Updating Typical XML Views

Jixue Liu[1], Chengfei Liu[2], Theo Haerder[3], and Jeffery Xu Yu[4]

[1] School of CIS, University of South Australia
jixue.liu@unisa.edu.au
[2] Faculty of ICT, Swinburne University of Technology
cliu@swin.edu.au
[3] Dept. of CS, Technical University of Kaiserslautern
haerder@informatik.uni-kl.de
[4] Dept. of Sys. Eng. and Eng. Management, Chinese University of HK
yu@se.cuhk.edu.hk

Abstract. View update is the problem of translating an update to a view to some updates to the source data of the view. In this paper, we formally define the problem, show the factors determining XML view update translation, and propose a translation solution for two specific but typical settings of the view update problem. We prove that the translated source updates are precise and they generalize the solutions to the problem with similar settings in the relational database.

Keywords: XML data, view update, update translation, virtual views.

1 Introduction

A (virtual) view is defined with a query over some source data of a database. The query is called the view definition which determines what data appears in the view. The data of the view, called a **view instance**, is often not stored in the database but is derived from the source data on the fly using the view definition every time when the view is selected.

In database applications, many users do not have privileges to access all the data of a database. They are often given a view of the database so that they can retrieve only the data in the view. In data integration applications, user's access to the source data becomes even more impractical because of security. When these users need to update the data of the database, they put their updates against the view, not against the source data, and expect that the view instance is changed when it is accessed next time. This type of updates is called a view update. *Because of its important use*, view update has a long research history [1,9,11,12,6,4,13]. The work in [5] discusses detailed semantics of view updates in many scenarios.

Unfortunately, view updates cannot be directly applied to the view instance as it is not stored physically and is derived on the fly when required (virtual view). Even in the cases where the view instance is stored (materialized view), which is not the main focus of this paper, applying updates to the instance may cause inconsistencies between the source data and the instance. To apply a view update to a virtual view, a translation process is required to translate the view update to some *source updates*. When the source data is changed, the data in the view will be changed next time when the view is selected. To the user of the view, it seems that the view update has been successfully applied to the view instance.

Let V be a view definition, V^i the view instance, S^i the source data of the view, $V(S^i)$ the evaluation of V against S^i. Then $V^i = V(S^i)$. Assume that the user wants to apply a view update δV to V^i as $\delta V(V^i)$. View update translation is to find a process that takes V and δV as input and produces a source update δS to S^i such that next time when the user accesses the view, the view instance appears changed and is as expected by the user. That is, for any S^i and $V^i = V(S^i)$,

$$V(\delta S(S^i)) = \delta V(V^i) \tag{1}$$

Two typical anomalies, view side-effect and source document over-update, are easily introduced by the translation process although they are update policy dependent [9]. View side-effect [13] occurs if the translated source update causes more-than-necessary change to the source data which leads to more-than-expected change to the view instance. View side-effect makes Equation (1) violated.

Over-updates may also happen to a source document. An over-update to a source document causes the source data irrelevant to the view to be changed, but keeps the equation satisfied. A source document over-update is incorrect as it changes information that the user did not expect to change.

A **precise** translation of a view update should produce source updates that (1) result in necessary (as the user expects) change to the view instance, (2) do not cause view side-effect, and (3) do not cause over-updates to the source documents.

In the *relational database*, much work has been done on view update and the problem has been well understood [1,9,11]. In case of updating *XML views over relational databases*, updates to XML views need to be translated to updates to the base relational tables. The works in [4,13] propose two different approaches to the problem. The work in [4] translates an XML view to some relational views and an update to the XML view to updates to the relational views. It then uses the relational approach to derive updates to the base tables. The work in [13] derives a schema for the XML view and annotates the schema based on keys of relational tables and multiplicities. An algorithm is proposed to use the annotation to determine if a translation is possible and how the translation works. Both works assume keys, foreign keys and the join operator based on these two types of constraints. Another work, technical report [6], proposes brief work on updating hypertext views defined on relational databases. In the case of updating *pure XML views*, the views where the views and their source data are all modelled in XML, no direct work has been done. To the best of our knowledge, the only work relating to XML view update is [8] which proposes a middle language and a transformation system to derive view instance from source data, and to derive source data from a **materialized** view instance, and assumes XQuery as the view definition language. We argue that with the view update problem, only view updates are available but not the view instance (not materialized). Consequently view update techniques are still necessary. The work in [3] is on the exact topic as this paper, but it restricts its views only to node relabeling and selection; no restructuring is allowed in the view definitions.

In this paper, we look into the view update problem in pure XML context. This means that both source data and the view are in XML format. We assume that base XML documents have no schema and no constraints information available. Assuming no constraints makes the solutions developed more general.

The view update problem in the relational database is difficult as not all view updates are translatable. For example, if a view V is defined by a Cartesian product of two tables R and S, an update inserting a new tuple to the view instance is not translatable because there is no unique way to determine the change(s) to R and S. The view update problem in XML becomes much harder. The main reason is that the source data and view instances are modeled in trees and trees can nest in arbitrary levels. This fundamental difference makes the methods of translating view updates in the relational database not applicable to translating XML view updates. A typical example is that the selection and the projection operations in the relational database do not have proper counterparts in XML. The view update problem in XML has many distinct cases that do not exist in the view update problem in the relational database (see Sections 3 and 4 for details). To the best of our knowledge, our work is the first proposing a solution to the view update problem in XML.

We notice that the view update problem is different from the view maintenance problem. The former aims to translate a view update to a virtual view to a source update while the latter aims to translate a source update to a view update to a materialized view. The methods for one do not work for the other.

We make the following contributions in this paper. Based on the view definition and the update language presented later, we present a formal definition of the view update problem and identify the factors determining the view update problem. Secondly, we propose a translation solution to translate 'typical' view updates. We prove that the translated source update from the algorithm is precise in the typical settings. Furthermore, we show that the solution we propose generalizes the solution to the problem in the relational database in similar settings.

The paper is organized as follows. Section 2 shows the view definition language, the update language, and the preciseness of view update translation. In Section 3, we propose an algorithm and show that the translation obtained by the algorithm is a precise translation. In Section 4, we identify a 'join' case where a translated update is precise. Section 5 concludes the paper.

2 Preliminaries

In this section, we define basic notation, introduce the languages for view definitions and updates, and define the XML view update problem.

Definition 1 (tree)**.** An XML document can be represented as an ordered tree. Each node of the tree has a unique identifier v_i, an element name *ele* also called a **label**, and either a text string *txt* or a sequence of child trees $T_{j_1}, \cdots, T_{j_n}$. That is, a node is either $(v_i : ele : txt)$ or $(v_i : ele : T_{j_1}, \cdots, T_{j_n})$. When the context is clear, some or all of the node identifiers of a tree may not present explicitly in this paper. A tree without all node identifiers is called a **value tree**. Two trees T_1 and T_2 are (value) **equal**, denoted by $T_1 = T_2$, if they have identical value trees. If a tree T_1 is a subtree in T_2, T_1 is said **in** T_2 and denoted by $T_1 \in T_2$. □

For example, the document `<root><A><B>1</B></A><A><B>2</B></A></root>` is represented by $T = (v_r\!:\!root\!:\ (v_0\!:\!A\!:\!(v_1\!:\!B\!:\!1)), (v_2\!:\!A\!:\!(v_3\!:\!B\!:\!2)))$. The value tree of T is $(root\!:\ (A\!:\!(B\!:\!1)), (A\!:\!(B\!:\!2)))$.

Definition 2. A **path** p is a sequence of element names $e_1/e_2/\cdots/e_n$ where all names are distinct. The function $L(p)$ returns the last element name e_n.

Given a path p and a sequence of nodes $v_1, \cdots, v_n$ in a tree, if for every node $v_i \in [v_2, \cdots, v_n]$, v_i is labeled by e_i and is a child of v_{i-1}, then $v_1/\cdots/v_n$ is a **doc path** conforming to p and the tree rooted at v_n is denoted by $T^p_{v_n}$. □

2.1 View Definition Language

We assume that a view is defined in a dialect of the *for-where-return* clauses of XQuery [2].

Definition 3 (V)**.** A view is defined by

```
<v>{ for $x_1$ in $p_1$,   $\cdots$,   $x_n$ in $p_n$
     where  $cdn(x_1, \cdots, x_n)$
     return $rtn(x_1, \cdots, x_n)$   }</v>
```

where $p_1, \cdots, p_n$ are paths (Definition 2) proceeded by $doc()$ or x_i;
$cdn(x_1, \cdots, x_n) ::= x_i/\mathcal{E}_i = x_j/\mathcal{E}_j$ and $\cdots$ and $x_k/\mathcal{E}_k = strVal$ and $\cdots$;
$rtn(x_1, \cdots, x_n) ::=$ <$\mathfrak{e}$> $\{x_u/\gamma_u\} \cdots \{x_v/\gamma_v\}$ </$\mathfrak{e}$>;
$\gamma, \mathcal{E}$ are paths, and the last elements of all $x_u/\gamma_u, \cdots, x_v/\gamma_v$ are distinct. □

We note that the paths in the *return* clause are denoted by x_i/γs because these expressions are specially important in view update translation. We purposely leave out the \$ sign proceeding a variable in the XQuery language.

Definition 4 (context-based production)**.** By the formal semantics of XQuery [7], the semantics of the language is

```
for $x_1$ in $p_1$ return
   for $x_2$ in $p_2$ return
      ...
         for $x_n$ in $p_n$ return
            if cdn($x_1$,...,$x_n$)=true
               return rtn($x_1$,...,$x_n$)
```

The for-statement produces tuples $<x_1, ..., x_n>$, denoted by $fortup(V)$, where the variable x_i represents a binding to one of the sub trees located by p_i within the context defined by $x_1, \cdots, x_{i-1}$. This process is called **context-based production**. □

For each tuple satisfying the condition $cdn(x_1, ..., x_n)$, the function $rtn(x_1, \cdots, x_n)$ produces a tree, called an $\mathfrak{e}$-tree, under the root node of the view. That is, V maps a tuple to an $\mathfrak{e}$-tree. The children of the $\mathfrak{e}$-tree are the **γ-trees** selected by all the expressions x_i/γ_is (for all i) from the tuple. A tuple is mapped to one and only one $\mathfrak{e}$-tree and an $\mathfrak{e}$-tree is for one and only one tuple. A γ-tree of a tuple is uniquely mapped to a child of the $\mathfrak{e}$-tree of the tuple and a child of an $\mathfrak{e}$-tree is for one and only one γ-tree of its tuple. This is illustrated in Figure 1(a) where x_c and x_t are two variables, T^{x_t/γ_t} and T^{x_c/γ_c} are γ-trees, and the γ-trees appear as children of the $\mathfrak{e}$ node. We note that the one-to-one mapping is between a tree in the tuple (not the source document) and a tree under an $\mathfrak{e}$ node in the view.

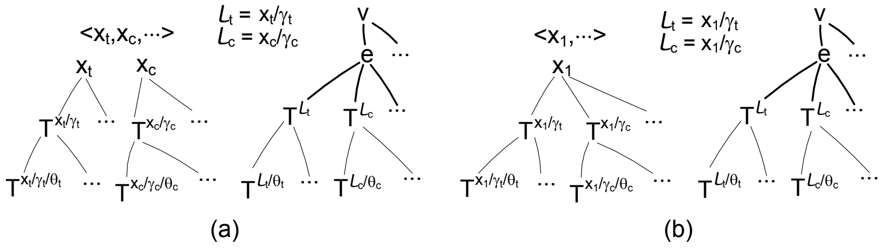

Fig. 1. Each of tuples is mapped to an $\mathfrak{e}$-tree

The path of a node s in the view has the following format:

$$v/\mathfrak{e}/\mathcal{L}_i/\theta_i \tag{2}$$

where

$$\mathcal{L}_i = L(x_i/\gamma_i) \tag{3}$$

returns the last element name $\mathcal{L}_i$ of x_i/γ_i, an expression in $rtn(x_1, ..., x_n)$, and θ_i is a path following $\mathcal{L}_i$ in the view. When $\mathcal{L}_i/\theta_i$ is not empty, the path in the source document corresponding to $v/\mathfrak{e}/\mathcal{L}_i/\theta_i$ is

$$x_i/\gamma_i/\theta_i \tag{4}$$

The view definition has some properties important to view update translation. Firstly because of context-based production, a binding of variable x_i may be copied into $x_i^{(1)}, \cdots, x_i^{(m)}$ to appear in multiple tuples:

$$\begin{array}{l} < \cdots, x_i^{(1)}, \cdots, x_{j[1]}, \cdots > \\ \cdots \\ < \cdots, x_i^{(m)}, \cdots, x_{j[m_j]}, \cdots > \end{array}$$

where $x_{j[1]}, \cdots, x_{j[m_j]}$ are different bindings of x_j. Each tuple satisfying the condition $cdn(x_1, \cdots, x_n)$ is used to build an $\mathfrak{e}$-tree. As a result of x_i being copied, the subtrees of x_i will be copied accordingly to appear in multiple $\mathfrak{e}$-trees in the view.

Secondly, a tree may have zero or many sub trees located by a given path p. That is, given a tree bound to x_i, the path expression x_i/p may locate zero or many sub trees $T_1^{x_i/p}, \cdots, T_{n_p}^{x_i/p}$ in x_i. This is true both in the source documents and in the view.

Thirdly, two path expressions x_i/γ_i and x_j/γ_j generally may have the same last element name, i.e., $L(x_i/\gamma_i) = L(x_j/\gamma_j)$. For example, if x_i represents an employee while x_j represents a department, then $x_i/name$ and $x_j/name$ will present two types of names in the same $\mathfrak{e}$-tree. This make the semantics of the view data not clear. This is the reason why we assume that all $L(x_i/\gamma_i)$s are distinct.

Example 1. Consider the view definition below and the source document shown in Figure 2(a). The view instance is shown in Figure 2(b).

```
<v>{for x in doc("r")/r/A,   y in x/C,   z in x/H
     where y/D=z and z="1"
     return <e>{x/B}{x/C}{y/F/G}{z}</e>
}</v>
```

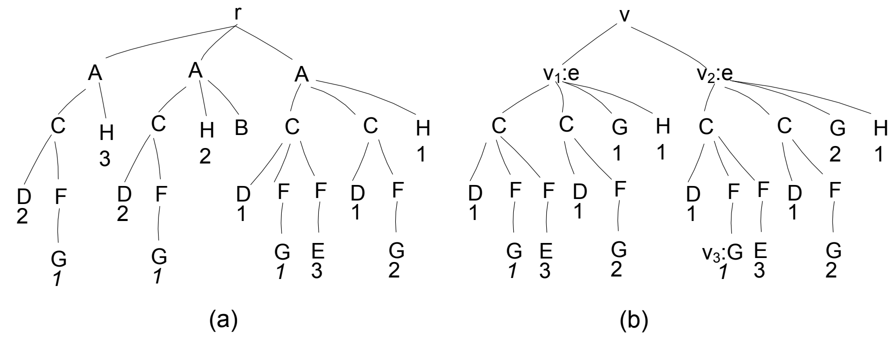

Fig. 2. Source document r and view v

From the view definition, $\gamma_1 = B$, $\gamma_2 = C$, $\gamma_3 = F/G$, and $\gamma_4 = \phi$. $L(x/\gamma_1) = \mathcal{L}_1 = B$, $L(x/\gamma_2) = \mathcal{L}_2 = C$, $L(y/\gamma_3) = \mathcal{L}_3 = G$, and $L(z/\gamma_4) = \mathcal{L}_4 = H$.

Formula (2) is exemplified as the following. The node v_3 in the view has the path $v/e/C/F/G$ where C is $\mathcal{L}_2 = L(x/\gamma_2)$ and F/G is θ. The node v_2 is an $\mathfrak{e}$ node and its path is v/e where $\mathcal{L}_i/\theta_i$ is ϕ.

The example shows the following.

- The expression x/B $(=x/\gamma_1)$ of the *return* clause has no tree in the two $\mathfrak{e}$-trees.
- The path expression x/C $(=x/\gamma_2)$ has multiple trees in each $\mathfrak{e}$-tree.

- The trees of x/C are duplicated in the view and so are their sub trees.
- Each of some x/C trees has more than one $x/C/F$ ($=x/\gamma_2/\theta$) sub trees.

2.2 The Update Language

The update language we use follows the proposal [10] extended from XQuery.

Definition 5 (δV)**.** A view update statement has the format of

for $\bar{x}_1$ in $\bar{p}_1$, $\cdots$, $\bar{x}_u$ in $\bar{p}_u$
where $\bar{x}_c/\bar{p}_c = strValu$
update $\bar{x}_t/\bar{p}_t$ (delete T | insert T)

where $\bar{x}_c, \bar{x}_t \in [\bar{x}_1, \cdots, \bar{x}_u]$, $\bar{p}_1, \cdots, \bar{p}_u$ are paths (Definition 2) proceeded by v or $\bar{x}_i$; $\bar{p}_c, \bar{p}_t$ are paths; all element names in the paths are elements names in the view. $\bar{x}_c/\bar{p}_c$ and $\bar{x}_t/\bar{p}_t$ are called the **(update) condition path** and **(update) target path** respectively. □

The next procedure maps a path in the update statement to a path in the source document.

Procedure 1 (mapping)**.**
(i). Replace the variables in $\bar{x}_c/\bar{p}_c$ and $\bar{x}_t/\bar{p}_t$ by their paths in the *for*-clause until the first element name becomes v. Thus the full paths of $\bar{x}_c/\bar{p}_c$ and $\bar{x}_t/\bar{p}_t$ in the view are built and will have the format of $v/\mathfrak{e}/\mathcal{L}_c/\theta_c$ and $v/\mathfrak{e}/\mathcal{L}_t/\theta_t$ as shown in Formula (2).
(ii). Search in the *return* clause of V using $\mathcal{L}_c$ and $\mathcal{L}_t$ to identify the expressions x_c/γ_c and x_t/γ_t. Append θ_t and θ_c to them respectively as $x_c/\gamma_c/\theta_c$ and $x_t/\gamma_t/\theta_t$. Thus, $x_c/\gamma_c/\theta_c$ and $x_t/\gamma_t/\theta_t$ are the source paths of $v/\mathfrak{e}/\mathcal{L}_c/\theta_c$ and $v/\mathfrak{e}/\mathcal{L}_t/\theta_t$. □

With this mapping, the update statement δV can be represented by the following abstract form:

$$(\bar{p}_s;\ \ v/\mathfrak{e}/\mathcal{L}_c/\theta_c = strValu;\ \ v/\mathfrak{e}/\mathcal{L}_t/\theta_t;\ \ del(T)|ins(T)) \qquad (5)$$

where
- $v/\mathfrak{e}/\mathcal{L}_c/\theta_c$ is the **full** update condition path (int the view) for $\bar{x}_c/\bar{p}_c$, $v/\mathfrak{e}/\mathcal{L}_t/\theta_t$ the **full** target path for $\bar{x}_t/\bar{p}_t$;
- $\bar{p}_s$ is the maximal common front part of $v/\mathfrak{e}/\mathcal{L}_c/\theta_c$ and $v/\mathfrak{e}/\mathcal{L}_t/\theta_t$.

The semantics of an update statement is that under a context node identified by $\bar{p}_s$, if a sub tree identified by $v/\mathfrak{e}/\mathcal{L}_c/\theta_c$ satisfies the update condition, all the sub trees identified by $v/\mathfrak{e}/\mathcal{L}_t/\theta_t$ will be applied the update action (del(T) or ins(T)). The sub tree $T^{v/\mathfrak{e}/\mathcal{L}_c/\theta_c}$ is called the **condition tree** of $T^{v/\mathfrak{e}/\mathcal{L}_t/\theta_t}$. A sub tree is updated only if it has a condition tree and the condition tree satisfies the update condition. An update target and its condition trees are always within a tuple when the view definition is evaluated and are in an $\mathfrak{e}$-tree in the view after the evaluation.

We note that because of the context-based production in the view definition, the same update action may be applied to a target node for multiple times. For example, if x is a binding and the context-based production produces two tuples as $< x^{(1)}, \cdots >$ and $< x^{(2)}, \cdots >$. If the update condition and target are all in x, x will be updated twice with the same action, each action being fired for each tuple. We assume that only the effect of the first application is taken and the effect of all other applications is ignored.

Based on the structure of the target path $tp = v/\mathfrak{e}/\mathcal{L}_t/\theta_t$, updates may happen to different types of nodes in the view.

- When $\mathcal{L}_t/\theta_t \neq \phi$, the update happens to the nodes within a γ-tree.
- When $tp = v/\mathfrak{e}$, the update will add or delete a γ-tree.
- When $tp = v$ (in this case, $\bar{p}_s = v$), the update will add or delete an $\mathfrak{e}$-tree.

In this paper, we only deal with the first case and leave the solutions to the last two cases to be future work.

2.3 The View Update Problem

Definition 6 (Precise Translation)**.** Let V be a view definition and S be the source of V. Let δV be an update statement to V. Let δS be the update statement to S translated from δV. δS is a precise translation of δV if, for any instance S^i of S and $V^i = V(S^i)$,

(1) δS is correct. That is, $V(\delta S(S^i)) == \delta V(V^i)$ is true; and

(2) δS is minimal. That is, there does not exist another translation $\delta S'$ such that ($\delta S'$ is correct, i.e., $V(\delta S'(S^i)) = V(\delta S(S^i)) = \delta V(V^i)$ and there exists a tree T in S^i and T is updated by δS but not $\delta S'$). □

We note that Condition (1) also means that the update δS will not cause view-side-effect. Otherwise, $V(\delta S(S^i))$ would contain more, less, or different updated trees than those in $\delta V(V^i)$.

Definition 7 (the view update problem)**.** Given a view V and a view update δV, the problem of view update is to (1) develop a translation process P, and show that the source update δS obtained from P is precise, or (2) prove that a precise translation of δV does not exist. □

3 Update Translation When $\mathcal{L}_t/\theta_t \neq \phi$ and $x_c = x_t$

In this section, we investigate update translation when the update is to change a γ-tree of the view and the mappings of the update condition path and the update target path refer to the same variable in the view definition. We present Algorithm 1 and the statement δS as the solution for view update translation in this case.

By the algorithm, the following source update is derived.

```
δS:  for $x_1$ in $p_1$,    $\cdots$,    $x_n$ in $p_n$
      where $cdn(x_1, \cdots, x_n)$    and    $x_c/\gamma_c/\theta_c = strValu$
      update $x_t/\gamma_t/\theta_t$  (insert $T$ | delete $T$)
```

Algorithm 1: A translation algorithm

```
Input: view definition $V$, view update $\delta V$
Output: translated source update $\delta S$
1 begin
2     make a copy of $V$ and reference the copy by $\delta S$ ;
3     remove $rtn()$ from $\delta S$ ;
4     from the view update $\delta V$, following Procedure 1, find mappings $x_c/\gamma_c/\gamma_c$
      and $x_t/\gamma_t/\gamma_t$ for the condition path $\bar{x}_c/\bar{p}_c$ and the target path $\bar{x}_t/\bar{p}_t$ ;
5     make a copy of $\delta V$ and reference the copy by $\delta V_c$ ;
6     in $\delta V_c$, replace $\bar{x}_c/\bar{p}_c$ and $\bar{x}_t/\bar{p}_t$ by $x_c/\gamma_c/\gamma_c$ and $x_t/\gamma_t/\gamma_t$ respectively ;
7     append the condition in the where clause of $\delta V_c$ to the end of the where
      clause in $\delta S$ using logic and ;
8     append the update clause of $\delta V_c$ after the where clause of $\delta S$
```

We now develop the preciseness of the translation. We recall the notation that $fortup(V)$ means the tuples of the context-based production (Definition 4) of V. The symbols x_c, $x_{c[1]}$ and $x_{c[2]}$ are three separate bindings of x_c. The symbols $x_c^{(1)}$ and $x_c^{(2)}$ are two copies of x_c.

Lemma 1. *Given a tuple $t = <x_t, x_c, \cdots> \in fortup(V)$ and its $\mathfrak{e}$-tree e, (1) if $T^{x_t/\gamma_t/\theta_t}$ (a tree for the path $x_t/\gamma_t/\theta_t$) in t is updated by δS, then all the trees identified by $x_t/\gamma_t/\theta_t$ in t are updated by δS, and all the trees identified by $\mathcal{L}_t/\theta_t$ in e are updated by δV. (2) if $T^{\mathcal{L}_t/\theta_t}$ (a tree for the path $\mathcal{L}_t/\theta_t$) in e is updated by δV, then all the trees identified by $\mathcal{L}_t/\theta_t$ in e are updated by δV, and all the trees identified by $x_t/\gamma_t/\theta_t$ in t are updated by δS.*

The lemma is correct because of the one-to-one correspondence between a tuple and an $\mathfrak{e}$-tree and between t's γ-trees and e's children, and because all the trees identified by $x_t/\gamma_t/\theta_t$ in t share the same condition tree(s) identified by $x_c/\gamma_c/\theta_c$ in x_c of t, and all the trees identified by $\mathcal{L}_t/\theta_t$ in e share the same condition tree(s) identified by $\mathcal{L}_c/\theta_c$ in e.

Lemma 2. *Assume a tuple $t = <x_t, x_c, \cdots> \in fortup(V)$. After a tree $T^{x_t/\gamma_t/\theta_t}$ in x_t is updated by δS, t becomes $t' = <x'_t, x_c, \cdots>$. If $x_t/\gamma_t/\theta_t$ is not a prefix of any of the paths in the where clause of δS and if t satisfies $cdn()$ of V, t' also satisfies $cdn()$ of V.*

The lemma is correct because the subtrees in the tuple used to test $cdn()$ are not changed by δS when the condition of the lemma is met.

Lemma 3. *Assume a tuple $t = <x_t, x_c, \cdots> \in fortup(V)$ and its $\mathfrak{e}$-tree e. If $T^{x_c/\gamma_c/\theta_c}$ in t satisfies $x_c/\gamma_c/\theta_c = strValu$, $T^{\mathcal{L}_c/\theta_c}$ in e satisfies $\mathcal{L}_c/\theta_c = strValu$ and vice versa.*

The correctness of the lemma is guaranteed by the one-to-one correspondence between t's γ-trees and e's children.

Lemma 4. *Assume a tuple $t = <x_t, x_c, \cdots> \in fortup(V)$, its $\mathfrak{e}$-tree e, $T^{x_t/\gamma_t/\theta_t}$ in x_t, and $T'^{\mathcal{L}_t/\theta_t}$ in e. Obviously $T = T'$. As δS and δV have the same update action, if x_c satisfies the update condition, $\delta S(T) = \delta V(T')$.*

Theorem 1. *Given view V and the view update δV where $\mathcal{L}_t/\theta_t \neq \phi$ and $x_c = x_t$, the update δS is a precise translation of the view update δV if and only if $x_t/\gamma_t/\theta_t$ does not proceed any path in the where clause of δS.*

Proof. We show only the 'if' proof. The proof of 'only if' can be done in a similar way. We follow Definition 6 to show that if the condition is true, the translation is precise. Without losing generality, we assume that $x_t = x_c = x_1$. Figure 1(b) illustrates the relationship between a variable binding x_1 in the tuple $< x_1, \cdots >$ and the $\mathfrak{e}$-tree built from the tuple. $T^{x_1/\gamma_t/\theta_t}$ and $T^{x_1/\gamma_c/\theta_c}$are an update target tree and a condition tree respectively. $T^{x_1/\gamma_t/\theta_t}$'s children will be deleted or a new child will be inserted.

(1) Correctness: $V(\delta S(S^i)) = \delta V(V(S^i))$

We firstly show that duplicated γ-trees are updated consistently in the view and then show that each side of the equation is contained in the other side. The reason why consistency is important here is that if the update changes one copy of a duplicated γ-tree without updating the others, the translation will have side effect.

Consistency: Consider two tuples $t_1 =< x_1^{(1)}, \cdots >$ and $t_2 =< x_1^{(2)}, \cdots >$ in $fortup(\delta S)$ where $x_1^{(1)}$ and $x_1^{(2)}$ are copies of x_1. Let e_1 and e_2 be two $\mathfrak{e}$-trees constructed from t_1 and t_2 respectively by V. Then, because of $x_t = x_c = x_1$, either both e_1 and e_2 are updated by δV or none is updated.

$\supseteq$: Let $T^{\mathcal{L}_t/\theta_t}$ be a tree in an $\mathfrak{e}$-tree e of $V(S^i)$ updated to $\bar{T}^{\mathcal{L}_t/\theta_t}$ by δV (e becomes e' after the update). We show that $\bar{T}^{\mathcal{L}_t/\theta_t}$ is in e' of $V(\delta S(S^i))$. In fact, that $T^{\mathcal{L}_t/\theta_t}$ is in $V(S^i)$ means that there exists one and only one tuple $t = <x_1, \cdots>$ in $fortup(V)$ satisfying $cdn()$, that in the tuple, $x_1/\gamma_t/\theta$ identifies the source tree $T^{x_1/\gamma_t/\theta_t}$ of $T^{\mathcal{L}_t/\theta_t}$. $T^{\mathcal{L}_t/\theta_t}$ being updated by δV means that there exists a condition tree $T^{\mathcal{L}_c/\theta_c}$ in e and the condition tree satisfies $v/\mathfrak{e}/\mathcal{L}_c/\theta_c = strValu$.

On the other side, because V and δS have the same *for* clause, t is in $fortup(\delta S)$. Because $T^{\mathcal{L}_c/\theta_c}$ makes $v/\mathfrak{e}/\mathcal{L}_c/\theta_c = strValu$ true, so $T^{x_1/\gamma_c/\theta_c}$ makes $x_1/\gamma_c/\theta_c = strVal$ true (Lemma 3). This means $T^{x_1/\gamma_t/\theta_t}$ is updated by δS and becomes $\hat{T}^{x_1/\gamma_t/\theta_t}$. Thus t becomes $t' =< \bar{x}_1, \cdots >$. Because of Lemma 4, $\bar{T}^{x_1/\gamma_t/\theta_t}$=$\hat{T}^{x_1/\gamma_t/\theta_t}$. Because of the condition of the theorem and Lemma 2, t' satisfies $cdn()$ and generalizes e' in the view. So $\bar{T}^{\mathcal{L}_t/\theta_t}$ is in $V(\delta S(S^i))$.

$\subseteq$: Let $T_1^{\mathcal{L}_t/\theta_t}$ and $T_2^{\mathcal{L}_t/\theta_t}$ be two trees in $V(\delta S(S^i))$ and their source tree(s) are updated by δS. We show that $T_1^{\mathcal{L}_t/\theta_t}$ and $T_2^{\mathcal{L}_t/\theta_t}$ are in $\delta V(V(S^i))$. There are three cases: (a) $T_1^{\mathcal{L}_t/\theta_t}$ and $T_2^{\mathcal{L}_t/\theta_t}$ share the same source tree $T^{x_1/\gamma_t/\theta_t}$ (they must appear in different $\mathfrak{e}$-trees in the view), and (b) $T_1^{\mathcal{L}_t/\theta_t}$ and $T_2^{\mathcal{L}_t/\theta_t}$ have different source trees $T_1^{x_1/\gamma_t/\theta_t}$ and $T_2^{x_1/\gamma_t/\theta_t}$. Case (b) has two sub cases: (b.1) $T_1^{\mathcal{L}_t/\theta_t}$ and $T_2^{\mathcal{L}_t/\theta_t}$ appear in the same $\mathfrak{e}$-tree in the view, and (b.2) $T_1^{\mathcal{L}_t/\theta_t}$ and $T_2^{\mathcal{L}_t/\theta_t}$ appear in different $\mathfrak{e}$-trees.

Case (a): That $T^{x_1/\gamma_t/\theta_t}$ is updated by δS means that there exist two tuples $<x_1^{(1)}, \cdots>$ and $<x_1^{(2)}, \cdots>$ in $fortup(\delta S)$ such that $x_1^{(1)} = x_1^{(2)}$, both tuples satisfy *cdn*(), and there exists condition tree $T^{x_1/\gamma_c/\theta_c}$ in each tuple satisfying $x_c/\gamma_c/\theta_c = strValu$, $T^{x_1/\gamma_t/\theta_t}$ is updated to $\bar{T}^{x_1/\gamma_t/\theta_t}$ by δS (two update attempts with the same action for the two tuples, only the effect of the first attempt is taken). After the update, the tuples become $t'_1 = <\bar{x}_1^{(1)}, \cdots>$ and $t'_2 = <\bar{x}_1^{(2)}, \cdots>$. By Lemma 2, t'_1 and t'_2 satisfy *cdn* of V and produce $e_1, e_2 \in V(\delta S(S^i))$ and $\bar{T}_1^{\mathcal{L}_t/\theta_t} \in e_1$ and $\bar{T}_2^{\mathcal{L}_t/\theta_t} \in e_2$.

On the other side, when V is evaluated against S^i, x_1 is copied to two tuples $t_1 = <x_1^{(1)}, \cdots>$ and $t_2 = <x_1^{(2)}, \cdots>$ in $fortup(V)$ and each of the tuples satisfies *cdn*(). They produce $\mathfrak{e}$-trees e'_1 and e'_2. Because each tuple has a condition tree $T^{x_1/\gamma_c/\theta_c}$ satisfying $x_c/\gamma_c/\theta_c = strValu$, by Lemma 3, each of e'_1 and e'_2 has $T^{\mathcal{L}_c/\theta_c}$ satisfying $\mathcal{L}_c/\theta_c = strValu$ and each has a $T^{\mathcal{L}_t/\theta_t}$. Thus $T_1^{\mathcal{L}_t/\theta_t} \in e'_1$ and $T_2^{\mathcal{L}_t/\theta_t} \in e'_2$ will be updated to $\bar{T}_1^{\mathcal{L}_t/\theta_t}$ and $\bar{T}_2^{\mathcal{L}_t/\theta_t}$ by δV. e'_1 and e'_2 become e_1 and e_2 in $\delta V(V(S^i))$.

Case (b.1): That $T_1^{x_1/\gamma_t/\theta_t}$ and $T_2^{x_1/\gamma_t/\theta_t}$ are updated by δS and that they appear in different $\mathfrak{e}$-trees mean that there are two tuples $<x_{1[1]}, \cdots>$ and $<x_{1[2]}, \cdots>$ where $x_{1[1]}$ and $x_{1[2]}$ are different bindings of x_1, $T_1^{x_1/\gamma_t/\theta_t} \in x_{1[1]}$, $T_2^{x_1/\gamma_t/\theta_t} \in x_{c[2]}$, and each of tuples satisfies *cdn*() and $x_c/\gamma_c/\theta_c = strValu$. $T_1^{x_1/\gamma_t/\theta_t}$ and $T_2^{x_1/\gamma_t/\theta_t}$ become $\bar{T}_1^{x_1/\gamma_t/\theta_t}$ and $\bar{T}_2^{x_1/\gamma_t/\theta_t}$ after the update and mapped to $\bar{T}_1^{\mathcal{L}_t/\theta_t}$ and $\bar{T}_2^{\mathcal{L}_t/\theta_t}$ in two different $\mathfrak{e}$-trees of $V(\delta S(S^i))$. Following the same argument of Case (a), $\bar{T}_1^{\mathcal{L}_t/\theta_t}$ and $\bar{T}_2^{\mathcal{L}_t/\theta_t}$ are in $\delta V(V(S^i))$.

Case (b.2): That $T_1^{x_1/\gamma_t/\theta_t}$ and $T_2^{x_1/\gamma_t/\theta_t}$ are updated by δS and that they appear in a single $\mathfrak{e}$-tree mean that there is one and only one tuple $<x_1, \cdots>$ where $T_1^{x_1/\gamma_t/\theta_t}, T_2^{x_1/\gamma_t/\theta_t} \in x_1$. The tuple satisfies *cdn*() and there is a tree $T^{x_1/\gamma_c/\theta_c}$ in the tuple satisfying $x_1/\gamma_c/\theta_c = strValu$. $T_1^{x_1/\gamma_t/\theta_t}$ and $T_2^{x_1/\gamma_t/\theta_t}$ become $\bar{T}_1^{x_1/\gamma_t/\theta_t}$ and $\bar{T}_2^{x_1/\gamma_t/\theta_t}$ after the update and mapped to $\bar{T}_1^{\mathcal{L}_t/\theta_t}$ and $\bar{T}_2^{\mathcal{L}_t/\theta_t}$ in a single $\mathfrak{e}$-tree of $V(\delta S(S^i))$. On the other side, as $T_1^{x_1/\gamma_t/\theta_t}$ and $T_2^{x_1/\gamma_t/\theta_t}$ are mapped to a single $\mathfrak{e}$-tree e and share the same condition tree $T^{x_1/\gamma_c/\theta_c}$, $T_1^{\mathcal{L}_t/\theta_t}$ and $T_2^{\mathcal{L}_t/\theta_t}$ share the same condition tree $T^{\mathcal{L}_c/\theta_c}$ in e and will be updated by δV. So $\bar{T}_1^{\mathcal{L}_t/\theta_t}$ and $\bar{T}_2^{\mathcal{L}_t/\theta_t}$ are in the $\mathfrak{e}$-tree of $\delta V(V(S^i))$.

(2) δS is minimal

We prove by contrapositive. Let $T^{\mathcal{L}_t/\theta_t}$ be a tree in the view updated by δV. Then from above proofs, $T^{x_1/\gamma_t/\theta_t}$ is updated by δS and there exists a tuple $<x_1, \cdots>$ such that $T^{x_1/\gamma_t/\theta_t}$ is in x_1 and x_1 has a condition tree $T^{x_1/\gamma_c/\theta_c}$ satisfying "*cdn*() and $x_1/\gamma_c/\theta_c = strValu$".

If $T^{x_1/\gamma_t/\theta_t}$ is not updated by $\delta S'$, either (a) x_1 is not a variable in the *for*-clause of $\delta S'$, i.e., x_1 is not in any tuple and neither is $T^{x_1/\gamma_t/\theta_t}$, or (b) x_1 is in the tuple $<x_1, \cdots>$ but $T^{x_1/\gamma_t/\theta_t}$ is not in x_1, or (c) x_1 is in the tuple $<x_1, \cdots>$ and $T^{x_1/\gamma_t/\theta_t}$ is in x_1 but one of "*cdn*()" and "$x_c/\gamma_c/\theta_c = strValu$" is not in $\delta S'$.

In Case (a), because x_1 is not a variable in $\delta S'$, so $T^{x_1/\gamma_t/\theta_t}$ will not be updated by $\delta S'$ (this does not prevent $T^{x_1/\gamma_t/\theta_t}$ from appearing in the view). This means that the $T^{\mathcal{L}_t/\theta_t}$ in $V(\delta S'(S^i))$ is different from the $T^{\mathcal{L}_t/\theta_t}$ in $\delta V(V(S^i))$ because the assumption assumes that the $T^{\mathcal{L}_t/\theta_t}$ in $\delta V(V(S^i))$ is updated. This contradicts the correctness of $\delta S'$.

In Case (b), because $T^{x_1/\gamma_t/\theta_t}$ is not in x_1, so $T^{x_1/\gamma_t/\theta_t}$ is not in $V(S^i)$. This contradicts the assumption that $T^{\mathcal{L}_t/\theta_t}$ is in the view.

In Case (c), if *cdn*() is violated, the tuple of $T^{x_1/\gamma_t/\theta_t}$ will not be selected by V, so $T^{x_1/\gamma_t/\theta_t}$ is not in $V(S^i)$ which contradicts the assumption. If $x_1/\gamma_c/\theta_c = strValu$ is violated, $T^{x_1/\gamma_t/\theta_t}$ will not be updated by δV. This contradicts the assumption that $T^{\mathcal{L}_t/\theta_t}$ is updated by δV.

This concludes that δS is a precise translation. □

The View Update Problem Here Generalizes the View Update Problem of the Relational Database in a Similar Setting. Suppose that there are three relations *Student*(*sid*, *name*, *tel*), *Course*(*cid*, *name*, *credit*), and *Enrolment*(*sid*, *cid*, *year*, *mark*), a view defined by $V_r = Student \bowtie Enrolment \bowtie Course$, and an update statement `update Vr set tel="22345" where name="John"`. The update condition attribute *name* and the update target attribute *tel* are from the same 'variable' *Student*, and the update does not change the values of the join condition attributes *sid* and *cid*. Consequently the theorem applies. It says that this update is translatable and the translated source update is `update Student set tel="22345" where name in (select name from Student join Enrolment join Course) and name="John"`. This source update is obviously precise because, if two students having the same name 'John' and different telephone numbers '21344' and '21345', and if the telephone numbers are all changed to '22345' in the view, they are also all changed in the source relation. Next time when the view is derived, the telephone numbers will appear changed and the same.

4 Update Translation When $\mathcal{L}_t/\theta_t \neq \phi$ and $x_c \neq x_t$

We look into the translation problem when the mappings of the update condition and the update target are from different variables. The results of this section generalize the view update problem in the relational views when they are defined with the join operator.

In general, **view updates are not translatable in the case of** $x_c \neq x_t$. Consider two tuples where the binding x_t is copied to $x_t^{(1)}$ and $x_t^{(2)}$ to combine with two bindings $x_{c[1]}$ and $x_{c[2]}$ of x_c by the context-based production as

$$< \cdots, x_t^{(1)}, \cdots, x_{c[1]}, \cdots >$$
$$< \cdots, x_t^{(2)}, \cdots, x_{c[2]}, \cdots >$$

Assume that in the view, the update condition $x_c/\gamma_c/\theta_c$ is satisfied in $x_{c[1]}$ but violated in $x_{c[2]}$. Then, the copy of x_t for the first tuple will be updated but the one for the second will not. In the source, if x_t is updated, not only the first copy of x_t changes, but also the second copy. In other words, the translated source update has view side-effect. However, if x_t in the source is not updated, all its copies in the view will not be changed.

Although generally view updates, when $x_c \neq x_t$, are not translatable, for the following view and the view update, a precise translation exists.

```
V:      <v>{ for x₁ in p₁,    ···,      xₙ in pₙ
             where ··· and x_c/γ_c/θ_c = x_{c+1}/γ_{c+1}/θ_{c+1} and ···
             return rtn(x₁, ···, xₙ)  }</v>
```

V: $<v>\{$ `for` x_1 `in` p_1, $\cdots$, x_n `in` p_n `where` $\cdots$ `and` $x_c/\gamma_c/\theta_c = x_{c+1}/\gamma_{c+1}/\theta_{c+1}$ `and` $\cdots$ `return` $rtn(x_1, \cdots, x_n)$ $\}</v>$

where x_c/γ_c is in $rtn(x_1, \cdots, x_n)$.

δV: $(\bar{p_s},\ \ v/\mathfrak{e}/\mathcal{L}_c/\theta_c = strValu,\ \ v/\mathfrak{e}/\mathcal{L}_t/\theta_t,\ \ del(T)|ins(T))$
where x_t is either x_c or x_{c+1}.

The conditions of the setting require that the condition path $x_c/\gamma_c/\theta_c$ must be a join path in the view definition and the γ-expression x_c/γ_c must be a prefix of this join path. At the same time, the variable of the target path must be x_c or x_{c+1}, the variable of the path joined to the update condition path.

Theorem 2. *Given view V and view update δV defined above where $\mathcal{L}_t/\theta_t \neq \phi$, update δS (in Section 3) is a precise translation of the view update δV if and only if $x_c/\gamma_t/\theta_t$ does not proceed any path in the where clause of δS.*

Proof. The notation of this proof follows that of the proof for Theorem 1 and Figure 1(a). Consider two tuples $t_1 = <x_t^{(1)}, x_{c[1]}, \cdots>$ and $t_2 = <x_t^{(2)}, x_{c[2]}, \cdots>$ in $fortup(\delta S(S))$ where $x_t^{(1)}$ and $x_t^{(2)}$ are copies of x_t and $x_{c[1]}$ and $x_{c[2]}$ can be the same. If one is updated by δS, the other is updated too. The reason is that for $T_1^{x_t/\gamma_t/\theta_t} \in x_t^{(1)}$ and $T_2^{x_t/\gamma_t/\theta_t} \in x_t^{(2)}$, because of the join condition in the view definition V: $x_c/\gamma_c/\theta_c = x_{c+1}/\gamma_{c+1}/\theta_{c+1}$ and because $x_{c+1} = x_t$ and $x_t^{(1)} = x_t^{(2)}$, a condition tree $T_1^{x_c/\gamma_c/\theta_c}$ exists for $T_1^{x_t/\gamma_t/\theta_t}$ and $T_2^{x_c/\gamma_c/\theta_c}$ exists for $T_2^{x_t/\gamma_t/\theta_t}$ and $T_1^{x_c/\gamma_c/\theta_c} = T_2^{x_c/\gamma_c/\theta_c}$. Consequently if $T_1^{x_c/\gamma_c/\theta_c}$ satisfies the update condition, so does $T_2^{x_c/\gamma_c/\theta_c}$. So either both $T_1^{x_t/\gamma_t/\theta_t}$ and $T_2^{x_t/\gamma_t/\theta_t}$ are updated or none is updated. Following Lemma 4, if e_1 and e_2 are mapped from $T_1^{x_t/\gamma_t/\theta_t}$ and $T_2^{x_t/\gamma_t/\theta_t}$ respectively, if one is updated, the other is updated too.

The remaining proof can be completed by following the argument of the proof of Theorem 1. □

Example 2. Consider the view definition in Example 1 and the view instance in Figure 2(b). If the view update is

δV: `for` s `in` v/e
`where` $s/H = 1$
`update` s/C `(insert` $(K\ 5)$`)`

By the theorem, this update is translatable because the full update condition path is $v/e/H$ and its source mapping, z, is a join path. The update target path is $/v/e/C$ and its source path, y, is the other end of the join condition $y/D = z$ in the view. Thus, the translated source update is below.

```
δS:  for x in doc("r")/r/A, y in x/C, z in x/H
      where y/D=z and z="1"
      update y (insert (K 5))
```

It is easy to check that the translated update works correctly. That is, $V(\delta S(S) = \delta V(V(S))$.

The View Update Problem Here Also Generalizes the Following View Update Problem of the Relational Database. Suppose that there are three relations $Student(sid, name, tel)$, $Course(cid, name, credit)$, and $Enrolment(stud, crs, year, semester, mark)$, a view defined by $Vr = Student \bowtie_{sid=stud} Enrolment \bowtie_{crs=cid} Course$, and an update statement `update Vr set mark="90" where sid="s01"`. The update condition attribute *sid* and the target attribute *mark* are from different 'variables' *Student* and *Enrolment* and they are bound to equal by the join condition, and the update does not change the values of the join condition attributes *sid*, *stud*, *cid* and *crs*. Consequently the theorem says that this update is translatable and the translated source update is `update Enrolment set mark="90" where stud in (select stud from Student join Enrolment on sid=stud join Course on crs=cid) and sid="s01"`. This source update is precise as if a student has two courses with the marks of '60' and '70', if they are changed to '90' in the view, they are also all changed to '90' in the source relation. Next time when the view is derived, the marks of the student will be '90' in the view.

5 Conclusion

In this paper, we invested two cases of the view update problem when the update target path is longer than the roots of 𝔢-trees. In the first case where the update target and the update condition are from the same variable, a solution is proposed and the translation obtained from the algorithm is proved to be precise. In the second case where the update target and the update condition are from the different variables, although in general view updates are not translatable, we discovered a specific type of view and a specific type of updates and derived a precise translation for the case. There are a few more cases that have not been investigated in the paper. These will be our future work.

References

1. Bancilhon, F., Spyratos, N.: Update semantics of relational views. TODS 6(4), 557–575 (1981)
2. Boag, S., Chamberlin, D., Fernndez, M.F., Florescu, D., Robie, J., Simon, J.: Xquery 1.0: An xml query language (2007), `http://www.w3.org/TR/xquery/`

3. Boneva, I., Groz, B., Tison, S., Caron, A.-C., Roos, Y., Stawork, S.: View update translation for xml. In: ICDT, pp. 42–53 (2011)
4. Braganholo, V.P., Davidson, S.B., Heuser, C.A.: From xml view updates to relational view updates: old solutions to a new problem. In: VLDB Conference, pp. 276–287 (2004)
5. Cong, G.: Query and Update through XML Views. In: Bhalla, S. (ed.) DNIS 2007. LNCS, vol. 4777, pp. 81–95. Springer, Heidelberg (2007)
6. Falquet, G., Nerima, L., Park, S.: Hypertext view update problem. Technical Report, University of Geneva (2000), `www.cui.unige.ch/isi/reports/hvu.ps`
7. Fankhauser, P.: Xquery formal semantics state and challenges. SIGMOD Record 30(3), 14–19 (2001)
8. Liu, D., Hu, Z., Takeichi, M.: Bidirectional interpretation of xquery. In: PEPM, pp. 21–30 (2007)
9. Llasunaga, Y.: A relational database view update translation mechanism. In: VLDB Conference, pp. 309–320 (1984)
10. Tatarinov, I., Ives, Z.G., Halevy, A.Y., Weld, D.S.: Updating xml. In: SIGMOD Conference, pp. 413–424 (2001)
11. Tomasic, A.: View Update Translation Via Deduction and Annotation. In: Gyssens, M., Van Gucht, D., Paredaens, J. (eds.) ICDT 1988. LNCS, vol. 326, pp. 338–352. Springer, Heidelberg (1988)
12. Tomasic, A.: Determining correct view update translations via query containment. In: Workshop on Deductive Databases and Logic Programming, pp. 75–83 (1994)
13. Wang, L., Rundensteiner, E.A., Mani, M.: Updating xml views published over relational databases: towards the existence of a correct update mapping. DKE 58, 263–298 (2006)