roger and Claire.

This insertion must be reflected in the database by inserting a tuple in the Department table corresponding to the Marketing department. Besides that, two tuples must be inserted in the Employee table, one for each employee inserted in the XML instance. This is done by the following SQL statements:

```
INSERT INTO Department (DeptCode, DeptName) VALUES (4, 'Marketing')
INSERT INTO Employee (EmpCode, DeptCode, EmpName) VALUES (7, 4, 'Roger')
INSERT INTO Employee (EmpCode, DeptCode, EmpName) VALUES (8, 4, 'Claire')
```

Notice that the value of the Employee.DeptCode column is inferred from the fact that an Employee element is nested in a Department element.

```
<Departments>
  <Department id="1">
    <DeptCode> 1 </DeptCode>
    <DeptName> Sales </DeptName>
    <Employees>
    ...
    </Employees>
  </Department>
  ...
  <Department>
    <DeptCode>4</DeptCode>
    <DeptName>Marketing</DeptName>
    <Employees>
      <Employee>
        <EmpCode>7</EmpCode>
        <EmpName>Roger</EmpName>
      </Employee>
      <Employee>
        <EmpCode>8</EmpCode>
        <EmpName>Claire</EmpName>
      </Employee>
    </Employees>
  </Department>
</Departments>
```

Figure 3.21: Modified XML instance: complex element Department inserted

Figure 3.22 shows the modified instance resulting of a DELETE operation over the example instance of figure 3.18. In this case, the Department with attribute id="5" (the Human Resources department) was removed.

```
<Departments>
  <Department id="1">
    <DeptCode> 1 </DeptCode>
    <DeptName> Sales </DeptName>
    <Employees>
      <Employee id="2">
        <EmpCode> 1 </EmpCode>
        <EmpName> Bill </EmpName>
      </Employee>
      <Employee id="3">
        <EmpCode> 2 </EmpCode>
        <EmpName> Ann </EmpName>
      </Employee>
      <Employee id="4">
        <EmpCode> 5 </EmpCode>
        <EmpName> John </EmpName>
      </Employee>
    </Employees>
  </Department>
</Departments>
```

Figure 3.22: Modified XML instance: element Department id="5" deleted

The result of the removal of an XML sub-tree is that the tuples corresponding to the removed elements will be deleted from the database. In the example, the Department element contained two Employee elements, referring to the employees Marc and Rachel. Thus, not only the tuple corresponding to the removed department will be deleted, but also the employee tuples.

```
DELETE FROM Employee WHERE EmpCode=4
DELETE FROM Employee WHERE EmpCode=6
DELETE FROM Department WHERE DeptCode=2
```

The UPDATE operation is defined only on atomic elements. Figure 3.23 shows the result of a UPDATE operation on the example instance of figure 3.18. The content of the EmpName element of the Employee with id="3" has been modified.

```
<Departments>
   <Department id="1">
      <DeptCode> 1 </DeptCode>
      <DeptName> Sales </DeptName>
      <Employees>
         <Employee id="2">
            <EmpCode> 1 </EmpCode>
            <EmpName> Bill </EmpName>
         </Employee>
         <Employee id="3">
            <EmpCode> 2 </EmpCode>
            <EmpName> Mary Ann </EmpName>
         </Employee>
  ...
</Departments>
```

Figure 3.23: Modified XML instance: value of element EmpName updated

In this case, an UPDATE SQL statement must be generated.

```
UPDATE Employee SET EmpName="Mary Ann"
WHERE EmpCode=2
```

The result of a MOVE operation is illustrated in Figure 3.24. In this example, the Employee named Ann former belonging to the Sales department was moved to the Human Resources department. The id attribute of the *tuple element* being moved must remain the same.

```
<Departments>
   <Department id="1">
      <DeptCode> 1 </DeptCode>
      <DeptName> Sales </DeptName>
      <Employees>
         <Employee id="2">
            <EmpCode> 1 </EmpCode>
            <EmpName> Bill </EmpName>
         </Employee>
         <Employee id="4">
            <EmpCode> 5 </EmpCode>
            <EmpName> John </EmpName>
         </Employee>
      </Employees>
   </Department>
   <Department id="5">
      <DeptCode> 2 </DeptCode>
      <DeptName> Human Resources </DeptName>
      <Employees>
         <Employee id="6">
            <EmpCode> 4 </EmpCode>
            <EmpName> Marc </EmpName>
         </Employee>
         <Employee id="7">
            <EmpCode> 6 </EmpCode>
            <EmpName> Rachel </EmpName>
         </Employee>
         <Employee id="3">
            <EmpCode> 2 </EmpCode>
            <EmpName> Ann </EmpName>
         </Employee>
      </Employees>
   </Department>
</Departments>
```

Figure 3.24: Modified XML instance: Employee Ann moved

This operation has the following semantics. In our approach, a child *tuple element* represents a tuple related by a foreign key constraint to the tuple represented by the parent tuple element. Therefore, moving a *tuple 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 the Employee table (corresponding to the child element) with a tuple in the Department table (corresponding to the parent element).

The generated UPDATE operation on the database is

```
UPDATE Employee SET DeptCode=2 WHERE EmpCode=2
```

Not all elements of an XML instance may be updated. We classify the elements of the result of an 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 is 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. This happens when a database object is represented several times in the XML instance. In an R2X query, this will occur only when the nesting of a tuple 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.

The update operations explained above consider only *updatable elements*. From now on, we present the semantics of each update operation for *non-updatable* elements.

```
<R2X>
  <tableElement name="Employees" source="Employee" alias="e">
    <tupleElement name="Employee">
      <atomElement name="EmpCode" source="EmpCode"/>
      <atomElement name="EmpName" source="EmpName"/>
      <tableElement name="Departments" source="Department" join="EmpDept" alias="d">
        <tupleElement name="Department">
          <atomElement name="DeptCode" source="DeptCode"/>
          <atomElement name="DeptName" source="DeptName"/>
        </tupleElement>
      </tableElement>
    </tupleElement>
  </tableElement>
</R2X>
```

Figure 3.25: R2X query that results an XML Instance with non-updatable elements

In Figure 3.25 an R2X query that constructs an XML instance that contains non-updatable elements is shown. The result of this query is shown in Figure 3.26. This query contains two nested TABLE constructors, one for Employees (outermost) and another one for Departments (embedded). The embedded TABLE constructor (the one that refers to the Department table) holds the primary key of the EmpDept constraint, and the outermost TABLE constructor holds the foreign 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="4" represent the same object in the database. These elements are called non-updatable.

```
<Employees>
  <Employee id="1">
    <EmpCode>1</EmpCode>
    <EmpName>Bill</EmpName>
    <Departments>
      <Department id="2">
        <DeptCode>1</DeptCode>
        <DeptName>Sales</DeptName>
      </Department>
    </Departments>
  </Employee>
  <Employee id="3">
    <EmpCode>2</EmpCode>
    <EmpName>Ann</EmpName>
    <Departments>
      <Department id="4">
        <DeptCode>1</DeptCode>
        <DeptName>Sales</DeptName>
      </Department>
    </Departments>
  </Employee>
  <Employee id="5">
    <EmpCode>3</EmpCode>
    <EmpName>Peter</EmpName>
    <Departments>
      <Department id="6">
        <DeptCode>3</DeptCode>
        <DeptName>Buys</DeptName>
      </Department>
    </Departments>
  </Employee>
  ...
</Employees>
```

Figure 3.26: XML instance with non-updatable elements

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.

However, in the special case of a non-updatable tuple 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 key that refers to this tuple and are contained in the tuple represented by the updatable parent element.

```
<Employees>
  <Employee id="1">
    <EmpCode>1</EmpCode>
    <EmpName>Bill</EmpName>
    <Departments>
    </Departments>
  </Employee>
 ...
</Employees>
```

Figure 3.27: Result of a DELETE operation on non-updatable elements

The operation semantics for the non-updatable elements are the following:

```
<Employees>
   <Employee id="1">
      <EmpCode>1</EmpCode>
      <EmpName>Bill</EmpName>
      <Departments>
         <Department id="6">
            <DeptCode>3</DeptCode>
            <DeptName>Buys</DeptName>
         </Department>
      </Departments>
   </Employee>
   ...
   <Employee id="5">
      <EmpCode>3</EmpCode>
      <EmpName>Peter</EmpName>
      <Departments>
      </Departments>
   </Employee>
   ...
</Employees>
```

Figure 3.28: Result of a MOVE operation on non-updatable elements

```
<Employees>
   <Employee id="1">
      <EmpCode>1</EmpCode>
      <EmpName>Bill</EmpName>
      <Departments>
         <Department>
            <DeptCode> 2 </DeptCode>
         </Department>
      </Departments>
   </Employee>
  ...
</Employees>
```

Figure 3.29: Result of an INSERT 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 3.27 shows the result of a DELETE operation on the XML instance shown in Figure 3.26. The department of the "Bill" Employee (Employee id="1") was removed. The SQL statement produced by this operation is:

```
UPDATE Employee SET DeptCode=NULL WHERE EmpCode=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 3.28 shows an example of that. This operation was executed on the XML instance of Figure 3.27. The "Buys" department (Department id="6") was moved from the Employee id="5" to Employee id="1". The generated SQL statements are:

```
UPDATE Employee SET DeptCode=NULL WHERE EmpCode=3
UPDATE Employee SET DeptCode=3 WHERE EmpCode=1
```

INSERT: The insertion of a non-updatable element means an update in the key that implements the foreign key constraint. Figure 3.29 shows the result of the insertion of a non-updatable element in the XML instance of Figure 3.27 . It inserts one Department in the "Bill" Employee (Employee id="1"). The semantics of this INSERT operation is to connect this employee with the "Human Resources" department. The generated SQL statement is the following:

```
UPDATE Employee SET DeptCode=2 WHERE EmpCode=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 3.29

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 3.28 results from the XML instance in Figure 3.26. 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.

3.5 Mapping R2X queries to uR2X queries

To query the database the R2X wrapper uses the uR2X language [VIT 2001]. The uR2X wrapper is used (a demo of this wrapper can be found in `http://143.54.8.145:8080/xmlsql.html`).

To transform an R2X query into an uR2X query, the R2X wrapper needs to generate the necessary SQL statements and to create a CONSTRUCT clause that holds the TABLE, TUPLE and ATOM constructors. This is done by a R2X wrapper module called *Translator* (see Figure 3.2).

SQL Generation

The criterion used to determine the number of SQL statements that must be generated takes into consideration the TABLE constructors. There will be one SQL statement for each inner most *table element*. Figure 3.30 shows a graphical representation of queries in Figure 3.17 and 3.20.

Based in the number of inner most tableElement present in each query, it's possible to conclude that the query in Figure 3.17 will generate one SQL statement, and the query in Figure 3.20 will generate two SQL statements.

Also, each SQL statement will consider an inner most *table element* and its parents, that is, each SQL statement will join an inner most *table element* with all the *table elements* in the way to the document root (observe the doted lines in Figure 3.30).

In order to exemplify the translation of an R2X query into an SQL query, the query of Figure 3.17 is used.

The R2X query is processed to extract its constructors. All the column names specified in the ATOM constructors are listed in the SELECT clause. The alias provided in the TABLE constructor is used to identify the source table of each column.

```
SELECT d.DeptName, d.DeptCode, e.EmpName, e.EmpCode
```

The SQL FROM clause is constructed with the source and alias attributes of each TABLE constructor.

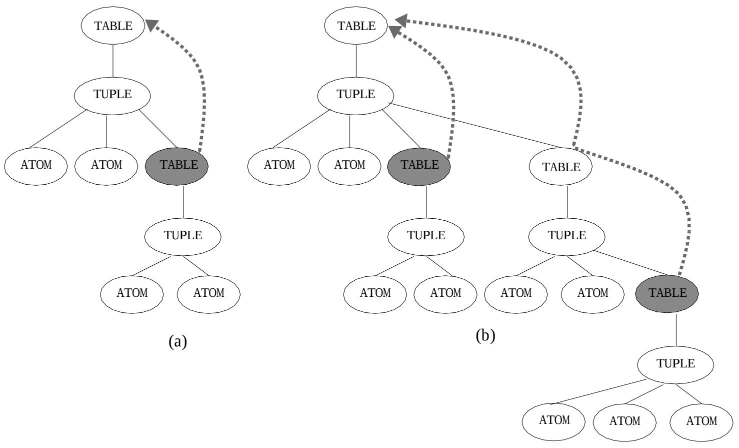


Figure 3.30: Graphical representation of queries in Figures 3.17 (a) and 3.20 (b)

```
FROM Department d, Employee e
```

In the WHERE clause, the join criteria must be explicitly written. The WHERE clause is generated by consulting the database description. The Translator looks for the description of the joined table specified in the source attribute of the TABLE constructor. Then, the translator finds the constraint with the name specified in the tableElement join attribute. The constraint specifies the name of the referenced table and the name of the foreign key that will be used to join the tables. The process is repeated recursively for the next tableElement in the way to the document root, until it reaches the tableElement nearest to the document root. This *table element* has no join attribute.

To finalize the WHERE clause, the filters of each TABLE constructor are added. The filter can be a complex SQL clause that refers to any table in the database.

```
WHERE  e.DeptCode= d.DeptCode
AND (d.DeptCode=1 or d.DeptCode=2)
```

Finally, there is always an ORDER BY clause that holds the primary keys of each table in the query. They are listed in the same order as the TABLE constructors appear in the query. This clause is a requirement of uR2X queries.

```
ORDER BY e.DeptCode, e.EmpCode
```

Through this process, the SQL statement above is generated from the query in Figure 3.17.

```
SELECT d.DeptName, d.DeptCode, e.EmpName, e.EmpCode
FROM Department d, Employee e
WHERE  e.DeptCode= d.DeptCode
AND (d.DeptCode=1 or d.DeptCode=2)
ORDER BY e.DeptCode, e.EmpCode
```

The query in Figure 3.20 generates two SQL queries. The first query selects Departments and Equipment. The second one selects Departments, Employees and Dependents.

```
SELECT d.DeptName, d.DeptCode, eq.EquipDescription, eq.EquipCode
FROM Department d, Equipment eq
WHERE eq.Department=d.DeptCode
ORDER BY d.DeptCode, eq.EquipCode

SELECT d.DeptName, d.DeptCode, e.EmpName, e.EmpCode,
    dep.DepName, dep.DepCode, dep.BirthDate
FROM Department d, Employee e, Dependent dep
WHERE dep.EmpCode=e.EmpCode AND e.DeptCode=d.DeptCode
ORDER BY d.DeptCode, e.EmpCode, dep.DepCode
```

Note that the second query joins three database tables. The number of tables joined will correspond to the number of nested *table elements* in the R2X query.

CONSTRUCT Generation

The CONSTRUCT clause is generated by a set of XSLT rules [XSL 99]. This set of rules is shown below.

The rules specify the construction of an XML document containing an uR2X element with a child construct.

```
<!-- R2X rule -->
<xsl:template match="R2X">
<uR2X>
   <construct>
      <xsl:apply-templates/>
   </construct>
</uR2X>
</xsl:template>
```

For each tableElement in the R2X query, a tableElement is generated in the resulting document, with attributes name, source and nestBy. The name value is copied from the R2X query name attribute. The values of the source and nestBy attributes are set later.

```
<xsl:template match="tableElement" >
<tableElement>
   <xsl:attribute name="name">
      <xsl:value-of select="./@name"/>
   </xsl:attribute>
   <xsl:attribute name="source"/>
   <xsl:attribute name="nestBy"/>
   <xsl:apply-templates/>
</tableElement>
</xsl:template>
```

For each tupleElement in the R2X query, a tupleElement is generated in the output, and the attribute name is set to be the same of the one specified in the R2X query.

```
<xsl:template match="tupleElement">
<tupleElement>
   <xsl:attribute name="name">
      <xsl:value-of select="./@name"/>
   </xsl:attribute>
   <xsl:apply-templates/>
</tupleElement>
</xsl:template>
```

A similar process occurs for each atomElement in the R2X query. An atomElement is generated in the resulting document, with two attributes: source and name. The values of these attributes are copied from the R2X query.

```
<xsl:template match="atomElement">
<atomElement>
   <xsl:attribute name="name">
      <xsl:value-of select="./@name"/>
   </xsl:attribute>
   <xsl:attribute name="source">
      <xsl:value-of select="./@source"/>
   </xsl:attribute>
</atomElement>
</xsl:template>
```

Considering the sample queries shown in this paper, the result of the mapping process can be verified in Figures 3.20 and 3.31. Figure 3.31 shows the corresponding uR2X query of R2X query in Figure 3.20.

```
<uR2X>
  <query>
    <sql viewName="q1">SELECT d.DeptName, d.DeptCode, eq.EquipDescription, eq.EquipCode
                FROM Department d, Equipment eq
                WHERE  eq.Department= d.DeptCode
                ORDER BY d.DeptCode, eq.EquipCode</sql>
    <sql viewName="q2">SELECT d.DeptName, d.DeptCode, e.EmpName, e.EmpCode,
                dep.DepName, dep.DepCode, dep.BirthDate
                FROM Department d, Employee e, Dependent dep
                WHERE  dep.EmpCode= e.EmpCode AND e.DeptCode= d.DeptCode
                ORDER BY d.DeptCode, e.EmpCode, dep.DepCode</sql>
  </query>
  <construct>
    <tableElement name="Departments" source="q1" nestBy="DeptCode">
      <tupleElement name="Department">
        <atomElement name="DeptCode" source="DeptCode"/>
        <atomElement name="DeptName" source="DeptName"/>
        <tableElement name="Equipment" source="q1" nestBy="EquipCode">
          <tupleElement name="Equip">
            <atomElement name="EquipCode" source="EquipCode"/>
            <atomElement name="EquipDescription" source="EquipDescription"/>
          </tupleElement>
        </tableElement>
        <tableElement name="Employees" source="q2" nestBy="EmpCode">
          <tupleElement name="Employee">
            <atomElement name="EmpCode" source="EmpCode"/>
            <atomElement name="EmpName" source="EmpName"/>
            <tableElement name="Dependents" source="q2" nestBy="DepCode">
              <tupleElement name="Dependent">
                <atomElement name="DepCode" source="DepCode"/>
                <atomElement name="DepName" source="DepName"/>
                <atomElement name="BirthDate" source="BirthDate"/>
              </tupleElement>
            </tableElement>
          </tupleElement>
        </tableElement>
      </tupleElement>
    </tableElement>
  </construct>
</uR2X>
```

Figure 3.31: uR2X query corresponding to R2X query in Figure 3.20

3.5.1 ID and DTD generator

The ID and DTD generator is responsible for generating the DTD of the query result and to include the ids in the resulting XML document.

Generating IDs

The Update Module of the R2X query identifies the operations performed by a user application in the XML instance by comparing the original instance and the modified one. Each element generated by a TUPLE constructor contains an id attribute that identifies it in the instance. By comparing the id attributes in the original XML view and in the modified XML view, the Update Module is able to determine which XML elements were deleted, updated, moved or inserted. As each tupleElement corresponds to a database tuple, it is possible to map updates on the XML view into tuple deletions, insertions and updates.

To generate these ids, we use a set of XSLT rules. These rules are applied in the XML document returned by the uR2X wrapper. The strategy used to implement these set of rules includes counting the number of ancestors of a given node. If the node has an even number of ancestors, this node corresponds to an element generated by a TUPLE constructor, and thus needs the id attribute.

The set of rules are shown bellow:

```
<xsl:template match="node()">
   <xsl:copy>
      <!-- Counts the number of ancestors of the current node -->
      <xsl:variable name="level" select="count(ancestor::*)"/>
         <!-- Creates the ID attribute -->
         <xsl:if test="($level mod 2) != 0">
            <xsl:attribute name="id">
               <xsl:number count="* " format="1" level="any" from="/"/>
            </xsl:attribute>
         </xsl:if>
      <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>
```

Generating the DTD

As stated before, it is necessary to generate the DTD of the resulting XML document of an R2X query. This DTD is passed to the user application. In this way, the user application modifies the XML document according to the DTD. In other words, no structural changes will be allowed.

To generate the DTD, we use a set of XSLT rules that transforms the R2X query into a DTD.

The first rule transforms each TABLE constructor into an ELEMENT declaration. The content of this element is the name specified in the tupleElement (child of the tableElement). This rule is shown bellow:

```
<xsl:template match="tableElement">
   &lt;!ELEMENT <xsl:value-of select="./@name"/>(<xsl:value-of select="./tupleElement/@name"/>
       <xsl:text>)+&gt;</xsl:text>
   <xsl:apply-templates/>
</xsl:template>
```

There is also a rule for the TUPLE constructor. This rule generates an ELEMENT declaration whose content are the children of this TUPLE constructor. The generated element also contains an attribute id used to identify this element. This attribute is needed by the R2X *Update Module*.

```
<xsl:template match="tupleElement">
   &lt;!ELEMENT  <xsl:value-of select="./@name"/> (
   <xsl:for-each select="./atomElement">
      <xsl:if test="not (position()=1)">, </xsl:if> <xsl:value-of select="./@name"/>
   </xsl:for-each>
   <xsl:for-each select="./tableElement">
      ,  <xsl:value-of select="./@name"/>
   </xsl:for-each>
   <xsl:text>) &gt;</xsl:text>
   &lt;!ATTLIST  <xsl:value-of select="./@name"/> <xsl:text> id ID #IMPLIED &gt;</xsl:text>
   <xsl:apply-templates />
</xsl:template>
```

The last rule concerns the ATOM constructor. It generates a #PCDATA element for each atomElement found in the query.

```
<xsl:template match="atomElement">
   &lt;!ELEMENT
   <xsl:value-of select="./@name"/> (#PCDATA) &gt;
   <xsl:apply-templates />
</xsl:template>
```

Following this process, the DTD generated for the R2X query in Figure 3.17 is the following:

```
<!ELEMENT Departments (Department)+>
<!ELEMENT Department (DeptCode, DeptName, Employees)>
<!ATTLIST Department id ID #IMPLIED>
<!ELEMENT DeptCode (#PCDATA)>
<!ELEMENT DeptName (#PCDATA)>
<!ELEMENT Employees (Employee)+>
<!ELEMENT Employee (EmpCode, EmpName)>
<!ATTLIST Employee id ID #IMPLIED>
<!ELEMENT EmpCode (#PCDATA)>
<!ELEMENT EmpName (#PCDATA)>
```

Chapter 4

Thesis Contributions and Plan

The previous chapter shows how the R2X implementation model uses NF^2 relations to build the XML views. However, the update operations defined over these views suppose that R2X generates only updatable views. Consequently, it is necessary to *guarantee* that R2X produces only updatable views. How to enforce this feature constitutes the main problem that needs to be solved by the thesis proposed here.

4.1 Thesis Contributions

4.1.1 Main Contributions

The main contributions of this thesis constitute of a mechanism to update relational databases through XML views. This mechanism is based in a query language that builds updatable XML views from relational databases, and in a concept of normalization for XML views. To accomplish this, the following tasks were executed.

Definition of a query language to build updatable XML views from relational databases: Given a relational database, it is necessary to specify how to obtain XML views from the data stored in its tables. In the design of R2X, we always had in mind that the views produced by the language should be updatable. In fact, the work in [FUR 85] states that a common tendency is to define views only for query and authorization purposes, and afterwards try to solve the problems created by view updates. Furtado and Casanova opinion is that update requirements should be considered from the start.

Definition of a wrapper architecture that implements this language: User applications will use the defined query language through a wrapper. The wrapper architecture must be defined.

Definition of the update operations that can be performed over the XML view: Given an XML view, the update operations that can be performed over it must be defined.

The big problem in updating relational databases through views resides in the ambiguity that views can hold [DAT 2000]. By removing the ambiguity, we can guarantee that the view is updatable. One alternative to obtain this guarantee is to study normal forms for NF^2 relations [MAK 77, MOK 96, ROT 88, LIN 89], and among them, to choose one that best fits in the language. The properties of the chosen normal form must be checked against the obtained view, making that only views that follow these properties be accepted.

More specifically, to guarantee that R2X produces only updatable views, the following tasks must be accomplished.

Definition of functional dependency: The first task consists in defining the functional dependency that will be used in the normal form definition. There are several definitions in literature. Among them are the ones used in the normal form definitions [MAK 77, MOK 96, ROT 88, LIN 89] and the *Nested Functional Dependency*, defined by [HAR 99].

Definition of a concept of normal form: Based in the functional dependency defined in the previous step, the concept of normal form that will be used in the thesis must be defined. Several normal forms for NF^2 relations were proposed in literature: *Normal Form* (NF) [MAK 77], *Nested Normal Form* (NNF) [MOK 96], *Partitioned Normal Form* (PNF) [ROT 88] and NF-N3 [LIN 89], among others. There are also proposals of normal forms for semistructured schemas: *Normal Form for Semistructured Schema* (NF-SS) [WU 2001] e *XML Normal Form* (XNF) [EMB 2001]. These proposals must be analyzed to check if they can be adapted to NF^2 relations.

Definition of the relation between XML and NF^2: In order to use NF^2 in the language specification, it is necessary to formally define the relation between XML and NF^2. This step was already accomplished (section 3.2).

Definition of what kinds of views are updatable: Using the concept of normal form defined in the second step, a method of identifying classes of updatable views and classes of non-updatable views must be defined. More specifically, it is necessary to verify what classes of views accomplish the properties of the normal form defined. Besides that, other features of updatable views must be observed. In literature, several work define classes of updatable views, and mechanisms to update databases through views. A classical work is presented in [KEL 86]. In addition, the properties mentioned in section 2.4 must be considered.

Moreover, the approach must be complemented in several aspects.

Specify the language using relational algebra: The specification made until now uses relational algebra. However, this specification must be complemented in order to make the language semantics clearer.

Allow the generation of views with additional semantics: There are several applications in which a view has a specific semantic. They usually construct views according to the value of an attribute. A classical example [KEL 86, ABI 96] is a view that shows the employees that plays in the company soccer time. In this case, the criteria for the view construction is *"play-soccer-time = yes"*. This makes the semantic of the update operations to be different from the usual. The exclusion of an employee must be mapped to an update in the value of *"play-soccer-time"* to *"no"* rather than to the exclusion of the employee.

These tasks constitute the main contribution of the thesis.

4.1.2 Operational Mechanisms

Despite the main contributions, there are some aspects that must be decided. They are related to the wrapper operational mechanism, and are listed bellow.

Definition of the update mechanism to be used: It is necessary to make a detailed study in the update mechanism adopted by the R2X language. The kind of exclusion adopted must be defined (*cascade* or *restrict*).

Error messages: The user application must be aware of error messages. The R2X wrapper must notify the user application about update errors like key violation, foreign key constraint violation, etc.

Communication with the user application: The communication of the R2X *wrapper* with the user application must be defined. An alternative is to design an API that will be used by the user application to execute queries and return the updated view to the wrapper. Additional alternatives must be studied to find an efficient solution.

Allow large views to be partitioned: It is known that large XML documents are difficult to be manipulated by the DOM API [WOO 98]. This is because the DOM API requires that the view be entirely loaded in the memory. To minimize this problem, the language could return the view in "pieces" (a record at a time, for instance). This certainly reduces the size of the generated view.

Transaction control: The transaction control system to be used in the wrapper must be defined. It must avoid classical database problems like the lost update problem and the dirty-read problem [ELM 2000], among others.

Generation of content according to its necessity: This problem is closely related to the transaction control problem. A user application must be allowed to work with several XML views at the same time. An example of this occurs when the application requests a view X, and, analyzing this view, it finds out that more data is needed. In this case, the extra data can be a new XML view, or just a complement of the view currently being analyzed. Both cases must be considered in the proposed approach.

4.2 Accomplished tasks

A mechanism to update relational databases through XML views was defined. It is based in the R2X language and in the R2X wrapper architecture. Until now, the following tasks were accomplished:

✓ Definition of an NF^2 implementation model for uR2X

✓ Definition and design of the R2X language

✓ Definition of the R2X wrapper architecture

✓ Definition of an NF^2 implementation model for R2X

✓ Definition of the R2X update operations

✓ Mapping from NF^2 relations to XML

✓ Mapping from R2X queries to uR2X queries

4.3 Schedule

This section presents the schedule of the activities defined in section 4.1. The schedule includes a one year stage at University of Pennsylvania with Professor Susan Davidson. The tasks to be executed were divided in two tables. The first one (4.1) concerns the stage abroad, and the second one (4.2) shows the activities that will be executed after returning to Brazil. The tables present year/month in the columns, and activities in the lines.

The activities are numbered in the list bellow.

1. Definition of a normal form to be used to guarantee the generation of updatable views.
   (a) Definition of functional dependency.
   (b) Definition of a concept of normal form.
   (c) Definition of what views are updatable.
2. Article describing the use of the defined normal form in the R2X language.
3. Allow the generation of view with additional semantics.
4. Article describing the new version of the R2X language, now supporting new classes of views.
5. Language specification.
6. Return to Brazil.
7. Definition of the update mechanism to be used.
8. Error messages.
9. Communication with the user application.
10. Transaction control.
11. Generation of content according to its necessity.
12. Allow large views to be partitioned.
13. Paper describing the architecture.
14. Implementation.
15. Thesis writing.
16. Thesis presentation.

Table 4.1: Schedule of the activities to be executed abroad

	2002		2003									
Task	11	12	01	02	03	04	05	06	07	08	09	10
1(a)	⋆	⋆										
1(b)		⋆	⋆	⋆								
1(c)					⋆	⋆						
2					⋆	⋆						
3							⋆	⋆	⋆	⋆		
4										⋆	⋆	
5											⋆	⋆
6												⋆

Table 4.2: Schedule of activities to be executed after returning to Brazil

	2003		2004											
Task	11	12	01	02	03	04	05	06	07	08	09	10	11	12
7	⋆	⋆	⋆											
8				⋆										
9					⋆	⋆								
10						⋆	⋆							
11						⋆	⋆	⋆						
12							⋆	⋆						
13								⋆	⋆					
14									⋆	⋆	⋆	⋆		
15				⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
16														⋆

Appendix A

DTDs

A.1 uR2X DTD

```
<!ELEMENT uR2X (query, construct)>
<!ELEMENT query (sql)+>
<!ELEMENT sql (#PCDATA)>
<!ATTLIST sql viewName ID #REQUIRED>
<!ELEMENT construct (tableElement)>
<!ELEMENT tableElement (atomElement | tupleElement | tableElement)>
<!ATTLIST tableElement name CDATA #REQUIRED
                    source IDREF #REQUIRED
                    nestBy NMTOKENS #IMPLIED>
<!ELEMENT tupleElement (atomElement | tupleElement | tableElement)+>
<!ATTLIST tupleElement name CDATA #REQUIRED>
<!ELEMENT atomElement EMPTY>
<!ATTLIST atomElement name CDATA #REQUIRED
                   source CDATA #IMPLIED
                   mandatory (yes | no) "no">
```

A.2 R2X DTD

```
<!ELEMENT R2X (tableElement)>
<!ELEMENT tableElement (tupleElement)>
<!ATTLIST tableElement name CDATA #REQUIRED
                    source CDATA #REQUIRED
                    join CDATA #IMPLIED
                    filter CDATA #IMPLIED
                    alias ID #REQUIRED>
<!ELEMENT tupleElement (atomElement+, tableElement*)>
<!ATTLIST tupleElement name CDATA #REQUIRED>
<!ELEMENT atomElement EMPTY>
<!ATTLIST atomElement name CDATA #REQUIRED
                   source CDATA #REQUIRED>
```

Appendix B

Database Schema in XML

```xml
<DatabaseSchema>
    <Table Name="Department">
        <Attributes>
            <Attribute Name="DeptCode" PrimaryKey="yes" Type="Integer"/>
            <Attribute Name="DeptName" Type="String"/>
            <Attribute Name="Acr" Type="String"/>
        </Attributes>
    </Table>
    <Table Name="Employee">
        <Attributes>
            <Attribute Name="EmpCode" PrimaryKey="yes" Type="Integer"/>
            <Attribute Name="EmpName" Type="String"/>
            <Attribute Name="BirthDate" Type="Date"/>
            <Attribute Name="DeptCode" Type="Integer"/>
        </Attributes>
        <Constraints>
            <Constraint Name="EmpDept" ForeignKey="DeptCode" References="Department"/>
        </Constraints>
    </Table>
    <Table Name="Dependent">
        <Attributes>
            <Attribute Name="DepCode" PrimaryKey="yes" Type="Integer"/>
            <Attribute Name="DepName" Type="String"/>
            <Attribute Name="BirthDate" Type="Date"/>
            <Attribute Name="EmpCode" Type="Integer"/>
        </Attributes>
        <Constraints>
            <Constraint Name="DepEmp" ForeignKey="EmpCode" References="Employee"/>
        </Constraints>
    </Table>
    <Table Name="Equipment">
        <Attributes>
            <Attribute Name="EquipCode" PrimaryKey="yes" Type="Integer"/>
            <Attribute Name="EquipDescription" Type="String"/>
            <Attribute Name="DeptCode" Type="Integer"/>
        </Attributes>
        <Constraints>
            <Constraint Name="EquipDept" ForeignKey="DeptCode" References="Department"/>
        </Constraints>
    </Table>
</DatabaseSchema>
```

Bibliography

[ABI 84] ABITEBOUL, S.; BIDOIT, N. Non first normal form relations to represent hierarchically organized data. In: PODS, 1984. **Proceedings** . . . [S.l.: s.n.], 1984. p.191–200.

[ABI 86] ABITEBOUL, S.; BIDOIT, N. Non-first normal form relations: an algebra allowing data restructuring. **Journal of Computer and System Sciences**, v.33, p.361–393, 1986.

[ABI 96] ABITEBOUL, S.; HULL, R.; VIANU, V. **Foundations of databases**. 1.ed. [S.l.]: Addison-Wesley, 1996.

[ARI 83] ARISAWA, H.; MORIYA, K.; MIURA, T. Operations and the properties on non-first-normal-form relational databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 9., 1983, Florence, Italy. **Proceedings** . . . Morgan Kaufmann, 1983. p.197–204.

[BOA 2001] BOAG, S. et al. **Xquery 1.0**: an xml query language. Available at `http://www.w3.org/TR/xquery/`, W3C Working Draft.

[BRA 2000] BRADLEY, N. **The xml companion**. 2.ed. [S.l.]: Addison-Wesley, 2000.

[BRA 2001] BRAGANHOLO, V. **A study of non-first normal form relations and update of relational databases through views**. Porto Alegre: PPGC-UFRGS, 2001. Trabalho Individual II. (1040).

[BRA 2001a] BRAGANHOLO, V.; HEUSER, C.; VITTORI, C. Updating relational databases through xml views. In: PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON INFORMATION INTEGRATION AND WEB-BASED APPLICATIONS & SERVICES - IIWAS 2001, 2001, Linz, Austria. **Proceedings** . . . [S.l.: s.n.], 2001. p.85–94.

[CAR 2000] CAREY, M. J. et al. Xperanto: Publishing Object-Relational Data as XML. In: THIRD INTERNATIONAL WORKSHOP ON THE WEB AND DATABASES, 2000, Dallas, Texas. **Proceedings** . . . [S.l.: s.n.], 2000.

[CHA 75] CHAMBERLIN, D. D.; GRAY, J. N.; TRAIGER, I. L. Views, authorization, and locking in a relational data base system. In: AFIPS - NATIONAL COMPUTER CONFERENCE, 1975. **Proceedings** . . . [S.l.: s.n.], 1975. p.425–430.

[CHE 2000] CHENG, J.; XU, J. Ibm db2 xml extender: an end-to-end solution for storing and retrieving xml documents. In: PROCEEDINGS OF ICDE'00, 2000. **Proceedings** . . . [S.l.: s.n.], 2000.

[COD 71] CODD, E. F. Normalized data base structure: a brief tutorial. In: ACM-SIGFIDET, 1971. **Proceedings** . . . [S.l.: s.n.], 1971. p.1–17.

[COD 74] CODD, E. F. Recent investigations in relational data base systems. In: IFIP, 1974, Stockholm, Sweden. **Proceedings** . . . [S.l.: s.n.], 1974. p.1017–1021.

[COL 89] COLBY, L. A recursive algebra and query optimization for nested relations. In: PROCEEDINGS OF THE ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1989, Portland, Oregon. **Proceedings** . . . [S.l.: s.n.], 1989. p.273–283.

[CON 2001] CONRAD, A. **A survey of microsoft sql server 2000 xml features**. MSDN Library. Available at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxm%l/html/xml07162001.asp`.

[DAT 2001] DATA MIRROR. **Db/xml transform**. Available at <`http://www.datamirror.com/resourcecenter/dbxmltransform.asp`> (Mai.2002), Technical White Paper.

[DAT 2000] DATE, C. J. **An introduction to database systems**. 7th.ed. [S.l.]: Addison Wesley, 2000.

[DEU 98] DEUTSCH, A. et al. **Xml-ql**: a query language for xml. Submission to the World Wide Web Consortium.

[ELM 2000] ELMASRI, R.; NAVATHE, S. B. **Fundamentals of database systems**. 3.ed. [S.l.]: Addison-Wesley, 2000.

[EMB 92] EMBLEY, D.; NG, Y.-K.; MOK, W. Y. **Unifying normalization theory under a new definition for nested normal form**. [S.l.]: Brigham Young University and Hong Kong Polytechnic, 1992. Research Report, Available at <`http://osm7.cs.byu.edu/Papers.html`>. (December 2001).

[EMB 2001] EMBLEY, D. W.; MOK, W. Y. Developing xml documents with guaranteed ’good’ properties. In: CONCEPTUAL MODELING - ER 2001, 2001, Yokohama, Japan. **Proceedings** . . . Springer, 2001. p.426–441.

[FAN 2002] FAN, C. et al. **Xtables**: bridging relational technology and xml. Available at `<http://www7b.software.ibm.com/dmdd/library/techarticle/0203shekita/020%3shekita.pdf>` (May 2002), IBM web site.

[FER 2000] FERNÁNDEZ, M.; TAN, W.-C.; SUCIU, D. Silkroute: trading between relations and xml. In: PROCEEDINGS OF THE NINTH INTERNATIONAL WORLD WIDE WEB CONFERENCE, 2000. **Proceedings** . . . [S.l.: s.n.], 2000.

[FIS 85] FISCHER, P. C.; GUCHT, D. V. Determining when a structure is a nested relation. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 1985. **Proceedings** . . . [S.l.: s.n.], 1985.

[FUR 85] FURTADO, A. L.; CASANOVA, M. A. Updating relational views. In: KIM, W.; REINER, D. S.; BATORY, D. S. (Eds.). **Query processing in database systems**. Berlin, Heidelberg: Springer, 1985. p.127–142.

[HAR 99] HARA, C.; DAVIDSON, S. Reasoning about nested functional dependencies. In: ACM SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS (PODS), 1999, Philadelphia, Pennsylvania. **Proceedings** . . . [S.l.: s.n.], 1999.

[HIG 2001] HIGGINS, S. **Oracle9i application developer's guide - xml. release 1 (9.0.1)**. ORACLE Corporation. Available at `http://download-west.oracle.com/otndoc/oracle9i/901_doc/appdev.901/a888%94/toc.htm`.

[HUL 90] HULIN, G. On restructuring nested relations in partitioned normal form. In: VLDB CONFERENCE, 16., 1990, Brisbane, Australia. **Proceedings** . . . [S.l.: s.n.], 1990. p.626–636.

[IBM 2000] IBM. **Xml extender**: administration and programming (version 7). Technical Manual.

[INT 2001] INTELLIGENT SYSTEM RESEARCH. **Odbc2xml**: merging odbc data into xml documents. Available at `http://www.intsysr.com/odbc2xml.htm`.

[ISH 98] ISHIKAMA, H. et al. Xql: A Query Language for XML Data. In: THE QUERY LANGUAGES WORKSHOP, 1998. **Proceedings...** . . . [S.l.: s.n.], 1998.

[JAE 82] JAESCHKE, G.; SCHEK, H. J. Remarks on the algebra of non first normal form relations. In: ACM SIGACT-SIGMOD SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, 1982. **Proceedings** . . . [S.l.: s.n.], 1982. p.124–138.

[KEL 85] KELLER, A. M. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In: FOURTH ACM SIGACT-SIGMOD SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, 1985, Portland, Oregon. **Proceedings** . . . ACM, 1985. p.154–1i63.

[KEL 86] KELLER, M. The role of semantics in translating view updates. **IEEE Computer**, v.19, n.1, p.63–73, Jan. 1986.

[LAD 2001] LADDAD, R. **Xml apis for databases**: blend the power of xml and databases using custom sax and dom apis. Available at `http://www.javaworld.com/javaworld/jw-01-2000/jw-01-dbxml.html`.

[LIN 89] LING, T. W. A normal form for sets of not-necessarily normalized relations. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 22., 1989, Kailua-Kona, Hawaii, United States. **Proceedings** . . . IEEE Computer Society Press, 1989. p.578–586.

[MAK 77] MAKINOUCHI, A. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In: VLDB, 1977, Tokio, Japan. **Proceedings** . . . [S.l.: s.n.], 1977. p.447–453.

[MOK 96] MOK, W. Y.; NG, Y.; EMBLEY, D. W. A normal form for precisely characterizing redundancy in nested relations. **ACM TODS**, v.21, n.1, p.77–106, 1996.

[ORA 2001] ORACLE CORPORATION. **Oracle xml sql utility**. Available at `http://technet.oracle.com/docs/tech/xml/oracle_xsu/doc_library/adx04xsu%.html`, Oracle Corporation.

[OZS 87] OZSOYOGLU, Z. M.; YUAN, L.-Y. A new normal form for nested relations. **ACM Transactions on Database Systems**, v.12, n.1, p.111–136, Mar. 1987.

[OZS 89] OZSOYOGLU, Z.; YUAN, L.-Y. On the normalization in nested relational databases. **Lecture Notes in Computer Science**, v.361, p.243–271, 1989.

[RAM 2001] RAMAKRISHNAN, R. **Database management systems**. 2nd.ed. [S.l.]: Mc Graw Hill, 2001.

[ROT 87] ROTH, M. A.; KORTH, H. F. The design of non-1nf relational databases into nested normal form. In: PROCEEDINGS OF ACM-SIGMOD CONFERENCE, 1987, San Francisco. **Proceedings** . . . [S.l.: s.n.], 1987. p.143–159.

[ROT 87a] ROTH, M. A.; KORTH, H. F.; BATORY, D. S. Sql/nf: a query language for ¬1nf relational databases. **Information Systems**, v.12, n.1, p.99–114, 1987.

[ROT 88] ROTH, M. A.; KORTH, H. F.; SILBERSCHATZ, A. Extended algebra and calculus for nested relational databases. **ACM Transactions on Database Systems**, v.13, n.4, p.389–417, Dec 1988.

[SIL 2002] SILBERSCHATZ, A.; KORTH, H.; SUDARSHAN, S. **Database system concepts**. 4th.ed. [S.l.]: Mc Graw Hill, 2002.

[SIL 2002a] SILVA, A. S. da et al. Using nested tables for representing and querying semistructured data. In: PROCEEDINGS OF 18TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 2002, San Jose, California. **Proceedings** . . . [S.l.: s.n.], 2002.

[TAT 2001] TATARINOV, I. et al. Updating xml. In: PROCEEDINGS OF SIGMOD 2001, 2001, Santa Barbara, California. **Proceedings** . . . [S.l.: s.n.], 2001.

[THO 86] THOMAS, S.; FISCHER, P. C. Nested relational structures. **Advances in Computing Research**, v.3, p.269–307, 1986.

[TUR 99] TURAU, V. **Making legacy data accessible for xml applications**. Available at `http://www.informatik.fh-wiesbaden.de/$\sim$turau/`.

[TUR 2001] TURAU, V. **Db2xml 1.4**: transforming relational databases into xml documents. Available at `http://www.informatik.fh-wiesbaden.de/$\sim$turau/DB2XML/index.html`.

[VIT 2001] VITTORI, C.; DORNELES, C.; HEUSER, C. Creating xml documents from relational data sources. In: PROCEEDINGS OF EC-WEB 2001, 2001, Munich, Germany. **Proceedings** . . . [S.l.: s.n.], 2001. p.60–70.

[WAH 2000] WAHLIN, D. Leveraging sql server's xml features. **XML Magazine**, v.1, n.5, 2000. `http://www.xmlmag.com/upload/free/features/xml/2000/05win00/dw0005/dw00%05.asp`.

[WOO 98] WOOD, L. **Document object model (dom) level 1 specification**. Available at `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`, W3C Recommendation.

[WU 2001] WU, X. et al. Nf-ss: a normal form for semistructured schema. In: INTERNATIONAL WORKSHOP ON DATA SEMANTICS IN WEB INFORMATION SYSTEMS - DASWIS 2001, 2001, Yokohama, Japan. **Proceedings** . . . [S.l.: s.n.], 2001. p.146–159.

[XML 2001] XML schema part 0: primer. W3C Recommendation. Available at `http://www.w3.org/TR/xmlschema-0`.

[XSL 99] XSL transformations version 1.0. W3C Recommendation. Available at `http://www.w3.org/TR/xslt`.