Updating Multidimensional XML Documents

Nikolaos Fousteris[1], Manolis Gergatsoulis[1], Yannis Stavrakas[2]
[1]*Department of Archive and Library Science*
Ionian University
Ioannou Theotoki 72, 49100 Corfu, Greece
{nfouster, manolis}@ionio.gr
[2]*Institute for the Management of Information Systems (IMIS)*
R. C. Athena
G. Mpakou 17, 11524, Athens, Greece
yannis@inmis.gr

Abstract:
Purpose – In a wide spectrum of applications it is desirable to manipulate semistructured information that may present variations according to different circumstances. Multidimensional XML (MXML) is an extension of XML suitable for representing data that assume different facets, having different value and / or structure under different contexts. The purpose of this research is to develop techniques for updating MXML documents.
Design/methodology/approach – Updating XML has been studied in the past, however, updating MXML must take into account the additional features which stem from incorporating context into MXML. We investigate the problem of updating MXML in two levels: a) at the graph level, i.e. in an implementation independent way, and b) at the relational storage level.
Findings – We introduce six basic update operations, which are capable of any possible change. We specify those operations in an implementation independent way, and explain their effect in the document through examples. We give algorithms that implement those operations using SQL on a specific storage method that employs relational tables for keeping MXML. We also give an overview of MXPath, an extension of XPath that incorporates context, and show how to translate MXPath queries to "equivalent" SQL queries.
Research limitations/implications – Though the proposed operations solve the problem of updating MXML documents, several problems, such as formally define MXPath and its translation to SQL, remain to be investigated in the future in order to implement a system that stores, queries and updates MXML documents through a relational database infrastructure.
Practical implications – MXML is suitable for representing, in a compact way, data that assume different facets, having different value or structure, under different contexts. In order for MXML to be applicable in practice, it is vital to develop techniques and tools for storing, updating and querying MXML documents. The techniques proposed in this paper form a significant step in this direction.
Originality/value – This article presents a novel approach for updating Multidimensional XML documents by proposing update operations on both, the graph level and the (relational) storage level.
Keywords: Multidimensional XML, extensions of XML, (M) XML databases, (M)XML updates, (M)XML storage, (M)XML query.
Article Type: Research paper

1 Introduction

Multidimensional XML (MXML) (Gergatsoulis, Stavrakas and Karteris, 2001) is an extension of XML which allows context to qualify element and attribute values, and specify the conditions under which the document components have meaning.

MXML is therefore suitable for representing data that assume different facets, having different value or structure, under different contexts. Contexts are specified by giving values to one or more user defined *dimensions*.

MXML is intended to be a natural way of handling context-dependent information. Managing such information is very important for a wide range of applications that need an abstraction mechanism to deal with the complexity of multi-faceted data. In Gergatsoulis et al. (2001) we showed how MXML can be used together with Multidimensional XSL (MXSL) stylesheets in order to present in a personalized way relevant parts of information to users. In Gergatsoulis and Stavrakas (2003) we considered time as context, and discussed how MXML can be used to represent data that evolve over time in value and / or in structure.

In this paper we address the problem of updating MXML. We introduce six basic change operations that are necessary and sufficient to perform any possible change in a MXML document. We define those operations based on a MXML graph model, called MXML graph. Updating MXML must take into account context, which appear as a special type of nodes in MXML graph. The definitions of the operations are independent of any specific storage approach for MXML. We discuss in detail the effect of each operation on MXML documents through examples. Moreover, we give an overview of MXPath, an extension of XPath that incorporates context and is suitable for navigating in MXML graphs. MXPath is used to specify the nodes that update operations act upon.

As a practical approach for managing MXML and implementing MXPath we consider the relational database framework. In previous work (Fousteris, Gergatsoulis and Stavrakas, 2007), we have proposed techniques for storing MXML in relational databases. This problem has been extensively studied in the past (Deutsch, Fernandez and Suciu, 1999; Shanmugasundaram et al., 2001b; Shanmugasundaram et al., 1999; Tatarinov et al., 2002) for XML. The problem of updating XML documents stored in relational DBs has also been studied in the past (Braganholo, Davidson and Heuser, 2003b; Braganholo, Davidson and Heuser, 2004; Braganholo, Davidson and Heuser, 2003a). Consequently, we use relational infrastructure to specify a possible implementation of the MXML update operations we introduce. First, we outline a simple method from Fousteris, Gergatsoulis and Stavrakas (2007) for storing MXML in relational tables. Then, we show how it is possible to translate MXPath expressions to SQL queries on the aforementioned simple storage solution. Finally, we give algorithms that implement the MXML update operations using SQL.

In summary, the goal of this paper is to develop a framework for updating context-dependent semistructured information using MXML. The main contributions are:

- An overview of MXPath, an extension of XPath that incorporates context.
- A set of MXML update operations, defined in a storage independent way.
- An overview of the MXPath translation to SQL, for a specific MXML storage method.
- A set of algorithms implementing the MXML update operations, using SQL on a specific MXML storage method.

The paper is organized as follows. In Section 2 we give some preliminaries on context, MXML, its properties, and introduce MXML graph through an example. In Section 3 we explain briefly MXPath. In Section 4 we present the basic MXML update operations on MXML graph. In Section 5 we consider a method for storing MXML in relational tables. We show how MXPath can be translated to SQL, and specify algorithms that implement the MXML update operations using SQL. In Section 6 we discuss related work. Finally, Section 7 concludes the paper and presents some directions for future work.

2 Preliminaries

2.1 Mutidimensional XML (MXML)

In Mutidimensional XML (MXML), data assume different facets, having different value or structure, under different contexts (Gergatsoulis, Stavrakas and Karteris, 2001). Contexts are described through syntactic constructs called *context specifiers* and are used to specify sets of *worlds* by imposing constraints on the values that a set of user defined *dimensions* can take. A *world*, which represents an environment under which data obtain meaning, is determined by assigning a single value to every dimension, taken from the *domain* of the dimension. The elements/attributes that have different facets under different contexts are called *multidimensional elements/attributes* while their facets are called *context elements/attributes*, and are accompanied with a corresponding context specifier denoting the set of worlds under which each facet is the holding one. MXML documents can be represented in a node-based graphical model called *MXML-graph* (Fousteris, Gergatsoulis and Stavrakas, 2007). The syntax of MXML is shown in Example 2.1. More details on MXML syntax can be found in Gergatsoulis, Stavrakas and Karteris (2001).

Example 2.1. The MXML document shown below represents a book in a book store. Two dimensions are used in the document. The dimension edition whose domain is {greek, english}, and the dimension customer_type whose domain is {student, library}.

```
<book isbn=[edition=english]"0-13-110362-8"[/]
             [edition=greek]"0-13-110370-9"[/]>
  <title>The C programming language</title>
  <authors>
     <author>Brian W. Kernighan</author>
     <author>Dennis M. Ritchie</author>
  </authors>
  <@publisher>
     [edition = english] <publisher>Prentice Hall</publisher>[/]
     [edition = greek] <publisher>Klidarithmos</publisher>[/]
  </@publisher>
  <@translator>
     [edition = greek] <translator>Thomas Moraitis</translator>[/]
  </@translator>
  <@price>
     [edition=english]<price>15</price>[/]
     [edition=greek,customer_type=student]<price>9</price>[/]
     [edition=greek,customer_type=library]<price>12</price>[/]
  </@price>
  <@cover>
     [edition=english]<cover><material>leather</material></cover>[/]
     [edition=greek]
        <cover>
           <material>paper</material>
           <@picture>
```

```
                [customer_type=student]<picture>student.bmp</picture>[/]
                [customer_type=library]<picture>library.bmp</picture>[/]
            </@picture>
        </cover>
    [/]
  </@cover>
</book>
```

Notice that the name of a multidimensional elements is preceded by the symbol @ while the corresponding context elements have the same element name but without the symbol @. The MXML-graph representing the MXML document presented above is shown in Fig. 1. For saving space, obvious abbreviations for dimension names and values are used. IDs assigned to nodes are used in this paper to facilitate the reference to the nodes. However, these IDs are also used when the MXML tree is stored in relational tables as explained in Fousteris, Gergatsoulis and Stavrakas (2007).

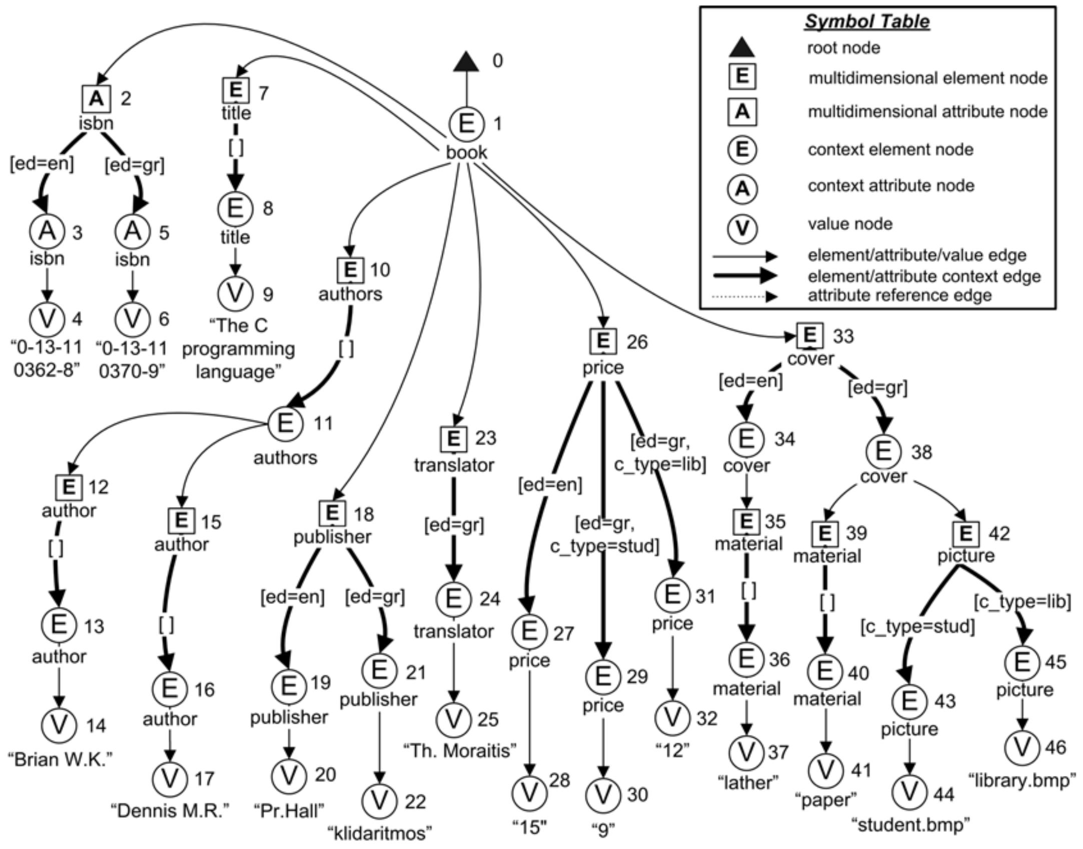

Figure 1. Graphical representation of MXML (MXML graph)

2.2 Properties of contexts

Context specifiers qualifying context edges give the *explicit contexts* of the nodes to which the edges lead. The explicit context of all the other nodes is by definition the *universal context* represented by [ ], denoting the set of all possible worlds. When elements and attributes are combined to form a MXML document, their explicit contexts do not alone determine the worlds under which each element/attribute holds, since when an element/attribute e_2 is part of another element e_1, then e_2 have substance only under the worlds that e_1 has substance. This is conceived as if the context of e_1 is inherited to e_2. The context propagated in that way is combined with the explicit context of a node to give its *inherited context*. Formally, the inherited context *ic(q)* of a node *q* is $ic(q) = ic(p) \cap^c ec(q)$, where *ic(p)* is the inherited context of its parent node *p* and $\cap^c$ is the *context intersection* defined in Stavrakas and Gergatsoulis (2002). $\cap^c$ combines two context specifiers and computes a new one representing the intersection of the worlds specified by the original specifiers. The evaluation of the inherited context starts from the root of the MXML-graph. By definition, the inherited context of the root is the universal context [ ]. Contexts are not inherited through attribute reference edges. As in conventional XML, the leaf nodes of MXML-graphs must be value nodes. The *inherited context coverage* (icc) of a node further constraints its inherited context, so as to contain only the worlds under which the node has access to some value node. This property is important for navigation and querying, but also for the *reduction* of MXML to XML (Gergatsoulis et al., 2001; Fousteris, Gergatsoulis and Stavrakas, 2007). The *icc(n)* of a node *n* is defined as follows: if *n* is a value node then $icc(n) = ic(n)$; else if *n* is a leaf node but not a value node then $icc(n)$ = [-]; otherwise $icc(n) = icc(n_1) \cup^c icc(n_2) \cup^c \dots \cup^c icc(n_k)$, where $n_{1,\dots,}$ n_k are the child element nodes of *n*. [-] stands for the *empty context specifier* which represents the empty set of worlds. $\cup^c$ is the *context union* operator defined in Stavrakas and Gergatsoulis (2002) which combines two context specifiers and computes a new one representing the union of the worlds specified by the original context specifiers.

Example 2.2. In Fig. 2 we see a fragment of the MXML graph of Fig. 1 in which it is shown the value of the inherited context coverage (icc) for every node of the subtree.

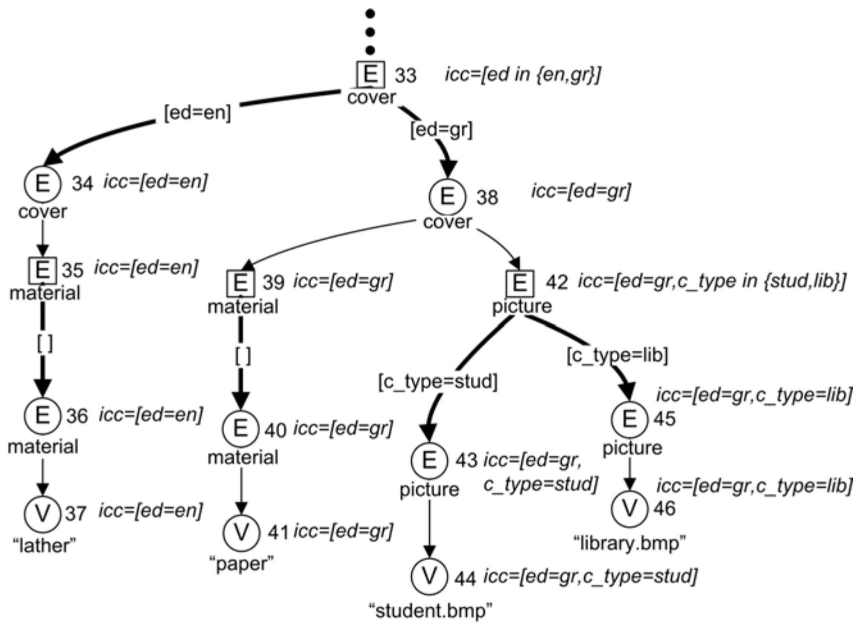
E 33 cover icc=[ed in {en,gr}]
[ed=en]
[ed=gr]
E 34 cover icc=[ed=en]
E 38 cover icc=[ed=gr]
E 35 material icc=[ed=en]
[ ]
E 36 material icc=[ed=en]
V 37 "lather" icc=[ed=en]
E 39 material icc=[ed=gr]
[ ]
E 40 material icc=[ed=gr]
V 41 "paper" icc=[ed=gr]
E 42 picture icc=[ed=gr,c_type in {stud,lib}]
[c_type=stud]
[c_type=lib]
E 43 picture icc=[ed=gr, c_type=stud]
V 44 "student.bmp" icc=[ed=gr,c_type=stud]
E 45 picture icc=[ed=gr,c_type=lib]
V 46 "library.bmp" icc=[ed=gr,c_type=lib]

Figure 2. Representing the Inherited Context Coverage

3 A Multidimensional extension to XPath

Multidimensional XPath (MXPath) is an extension of XPath that uses the inherited context coverage and the explicit context of MXML to specify navigation patterns over the additional MXML features. MXPath is used in subsequent sections to specify the nodes to be modified when MXML graphs are updated. MXpath can also be used in query and transformation languages for MXML. A detailed discussion of MXPath is out of the scope of this paper, however in this section we provide a brief overview.

As an example consider the following query given in natural language: *Find the ISBN of the greek edition of the book with title "The C Programming Language"*. The corresponding MXPath expression is:

[icc()>= "ed=gr"], /child::book[title= "The C Programming Language"]/attribute::isbn

The [icc()>= "ed=gr"] is the *icc qualifier*, which is the first part of every MXPath, and denotes a condition for the *icc* of the result nodes. In this example, we demand that the *icc* of the results (returned by the function icc()) is context superset (>=) of the context [ed=gr]. The rest of the syntax in this example is similar to conventional XPath. However, each element of the form axis::label describes paths that include two MXML nodes, one multidimensional node and one context node. For example, the attribute::isbn looks for attributes of node 1, crosses the multidimensional node labeled "isbn" with ID 2 and returns the context nodes labeled "isbn" with IDs 3 and 5. There are two main additional features of MXPath: (a) it allows conditions on the explicit context of a node, by using the *ec qualifier* predicate, and (b) it is possible to use the characters (->) instead of (::) in order to return multidimensional nodes instead of context nodes. For example, consider the following MXPath:

[icc()>= "-"], /child::book/child::cover[ec()>= "ed=en"]/child -> material

It returns the "material" multidimensional nodes of covers of the english edition. On the tree of Fig. 1, it will evaluate to node with ID 35. Note that [icc()>= "-"] denotes that the inherited context coverage of the results must be context superset of the empty context [-]. As this is always true, in this case the inherited coverage qualifier may be omitted and is implied. The predicate [ec()>= "ed=en"] states that the explicit context of "cover"' context nodes must be superset of [ed=en], thus leading the navigation to node 34. Then, child->material evaluates to the multidimensional nodes labeled "material" which are children of node 34. As in XPath, there are shorthands in MXPath. The above MXPath can take the form:

/book/cover[ec()>= "ed=en"]/->material

4 MXML Update Operations

In this section we define a set of basic update operations namely *delete*, *insert*, *update_label*, *update_context*, *update_value* and *replace* that are used to modify MXML documents (MXML graphs). These operations can be combined to perform any possible change on a document, and when they apply on a well-formed MXML document, the document obtained is well-formed. All operations have been defined in this section at the graph level. In the next section we will show how these operations can be implemented using a relational framework.

4.1 Deleting subtrees

The operation *delete(P)* is used to delete subtrees rooted at specific (context or multidimensional) nodes specified by the MXPath *P*. The operation is defined by:

```
delete(P):
Input:    P: MXPath expression.
BEGIN
    1. Let $N_P$ be the set of nodes returned by evaluating P.
    2. For each $n \in N_P$ do
```

```
            - Let p be the parent node of n.
            - Delete the subtree rooted at n and the edge leading to n.
            - Recalculate the icc of each node in the path from p to the root of the tree.
            - Delete the subtree rooted at each node m for which icc(m)= [-] as well as the edges leading to m.
        End For
    END
```

Example 4.1. The tree in Fig.3(b) is obtained by deleting the subtree rooted at the node 45 from the tree in Fig.3(a) (which is a fragment of the tree in Fig.1). By recalculating the icc's of the nodes from the node 42 to the root we get icc(42)=[ed=gr,c_type=stud], icc(38)=[ed=gr], icc(33)=[ed in {gr,en}] and icc(1)=[]. Notice that the MXPath *P* returning node 45 may be: /book/cover[ec()= "ed=gr"]/picture[ec()= "c_type=lib"]. The last part [ec()= "c_type=lib"] is a predicate on the explicit contexts of "picture" context nodes. It says that the explicit context of the results should be equal to [c_type=lib]. The valuation of this path returns the "library" pictures for greek editions. If we wanted the "library" pictures for any edition, we would omit the predicate [ec()= "ed=gr"], thus implying [ec()>= "-"] in its place. Note that in this case we get the nodes 43 and 45. By deleting both nodes the icc of node 42 becomes [-] and therefore node 42 will also be deleted.

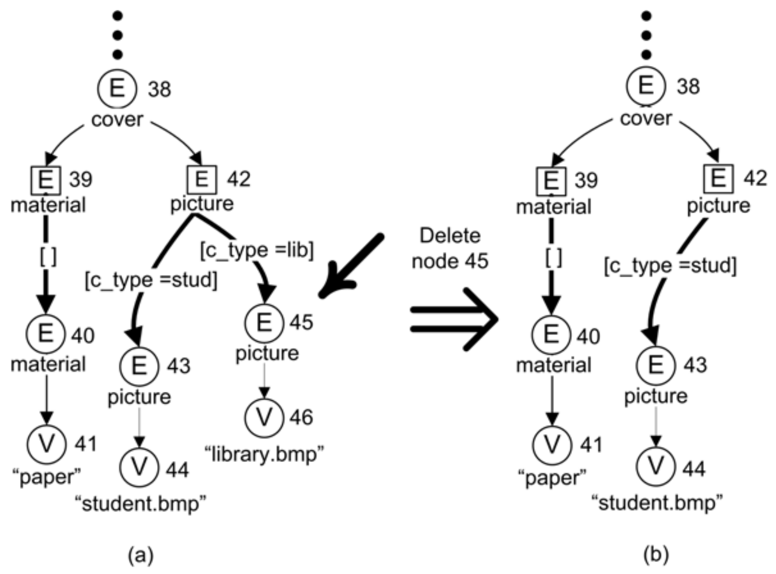

Figure 3. Deleting a node

4.2 Inserting subtrees

Inserting a (well-formed) tree *T* at specific points of the MXML tree, specified by an MXPath expression *P*, is done through the operation *insert(P, T)*:

```
insert(P, T):
Input:  P: MXPath expression.
        T: A well-formed MXML tree.
BEGIN
    1. Let $N_P$ be the set of nodes returned by evaluating P.
    2. For each n $\in N_P$ do
        If n and root(T) are multidimensional nodes and label(n) = label(root(T)) then
```

```
            - Let $C_n$ be the set of context specifiers of all context edges departing from n.
            - Let $C_T$ be the set of context specifiers of all context edges departing from the root of T.
            - **If** $C_n \cup C_T$ is a context deterministic set of context specifiers **then**
                - Hang T on the node n (unify n with the root of T).
                - Recalculate the icc of n and of each ancestor or descendant of n in the resulted tree.
            **End If**
        **else If** n and root(T) are context nodes **and** label(n) = label(root(T)) **then**
            - Hang T on the node n (unify n with the root of T).
            - Recalculate the icc of n and of each ancestor or descendant of n in the resulted tree.
        **End If**
    **End For**
    3. Delete the subtree rooted at each node m for which icc(m)= [-] as well as the edges leading to m.
**END**
```

Notice that: (a) Insertion applies only to nodes that are of the same type and have the same label with the root node of *T*. (b) If the root of *T* is multidimensional, the algorithm ensures that the tree obtained is context deterministic. (c) If *P* returns multiple nodes, the insertion of *T* is applied for each node separately. The sequence of the nodes does not play any role.

Example 4.2. The tree in Fig.4(c) is obtained by inserting the tree *T* shown in Fig.4(b) at the node 34 of the tree in Fig.4(a) (which is a fragment of the tree in Fig.1).

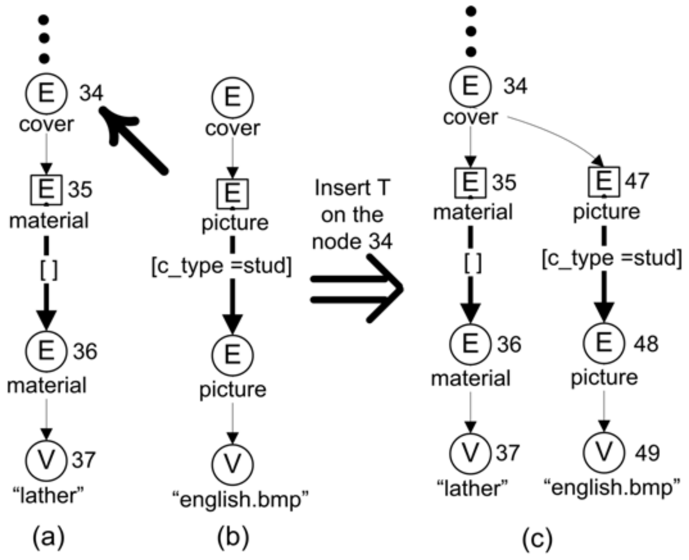

Figure 4. Inserting a subtree

By recalculating the icc's of the nodes we get icc(49)=icc(48)=icc(47) =[ed=en,c_type=stud] and icc(34)=[ed=en]. Note that *P* is the MXPath: /book/cover[ec()= "ed=en"].

4.3 Updating nodes

Updating a node in a MXML graph, can be achieved in three ways. By updating its label (if the node is a multidimensional one), updating its context, or updating its value. Concerning context update, we must check the necessary conditions so as to retain context determinism of the resulted tree.

4.3.1 Updating Label

The *update_label(P, L)* operation is used to update the label of one or more nodes. This operation is defined as follows:

```
update_label(P, L):
Input:  P: MXPath expression.
        L: A node label.
BEGIN
  1. Let $N_P$ be the set of nodes returned by evaluating P.
  2. For each multidimensional node n $\in N_P$ do
        - Replace the label of n by L.
        - Replace the label of each child context node of n by L.
     End For
END
```

Note that *update_label* applies only to multidimensional nodes. This is not a restriction when the MXML tree is in *canonical form* (Stavrakas, 2003). Besides, it prevents inconsistent situations in which a child context element has different label from its parent multidimensional element.

Example 4.3. In Fig.5 we see how *update_label* operation applies to node 42 and changes the label picture of node 42 (and of its child context nodes 43 and 45), to the new label image.

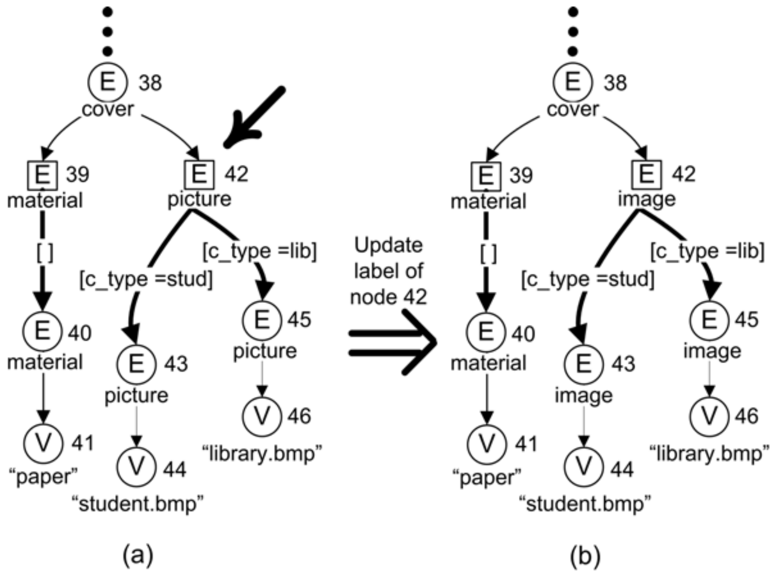

Figure 5. Updating label

The argument *P* of the function *update_label* is the MXPath expression:

/child::book /child::cover[ec() = “ed=gr”] /child->picture

which evaluates to the multidimensional node 42. A simpler form of the above MXPath is:

/book/cover[ec() = “ed=gr”]/->picture

Note that if we wanted to change the labels of all picture nodes, irrespective of edition, we would omit [ec() = “ed=gr”].

4.3.2 Updating Context

The operation *update_context(P, E)* is used to update the explicit context of the nodes returned by evaluating the MXPath expression *P*. *E* is a *context expression* which specifies, through the use of operations on contexts (e.g. context union, context intersection etc.), how the new explicit contexts will be constructed. The *update_context* operation is defined as follows:

```
update_context(P, E):
Input:  P: MXPath expression.
        E: A context expression.
BEGIN
  1. Let $N_P$ be the set of nodes returned by evaluating P.
  2. Discard all non-context nodes from $N_P$. Let M be a partition of $N_P$
     such that each member of the partition consists of all nodes in $N_P$ which are siblings.
  3. For each $m \in M$ do
         - Collect in a set $S_m$ all explicit contexts of the nodes in m
           and in S the explicit contexts of all siblings of the nodes in m which are not in m.
         - Let $E(S_m)$ be the set of context specifiers obtained by applying E to the elements of $S_m$.
         - If $E(S_m) \cup S$ is a context deterministic set of context specifiers then
              - Replace the explicit context of each node in m by the corresponding element of $E(S_m)$.
              - Recalculate the icc of each node in m and the icc of each ancestor or descendant of
                   each node in m in the resulted tree.
         End If
  End For
  4. Delete the subtree rooted at each node n for which icc(n)= [-] as well as the edges leading to n.
END
```

The *update_context* operation applies only to context element/attribute nodes as these nodes possess a (user defined) explicit context.

Concerning the context expression *E*, its syntax allows specifying a number of operations on contexts. The syntax of context expressions is as follows:

Context_Operation(Context_Modifier)

The *Context_Operation* is either *new* or one of +, -, and * representing the *context union* ($\cup^c$), the *context difference* ($-^c$) and the *context intersection* ($\cap^c$) respectively. The *Context_Modifier* is a context specifier. The new explicit context obtained by applying a context expression *E = Context_Operation(Context_Modifier)* to an explicit context *Explicit_Context* is either the *Context_Modifier* if the context operation is *new*, or the result of evaluating the expression:

Explicit_Context Context_Operation Context_Modifier.

The context operations *context union* ($\cup^c$), *context difference* ($-^c$) and *context intersection* ($\cap^c$) have been formally defined in Stavrakas and Gergatsoulis (2002) and Stavrakas (2003).

Example 4.4. The tree in Fig.6(b) is obtained by updating the context of the node 45 in the tree of Fig.6(a) (which is a fragment of the tree in Fig.1). The MXPath expression *P* is:

/child::book/child::cover[ec()= “ed=gr”]/child::picture[ec()= “c_type=lib”]

The context expression E = [+(c_type=museum)] adds “*museum*” to the values of the dimension “*c_type*” to form the new explicit context of node 45 which becomes [c_type in {lib,museum}]. Notice that the new context specifier is the result of the operation [c_type=lib] $\cup^c$ [c_type=museum]. Recalculating the iccs we get icc(46)=icc(45)=[c_type in {lib,museum}, ed=gr] and icc(42)=[ed=gr,c_type in {lib,stud,museum}].

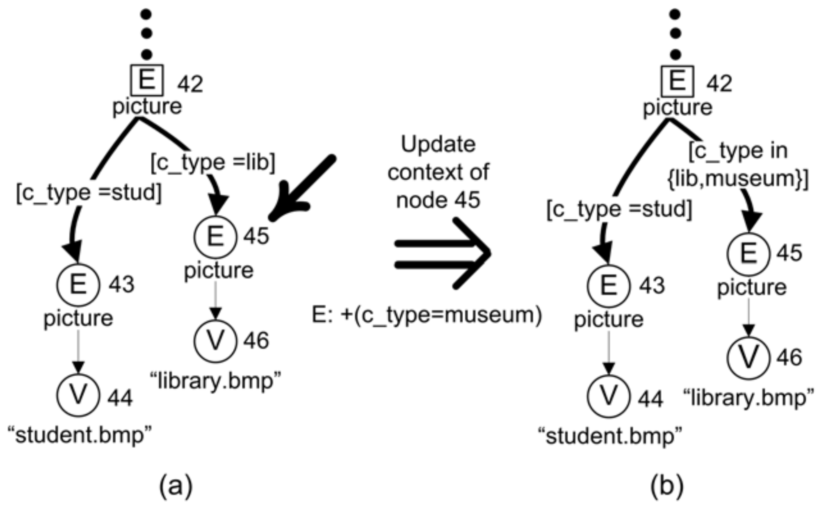

Figure 6. Updating the context of a node

Note that when the evaluation of *P* returns two or more context nodes which are all children of the same multidimensional node, these nodes are updated simultaneously because the final tree must be context deterministic.

Example 4.5. (Continued from Example 4.4) Notice that if the expression E used in Example 4.4 was E = [+(c_type=stud)], the update operation would not allowed as it violates the requirement to leave the tree in a context deterministic state. This is shown in Fig.7.

It is easy to prove that if the *Context_Operation* of an *update_context* operation is either * (context intersection), or - (context difference), then the set of the context specifiers obtained by this *update_context* operation is always context deterministic. In this case we can improve the algorithm presented above by omitting the test which checks if the set $E(S_m) \cup S$ is context deterministic. Notice however, that this does not hold if the *Context_Operation* is either + (context union) or *new*. Therefore, in this case the test cannot be eliminated.

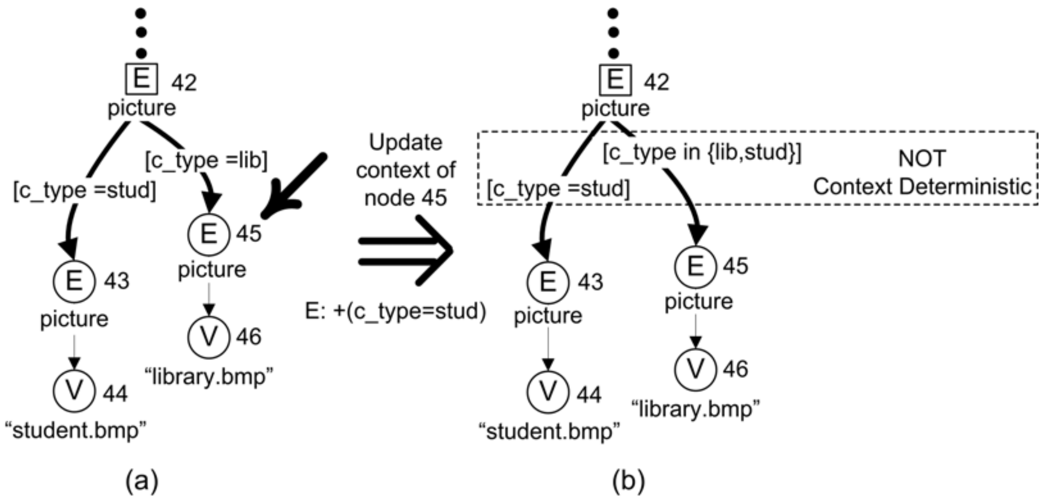

Figure 7. Forbidden context update

4.3.3 Updating Value

To update a node's value we use the operation *update_value(P, C)*, which replaces the value of one or more leaf nodes, specified by the MXPath expression *P*, by a new value obtained by evaluating the value expression *C*. This operation is defined as follows:

```
update_value(P, C):
Input:
    P: MXPath expression.
    C: A value expression.
BEGIN
  1. Let $N_P$ be the set of nodes returned by evaluating P.
  2. For each $N \in N_P$ do
       If N is a leaf node then
           Replace the value of N by the result of evaluating expression C.
       End If
  End For
END
```

Example 4.6. In Fig.8(a) we see a fragment of the MXML graph of Fig.1. In Fig.8(b) we see the tree obtained from the tree in Fig.8(a) after updating the value of node with id=9. According to this update operation, the old value ("The C programming language") of node 9 was replaced by the value "The Java programming language" by altering the string "C" to "Java" (C expression). The MXPath *P* is: /book/title.

This MXPath actually returns the node with ID 8, however the update_value function understands that it has to use the corresponding value node with ID 9. Note that in this case the MXPath looks like a conventional XPath.

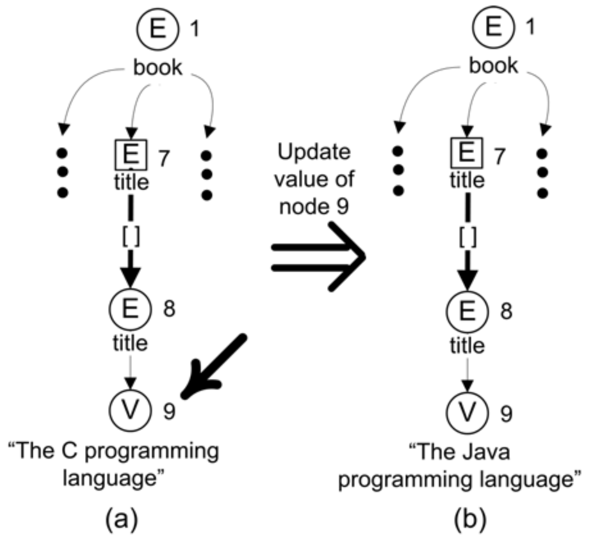

Figure 8. Updating value

4.4 Replacing subtrees

The operation *replace(P, T)*, replaces the subtrees rooted at the (context or multidimensional) nodes returned by evaluating *P*, by the MXML tree *T*. The root nodes of the replaced subtrees must match in type and in label with the root node of *T*.

```
replace(P, T):
Input:   P: MXPath expression.
         T: A well-formed MXML tree.
BEGIN
   1. Let $N_P$ be the set of nodes returned by evaluating P.
   2. For each $n \in N_P$ do
         If n and root(T) are nodes of the same type (context or multidimensional) and label(n) = label(root(T)) then
            - Replace the subtree with root node n with the subtree T.
            - Recalculate the icc of n and of each ancestor or descendant of n in the resulted tree.
         End If
   End For
   3. Delete the subtree rooted at each node m for which icc(m)= [-] as well as the edges leading to m.
END
```

Example 4.7. In Fig.9 we see the tree obtained (see Fig.9(c)) by replacing the tree rooted at node 42 (see Fig.9(a) which is a fragment of the MXML graph of Fig.1) by the tree shown in Fig.9(b). By recalculating the icc's we get icc(48)=icc(47)=icc(42)=icc(38)=[ed=gr].

5 Updating MXML stored in Relational Databases

In previous sections we showed how to specify nodes in MXML graphs using MXPath, and gave a set of update operations on MXML. In this section we illustrate how to use a Relational Database (RDB) infrastructure in order to update MXML documents. First, we give an overview of a straightforward method for storing MXML graphs in RDBs. Then we show how it is possible to translate MXPath expressions to SQL queries. Finally, we explain how the update operations introduced previously can be implemented on relational tables.

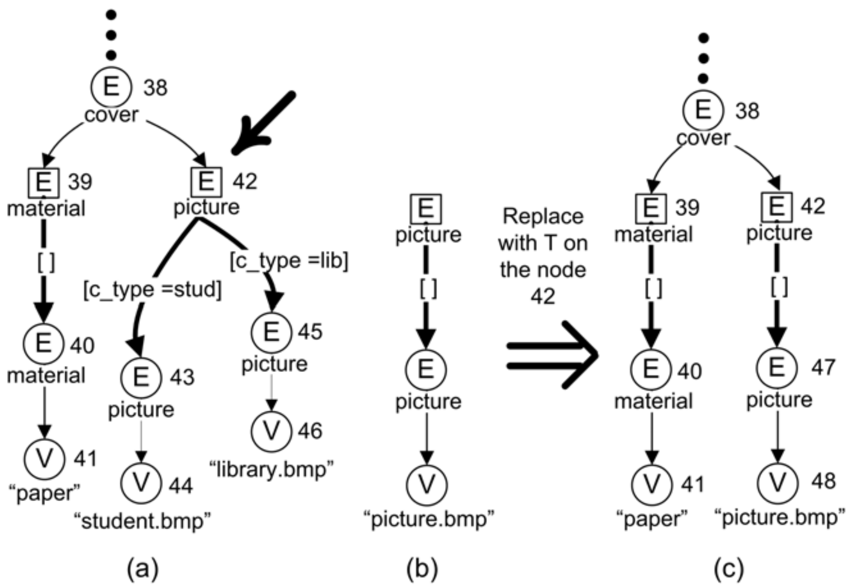

Figure 9. Replacing a subtree

5.1 Storing MXML in Relational Databases

In this subsection we present a technique for storing MXML documents in relational databases. The proposed technique, called *naive approach*, uses a single table (*Node Table*), to store all information concerning the nodes (of all types) contained in a MXML document. Node Table contains all the information which is necessary to reconstruct the MXML document(graph). Each row of the table represents a MXML node. The attributes of Node Table are: node_id stores the id of the node, parent_id stores the id of the parent node, ordinal stores a number denoting the order of the node among its siblings, tag stores the label (tag) of the node or NULL (denoted by "-") if it is a value node, value stores the value of the node if it is a value node or NULL otherwise, type stores a code denoting the node type (CE for context element, CA for context attribute, ME for multidimensional element, MA for multidimensional attribute, and VN for value node), and explicit_context stores the explicit context of the node (as a string). Note that the explicit context is kept here for completeness, and does not serve any retrieval purposes. Instead, as we will see in the following, the correspondence of nodes to the worlds under which they hold is encoded in separate tables.

Example 5.1. Fig.10 shows how the MXML Graph of Fig.1 is stored in the Node Table. Some of the nodes have been omitted, denoted by (....), for brevity.

We now explain how context is stored in such a way so as to facilitate the formulation of context-aware queries. We introduce three additional tables, as shown in Fig.11. The *Possible Worlds Table* which assigns a unique ID (attribute word_id) to each possible combination of dimension values (i.e. to each possible world). Each dimension in the MXML document has a corresponding attribute in this table. The *Explicit Context Table* keeps the correspondence of each node with the worlds represented by its explicit context. Finally, the *Inherited Coverage Table* keeps the correspondence of each node with the worlds represented by its inherited context coverage.

Node Table						
node_id	parent_id	ordinal	tag	value	type	explicit_context
1	0	1	book	-	CE	-
2	1	-	isbn	-	MA	-
3	2	1	isbn	-	CA	[ed=en]
4	3	1	-	0-13-110362-8	VN	-
5	2	2	isbn	-	CA	[ed=gr]
6	5	1	-	0-13-110370-9	VN	-
….	….	….	….	….	….	….
26	1	4	price	-	ME	-
27	26	1	price	-	CE	[ed=en]
28	27	1	-	15	VN	-
….	….	….	….	….	….	….

Figure 10. Storing the MXML-graph of Fig.1 in a Node Table

Example 5.2. Fig. 11, depicts (parts of) the Possible Worlds Table, the Explicit Context Table, and the Inherited Coverage Table obtained by encoding the context information appearing in the MXML-graph of Fig. 1.

For example, the inherited context coverage of the node with node_id=28 includes the worlds {(edition, english), (customer_type, student)} and {(edition, english), (customer_type, library)}. This is encoded in the Inherited Coverage Table as two rows with node_id=28 and the world ids 2 and 4. In the Explicit Context Table the same node corresponds to all possible worlds (ids 1, 2, 3 and 4).

Possible Worlds Table		
world_id	edition	customer_type
1	gr	stud
2	en	stud
3	gr	lib
4	en	lib

Explicit Context Table	
node_id	world_id
1	1
1	2
1	3
1	4
….	….
28	1
28	2
28	3
28	4
….	….

Inherited Coverage Table	
node_id	world_id
1	1
1	2
1	3
1	4
….	….
28	2
28	4
….	….

Figure 11. Mapping MXML nodes to worlds

5.2 Translating MXPath to SQL

It is possible to translate MXPath context aware expressions (queries), outlined in Section 3, to SQL queries over the aforementioned storage approach. The key is representing context in a way that SQL can understand and handle. Encoding the explicit context and the inherited context coverage as previously described in relational tables allows us to construct SQL queries which use both the explicit context and the inherited context coverage of nodes. As an example consider the following query on the MXML document of Example 2.1 given in natural language: *Find the material of the cover of the greek edition of the book with title "The C Programming Language"*. The corresponding abbreviated MXPath expression is:

```
[icc( )>= "ed=gr"], /book[title= "The C programming language"]/cover/material/text()
```

which returns the material of the Greek edition of the book with the title "The C programming language". This query would be translated into the following SQL query for the "naive" storage approach as follows:

```
Select N41.value from Node as N41, Node as N40, Node as N39, Node as N38, Node as N33,
     Node as N9, Node as N8, Node as N7, Node as N1
where N9.type="VN" and N9.value="The C programming language" and N9.parent_id=N8.id and
     N8.type="CE" and N8.tag="title" and N8.parent_id=N7.node_id and
     N7.type="ME" and N7.tag="title" and N7.parent_id=N1.node_id and
     N1.type="CE" and N1.tag="book" and N1.node_id=N33.parent_id and
     N33.type="ME" and N33.tag="cover" and N33.node_id=N38.parent_id and
     N38.type="CE" and N38.tag="cover" and N38.node_id=N39.parent_id and
     N39.type="ME" and N39.tag="material" and N39.node_id=N40.parent_id and
     N40.type="CE" and N40.tag="material" and N40.node_id=N41.parent_id and
     N41.type="VN" and N41.node_id in (Select IC1.node_id
     from Inherited_Coverage as IC1, Inherited_Coverage as IC2
     where IC1.world_id=1 and IC2.world_id=3 and IC1.node_id=IC2.node_id)
```

The "where" clause implements the navigation on the tree of Fig. 1, while the nested query implements the constraints related to context, in order to finally return node 41 but not node 37. Note that to make the query more readable we have named the table variables after corresponding node ids. Besides, we have included in the query some conditions, which are redundant as they are deduced from the properties of the MXML graph. For example, as we know that the child elements of a multidimensional element are necessarily context elements with the same tag, the conditions N38.type="CE" and N38.tag="cover" can be eliminated. Observe that, the "greek edition" context contains both the worlds with ids 1 and 3 according to the Possible Worlds table, which has not been used in the SQL query for brevity. Finally, notice the large number of self-joins which is proportional to the number of nodes participated in the MXPath query expression.

5.3 Implementing MXML update operations at the relational storage level

In this subsection we present how the MXML update operations presented in Section 4 can be implemented at the storage level. In particular, we will show how the operations *delete(P)* and *update_label(P,L)* can be implemented using SQL embedded in pseudocode. It is straightforward to implement the remaining operations in a similar way.

5.3.1 Deleting subtrees

In the following we present an algorithm for implementing this operation at the relational database storage level. The algorithm consists of the procedures *Delete(P)*, *Delete_subtree(N)*, and *Update_context_path(E)* defined as shown bellow. The procedure *Delete(P)* is defined as follows:

```
Delete(P)
Input: P: MXPath expression
BEGIN
    1. Let $N_P$ be the set of node ids returned by evaluating P
    2. For each n $\in N_P$ do
        - Let p be the node id returned by "SELECT parent_id FROM Node_Table WHERE node_id = n"
        - Delete_subtree(n)
        - Update_context_path(p)
       End For
    3. Let D be the result set of "SELECT node_id FROM Node_Table WHERE node_id NOT IN
              (SELECT DISTINCT node_id FROM Inherited_Coverage_Table)"
    4. For each m $\in$ D do
        - Delete_subtree(m)
       End For
END
```

The procedure *Delete_subtree(N)* is responsible for the deletion of all records in *Node_table*, in *Explicit_Context_Table* and in *Inherited_Coverage_Table* corresponding to all nodes of the subtrees that should be deleted. *Delete_subtree(N)* is implemented as follows:

```
Delete_subtree(N)
Input:   N: a node id
BEGIN
    1. Let $N_C$ be the result set of "SELECT node_id FROM Node_Table WHERE parent_id = N"
    2. For each c $\in N_C$ do
           Delete_subtree(c)
       End For
    3. DELETE FROM Node_Table WHERE node_id = N
    4. DELETE FROM Explicit_Context_Table WHERE node_id = N
    5. DELETE FROM Inherited_Coverage_Table WHERE node_id = N
END
```

The procedure *Update_context_path(E)* is responsible for the the calculation of the new iccs of the nodes of the path from the parent node of the deleted node to the root of the tree. *Update_context_path(E)* is implemented as follows:

```
Update_context_path(E)
Input: E: a node id
```

```
BEGIN
    1. Let i = E
    2. While i ≠ 0 do
           - DELETE FROM Inherited_Coverage_Table WHERE node_id = i
           - Let $N_C$ be the result set of “SELECT node_id FROM Node_Table WHERE parent_id = i”
           - Let W be the set obtained by
                 “SELECT world_id FROM Inherited_Coverage_Table WHERE node_id in $N_C$”
           - For each $w \in W$ do
                 INSERT INTO Inherited_Coverage_Table VALUES (i, w)
             End For
           - Let i = “SELECT parent_id FROM Node_Table WHERE node_id = i”
      End While
END
```

5.3.2 Updating labels

The algorithm for implementing the operation *update_label(P, L)* is as follows:

```
Update_label(P,L)
Input: P: MXPath expression
       L: A node label
BEGIN
    1. Let $N_P$ be the set of node ids returned by evaluating P
    2. For each $n \in N_P$ do
           - Let t be the value returned by “SELECT type FROM Node_Table WHERE node_id=n”
           - If (t = `ME' OR t = `MA') then
                 UPDATE Node_Table SET tag = L WHERE (node_id = n) OR (parent_id = n)
             End If
      End For
END
```

6 Related Work

MXML is a special case of a more general family of data, called *Multidimensional Semistructured Data* (MSSD) defined in Stavrakas and Gergatsoulis (2002) and Stavrakas (2003). MXML (and MSSD) was influenced by *Intensional HTML* (IHTML) (Wadge et al., 1998), a Web authoring language, based on and extending ideas proposed for a software versioning system (Plaice and Wadge, 1993). IHTML allows a single Web page to have different variants and to dynamically adapt itself to a given context. Related work has also been directed towards temporal extensions of XML (Wang, Zhou and Zaniolo, 2006; Amagasa, Yoshikawa and Uemura, 2000; Amagasa, Yoshikawa and Uemura, 2001).

MXML has been proved useful in various applications such as in representing the history of conventional XML documents (Gergatsoulis and Stavrakas, 2003). Moreover, a method of constructing multidimensional web pages based on MXML is presented in Gergatsoulis et al. (2001).

Concerning the storage of XML data, many researchers have investigated the problem of using an RDBMS to store and query XML data. The techniques proposed for XML storage can be divided in two categories: (a) Schema-Based XML Storage techniques: the task of finding a relational schema is guided by the structure of a schema for that document (Ramanath et al., 2003; Tatarinov et al., 2002; Du, 2004; Tian et al., 2002; Shanmugasundaram et al., 1999; Shanmugasundaram et al., 2001b; Bohannon et al. 2002) (b) Schema-Oblivious XML Storage techniques: the relational schema is designed independently of the presence or absence of a schema for the XML data (Tatarinov et al., 2002; Du, Amer-Yahia, and Freire, 2004; Tian et al., 2002; Yoshikawa et al., 2001; Shanmugasundaram et al., 2001b; Florescu and Kossmann, 1999; Deutsch, Fernandez and Suciu, 1999). Note that Schema-Oblivious techniques are more general than Schema-Based as they do not require the existence of a DTD or XML Schema. The proposed approach presented in this paper (see Subsection 5.1), belongs to the Schema-Oblivious category.

Additionally to the above researching area, the field of exporting and querying relational data as XML views (Fernandez, Tan and Suciu, 1999; Shanmugasundaram et al., 2000; Shanmugasundaram et al., 2001a; Chaudhuri, Kaushik and Naughton, 2003) has been studied a lot. On the other hand, several work has been done on the problem of updating XML views. According to this approach, XML views can be mapped to the relational schema where the XML data are stored. Then, updates on XML views are mapped to updates on the corresponding relational schema (Braganholo, Davidson and Heuser, 2003b; Braganholo, Davidson and Heuser, 2004; Braganholo, Davidson and Heuser, 2003a; Wang, Mulchandani and Rundensteiner 2003).

7 Conclusions and Future Work

The present work is part of a framework aiming at storing, querying and updating MXML documents using relational databases. Alternative approaches for storing MXML in relational tables are presented in Fousteris, Gergatsoulis and Stavrakas (2007). In those approaches, specific tables are employed to store the contexts associated with each node. In this paper we investigated the problem of updating MXML documents. We presented a set of basic operations which can perform any possible change in a document. We defined those operations in a storage-independent way, and gave algorithms that implement those operations using SQL over a specific solution for storing MXML. We also outlined MXPath, an extension of XPath that incorporates context, and showed how to translate MXPath queries to SQL queries. The direction of our future work is twofold: (a) to formally define MXPath and its translation to SQL, and (b) to continue our ongoing work on the implementation of a system that stores, queries and updates MXML documents through a relational database infrastructure.

Acknowledgments

Part of this work was co-funded by the European Social Fund (75%) and National Resources (25%) - Operational Program for Educational and Vocational Training (EPEAEK II) and particularly by the Research Program “PYTHAGORAS II”.

References

Amagasa, T., Yoshikawa, M. and Uemura, S., (2000), “A Data Model for Temporal XML Documents”, *In: Proceedings of the Database and Expert Systems Applications, 11th International Conference DEXA 2000*, London, UK, pp. 334–344.

Amagasa, T., Yoshikawa, M. and Uemura, S., (2001), “Realizing Temporal XML Repositories using Temporal Relational Databases”, *In: Proceedings of the Third International Symposium on Cooperative Database Systems and Applications*, Beijing, China, pp 63-68.

Bohannon, P., et al., (2002), “From XML Schema to Relations: A Cost-Based Approach to XML Storage”, *In: Proceedings of the 18th International Conference on Data Engineering*, CA, San Jose, pp. 64-75.

Braganholo, V.P., Davidson, S.B. and Heuser, C.A., (2003a), "On the updatability of XML views over relational databases", *In: Proceedings of the International Workshop on Web and Databases*, San Diego, California, pp. 31-36.

Braganholo, V.P., Davidson, S.B. and Heuser, C.A., (2003b), "UXQuery: Building Updatable XML Views over Relational Databases", *In: Proceedings of the XVIII Simpósio Brasileiro de Bancos de Dados*, Amazonas, Brasil, pp. 26-40.

Braganholo, V.P., Davidson, S.B. and Heuser, C.A., (2004), "From XML View Updates to Relational View Updates: old solutions to a new problem", *In: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, Toronto, Canada, pp. 276-287.

Chaudhuri, S., Kaushik, R. and Naughton, J.F., (2003), "On Relational Support for XML Publishing: Beyond Sorting and Tagging", *In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, pp. 611-622.

Deutsch, A., Fernandez, M.F. and Suciu, D., (1999), "Storing Semistructured Data with STORED", *In: Proceedings of the ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania, USA, pp. 431-442.

Du, F., Amer-Yahia, S. and Freire, J., (2004), "ShreX: Managing XML Documents in Relational Databases", *In: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, Toronto, Canada, pp. 1297-1300.

Fernandez, M.F., Tan, W.C. and Suciu, D., (1999), "SilkRoute: trading between relations and XML", *The Computer Networks Journal*, vol. 33, no. 1-6, pp. 723-745.

Florescu, D. and Kossmann, D. (1999), "Storing and Querying XML Data using an RDMBS", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 22, no. 3, pp. 27-34.

Fousteris, N., Gergatsoulis, M. and Stavrakas, Y., (2007), "Storing Multidimensional XML documents in Relational Databases", *In: Proceedings of the Database and Expert Systems Applications, 18th International Conference on Database and Expert Systems Applications, DEXA 2007*, Regensburg, Germany, pp. 23-33.

Gergatsoulis, M. and Stavrakas, Y., (2003), "Representing Changes in XML Documents using Dimensions", *In: Proceedings of the Database and XML Technologies, First International XML Database Symposium, XSym 2003*, Berlin, Germany, pp. 208-222.

Gergatsoulis, M., Stavrakas Y. and Karteris, D., (2001), "Incorporating Dimensions in XML and DTD", *In: Proceedings of the Database and Expert Systems Applications, 12th International Conference, DEXA 2001*, Munich, Germany, pp. 646-656.

Gergatsoulis, M., et al., (2001), "A Web-based System for Handling Multidimensional Information through MXML", *In: Proceedings of the Advances in Databases and Information Systems 5th East European Conference, ADBIS 2001*, Vilnius, Lithuania, pp. 352-365.

Plaice, J. and Wadge, W.W., (1993), "A New Approach to Version Control", *IEEE Trans. Software Eng*., vol. 19, no. 3, pp. 268-276.

Ramanath, M., et al., (2003), "Searching for Efficient XML-to-Relational Mappings", *In: Proceedings of the First International XML Database Symposium, XSym 2003*, Berlin, Germany, pp. 19-36.

Shanmugasundaram, J., et al., (2001a), "Querying XML Views of Relational Data", *In: Proceedings of the 27th International Conference on Very Large Data Bases*, Roma, Italy, pp. 261-270.

Shanmugasundaram, J., et al., (2000), "Efficiently Publishing Relational Data as XML Documents", *In: Proceedings of the 26th International Conference on Very Large Data Bases*, Cairo, Egypt, pp. 65-76.

Shanmugasundaram, J., et al., (2001b), "A General Technique for Querying XML Documents using a Relational Database System", *SIGMOD Record*, vol. 30, no. 3, pp. 20-26.

Shanmugasundaram, J., et al., (1999), "Relational Databases for Querying XML Documents: Limitations and Opportunities", *In: Proceedings of the 25th International Conference on Very Large Data Bases*, Edinburgh, Scotland, pp. 302-314.

Stavrakas, Y., (2003), "Multidimensional Semistructured Data: Representing and Querying Context-Dependent Multifaceted Information on theWeb", PhD thesis, Department of Electrical and Computer Engineering, National Technical University of Athens, Greece.

Stavrakas, Y. and Gergatsoulis M., (2002), "Multidimensional Semistructured Data: Representing Context-Dependent Information on the Web", *In: Proceedings of the 14th International Conference in Advanced Information Systems Engineering*, Toronto, Canada, pp. 183-199.

Tatarinov, I., et al., (2002), "Storing and querying ordered XML using a relational database system", *In: Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, pp. 204-215.

Tian, F., et al., (2002), "The Design and Performance Evaluation of Alternative XML Storage Strategies", *SIGMOD Record*, vol. 31, no. 1, pp. 5-10.

Wadge, W.W., et al., (1998), "Intensional HTML", *In: Principles of Digital Document Processing, 4th International Workshop, PODDP'98*, Saint Malo, France, pp. 128-139.

Wang, F., Zhou, X. and Zaniolo, C., (2006), "Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases", *In: Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006*, Atlanta, GA, USA, pp. 131.

Wang, L., Mulchandani, M. and Rundensteiner, E.A., (2003), "Updating XQuery Views Published over Relational Data: A Roundtrip Case Study", *In: Proceedings of the Database and XML Technologies, First International XML Database Symposium, XSym 2003*, Berlin, Germany, pp. 223-237.

Yoshikawa, M., et al., (2001), "XRel: a path-based approach to storage and retrieval of XML documents using relational databases", *ACM Transactions on Internet Technology*, vol. 1, no. 1, pp. 110-141.

Nikolaos Fousteris (nfouster@ionio.gr) received his Diploma in Informatics from the Technological Educational Institute of Athens, Greece, and his "M.Sc. with Distinction in High Speed Networks and Distributed Systems" from Oxford Brookes University, UK. From 2004 he is working as a system administrator for the Greek Research Network. He is also a PhD candidate student at the Department of Archive & Library Sciences of the Ionian University in Greece (address: Ioannou Theotoki 72, 49100 Corfu, Greece). His research has recently been focused on storing, querying and updating XML documents in relational databases. More information about his research and publications is online: http://www.ionio.gr/~nfouster/

Prof. Manolis Gergatsoulis (manolis@ionio.gr) received his Diploma in Physics in 1983, the M.Sc. and the Ph.D. degrees in computer science in 1986 and 1994 respectively all from the University of Athens, Greece. From 1996 to 2000 he worked as a Research Associate in the Institute of Informatics and Telecommunications, NCSR `Demokritos', Athens. From 2000 to 2002 he worked as a Visiting Professor at the Department of Technology Education and Digital Systems and the Department of Informatics, of the University of Piraeus. He has also worked as a Visiting Professor at the Department of Mathematics of the Technological Educational Institute of Piraeus and the Department of Informatics of the Technological Educational Institute of Athens. He is currently an Assistant Professor at the Department of Archive & Library Sciences of the Ionian University in Greece (address: Ioannou Theotoki 72, 49100 Corfu, Greece). His research interests include XML and semistructured databases, context-aware data management, web databases and semantic web, logic programming, program and query transformations and synthesis, and theory of programming languages. Web page: http://www.ionio.gr/~manolis

Dr. Yannis Stavrakas (yannis@imis.athena-innovation.gr) received his Diploma in Physics from the University of Athens, Greece, and his "M.Sc. with Distinction in Computer Science" from University College London, UK. From 1993 to 1997 he worked as a software engineer and technical manager. From 1998 to 2003 he worked as a research fellow at the Institute of Informatics and Telecommunications of the National Center for Scientific Research "Demokritos". In 2003 he completed his Ph.D. at the National Technical University of Athens, Greece. He is currently a researcher at the Institute for the Management of Information Systems – R.C. "Athena" (G.Mpakou 17, 11524 Athens GREECE). His research interests include context-aware data management, web data management, database preservation, data provenance, digital libraries, management of scientific data, and situation-aware mobile computing. Web page: http://www.ipsyp.gr/en/imis/home.html