View update translation for XML[*]

Iovka Boneva[†]
University Lille 1

Benoît Groz[‡]
University Lille 1

Sophie Tison
University Lille 1

Anne-Cécile Caron
University Lille 1

Yves Roos
University Lille 1

Sławek Staworko
University Lille 3

ABSTRACT

We study the problem of update translation for views on XML documents. More precisely, given an XML view definition and a user defined view update program, find a source update program that translates the view update without side effects on the view. Additionally, we require the translation to be defined on all possible source documents; this corresponds to Hegner's notion of uniform translation. The existence of such translation would allow to update XML views without the need of materialization.

The class of views we consider can remove parts of the document and rename nodes. Our update programs define the simultaneous application of a collection of atomic update operations among insertion/deletion of a subtree and node renaming. Such update programs are compatible with the XQuery Update Facility (XQUF) snapshot semantics. Both views and update programs are represented by recognizable tree languages. We present as a proof of concept a small fragment of XQUF that can be expressed by our update programs, thus allows for update propagation.

Two settings for the update problem are studied: without source constraints, where all source updates are allowed, and with source constraints, where there is a restricted set of authorized source updates. Using tree automata techniques, we establish that without constraints, all view updates are uniformly translatable and the translation is tractable. In presence of constraints, not all view updates are uniformly translatable. However, we introduce a reasonable restriction on update programs for which uniform translation with constraints becomes possible.

General Terms

Theory, Algorithms, Security

Keywords

Update, view, translation, automata, editing script, tree alignment, composition

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

1. INTRODUCTION

Since its standardisation by the W3C [38], XML has become the gold standard as a format for exchanging and representing data. Along with this format, several query languages like XPath [39] and XQuery [40] were devised to query XML documents. While some databases store data into traditional DBMS and use XML only for exporting information, more and more DBMS provide a XQuery engine. Storing data in XML format (the so called *native XML databases*) avoids the conversion cost.

The XQuery Update Facility [41] standard from W3C extends XQuery with an important feature present in all DBMS: the ability to update an XML document. Another important feature of DBMS is the *view* facility, provided either for security reasons in the frame of access control, or just for commodity purposes. Even though there is no specific standard for defining XML views, one could use the general transformation language XSLT, or even XQuery, for this purpose. The possibility to both update and define views in the same DBMS leads to the question of updating data through views. Generally speaking, the *view update problem* is, given a database instance t and a view v of t, and given an update p to be applied on v, how to "correctly" propagate on t the changes of v made by p, where the meaning of "correctly" is to be defined.

A generalization of the view update problem is the problem of *translating update programs*. It can be stated as follows: given a view definition v and a view update program f_v, provide a source update program f s.t. for all document t, $v(f(t)) = f_v(v(t))$. That is, applying f on the source document t and then computing the view gives the same result as applying f_v on the view of t. This problem is to be related to update translation for closed views, as it has been studied for relational databases [24]. The user only has access to the view, and can define read and update queries without knowing anything about the underlying source document stored in the database. Thanks to the automatic

[*] All authors are members of MOSTRARE, joint team of INRIA Lille and LIFL

[†] Contact author: iovka.boneva@lifl.fr, Phone: +33 3 59 57 78 58, Fax: +33 3 59 57 78 50

[‡] Contact author: benoit.groz@inria.fr, Phone: +33 3 59 57 79 88, Fax: +33 3 59 57 78 50

ICDT 2011, March 21–23, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0529-7/11/0003 ...$10.00

translation of view updates, the DBMS can handle user defined updates on views without any visible side effect for the user, and without any need of materialization. This however does not prevent potentially undesirable side effects on the part of the document which is hidden from the user, as for instance deleting some hidden nodes, or adding random hidden nodes. In order to tackle this problem, we consider view updates translation *in presence of constraints*. A constraint is a set of authorized updates on the source. Then f is a correct translation of f_v if it performs only authorized updates.

In our framework, XML documents are modelled as finite unranked ordered node-labeled trees. The views that we consider can relabel nodes and hide some nodes together with the underlying subtree but cannot restructure the document otherwise. This view framework is quite simple, but still useful, e.g., for defining security views, in which the user is allowed to access only to a part of the data. Programs are allowed to perform three kinds of atomic update operations: relabeling of nodes, deletion of nodes together with the whole underlying subtree, and insertion of nodes or constant subtrees. An update defines the simultaneous application of a set of such atomic operations on its input tree, by specifying what atomic operation is to be applied on each node. This definition allows to model the snapshot semantics of the XQUF. An update program is a (recognizable) set of updates; it specifies how each of the trees in its domain should be updated. As our programs are functional, they are called *update functions*. Both update programs and views are represented by tree automata.

Contributions.

We establish that, for the class of views and updates we consider, update translation is always possible when no constraints are given. Moreover, the translation can be done in polynomial time under the assumption of inclusion of the co-domain of the view update into the co-domain of the view. We show that the family of update programs we present subsumes a small fragment of XQUF. This in particular implies that XQUF updates in that fragment allow for view update translation.

In presence of constraints, update translation is impossible in general. It is even undecidable to test whether a given source update program is a correct translation of a given view update program. However, the translation of a single update (i.e., an update function which domain is a singleton) is possible even in the presence of constraints. We also show that update translation in presence of constraints is possible for a subclass of update programs, namely those that cannot insert an unbounded number of subtrees between two sibling nodes. These are called k-synchronized update programs.

Related work.

In [35], we have studied the view update problem for a less expressive class of views. We have proposed a method for computing an optimal propagation of a single view update, where the optimality criterion is restraining side effects on the hidden part of the source. An important difference with the current work is that in [35], views are materialized.

The translation of view updates for relational databases has been an active research area for several years, e.g. [10, 3, 13, 21, 12]. Different criteria for correctness of a translation have been considered, as e.g. the constant complement criterion [3], which requires for a complement view to remain constant through updates. The authorized source updates that we define are in the same spirit. In [24], it is additionally required for the translation to be applicable on any possible source document, this is directly implied by our definition of update translation. In [21] are introduced so called dynamic views, which are view definitions coming together with an update translator. A translation is correct (called consistent in the paper) if the view update uniquely determines the source update. The authors characterize the class of correct translations, and show that these are a superset of translations under a constant complement defined in [3].

More recently, in the context of XML, [19, 18] study so called *lenses*. These are bi-directional tree transformers (view definitions) that provide two operations: get and put. The *get* operation allows to compute an abstract view of a concrete tree. The *put* operation takes an updated version of the abstract view, together with the original concrete tree, and correspondingly updates the original tree. This way the view definition itself allows to compute the update propagation. In contrast with our approach, views are always materialized. The expressiveness of lenses and of the views defined in our framework are incomparable. Lenses allow e.g. reordering of siblings, which is not possible for our approach. On the other hand, the visibility of a node in our approach is defined by any recognizable condition on the tree, whereas it only depends on a bounded neighbourhood for lenses.

Several authors consider updating XML views of relational databases [42, 7, 9, 16]. For instance, [7] focuses on translating XML view updates to relational view updates and delegating the problem to the relational DBMS, [42] studies the conditions under which a view update is translatable, and [9] provides algorithms for the translation of a rich class of view updates. There exist numerous approaches storing XML documents in relational databases, e.g. [6, 36], and one could attempt to combine them with the view propagation solutions. However, the complexity of view definitions required to *reconstruct* the XML documents is beyond the capabilities of the existing propagation solutions.

In [28], the authors propose to propagate view updates by defining a *backward* semantics of XQuery expressions. Essentially, the backward semantics of an XQuery expression used to define a view is a function which takes the original source document with the modified view and returns an updated source document. The class of views is incomparable with ours, as for instance it allows copying. Because of copying, update propagation is not necessarily side-effect free. Moreover, as for lenses, it requires materialization.

The update translation problem is to be related to so called *query rewriting*, which is the problem of translating read-only queries. Query rewriting for XML has been studied in e.g. [37, 15, 33, 23].

Organization of the paper.

In Section 2 we outline generalities on XQUF and tree automata. In Section 3 we define how updates are represented and give some basic properties of updates and editing scripts. We also introduce update functions and present an XQUF fragment that is translatable into our framework. In Section 4 we define views, formally state the update translation problem and solve this problem when no source constraint is given. Section 5 treats the update translation problem with source constraints; we show that this problem is not solvable in general. Then we introduce k-synchronized updates and show that, in this case, update translation is possible in presence of constraints. Section 6 concludes and gives directions for future research.

2. PRELIMINARIES

2.1 XQuery Update Facility

XQuery Update Facility (XQUF) is an extension of the XQuery language, for performing update operations on XML documents stored in an XML database. It means that XQUF is composed of non-updating expressions (classical XQuery) returning a result, and updating expressions returning nothing, like in SQL. It provides basic operations acting upon XML nodes:

- insert a (sequence of) node(s) after/before/as a children a specified node
- delete a (sequence of) node(s)
- rename a node without affecting its content
- replace the children of a node with a sequence of nodes
- replace the value of a node by a string value

Updating expressions are evaluated following the *snapshot* semantics: the query selects the node(s) to update, and describes the update operations to apply on those nodes; update operations are accumulated into a Pending Update List, and are executed all at once. Consider for instance the update query in Figure 1. It is irrelevant whether the `delete` is written after or before the `insert` operation. This query will insert an `a(b)` subtree before every node on a path `/r/c[.//d]`, and delete all such nodes. See [20] for a short introduction to XQUF.

```
for $x in /r/c[.//d]
return
   delete $x ,
   insert a(b) before $x
```

Figure 1: An update defined with XQUF.

2.2 Trees, automata and morphisms

We consider finite unranked ordered node-labeled trees. Let Σ be a finite set of symbols. A *tree* over the alphabet Σ is a structure $t = (N_t, \lambda_t)$, where N_t, the set of nodes, is a finite non-empty prefix-closed subset of $\mathbb{N}^*$ such that $ni \in N_t$ whenever $n(i+1) \in N_t$, and $\lambda_t : N_t \to \Sigma$ is the labelling function. The node ϵ (the empty sequence) is called the root of the tree. $\mathcal{T}_\Sigma$ denotes the set of trees over the alphabet Σ. For convenience, we are sometimes going to present trees using terms. For instance, the tree in Figure 2(c) corresponds to $r(a, d(f))$.

We also define the child relation $ch_t \subseteq N_t \times N_t$ and the following-sibling relation $\prec_t \subseteq N_t \times N_t$ on nodes: for every $n, n' \in N_t$, n' is a child of n, i.e., $(n, n') \in ch_t$ iff $\exists i \in \mathbb{N}$, $n' = ni$, and $(n, n') \in \prec_t$ iff there are $n'' \in \mathbb{N}$, $i < j$ such that $n = n''i$ and $n' = n''j$. Note that the following-sibling relation is irreflexive.

Given an alphabet Σ, we denote by Σ_ε the alphabet $\Sigma \cup \{\varepsilon\}$, where $\varepsilon \notin \Sigma$. For all natural k, $\Sigma_{\text{edit},k}$ is the set $\Sigma_\varepsilon^k \setminus \{(\varepsilon, \ldots, \varepsilon)\}$. We write ε instead of $(\varepsilon, \ldots, \varepsilon)$ when the product alphabet Σ^k is clear from the context. We omit the k-subscript whenever it equals two i.e., $\Sigma_{\text{edit}} = \Sigma_{\text{edit},2}$.

We assume that the reader is familiar with basic notions of tree automata (see e.g. [11]). There exist two generally accepted approaches of defining automata for unranked trees [34]. The first is to represent unranked trees as ranked trees and use automata for ranked trees. The second is to define automata that work directly on unranked trees. Several equivalent models of automata have been proposed in this context, e.g., the model of hedge automaton is accepted as a natural and fundamental model for automata on unranked trees [29]. For sake of simplicity in several proofs, we will follow the first approach: encode unranked trees by binary trees and then use ranked automata. Several encodings have been investigated and in this paper we use the Rabin's first-child next-sibling encoding *fcns* [31, 26]. Basically, it encodes an unranked tree over Σ into a binary tree over the alphabet $\Sigma_\perp = \Sigma \uplus \{\perp\}$, all symbols from Σ having in $\Sigma_\perp$ arity 2, and $\perp$ being the sole constant symbol. For instance, $f(a, b, c)$ is encoded into $f(a(\perp, b(\perp, c(\perp, \perp))), \perp)$.

It is folklore result that an unranked tree language is recognizable by an hedge automaton iff its *fcns* encoding is recognizable by a binary tree automaton. We will call such a tree language *recognizable.* In the sequel, we will use indifferently the automaton and the language it represents: when a recognizable (tree) language is given as input of a problem, we suppose the input is a (tree) automaton. Let us note that representation by an hedge automaton is polynomially equivalent to representation by a binary tree automaton for the *fcns* encodings.

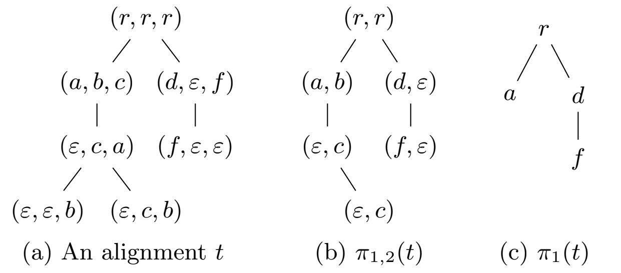
Given a function $f : \Sigma \to \Sigma'_\varepsilon$, a *morphism* induced by f is the function that maps every tree t over Σ into the tree t' over Σ'_ε s.t. t' is obtained from t by relabeling every node $n \in N_t$ as $f(\lambda_t(n))$, and then deleting the (resulting) subtrees whose root is labeled by ε. We do not distinguish the function f and the morphism it induces. This notion can be viewed as a very restricted adaptation of the notion of (ranked) tree homomorphism [1]. Indeed, the morphisms we have defined correspond to a subclass of linear alphabetic (non necessarily non-erasing) tree homomorphisms on the *fcns* encoding. More precisely, if we note f_e the linear alphabetic tree homomorphism [11] induced by f on the trees over $\Sigma_\perp$, then $f_e(\mathit{fcns}(t)) = \mathit{fcns}(f(t))$. We get easily the following result.

Proposition 2.1. *The image and the inverse image of a recognizable set of trees under a morphism are recognizable sets of trees.*

Proof. The proof is got easily by using the *fcns* encoding and the closure properties of recognizable ranked tree languages under inverse morphisms and linear morphisms. The constructions are polynomial. □

3. UPDATES

[1] This can also be viewed as a special case of morphism of forest algebras as defined in [5].



(a) An alignment t (b) $\pi_{1,2}(t)$ (c) $\pi_1(t)$

Figure 2: An alignment and two projections.

3.1 Updates as editing scripts

To represent updates on trees, we use tree alignments, commonly used for measuring similarities between trees [25].

Definition 1. A Σ-alignment – or *alignment* for short – is a tree t over $\Sigma_{\text{edit},k}$, for some natural k, and satisfying:

1. the label of the root of t is $(r, \ldots, r)$ for some $r \in \Sigma$;
2. if, for some node n, the i^{th} component of $\lambda_t(n)$ is ε, then for all node n' child of n, the i^{th} component of $\lambda_t(n')$ is also ε.

A Σ-alignment for $k = 2$ is called an *editing script*.

In order to understand how editing scripts define updates on trees, let us introduce *projections* for alignments over $\Sigma_{\text{edit},k}$. π_i^k is the usual projection on the i-th component, in which sub-trees labeled by ε are removed. More generally, for every m integers $i_1, i_2, \ldots, i_m$ in $\{1, \ldots, k\}$, we denote by $\pi^k_{i_1,i_2,\ldots,i_m}$ the morphism induced by the function that maps every symbol of $\Sigma_{\text{edit},k}$ onto its elements at positions $i_1, i_2, \ldots, i_m$, taken in that order. Sometimes k will be implicit and then $\pi^k_{i_1,i_2,\ldots,i_m}$ will be denoted by $\pi_{i_1,i_2,\ldots,i_m}$. Figure 2 represents an alignment t over $\Sigma_{\text{edit},3}$ and its projections $\pi_{1,2}(t)$ and $\pi_1(t)$.

Note that by Proposition 2.1, we get directly

Proposition 3.1. *Any projection of a recognizable set of alignments is also a recognizable set of alignments.*

One can associate each editing script t with the ordered pair of trees $(\pi_1(t), \pi_2(t))$. Therefore, an editing script t defines an update on a tree, taking as input $\pi_1(t)$ and producing $\pi_2(t)$ as output. For each node of the input tree $\pi_1(t)$, the editing script specifies whether it is to be deleted (label (a, ε)), renamed (label (a, b) with $a \neq b$, where $a, b \in \Sigma$), or kept unchanged (label (a, a)). A node labeled (ε, a) specifies an insertion of a node with label a. In that sense, condition 1 in Definition 1 ensures that the root is never changed, and condition 2 ensures that when a node is deleted, the underlying subtree is also deleted, and that we cannot insert internal nodes, but only whole subtrees.

We wish to make this notion of update more precise, by taking node identifiers into account. Different editing scripts can define the same ordered pair, but we still wish to distinguish them. For example, the three editing scripts $(r, r)((\varepsilon, b)(a, \varepsilon))$, $(r, r)((a, \varepsilon), (\varepsilon, b))$, and $(r, r)((a, b))$ define the same ordered pair $(r(a), r(b))$. Intuitively, the two first insert a b-labeled node and delete an a-labeled node, but they do it in different order. The latter renames an a-labeled node as b. Therefore, the first two are equivalent, and are different from the third one. This is formalized below.

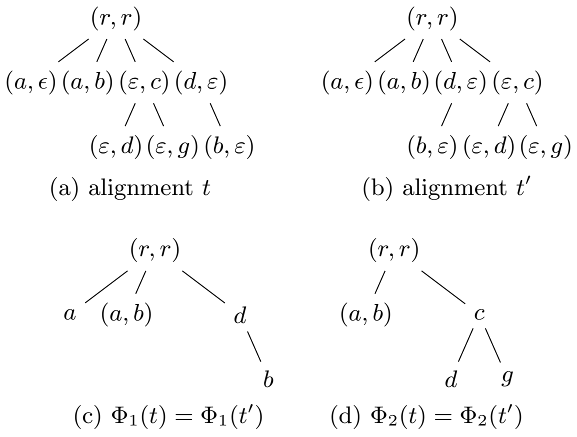
Consider the two morphisms $\Phi_1, \Phi_2 : \Sigma_{\text{edit}} \to \Sigma \cup \Sigma^2 \cup \{\varepsilon\}$ defined by:

$$\Phi_i(\alpha_1, \alpha_2) = \begin{cases} \alpha_i & \text{if } \alpha_{(3-i)} = \varepsilon \text{ or } \alpha_i = \varepsilon \\ (\alpha_1, \alpha_2) & \text{otherwise} \end{cases}$$

Definition 2. Two editing scripts t and t' are *equivalent*, written $t \sim t'$, if $\Phi_1(t) = \Phi_1(t')$ and $\Phi_2(t) = \Phi_2(t')$.

This definition emphasizes that $\sim$ is an equivalence relation. We define the equivalence class of an editing script t as $[t] = \{t' \mid t' \sim t\}$. We extend these definitions to sets of editing scripts: $[L] = \bigcup_{t \in L}[t]$, and $L \sim L'$ if $[L] = [L']$. Let us note that $\Phi_1(L) = \Phi_1(L')$ and $\Phi_2(L) = \Phi_2(L')$ does not imply $L \sim L'$. Intuitively, two editing scripts are equivalent if we can obtain each of them from the other by (repeatedly) commuting a subtree labeled with insertions with an adjacent subtree labeled with deletions.

Figure 3 represents two editing scripts t and t', and their images by the morphisms Φ_1 and Φ_2 as a witness for $t \sim t'$. In the case of words[2], the relation $\{(\pi_1(t), \pi_2(t)) \mid t \in L\}$ associated with a recognizable set of editing scripts L corresponds exactly to a rational transduction. As for tree languages, this model is closely related to the class of visibly pushdown transducers (restricted to trees). On one hand the relations expressible with a recognizable set of editing scripts form a strict subclass of synchronized visibly pushdown transducers (SVPT) [32]: intuitively SVPT are not restricted to insertions and deletions at the leaves. On the other hand, it is more expressive than the fully synchronized visibly pushdown transducers [32], since editing scripts allow insertions and deletions. The relations expressible with a recognizable set of editing scripts are incomparable with the VPT of [17] since, intuitively, editing scripts can insert unbounded nodes at the leaves, but cannot insert internal nodes.



(a) alignment t (b) alignment t'

(c) $\Phi_1(t) = \Phi_1(t')$ (d) $\Phi_2(t) = \Phi_2(t')$

Figure 3: Two equivalent trees t, t'

As seen before, a set of editing scripts L induces a binary relation of input and output trees $\{(\pi_1(u), \pi_2(u)) \mid u \in L\}$. If two editing scripts are equivalent, they induce the same

[2] A word is a tree in which, if we forget the root, the *ch* relation is empty, and the $\prec$ relation defines a total ordering.

relation, but the converse is false in the general case. However, it is true when the scripts contain no insertion (resp. no deletion, resp. no renaming). Equivalence of two recognizable sets of editing scripts is undecidable; this can be easily deduced from undecidability of equivalence of two word transducers [22] or undecidability results for trace languages [1]. Let us also note that even when L is a recognizable set of words, the set $[L]$ needs not even be context-free: consider the set of editing scripts $\{(r,r)(w) \mid w \in ((a,\varepsilon)(\varepsilon,b))^* \, ((c,\varepsilon)(\varepsilon,d))^*\}$.

Remark 1. If t, t' (resp. L, L') are editing scripts (resp. sets of editing scripts) over the alphabet $\Sigma \times \Sigma_\varepsilon$ or over the alphabet $\Sigma_\varepsilon \times \Sigma$, then equivalence coincides with equality. That is, $t \sim t'$ iff $t = t'$ (resp. $L \sim L'$ iff $L = L'$).

The inverse of an editing script is an editing script having the same tree structure but in which labels are inverted, that is, (α, β) becomes (β, α), for $\alpha, \beta \in \Sigma_\varepsilon$. This can be achieved with the morphism $\pi_{2,1}$.

Definition 3. For an editing script t, we denote by t^{-1} its inverse editing script defined by $t^{-1} = \pi_{2,1}(t)$. We extend this definition to sets of editing scripts: the inverse of a set L of editing scripts is $L^{-1} = \{s^{-1} \mid s \in L\}$.

Remark that if a set of editing scripts L is given by an automaton, then the automaton for L^{-1} is obtained by inverting every label.

Composition of updates.

In order to define compositions of updates, we define synchronization of editing scripts:

Definition 4. For $n \geq 2$ sets of editing scripts L_1, L_2, ... L_n, their *synchronization* $L_1 \bowtie L_2 \bowtie \ldots \bowtie L_n$ is the set of trees t over $\Sigma_{\mathrm{edit},n+1}$ such that for all $1 \leq i \leq n$ and $\pi_{i,i+1}(t) \in L_i$.

Figure 4 presents the synchronization of two editing scripts.

Remark 2. Consider two sets of editing scripts L_1, L_2 and let $u \in L_1 \bowtie L_2$. Let $u_1 \in L_1$ and $u_2 \in L_2$ be the witnesses for $u \in L_1 \bowtie L_2$, that is, $\pi_{1,2}(u) = u_1$ and $\pi_{2,3}(u) = u_2$. Remark that in this case $u \in u_1 \bowtie u_2$. Then $\pi_2(u_1) = \pi_1(u_2)$, as both are equal to $\pi_2(u)$. This intuitively means that the synchronization of two editing scripts u_1, u_2 (resp. of two sets of editing scripts L_1, L_2) is obtained by "gluing" the two trees (resp. the two sets of trees) around a common "middle" component $\pi_1(u_2) = \pi_2(u_1)$ (resp. $\pi_1(L_2) \cap \pi_2(L_1) \neq \varnothing$). This is actually where the term "synchronization" comes from.

Recognizability is preserved by synchronization, as established in the following proposition.

PROPOSITION 3.2. *Given recognizable sets of editing scripts L_1, L_2, ... L_n, their synchronization $L_1 \bowtie L_2 \bowtie \ldots \bowtie L_n$ is a recognizable set of alignments.*

PROOF. By definition, $L_1 \bowtie L_2 \bowtie \ldots \bowtie L_n$ is equal to $(\pi^{n+1}_{1,2})^{-1}(L_1) \cap (\pi^{n+1}_{2,3})^{-1}(L_2) \cap \ldots (\pi^{n+1}_{n,n+1})^{-1}(L_n)$ and this set is recognizable by Proposition 2.1. □

We denote by $L_1 \circ L_2$ the following set of editing scripts: $L_1 \circ L_2 = \pi_{1,3}\left((L_1 \bowtie L_2) \cap L_{corr}\right)$, where L_{corr} is the set of all trees having no nodes labeled with a tag in $\Sigma \times \{\varepsilon\} \times \Sigma$. Since composition is to represent composition of editing operations, we forbid to 'recover' a node once it has been deleted. Composition of editing scripts preserves recognizability, and the corresponding automaton is constructed in polynomial time.

PROPOSITION 3.3. *Operation $\circ$ is associative.*

In particular, taking S to be the set of all sets of editing scripts, $(S, \circ)$ is a monoid, with neutral element the set of all editing scripts over $\{(a,a) \mid a \in \Sigma\}$.

Remark 3. As we could expect, $(L_1 \circ L_2)^{-1} = L_2^{-1} \circ L_1^{-1}$ and the relation associated with L^{-1} is the inverse relation of the relation associated with L. However, $L \mapsto L^{-1}$ is not the inverse operation associated to the binary operation $\circ$. Indeed $(S, \circ)$ is not a group: not every set L has an inverse for operation $\circ$, whereas L^{-1} is always defined.

PROPOSITION 3.4. *Given editing scripts u and w, all editing scripts in $u \circ w$ are equivalent.*

The equivalence relation is stable under composition:

PROPOSITION 3.5. *Given editing scripts u, u', w, w', if $u \sim u'$ and $w \sim w'$, then $u \circ w \sim u' \circ w'$.*

Note that this property would not have been guaranteed if we had not intersected $\pi_{1,3}\,(L_1 \bowtie L_2)$ with L_{corr} in the definition of composition. For instance, let $u = (r,r)((b,\varepsilon),(\varepsilon,a))$, $u' = (r,r)((\varepsilon,a),(b,\varepsilon))$, and $v = (r,r)((a,d),(\varepsilon,c))$. We check easily that $u \sim u'$ whereas $\pi_{1,3}(u \bowtie v)$ is not equivalent to $\pi_{1,3}(u' \bowtie v)$.

3.2 Update functions

As seen before, an editing script u defines an update on its input tree $\pi_1(u)$. The notion of update function generalizes this, by defining how each tree in its domain is updated.

Definition 5. A set of editing scripts f is an *update function* iff for all $u, u' \in f$, $\pi_1(u) = \pi_1(u')$ implies $u \sim u'$. The set of trees $\{\pi_1(u) \mid u \in f\} \subseteq \mathcal{T}_\Sigma$ is called the *domain of f*.

Note that if L is an update function, then the induced relation $\{(\pi_1(u), \pi_2(u)) \mid u \in L\}$ is functional, but the converse is false. In this paper, we are only interested in update functions that are recognizable sets. These have a finite representation by means of tree automata over Σ_{edit}. The following proposition adapts to our setting a classical property, namely that testing equivalence of functional transductions can be reduced to testing functionality of the union of those transductions.

PROPOSITION 3.6. *Given two update functions f_1 and f_2, $f_1 \sim f_2$ iff $\pi_1(f_1) = \pi_1(f_2)$, i.e., they have same domain, and $f_1 \cup f_2$ is an update function.*

PROOF. For the *only if* part suppose $f_1 \sim f_2$. Then $\pi_1(f_1) = \pi_1(f_2)$. Let $u_1 \in f_1, u_2 \in f_2$ such that $\pi_1(u_1) = \pi_1(u_2)$. By hypothesis, there exists $u_1' \in f_1$ such that $u_1' \sim u_2$. Since $\pi_1(u_1) = \pi_1(u_2) = \pi_1(u_1')$, $u_1 \sim u_1'$ (f_1 is an update function). Hence, $u_1 \sim u_2$. Therefore, $f_1 \cup f_2$ is an update function.

For the *if* part, suppose $\pi_1(f_1) = \pi_1(f_2)$ and $f_1 \cup f_2$ is an update function. Then for every $u_1 \in f_1$, there exists $u_2 \in f_2$ such that $\pi_1(u_1) = \pi_1(u_2)$. By hypothesis, $u_1 \sim u_2$. Therefore $f_1 \sim f_2$. □

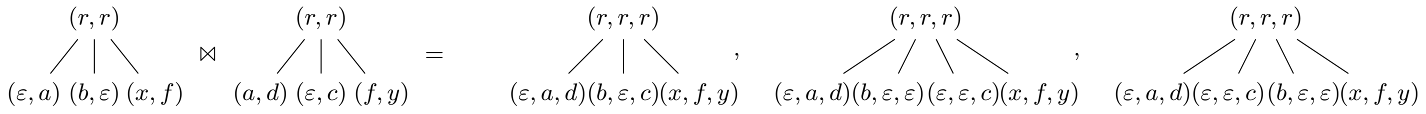

(r, r)
(ε, a) (b, ε) (x, f)
⋈
(r, r)
(a, d) (ε, c) (f, y)
=
(r, r, r)
(ε, a, d)(b, ε, c)(x, f, y)
,
(r, r, r)
(ε, a, d)(b, ε, ε)(ε, ε, c)(x, f, y)
,
(r, r, r)
(ε, a, d)(ε, ε, c)(b, ε, ε)(x, f, y)

Figure 4: Synchronization of two editing scripts.

Proposition 3.7. *The composition $f_1 \circ f_2$ of two update functions f_1 and f_2 is an update function.*

Proof. Fix $s, s' \in f_1 \bowtie f_2$. There are $u_1, u_1' \in f_1, u_2, u_2' \in f_2$ such that $s \in u_1 \bowtie u_2$ and $s' \in u_1' \bowtie u_2'$. Suppose $\pi_1(s) = \pi_1(s')$. Then $\pi_1(u_1) = \pi_1(u_1')$, hence $u_1 \sim u_1'$ (f_1 is an update function). Consequently, $\pi_1(u_2) = \pi_2(u_1) = \pi_2(u_1') = \pi_1(u_2')$, hence $u_2 \sim u_2'$. Thus, $u_1 \sim u_1'$ and $u_2 \sim u_2'$. We conclude the proof using propositions 3.4 and 3.5: $\pi_1(s) = \pi_1(s')$ implies $s \sim s'$, so $f_1 \circ f_2$ is an update function. □

Without intersecting $\pi_{1,3}(L_1 \bowtie L_2)$ with L_{corr} in the definition of composition, this property would not hold: given $f_1 = (a, \varepsilon)$ and $f_2 = (\varepsilon, b)$, $\pi_{1,3}(L_1 \bowtie L_2)$ comprises (a, b), $(a, \varepsilon)(\varepsilon, b)$ and $(\varepsilon, b)(a, \varepsilon)$, hence is not an update function.

Functionality and disambiguation.

Proposition 3.8. *Given a recognizable set L of editing scripts, it is decidable whether L is an update function in time polynomial in the size of the automaton defining L.*

Proof. By proposition 3.2, the language $L^{-1} \bowtie L$ is recognizable, so its linearization[3] is context-free. Furthermore, the morphisms (projections, inversions and Φ_1, Φ_2) we have defined on tree alignments can be viewed as word homomorphisms on the linearizations. We define morphisms f_1, f_1', f_2, f_2' on trees over $\Sigma_{\text{edit},3}$ as: $f_1 : t \mapsto \Phi_1\left((\pi_{1,2}(t))^{-1}\right)$, $f_1' : t \mapsto \Phi_1(\pi_{2,3}(t))$, and similarly for f_2, f_2' with Φ_2 instead of Φ_1. From the equality $\{(s, s') \in L^2 \mid \pi_1(s) = \pi_1(s')\} = \{\left((\pi_{1,2}(t))^{-1}, \pi_{2,3}(t)\right) \mid t \in L^{-1} \bowtie L\}$ we get $f_1(t) = f_1'(t)$ for every $t \in L^{-1} \bowtie L$ iff $\Phi_1(t_0) = \Phi_1(t_0')$ for every t_0 and $t_0' \in L$ such that $\pi_1(t_0) = \pi_1(t_0')$. We obtain the same condition for f_2, f_2' and Φ_2 and this implies $f_1(t) = f_1'(t)$ and $f_2(t) = f_2'(t)$ for every $t \in L^{-1} \bowtie L$ iff $t_0 \sim t_0'$ for every $t_0, t_0' \in L$ such that $\pi_1(t_0) = \pi_1(t_0')$. Therefore it suffices to use Plandowski's result that equivalence of morphisms on context-free languages is decidable in polynomial time [30] in order to verify that $t \mapsto \Phi_1\left((\pi_{1,2}(t))^{-1}\right)$ and $t \mapsto \Phi_1(\pi_{2,3}(t))$ are equivalent morphisms on $L^{-1} \bowtie L$, and similarly for Φ_2. □

Even if the user defines a functional update on the view, as we will see later, its propagation on the source may be ambiguous. Disambiguating a set of updates i.e., making it functional, is a key point.

Theorem 3.9. *Given a recognizable set of editing scripts L, we can effectively compute a recognizable update function L' such that $L' \subseteq L$ and the domains of L' and L are equal ($\pi_1(L) = \pi_1(L')$).*

The theorem gives a theoretical solution for disambiguating a recognizable set of editing scripts. The construction is non polynomial but could be applied on the fly to increase efficiency: rather than first constructing an unambiguous set of editing scripts and then apply it on the document, we can disambiguate on the fly by a two-pass run on the document.

3.3 Translation from XQUF

We introduce a small fragment of XQUF that can be compiled in automata on editing scripts. That is, for every update query Q in that fragment, one can construct an automaton on editing scripts A_Q representing Q. For example Figure 5 shows an editing script u belonging to the language $L(A_Q)$ recognized by the automaton A_Q obtained from the XQUF query Q of figure 1. More formally, let L_Q be the set of all editing scripts u such that $\pi_1(u)$ is transformed into $\pi_2(u)$ by Q. The editing scripts equivalent to editing scripts in the language $L(A_Q)$ form exactly the set L_Q. The reader is supposed to be familiar with the XPath 1.0 language, and to have some basic knowledge about XQUF and about querying trees using tree automata. We remind that CoreXPath 1.0 is the logical core of XPath 1.0, which corresponds to the navigational core of XPath 1.0. In what follows, (Core)XPath always stands for CoreXPath 1.0.

The following grammar defines the fragment of XQUF we are interested in.

Expr	::=	SingleExpr [, SingleExpr]*
SingleExpr	::=	IfExpr | ForExpr | UpdateExpr
IfExpr	::=	`if (` *AbsolutePath* `)` `then` Expr `else` Expr
ForExpr	::=	`for` $ *VarName* `in` *AbsolutePath* `return` UpdateExpr
UpdateExpr	::=	SingleUpdate [, SingleUpdate]*
SingleUpdate	::=	Insert | Delete | Rename | Replace
Insert	::=	`insert` Source [[[`as`[`first`|`last`]]?`into`]|`after`|`before`] Target
Delete	::=	`delete` Target
Rename	::=	`rename node` Target `as` *ElementName*
Replace	::=	`replace node` Target `with` Source
Target	::=	$ *VarName* | *AbsolutePath*
Source	::=	*ConstantSequence*

Here *AbsolutePath* means any CoreXPath 1.0 absolute path and *ConstantSequence* means any sequence of constant XML (sub)trees. Note that we basically distinguish two kinds of elementary update operations: those in which the target is specified by an XPath expression (e.g. `delete /a/b`; `insert a, b(a) as last into /r/c`), and those in which the target is specified by a variable (e.g. `rename node $var as e`; `replace node $var with a, b(), a`). The latter ones are to be used in the body of `for` update instructions (ForExpr rule). An expression in the fragment we consider (Expr) is

[3] The linearization *lin* is defined as usually by $lin(a(t_1, \ldots, t_n)) = (op; a) lin(t_1) \ldots lin(t_n)(cl; a)$.

a sequence of single expressions that can be update instructions (insert, delete, rename or replace), or a *for* update instruction (ForExpr rule), or an *if* update instruction (IfExpr rule). The body of the *for* update instruction can contain only a sequence of basic update operations (no nested for). The *if* expression can contain, in its then and else parts, any expression. Naturally, the target of basic update operations may be specified by a variable only when the variable is bound by a *for* expression.

Translation to Automata.

We give here an intuition of how the translation works. A complete construction is out of the scope of this paper. We know from [27, 8], that for every XPath query p, we can compute in exponential time an automaton B_p that accepts all trees t over $\Sigma \times \{0,1\}$ such that for every node $n \in u$, $\lambda_t(n) \in \Sigma \times \{1\}$ if and only if n is selected by p over $\pi_1(t)$.

Let Q be an XQUF update from the above fragment. We are going to decorate the editing scripts with intermediate results that represent the evaluation of the (absolute) path queries occurring in Q. Let $p_1, p_2, \ldots p_k$ be the absolute path queries appearing in Q.

First, we replace in Q every ForExpr expression `for` $\$x$ `in` p_i `return` U with update expression U in which we replace each occurrence of x with p_i. Technically speaking, the new expression is not exactly an XQUF update because in queries of the form `rename node` Target `as` *ElementName*, for instance, Target should evaluate into a single node. However, the intended semantics of such an extended expression is quite clear. Thanks to this transformation, we can suppose there are no variables in the SingleUpdate expressions of Q for the following construction.

Let $u_1, u_2, \ldots u_m$ be the SingleUpdate expressions in (transformed) update Q. For each u_i, we clearly can build an automaton B_{u_i} with the required property $L(B_{u_i}) = L_{u_i}$.
Decorations for trees: Let Σ_{deco} be the alphabet built from $\Sigma \times \Sigma_\varepsilon \times \{0,1\}^k \times (\Sigma_\varepsilon)^m \cup \{\varepsilon\} \times \Sigma \times \{\varepsilon\}^{k+1} \times (\Sigma_\varepsilon)^m$ with the additional restriction that for every $\alpha \in D$, if $\pi_1(\alpha) = \varepsilon$ then there is at most one $i \in \{k+3, \ldots k+m+2\}$ such that $\pi_i(\alpha) \neq \pi_1(\alpha)$.

Intuitively, the first two components will represent the final XQuery update, the k next components represent the nodes selected by the k XPath queries $p_1, \ldots p_k$, and the last m components the result of the SingleUpdates $u_1, \ldots u_m$. So, let D be the (recognizable) set of all trees t over Σ_{deco} such that
(1) for every $i \in \{3, \ldots, k+2\}$, $\pi_{1,i}(t) \in L(B_{p_i})$ and
(2) for every $i \in \{k+3, \ldots, k+m+2\}$, $\pi_{1,i}(t) \in L(B_{u_i})$.
Building the automaton: We can build by induction an automaton A_Q over D such that $\pi_{1,2}(L(A_Q)) = L_Q$. We just sketch the construction for a single update and an If expression.

(SingleUpdate): For a single update u_i, then A_{u_i} selects the trees from D such that $\pi_2(t) = \pi_{(k+2+i)}(t)$, so this case is trivial.

(IfExpr): Given a query q of the form `if (`p_i`) then` e_1 `else` e_2, suppose we have computed automata A_{e_1} and A_{e_2}. Then, given any tree t, automaton A_q tests whether there exists a node with label 1 on the $(2+i)^{th}$ component of t. If so, A_q runs automaton A_{e_1} on t, otherwise it runs automaton A_{e_2}. This concludes our sketch of proof.

4. THE VIEW UPDATE FRAMEWORK

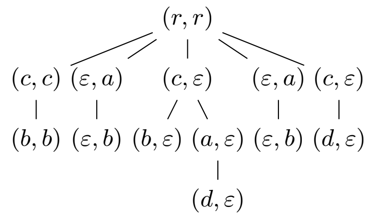


Figure 5: An editing script from the XQUF query of figure 1.

We begin with an illustrative example. Suppose we have a database containing documents of two kinds: drafts, and certified papers. There are two different authorities A_1, A_2 that certify the documents independently by attaching some certificate c_1 (resp. c_2) to the drafts in the database. Every paper possesses a certificate from each authority, otherwise it would be a draft, and once a draft has received both certificates, it becomes a certified paper. This database could be modelled with the following DTD:

```
docs    →   ( draft | paper )*
draft   →   c1?  | c2?
paper   →   c1 , c2
```

Since each authority should not know the status of the paper and work independently, authority A_1 gets only a view of the database, that hides certificates `c2` and renames both `draft` and `paper` elements into `doc`. That view process should be totally transparent, meaning that A_1 should not even be aware that it has only access to a view of the database instead of the whole database. This means in particular that A_1 gets a schema for its view, with rules `docs`→ `doc*` and `doc`→ `c1?`, and does not know DTD D.

Now, authority A_1 may wish to delete all its certificates, via an XQUF query like $Q_V =$ `delete /docs/doc/c1`. This update should not be applied directly on the database, since there are no `doc` elements in it. Besides, deleting `c1` element under a `paper` would lower the status of this document from `'paper'` to `'draft'`. The update function Q_V should thus be first translated into some query like

```
delete /docs/draft/c1,
delete /docs/paper/c1,
for $p in /docs/paper return rename node $p as draft
```

This section focuses on such update translation problems.

4.1 Views

Views are a particular kind of update functions that only delete and rename, but do not insert. Of course, in a database management system, computing a view does not imply modifying the source document. However, from a theoretical point of view, we can use the same representation via editing scripts to represent a view V as an update function without insertion: the first component of an editing script can represent a possible source document t, and the second component represents the “view” of that document.

Definition 6. A *view* V is an update function over the alphabet $\Sigma \times \Sigma_\varepsilon$.

Note that by Remark 1, for all views V_1 and V_2, $V_1 \sim V_2$ iff $V_1 = V_2$.

In the following, we denote by u_s and u_v editing scripts with the intended meaning that u_v should be applied on the view and u_s on the source, by V a view, by f_v an update function, and by L a set of editing scripts. Views have the following properties :

Lemma 4.1. *For all editing scripts u_s, u'_s and u_v, u'_v for all view V*

1. *$u_s \bowtie V$, $V^{-1} \bowtie u_s$, and $V^{-1} \bowtie u_s \bowtie V$ are singletons or empty.*
2. *$[V \circ u_v] = V \circ [u_v]$.*

Thus, by item 1, given an editing script u_s and a view V, $V^{-1} \bowtie u_s \bowtie V$ consists in a single tree t' (or is empty): $\pi_{1,4}(t')$ is the *editing script induced* by u_s on the view V. Let us note that $\pi_{1,4}(t') = V^{-1} \circ u_s \circ V$. We do not want updates to affect the visibility nodes. It seems to us that it would make little sense to translate an insertion on the view by showing a hidden node. Similarly, deleting a node should result in proper deletion, and not in hiding that node. Therefore, we call an editing script u_s *stable* w.r.t. view V if $V^{-1} \bowtie u_s \bowtie V$ is non empty and if no node of that tree $V^{-1} \bowtie u_s \bowtie V$ has label in $\{\varepsilon\} \times \Sigma \times \Sigma \times \Sigma$ or $\Sigma \times \Sigma \times \Sigma \times \{\varepsilon\}$. Let us note that the set of stable editing scripts w.r.t. a recognizable view V is recognizable as defined by $\pi_{2,3}(V^{-1} \bowtie \mathcal{T}_{\Sigma_{\text{edit}}} \bowtie V \cap \mathit{Correct})$ where $\mathit{Correct}$ is the recognizable set of tree alignments over the alphabet $\Sigma_{\text{edit},4}$ with no occurrences of $\{\varepsilon\} \times \Sigma \times \Sigma \times \Sigma$ or $\Sigma \times \Sigma \times \Sigma \times \{\varepsilon\}$. Furthermore, we note that the set of stable scripts is closed under $\sim$.

Lemma 4.2. *For every editing script u_v, every stable editing script u_s, for every view V, the following assumptions are equivalent:*

$$\begin{aligned} & \pi_{1,4}(V^{-1} \bowtie u_s \bowtie V) \sim u_v && (1)\\ \textit{iff}\quad & u_s \in \pi_{1,4}(V \bowtie [u_v] \bowtie V^{-1}) && (2)\\ \textit{iff}\quad & u_s \circ V \in [V \circ [u_v]] && (3)\\ \textit{iff}\quad & u_s \circ V \in V \circ [u_v] && (4) \end{aligned}$$

4.2 Update translation

Given an update on the view, we want to define which propagations on the source we allow. Roughly speaking, we will require a *propagation* of a view update to be side-effect free, i.e., induce the update defined by the user on the view, and to preserve visibility of nodes.

Definition 7. An editing script u_s is a *propagation* of editing script u_v w.r.t. view V iff u_v is equivalent to the editing script induced by u_s on the view V, and u_s is stable w.r.t. V. We denote by $\mathit{Prop}_V(u_v)$ the set of all propagations of u_v w.r.t. view V.

We extend this notion to sets of editing scripts: given a set of editing scripts L, the propagations of L are defined by $\mathit{Prop}_V(L) = \{\mathit{Prop}_V(u_v) \mid u_v \in L\}$. Now, for an update function f_v on the view V, we want to characterize which sets of editing scripts can be considered as correctly and completely propagating f_v on the source:

Definition 8. Given an update function f_v, and a set of editing scripts L, we say that L is a *translation* of update f_v w.r.t. view V if L consists in stable editing scripts w.r.t. view V and $L \circ V \sim V \circ f_v$.

$$\begin{tikzcd} {} \arrow[r, dotted, "\text{translation}"] \arrow[d, "V"'] & {} \arrow[d, "V"] \\ {} \arrow[r, "\text{update}"'] & {} \end{tikzcd}$$

Figure 6: the view update problem

Thus, a set of editing scripts is a translation if the diagram of figure 6 commutes. As we could expect, propagations and translations (as well as stability) are preserved under equivalence:

Remark 4. Observe that, by Proposition 3.5, an editing script equivalent to a propagation is a propagation, and a set of editing scripts equivalent to a translation is a translation.

There is an alternative characterization of translations:

Proposition 4.3. *L is a translation of update function f_v iff $L \subseteq \mathit{Prop}_V(f_V)$ and $\pi_1(L \circ V) = \pi_1(V \circ f_v)$.*

Proof. Let L a translation of update function f_v and u_s in L; as $L \circ V \sim V \circ f_v$, $u_s \circ V \sim v \circ u_v$ for some u_v in f_v and v in V; then $V^{-1} \circ u_s \circ V \sim u_v$ by Proposition 4.2: so $L \subseteq \mathit{Prop}_V(f_V)$; furthermore as $L \circ V \sim V \circ f_v$, $\pi_1(L \circ V) = \pi_1(V \circ f_v)$.

Conversely, let L s.t. $L \subseteq \mathit{Prop}_V(f_V)$ and $\pi_1(L \circ V) = \pi_1(V \circ f_v)$. As $L \subseteq \mathit{Prop}_V(f_V)$, L consists in stable editing scripts w.r.t. V and $V^{-1} \circ L \circ V \subseteq [f_v]$. So $L \circ V \subseteq V \circ [f_v]$ by Proposition 4.2, using (1) $\Longrightarrow$ (4). Then, by Proposition 4.1 item 2, $L \circ V \subseteq [V \circ f_v]$ as L consists of stable editing scripts. As $\pi_1(L \circ V) = \pi_1(V \circ f_v)$ and $V \circ f_v$ is functional, $L \circ V \sim V \circ f_v$. □

The update translation problems.

Problem 1 (Checking a translation). *Given a recognizable view V, a recognizable view update function f_v, and a recognizable set of source editing scripts L_s, answer whether L_s is a translation of f_v.*

Problem 2 (Finding a translation). *Given a recognizable view V and a recognizable view update function f_v, find a recognizable set of source editing scripts L_s s.t. L_s is a translation of f_v.*

4.3 Solution in the unconstrained case

From now on, we suppose w.l.o.g. that $\pi_1(f_v) \subseteq \pi_2(V)$. We further assume that $\pi_2(f_v) \subseteq \pi_2(V)$ otherwise there would be no translation for f_v. These assumptions are reasonable insofar as we can suppose the user to be provided a view schema. Besides, one can we verify those assumptions in time polynomial w.r.t. f_v. Then an update function is translatable iff its output remains in the view schema. Proposition 4.4 answers Problem 1 positively.

Proposition 4.4. *Given a recognizable view V, a recognizable update function f_v, and a recognizable set of source editing scripts L, testing whether L is a translation of f_v is decidable.*

Proof. First, we test whether L consists in stable updates. Next, we must check that $L \circ V \sim V \circ f_v$. We claim that $L \circ V \sim V \circ f_v$ iff $\pi_1(L \circ V) = \pi_1(V \circ f_v)$ and $L \circ V \cup V \circ f_v$ is an update function. Once we have tested the equality of the domains, namely $\pi_1(L \circ V) = \pi_1(V \circ f_v)$, we can use Proposition 3.8 and check that $L \circ V \cup V \circ f_v$ is an update function. Let us prove the claim: $V \circ f_v$ is an update function, by Proposition 3.7. Now, either $L \circ V$ is not an update function and then it is not equivalent to $V \circ f_v$, but $L \circ V \cup V \circ f_v$ is not an update function either. Or $L \circ V$ is an update function and the claim follows by Proposition 3.6. This concludes our proof. Furthermore, the algorithm is polynomial once we have checked equality of the domains. □

The following proposition answers Problem 2 positively.

Proposition 4.5. *Given a recognizable view V and a recognizable update function f_v, we can compute a translation L of f_v in polynomial time.*

Proof. By Propositions 3.2 and 3.1, we can compute in polynomial time an automaton for the set L of all stable editing scripts (w.r.t. V) from $\pi_{1,4}\left(V \bowtie f_v \bowtie V^{-1}\right)$. We must show that L is a translation of f_v. By Proposition 4.2, using (1) $\implies$ (2), L consists in propagations of f_v. The above assumptions ensure that $\pi_1(L \circ V) \subseteq \pi_1(V \circ f_v)$. □

Finally, using Theorem 3.9, we get

Corollary 1. *We can compute a functional translation L of f_v.*

5. SOLVING THE CONSTRAINED UPDATE PROPAGATION PROBLEMS

5.1 The general case

Our views impose only 'static' constraints on the state of the database. We wish to study constraints on the updates in the spirit of the "transition laws" of [14]. While in [14] the transition laws are treated as static constraints, using an extended database, our approach focuses on studying transitions, and the constraints we define on the updates cannot be expressed by static constraints within our framework.

In this section, we suppose a given recognizable set of editing scripts $\mathcal{U}_s$ representing the authorized source updates. Furthermore, we are going to consider only translations valid w.r.t. $\mathcal{U}_s$ (as formalized by Definition 9). Such restrictions can be most useful in the case of a database with multiple user profiles. One may require for instance that the updates of user 1 should not affect the view of user 2, or more permissively, the updates of user 1 should affect user 2's view only on nodes that are also visible in user 1's own view. A recognizable set $\mathcal{U}_s$ of authorized source updates can express that kind of restrictions on side effects. This approach is more flexible than the constant complement approach of [3] in the sense that we do not require the constant part to be a complement. Thus, the user can specify precisely the constraints he deems relevant, without the obligation to enforce a unique propagation. Such restrictions can also be used to protect the integrity of sensible data or to indicate some preference among possible propagations, as demonstrated in Theorem 5.13, in order to get a unique propagation. More generally, the possibility to define a set of authorized source updates allows the database administrator to specify which updates he thinks are reasonable.

Definition 9. A set of editing scripts L is a *valid translation* of an update function f_v w.r.t. a view V and $\mathcal{U}_s$ if L is a translation of f_v w.r.t. V and there exists a set of editing scripts $L' \subseteq \mathcal{U}_s$ s.t. $L' \sim L$.

A view editing script is called *uniform* if it admits a valid translation. We denote by *Unif* $(V, \mathcal{U}_s)$ the set of uniform (view) editing scripts.

An update function f_v is called *uniformly translatable* w.r.t. view V and $\mathcal{U}_s$ if it has a valid translation.

Let us note that even when f_v and V are recognizable, we impose in the definition neither recognizability of L nor recognizability of L'. For instance, if V is the identity and $\mathcal{U}_s = (r,r)((a,\varepsilon)^*(\varepsilon,b)^*)$, $f_v = (r,r)((a,b)^*)$ has a recognizable translation but there is no *valid* translation L such that $\exists L'.L' \sim L$ and $L' \subseteq \mathcal{U}_s$.

Proposition 5.1. *An update function f_v is uniformly translatable w.r.t. view V iff there exists some set of stable editing scripts $L \subseteq \mathcal{U}_s$ such that $L \circ V \sim V \circ f_v$.*

Proof. The result is immediate by remark 4. □

However, let us note that the preceding property is no longer valid when we require recognizability; f_v can have a recognizable valid translation but no *recognizable* valid translation included in $\mathcal{U}_s$.

The following adapts Problem 1 to the constrained setting.

Problem 3. *Given a recognizable view V, a recognizable set of authorized source editing scripts $\mathcal{U}_s$, a recognizable update function f_v, and a recognizable set of source editing scripts L, answer if L is a valid translation of f_v.*

While every update function admits a translation in the unconstrained setting, this is no longer the case in presence of constraints. The presence of source constraints raises two additional problems.

Problem 4. *Given a recognizable view V, a recognizable set of authorized source editing scripts $\mathcal{U}_s$, and a recognizable update function f_v, answer if f_v is uniformly translatable.*

Problem 5. *Given a recognizable view V, and a recognizable set of authorized source editing scripts $\mathcal{U}_s$, compute an automaton whose language is the set Unif $(V, \mathcal{U}_s)$.*

Let us resume with the illustrative example from section 4. We assume that once a document has acquired the `'paper'` status, it is published somewhere, so that no `'paper'` element should revert to the `'draft'` status. This in turn implies that authority A_1 cannot delete certificates c_1 under a document that has also been certified by the second authority. Such constraints can clearly be expressed via a regular set of editing scripts. However, if we do only forbid the above deletions, A_1 may face a strange behavior since it does not know about certificates c_2. Thus, A_1 will observe that it is sometimes allowed to delete its certificate c_1 under a document, and sometimes not. The uniform updates are those that avoid this kind of unpredictable behaviour. Here, the uniform updates forbid deleting any c_1 certificate altogether. Computing the set of uniform updates enables the database administrator to provide the user with the set of updates she is allowed to execute, which is the motivation for problem 5.

Negative results in the general setting.

Proposition 5.2. *Testing uniform translatability is undecidable, even when f_v is recognizable.*

Proof. Let V be the view that does not hide anything: $V = \left(\bigcup_{a \in \Sigma}(a,a)\right)^*$. Suppose f_v uses no relabelings, only deletions and insertions: $f_v \subseteq (\Sigma \times \{\varepsilon\} \cup \{\varepsilon\} \times \Sigma)^*$. Then the problem is equivalent to the problem of testing the inclusion of a functional word transducer (f_v) into an arbitrary word transducer ($\mathcal{U}_s$), which is undecidable [4]. This proves also that testing uniform translatability remains undecidable when we require translation to be recognizable. The question remains open when we require also recognizability of $L' \subseteq \mathcal{U}_s$ such that $L \sim L'$. □

Note that with the same proof, for $L = f_v$, we get the undecidability of problem 3. However, if the input is some $L \subseteq \mathcal{U}_s$, it becomes decidable in polynomial time once the domains are verified equal, using proposition 4.4.

Proposition 5.3. *Given a recognizable view V, Unif $(V, \mathcal{U}_s)$ is not recognizable, and its emptiness is undecidable.*

Consequently, computing the set of uniform editing scripts seems unfeasible in general, and therefore, we look for restrictions that allow to tackle these problems.

Single updates.

The simplest restriction will be to study translatability of single editing script instead of more general update functions. For that limited setting, the previous problems become decidable.

Proposition 5.4. *Testing uniform translatability of a view editing script u_v is decidable. Furthermore, we can compute in polynomial time a recognizable set L of editing scripts such that $[L]$ is the set of (valid) propagations of u_v.*

Proof. The set of valid propagations is equivalent to $\mathcal{U}_s \cap \pi_{1,4}(V \bowtie [u_v] \bowtie V^{-1})$ by Lemma 4.2, using (1)⇔(2). Those results can also be considered a consequence of Propositions 5.9 and 5.10. □

The previous results might however be misleading in the sense that one could suppose the difficulty to stem from $\mathcal{U}_s$'s not being closed under equivalence. The following undecidability result that holds for $\mathcal{U}_s$ over alphabet $\Sigma_\varepsilon \times \Sigma$ dismisses such misconceptions. Intuitively, even when $\mathcal{U}_s$ has no deletions, V may have deletions, so that $\mathcal{U}_s \circ V$ needs not be recognizable.

One could have supposed that solving problem 5 dynamically rather than statically would be easier: one does not need to compute all the uniform updates, but only those possible from the current state of the (non-materialized) view document. The following proposition puts paid to any such hope. We cannot tackle Problem 5 by fixing the initial document t and asking for the set of all uniform view editing scripts u such that $\pi_1(u) = t$. Fix a tree t over Σ, the tree that consists in a single node r for instance. Even when we require $\mathcal{U}_s$ to consist only in editing scripts without deletions, i.e., trees over $\Sigma_\varepsilon \times \Sigma$,

Proposition 5.5. *The problem (with input V and $\mathcal{U}_s$) of deciding 'universality' of the set $\{t' \in$ Unif $(V, \mathcal{U}_s) \mid \pi_1(t') = t\}$ (more exactly, testing whether it is equal to the co-domain of the view) is undecidable.*

Since our target is a translation of update functions, we look for a less drastic restriction than single updates. What makes translatability decidable for a single (view) editing script u_v is the possibility to compute a recognizable language for the equivalence class of u_v. The next section defines a class of update functions that guarantees that property, while remaining powerful enough to express most reasonable update functions.

5.2 K-synchronized updates

Equivalence of editing scripts deals with commuting consecutive insertions and deletions. For that reason, we must control those commutations to prevent the closure under equivalence from attaining languages that are not recognizable.

Definition 10. Given a natural $k \geq 1$, an editing script t is *k-synchronized* if for every sequence $n_1, n_2, \dots n_{k+1}$ of nodes in N_t such that for all $j \leq k$, $(n_j, n_{j+1}) \in \prec_t$, and for all $j \leq k+1, \lambda_t(n_j) \in \{\varepsilon\} \times \Sigma$, there is some node $n' \in N_t$ such that $(n_1, n') \in \prec_t$, $(n', n_{k+1}) \in \prec_t$, and $\lambda_t(n') \in \Sigma \times \Sigma$.

This means that, among the children of the same node, there cannot be more than k inserting nodes without a node tagged with a relabeling between them. A set of editing scripts is *k-synchronized* if it consists in k-synchronized editing scripts. A set of editing scripts is *synchronized* if it is k-synchronized for some k.

Remark 5. This notion is monotone: if a (set of) editing script(s) is k-synchronized, then it is k' synchronized for all $k' > k$.

General properties.

Proposition 5.6. *There exists a polynomial p such that a recognizable set L of editing scripts is synchronized iff it is $p(n)$-synchronized,*

This, with remark 5, allows to test in polynomial time whether a recognizable set of editing scripts is synchronized.

Proposition 5.7. *Fix $k \in \mathbb{N}$. Given a recognizable set L of k-synchronized editing scripts, $[L]$ is a recognizable set of k-synchronized editing scripts.*

Proof. This can be proved using classical constructions on automata. The core of the proof is that we must remember a finite information (corresponding to the insertions) between two siblings labeled with a relabeling. □

We could also define a normal form for the document, shifting all deletions to the left and insertions to the right as far as possible for instance: such a normalization of a recognizable set L of k-synchronized editing scripts would be recognizable.

Remark 6. When L is a recognizable set of editing scripts given by an automaton, one can compute an automaton for the set $\{t \in L \mid t \text{ is } k\text{-synchronized }\}$. We will denote this set by $KSync(L)$.

Notation 1. Given a recognizable view V and $k \geq 0$, we denote by $\mathcal{U}_V^k$ the set of all editing scripts u_s such that the editing script induced by u_s on view V is k-synchronized.

We have a result of the same flavour as the above remark:

PROPOSITION 5.8. *When L is a recognizable set of editing scripts given by an automaton, one can compute an automaton for the set $L \cap \mathcal{U}_V^k$.*

Uniform updatability for synchronized updates.

PROPOSITION 5.9 (PROBLEM 4). *Testing uniform translatability of an update function f_v of k-synchronized editing scripts is decidable.*

PROOF. In that setting, $[V \circ f_v]$ is a recognizable set of k-synchronized editing scripts. Furthermore, if we take $L_2 = KSync(\mathcal{U}_s \circ V)$, f_v is uniformly translatable iff $[V \circ f_v] \subseteq [L_2]$. Moreover, one can compute an automaton for $[L_2]$ by Remark 6 and Proposition 5.7. □

PROPOSITION 5.10. *When f_v is a recognizable set of k-synchronized editing scripts and V is a recognizable view, we can compute an automaton for its propagations.*

PROOF. This proposition holds whenever $[f_v]$ is a recognizable language. By proposition 5.7, this is the case when f_v is k-synchronized. The set of valid propagations is equivalent to $\mathcal{U}_s \cap \pi_{14}(V \bowtie [f_v] \bowtie V^{-1})$ by proposition 4.2, using (1)⇔(2). Note that in fact this result implies proposition 5.9 □

THEOREM 5.11 (PROBLEM 5). *When $\mathcal{U}_s \subseteq \mathcal{U}_V^k$, i.e., the set of authorized editing scripts is such that the editing scripts it induces on the views are k-synchronized, we can compute an automaton for the set of all uniform view editing scripts.*

We may be interested also in restricting the set of authorized editing scripts. For instance, given a fixed view V, one may wish the set $\mathcal{U}_s$ to be such that every view editing script u_v has a unique propagation from each source document:

Definition 11. A set of source editing scripts $\mathcal{U}_s$ is V-*unambiguous* if $\forall u_s, u'_s \in \mathcal{U}_s$ such that $\pi_1(u_s) = \pi_1(u'_s)$, either $u_s \sim u'_s$ or $u_s \circ V$ and $u'_s \circ V$ are not equivalent.

Given a recognizable set of (stable) editing scripts $\mathcal{U}_s$ and a view V, we would like to compute $\mathcal{U}'_s \subseteq [\mathcal{U}_s]$ such that $\mathcal{U}'_s \circ V = \mathcal{U}_s \circ V$ and $\mathcal{U}'_s$ is V-unambiguous.

In general, this disambiguation of $\mathcal{U}_s$ cannot be achieved.

PROPOSITION 5.12. *Testing whether $\mathcal{U}_s$ is V-unambiguous is undecidable.*

We can prove similarly there is no algorithm that can compute a (recognizable) set disambiguating $\mathcal{U}_s$. However, for k-synchronized editing scripts, the disambiguation can be achieved.

THEOREM 5.13. *Given a view V, and $\mathcal{U}_s \subseteq \mathcal{U}_V^k$, we can compute a recognizable set of editing scripts $\mathcal{U}'_s \subseteq \mathcal{U}_s$ such that $\mathcal{U}'_s \circ V = \mathcal{U}_s \circ V$ and $\mathcal{U}'_s$ is V-unambiguous.*

To conclude, let us add a few words about our definition for equivalence. Not only does this definition take better account of identifiers and data-values, it also helps incidentally to define the k-synchronized restriction. If we had defined the equivalence as the equality of the relations, i.e., letting t and t' be equivalent if and only if $\pi_1(t) = \pi_1(t')$ and $\pi_2(t) = \pi_2(t')$, we could have adapted most of our results except for this last local restriction on the number of insertions defining which we called "k-synchronized updates".

6. CONCLUSION AND FUTURE WORK

We have studied the problem of view update translation for a class of views and a class of update functions defined as recognizable sets of tree alignments. We have established conditions under which the translation is possible (absence of source update constraints and k-synchronized updates), and conditions under which it is not (presence of constraints on source updates). We also give the complexity for most of the translations. Finally, we have shown relationships between such update functions and XQuery Update Facility.

These theoretical results are our first results (together with [35]) on a more ambitious research objective which is to provide a complete framework for view update translation on XML documents. This should include, among others, a user-friendly specification of update functions, algorithms for translating updates, an interface for a system administrator that allows to specify preferences when several translations of the same user update are possible. Let us point out two concrete questions raised by the current work.

So far, we have not considered the actual application of update functions. That is, given a source document t and an update function f represented as a tree automaton A_t, how to compute the result of applying f on t. A simple solution is to see the automaton as a tree transducer, and use existing efficient algorithms (e.g. based on [2]'s algorithm for word transducers). However, such approach often requires to compute the product of the tree and the automaton A_t, which essentially is materialization. We rather intend to identify update functions that can apply an update "on the fly" while reading bounded parts of the input tree t at a time. This can be based on some locality criteria of the update, that is, guarantee that the propagation of any change in the document would not affect distant parts in the document. We believe that this is indeed often the case in real examples.

We also intend to consider more expressive view definition formalisms, allowing for instance for copying and insertion/deletion of internal nodes. We already have some preliminary results for views that delete internal nodes while remaining recognizable.[4]

7. REFERENCES

[1] I. J. Aalbersberg and H. J. Hoogeboom. Decision problems for regular trace languages. In *14th International Colloquium on Automata, languages and programming*, pages 250–259, London, UK, 1987. Springer-Verlag.

[2] C. Allauzen and M. Mohri. An optimal pre-determinization algorithm for weighted transducers. *Theoretical Computer Science*, 328(1–2):3–8, 2004.

[3] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[4] J. Berstel. *Transductions and context-free languages*, volume 38 of *Leitfäden der Angewandten Mathematik und Mechanik.* B. G. Teubner, Stuttgart, 1979.

[5] M. Bojańczyk and I. Walukiewicz. Forest algebras. In *"Logic and Automata"*. Amsterdam University Press, 2007.

[4] It is illustrated, e.g., in [23] that unlimited deletion of internal nodes leads to loss of recognizability.

[6] P. Boncz, T. Grust, M. Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *ACM SIGMOD International Conference on Management of Data*, pages 479–490, 2006.

[7] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. PATAXÓ: A framework to allow updates through XML views. *ACM Transactions on Database Systems (TODS)*, 31, 2006.

[8] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. An automata-theoretic approach to regular XPath. In *DBPL*, pages 18–35, 2009.

[9] B. Choi, G. Cong, W. Fan, and S. D. Viglas. Updating recursive XML views of relations. *Journal of Computer Science and Technology*, 23, 2008.

[10] E. F. Codd. Recent investigations in relational data base systems. In *IFIP Congress*, pages 1017–1021, 1974.

[11] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[12] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *J. ACM*, 31(4):742–760, 1984.

[13] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

[14] R. Fagin, J. D. Ullman, and M. Y. Vardi. On the semantics of updates in databases. In *PODS*, pages 352–365, 1983.

[15] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular XPath queries on XML views. In *ICDE*, pages 666–675, 2007.

[16] L. Fegaras. Propagating updates through XML views using lineage tracing. In *International Conference on Data Engineering (ICDE)*, 2010.

[17] E. Filiot, J.-F. Raskin, P.-A. Reynier, F. Servais, and J.-M. Talbot. Properties of visibly pushdown transducers. In *To appear in MFCS*, 2010.

[18] J. Foster, B. Pierce, and S. Zdancewic. Updatable security views. In *Computer Security Foundations Symposium*, 2009.

[19] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2007.

[20] X. Franc. XQuery update for the impatient. Available on `http://www.xmlmind.com/tutorials/XQueryUpdate/index.html`.

[21] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988.

[22] T. V. Griffiths. The unsolvability of the equivalence problem for lambda-free nondeterministic generalized machines. *J. ACM*, 15(3):409–413, 1968.

[23] B. Groz, S. Staworko, A.-C. Caron, Y. Roos, and S. Tison. XML security views revisited. In *International Symposium on Database Programming Languages (DBPL)*, volume 5708 of *Lecture Notes in Computer Science*. Springer, August 2009.

[24] S. J. Hegner. Foundations of canonical update support for closed database views. In *Third International Conference on Database Theory Paris (ICDT'90)*, 1990.

[25] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. *Theor. Comput. Sci.*, 143(1):137–148, 1995.

[26] C. Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *VLDB*, pages 249–260, 2003.

[27] L. Libkin and C. Sirangelo. Reasoning about XML with temporal logics and automata. In *Proceedings of the International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'08)*, 2008.

[28] D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of XQuery. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 21–30, 2007.

[29] M. Murata. Hedge automata: A formal model for XML schemata. Technical report, Fuji Xerox Information Systems, 1999.

[30] W. Plandowski. Testing equivalence of morphisms on context-free languages. In *ESA*, pages 460–470, 1994.

[31] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Bull. Amer. Math. Soc.*, 74:1025–1029, 1968.

[32] J.-F. Raskin and F. Servais. Visibly pushdown transducers. In *ICALP (2)*, pages 386–397, 2008.

[33] N. Rassadko. Query rewriting algorithm evaluation for XML security views. In *Secure Data Management (VLDB Workshop)*, volume 4721 of *Lecture Notes in Computer Science*, pages 64–80. Springer, 2007.

[34] T. Schwentick. Automata for XML - a survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.

[35] S. Staworko, I. Boneva, and B. Groz. The view update problem for XML. In *EDBT/ICDT workshop (XML Updates)*, March 2010.

[36] I. Tatarinov, K. Beyer, and J. Shanmugasundaram. Storing and querying ordered XML using a relational database system. In *ACM SIGMOD International Conference on Management of Data*, 2002.

[37] R. Vercammen, J. Hidders, and J. Paredaens. Query translation for XPath-based security views. In *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 250–263. Springer, 2006.

[38] W3C. Extensible markup language (XML) 1.0, 1999. `http://www.w3.org/TR/xml/`.

[39] W3C. XML path language (XPath) 1.0, 1999. `http://www.w3.org/TR/xpath`.

[40] W3C. XML query (XQuery), 2007. `http://www.w3.org/XML/Query`.

[41] W3C. XQuery update facility 1.0, 2009. `http://www.w3.org/TR/xquery-update-10/`.

[42] L. Wang, E. A. Rundensteiner, and M. Mani. Updating XML views published over relational databases: Towards the existence of a correct update mapping. *Data and Knowledge Engineering*, 58, 2006.