Schema Evolution for XML: A Consistency-preserving Approach

Béatrice Bouchou[1] Denio Duarte[⋆1] Mírian Halfeld Ferrari Alves[1]
Dominique Laurent[2] Martin A. Musicante[⋆⋆3]

[1] Université François Rabelais - LI/Antenne de Blois, France
{bouchou, mirian}@univ-tours.fr, denio.duarte@etu.univ-tours.fr
[2] Université de Cergy-Pontoise - Departement Informatique, France
dominique.laurent@dept-info.u-cergy.fr
[3] Universidade Federal do Paraná - Departamento de Informática, Brazil
mam@inf.ufpr.br

Abstract. This paper deals with updates of XML documents that satisfy a given schema, *e.g.*, a DTD. In this context, when a given update violates the schema, it might be the case that this update is accepted, thus implying to change the schema. Our method is intended to be used by a data administrator who is an expert in the domain of application of the database, but who is not required to be a computer science expert. Our approach consists in proposing different schema options that are derived from the original one. The method is consistency-preserving: documents valid with respect to the original schema remain valid. The schema evolution is implemented by an algorithm (called `GREC`) that performs changes on the graph of a finite state automaton and that generates regular expressions for the modified graphs. Each regular expression proposed by `GREC` is a choice of schema given to the administrator.

1 Introduction

We consider an XML-based data-exchange environment in which exist both ordinary users and administrators. We are interested in updates to *valid* XML documents, *i.e.*, those that satisfy some schema constraints. When a valid XML document is updated, we have to verify that the new document still conforms to the imposed constraints. *Invalid updates*, *i.e.*, updates resulting in invalid XML documents, can be treated in different ways, according to the kind of user performing them. Invalid updates performed by ordinary users are rejected, whereas invalid updates performed by administrators can be accepted, thus provoking changes on the schema.

We propose a method to enforce the validity of an update by means of changing the schema. Our approach aims to deal with the increasing demand for tools

[⋆] Supported by CAPES (Brazil) BEX0353/01-9.
[⋆⋆] Part of this work was done while the author was visitting Université François Rabelais. Supported by CAPES (Brazil) BEX1851/02-0.

specially designed for administrators not belonging to the computer science community, but capable of making decisions on the evolution of an application [8]. This kind of user needs a system that assures a consistent evolution of the schema in a incremental and interactive way. The main features of our method are: (a) If an update violates schema constraints then corrective actions are taken to restore validity. This is done by computing all relevant schema changes. Each option is obtained from the characteristics of the schema and documents being updated. (b) Valid documents w.r.t. the original schema remain valid w.r.t. the new one. (c) Different choices of schema are given to the administrator. The administrator can decide which schema is to be adopted, based on their knowledge about the semantics of the documents.

To our knowledge, our approach adopts a new strategy to deal with schema evolution. Research papers in a similar domain (such as [7, 10, 11]) propose to change XML documents but not the schema, in case of invalid updates. Notice that our aim is much less ambitious than automatic learning of automata: we propose `GREC` as a simple and directly usable solution to an interesting problem. Nevertheless, as much work has been done in the area of inference of regular grammars from examples [**?**,**?**], we are considering the comparison between our approach and these ones.

An XML document is seen as an unranked labeled tree t having different kinds of nodes (*data*, *elements* and *attributes*). We assume a schema (defining some element and attribute constraints) specified by an unranked bottom-up tree automaton $\mathcal{A}$ [1]. Checking if an XML document respects the constraints established by the schema is equivalent to run $\mathcal{A}$ with input t. Updates are seen as changes to be performed on XML tree representations and invalid updates are those that produce XML trees which cannot be recognized by $\mathcal{A}$. We focus on (changing the schema after invalid) insertions, since deletions are easy to treat.

An insert operation consists in the insertion of a sub-tree t' into a given position p of t. Before accepting an insertion, we have to test if the new tree respects the constraints established by the associated tree automaton $\mathcal{A}$. These tests are incremental [3]. If the tests fail, changes on the schema are proposed to the administrator. Changing the schema means changing $\mathcal{A}$. The following example illustrates (i) how to validate an XML document using $\mathcal{A}$ and (ii) how an insertion requested by an administrator can lead to changes to $\mathcal{A}$.

Example 1. Fig. 1(a) shows the labeled tree t, representing part of an XML document. Each node has a label (*e.g.*, *Author*) and a position (like 00, for an *Author* node and ϵ for the root). Let $\mathcal{A}$ be a tree automaton representing a schema. The execution of $\mathcal{A}$ with input t is represented by the labeled tree r (Fig. 1(b)). To illustrate how we obtain r, suppose that $\mathcal{A}$ contains the transition rule

$$Production, <\emptyset, \emptyset>, q_{Sub}\ (q_{Year}\ q_{JPaper}{}^{+})^{*} \rightarrow q_{Production} \quad (1)$$

The intended semantics of the regular expression $E = q_{Sub}\ (q_{Year}\ q_{JPaper}{}^{+})^{*}$ is that the production of a given author is stated by the area (or subject) of their research, followed by a list of journal papers, presented by year.
Rule (1) states that a position p, labeled *Production* in t, can be associated with

the state $q_{Production}$ in r if the constraints established by Rule (1) are respected[4]. As position 001 in t respects these constraints then node 001 in r is assigned to the corresponding state.
The tree automaton executes bottom-up by considering each position and the transition rule that applies to it. A tree automaton accepts a document tree if and only if the state associated to the root is *final*. In this case, we say that t is valid.

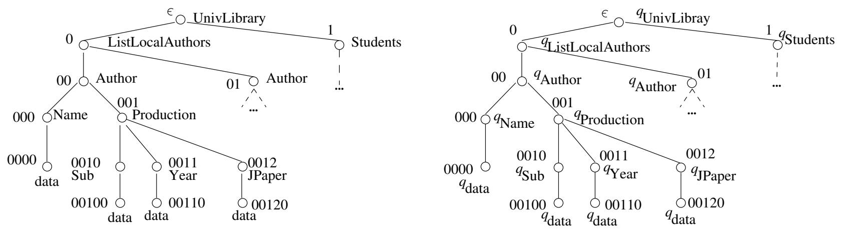

Fig. 1. (a) Labeled tree t. (b) Labeled tree r

We consider now an insertion of a tree t_a at position 0013 (*i.e.*, on the right of position 0012) of the valid tree t. We suppose that the execution of $\mathcal{A}$ over t_a results in a labeled tree whose root is associated to $q_{ConfPaper}$.
In our case, as the update concerns position 0013, we should check if the subtree rooted at position 001 (0013's father) still respects the validity conditions. In other words, we should verify whether the state associated to position 001 after the update is still $q_{Production}$. This is done by analyzing the behavior of Rule (1). As the transition rule defines the possible children of a node using regular expressions we should check if the word $q_{Sub}q_{Year}q_{JPaper}q_{ConfPaper}$, corresponding to the concatenation of the states associated to children of position 001 after the update, matches the regular expression E. In our case, this does not happen and thus we have an invalid update.
As our user is an administrator, the requested update will be taken as a request to change the schema. To understand this request we consider the new word $q_{Sub}q_{Year}q_{JPaper}q_{ConfPaper}$ and the original regular expression $E = q_{Sub}$ $(q_{Year}\ q_{JPaper}{}^{+})^{*}$ appearing in (1). The goal is to replace $\mathcal{A}$ by a new tree automaton $\mathcal{A}'$ having the following characteristics: (i) every XML document valid with respect to $\mathcal{A}$ is also valid with respect to $\mathcal{A}'$ and (ii) $\mathcal{A}'$ differs from $\mathcal{A}$ only in the regular expression affected by the update. The options are:
1. $(q_{Sub}(q_{Year}\ q_{JPaper}{}^{+})^{*}q_{ConfPaper}?)$ and $(q_{Sub}(q_{Year}\ q_{JPaper}{}^{+})^{*}(q_{ConfPaper})^{*})$:
Choices allowing the insertion of one or several conference papers to a given domain (not organized by year).
2. $(q_{Sub}(q_{Year}\ q_{JPaper}{}^{+}q_{ConfPaper}?)^{*})$ and $(q_{Sub}(q_{Year}\ q_{JPaper}{}^{+}q_{ConfPaper}{}^{*})^{*})$:
These options allow the insertion of one or several conference papers to a given domain per year.

[4] Here, the constraints are: p should have no attribute children (due to the tuple $<\emptyset, \emptyset>$) and the word formed by the concatenation of the states associated to the element children of p should match the regular expression q_{Sub} $(q_{Year}\ q_{JPaper}{}^{+})^{*}$.

3. $(q_{Sub}(q_{Year}(q_{JPaper}\ q_{ConfPaper}?)^+)^*)$ and $(q_{Sub}(q_{Year}(q_{JPaper}\ q_{ConfPaper}{}^*)^+)^*)$:
Choices allowing the insertion of several conference papers to a given domain every year. In the first case, each conference paper (if it exists) should be preceded by a journal paper while, in the second case, this restriction is dropped. In both cases, a conference paper exists only if at least one journal paper exists.
4. $(q_{Sub}(q_{Year}(q_{JPaper} \mid q_{ConfPaper})^+)^*)$:
Journal and conference papers can exist alone. However, at least one of them should exist per year, *i.e.*, authors must have one publication per year.

Given these options, the administrator can choose the one that fits the best the application. $\square$

From the above example, we can notice that our goal is to propose several choices of regular expressions, trying to foresee the needs of an application. Indeed, each candidate regular expression E' corresponds to a language $L(E')$ more general than $L(E) \cup \{w'\}$ (where E is the original regular expression and $w' \notin L(E)$). We are neither interested in the candidate $E|w'$ that adds just one word to $L(E)$, nor in candidates too general allowing any kind of updates[5]. Our interest concerns candidates E' such that they are as similar to E as possible. Notice that in the previous example, each proposed regular expression have just one alphabet symbol inserted, in relation to the original one. This condition will be reflected by a very simple notion of distance, defined in Section 2.

Each transition rule of $\mathcal{A}$ has the general form $a, S, E \rightarrow q$ where a is a label, S is a tuple of two disjoint sets of states establishing attribute constraints ($S = <S_{optional}, S_{compulsory}>$ with $S_{optional}$ for optional attributes and $S_{compulsory}$ for compulsory ones), E is a regular expression establishing element constraints and q is a state. A run of $\mathcal{A}$ on a tree starts its computation at the leaves and then simultaneously works up the paths of the tree. The tree automaton accepts a tree t if all the attributes and element constraints defined in $\mathcal{A}$ (via the transition rules) hold in t. The insertion of a labeled tree t_a at position p of tree t can provoke changes on S (if the root of t_a is an attribute) or on E (if the root of t_a is an element or data). We concentrate on the changes occurring on E, *i.e.*, we only consider the insertion of sub-elements. Moreover, we only consider the insertion of one sub-element at a time.

Given a transition rule $a, S, E \rightarrow q$. Let $w = \alpha\beta$ be the word formed by the concatenation of the states associated to the element children of the position p, in a valid XML tree t. Thus, w belongs to the language $L(E)$. The insertion of t_a as a child of position p, in t, corresponds to the construction of a new word $w' = \alpha n \beta$ (always associated to the sub-elements of p). Notice that n corresponds to the state that $\mathcal{A}$ associates to the root of t_a. If w' is not in $L(E)$ then the rule $a, S, E \rightarrow q$ cannot be applied after the update. Our approach consists in computing new regular expressions to replace E according to the structure imposed by E (*i.e.*, number of starred sub-expression, disjoint symbols, etc.) and the characteristics of the unrecognized word w'. Thus, we propose a method that,

[5] As, for instance, a method that gives $E' = a^*b^*$ as the result for $E = ab$, $w = ab$ and $w' = aab$.

given a regular expression E and a new word w' to accept, (i) computes a finite state automaton M_E associated to E; (ii) performs some modifications on M_E to obtain new automata M'_E that accepts w' and (iii) finds regular expressions E' associated to each M'_E. To avoid important changes in E, we propose new regular expressions that have the smallest distance from E, that preserve the syntactic nesting of E and that respect the structure of w'. To this end, we introduce an algorithm called `GREC` which is an extension of the transformation of a Glushkov automaton into a regular expression presented in [6].
Section 2 presents some theoretical notions necessary to understand the schema evolution method implemented by `GREC` which is introduced in Section 3. Proofs are omitted due to lack of space (see [2]).

2 Theoretical Background

In this section, we consider the method proposed in [6] to obtain a homogeneous[6] finite state automaton, called Glushkov automaton, that recognizes the language associated to a given regular expression. Given a regular expression, a Glushkov automaton is built by subscribing each alphabet symbol in this regular expression with its position. In a Glushkov automaton, each non initial state corresponds to a position in the regular expression. For instance, given the regular expression $E = (a(b|c)^*)^*d$, the subscribed regular expression is $\overline{E} = (a_1(b_2|c_3)^*)^*d_4$. The Glushkov automaton $M = (\Sigma, Q, \Delta, q_0, F)$, built from E, is such that: the alphabet is $\Sigma = \{a, b, c, d\}$, the set of states is $Q = \{0, 1, 2, 3, 4\}$, the initial state is $q_0 = 0$, the set of final states is $F = \{4\}$ and the transition relation Δ is defined by the edges of the graph in Fig. 2(a).

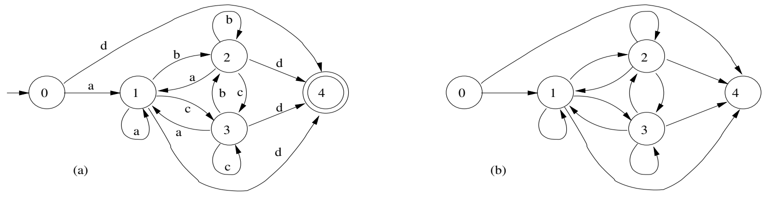

Fig. 2. (a) Pictorial representation of a FSA for $(a(b|c)^*)^*d$. (b) Its Glushkov graph

A *Glushkov graph* is the directed graph $G = (X, U)$ obtained from a Glushkov automaton such that each node in X corresponds to a state and each edge in U to a transition. Since Glushkov automata are homogeneous, their edges are not decorated as shown in Fig. 2(b).

Let $G = (X, U)$ be a graph. An edge between nodes r and s, denoted by $u = (r, s)$, is a *loop* iff $r = s$. A *path* is a sequence of nodes $x_0, \ldots, x_n$ such that, for every $0 \leq i < n$, (x_i, x_{i+1}) is an edge in G. A path with no edges is said to be trivial. A graph has a *root* node r (resp. an *antiroot*) if there exists a path from

[6] A finite state automaton is said to be *homogeneous* [6] if one always enters a given state by the same symbol.

r to any node in the graph (resp. from any node in the graph to r). A graph is a *hammock* if it has both a root (r) and an antiroot (s), with $r \neq s$.

Now we consider the graph properties taken from [6], that will be used later on in this work. A set $\mathcal{O} \subseteq X$ is said to be an *orbit* if for all x and x' in $\mathcal{O}$ there exists a non trivial path from x to x'. An orbit is *maximal* if for each node x of $\mathcal{O}$ and for each node x' out of $\mathcal{O}$, there does not exist a path from x to x' and a path from x' to x. Notice that an orbit is maximal if it is not contained in any other orbit. Let $\mathcal{O}$ be an orbit, we define: $In(\mathcal{O})=\{x \in \mathcal{O} \mid \exists x' \in (X \backslash \mathcal{O}), (x', x) \in U\}$ as the *input* of $\mathcal{O}$ and $Out(\mathcal{O})=\{x \in \mathcal{O} \mid \exists x' \in (X \backslash \mathcal{O}), (x, x') \in U\}$ as the *output* of $\mathcal{O}$. An orbit $\mathcal{O}$ is *stable* if $\forall x \in Out(\mathcal{O})$ and $\forall y \in In(\mathcal{O})$, the edge (x, y) exists. An orbit $\mathcal{O}$ is *transverse* if $\forall x, y \in Out(\mathcal{O})$, $\forall z \in (X \setminus \mathcal{O}), (x, z) \in U \Rightarrow (y, z) \in U$, and if $\forall x, y \in In(\mathcal{O})$, $\forall z \in (X \setminus \mathcal{O}), (z, x) \in U \Rightarrow (z, y) \in U$. An orbit $\mathcal{O}$ is *strongly stable* (resp. *strongly transverse*) if it is stable (resp. transverse) and if after deleting the edges in $Out(\mathcal{O}) \times In(\mathcal{O})$, every suborbit is strongly stable (resp. strongly transverse).

Example 2. The graph (a hammock) of Fig. 2(b) has 7 orbits. The orbit $\mathcal{O}_1 = \{1, 2, 3\}$, with $In(\mathcal{O}_1) = \{1\}$ and $Out(\mathcal{O}_1) = \{1, 2, 3\}$, is maximal. Orbits $\mathcal{O}_2 = \{1, 2\}$ and $\mathcal{O}_3 = \{2, 3\}$ are not maximal. Orbit $\mathcal{O}_3$ is stable since all the edges in $Out(\mathcal{O}_3) \times In(\mathcal{O}_3)$ are in $\mathcal{O}_3$. It is transverse since all the edges $(1, 2)$, $(1, 3)$, $(2, 4)$ and $(3, 4)$ exist in the graph. In fact, $\mathcal{O}_1$, $\mathcal{O}_2$ and $\mathcal{O}_3$ are strongly stable and strongly transverse. □

Given a graph G in which all orbits are strongly stable, we build a *graph without orbits* $SO(G)$ by recursively deleting, for each maximal orbit $\mathcal{O}$, all edges (x, y) such that $x \in Out(\mathcal{O})$ and $y \in In(\mathcal{O})$. The process ends when there are no more orbits. Notice that $SO(G)$ is defined in a unique way [6].

Let x be a node in a graph without orbits $G = (X, U)$. We denote by $Q^-(x) = \{y \in X \mid (y, x) \in U\}$ the set of immediate predecessors of x and by $Q^+(x) = \{y \in X \mid (x, y) \in U\}$ the set of immediate successors of x. The graph G is reducible if it is possible to reduce it to one node by successive applications of any of the rules $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3$ below (as illustrated in Fig. 3).

Rule $\mathbf{R}_1$: If two nodes x and y are such that $Q^-(y) = \{x\}$ and $Q^+(x) = \{y\}$, then replace node x by node xy and delete node y.

Rule $\mathbf{R}_2$: If two nodes x and y are such that $Q^-(x) = Q^-(y)$ and $Q^+(x) = Q^+(y)$, then replace node x by node $x|y$ and delete node y.

Rule $\mathbf{R}_3$: If a node x is such that $y \in Q^-(x) \Rightarrow Q^+(x) \subset Q^+(y)$, then replace node x by node $x?$ and delete the edges going from $Q^-(x)$ to $Q^+(x)$.

Notice that each node of the graph has a regular expression (which, initially, is just the position identifying the node). At the end of the process, the graph has just one node whose content is the regular expression corresponding to the original Glushkov automaton. In this way, we obtain a regular expression from a Glushkov automaton.

If x is optional, by $\mathbf{R}_3$ we build a regular expression in the following way: If the original regular expression associated to x is of the form E^+ (resp. E) then the new one will be E^* (resp. $E?$). From now on, we use the notation $E!$ to stand both for $E?$ and E^*. The node resulting from the application of one rule keeps

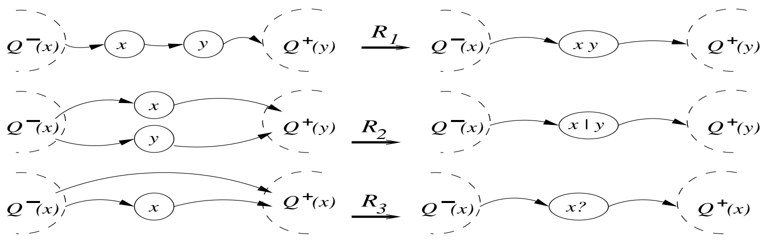

Fig. 3. Rules $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3$

the identities of its origins, responding for them. Notice that the above rules do not account to the construction of E^+ expressions. This kind of expression will appear by considering the orbits existing in the original graph. It is important to remark that the reduction process works from inside to outside of the nested maximal orbits. Maximal orbits are built during the construction of $SO(G)$ and they are hierarchically organized according to the set-inclusion relation.

The characterization[7] of Glushkov FSA is given by the following theorem. Moreover, from Lemma 1 below, we can associate an orbit in a Glushkov graph to a Kleene closure in the corresponding regular expression.

Theorem 1. *[6] $G = (X, U)$ is a Glushkov graph iff the following conditions are satisfied:* (1) *G is a hammock,* (2) *each maximal orbit in G is strongly stable and strongly transverse and* (3) *the graph without orbit of G is reducible.*

Lemma 1. *[6] Let $G = (X, U)$ be a graph that satisfies the properties* (2) *and* (3) *of Theorem 1. Let $\mathcal{O}$ be a maximal orbit in G. By iteration of $\boldsymbol{R}_1$, $\boldsymbol{R}_2$ and $\boldsymbol{R}_3$ in $SO(G)$, the orbit $\mathcal{O}$ is reduced to a unique node, under the assumption that $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ are only applied to pairs $(x, y) \in (\mathcal{O} \times \mathcal{O})$ or $(x, y) \in [(X \setminus \mathcal{O}) \times (X \setminus \mathcal{O})]$.*

We define now a very simple notion of the distance between two regular expressions, based on the number of positions of the subscribed expressions:

Definition 1. *Let E and E' be regular expressions. Let $\overline{E}$ and $\overline{E'}$ be subscripted expressions built from E and E', respectively, by using the Glushkov method. Let S^E (resp. $S^{E'}$) be the set of positions of $\overline{E}$ (resp. $\overline{E'}$). The* distance *between E and E', denoted by $\mathcal{D}(E, E')$, is $\mathcal{D}(E, E') = || S^E - S^{E'} ||$.*

3 Schema Evolution by Changing Glushkov Graphs

We recall that, in our work, a schema is defined by an unranked bottom-up tree automaton $\mathcal{A}$ obtained from an (unambiguous) DTD [1] and that updates are seen as changes to be performed on an XML tree as shown in Example 1. Invalid updates produce XML trees which cannot be recognized by $\mathcal{A}$. To deal with this kind of updates we present a method that proposes changes to the

[7] To characterize a Glushkov FSA, add an end mark (#) to every string [6].

schema. If the update corresponds to the insertion of an attribute then the new schema is obtained just by allowing the existence of a new optional attribute. No attributes can be inserted as compulsory, otherwise the validity of documents with respect to the original schema is not preserved. Schema evolution due to a delete operation is also straightforward to define, since it consists in rendering optional the deleted attribute or sub-element. The challenge in this context is to consider the evolution of the schema caused by the insertion of an element. In this case, the schema evolution is achieved by changing the regular expression E that constraints the sub-elements of a given node in the XML tree. In this context, our problem can be expressed as follows:

(1) The insertion of a labeled tree t_1 as a child of node a in tree t means changing the original word w obtained from the children of a. The new word w' is obtained from w by inserting the new state (associated to the root of t_1).

(2) Given the regular expression E (with $w \in L(E)$) and a word w', our problem is to propose new regular expressions E', such that $\mathcal{D}(E, E') = 1$, and whose languages contain, at least, the word w' and $L(E)$.

To work on words w and w', we use a Glushkov automaton M_E. This automaton is built by applying the method of [6], mentioned in Sect. 2, over each E that appears in the transition rules of $\mathcal{A}$. In M_E each state (but the initial one) corresponds to a position in the subscribed regular expression $\overline{E}$. The only final state of M_E is subscribed with the position of the end mark (#).

We consider now the execution of M_E over the new word w'. Let p be the position of w' where the new symbol is inserted. We define the *nearest left state* (s_{nl}) as a state in M_E reached after reading the first $p-1$ symbols in w' (or in w). Similarly, we define the *nearest right state* (s_{nr}) as a state in M_E that succeeds s_{nl} when reading the p-th position of w. Notice that to determine nodes s_{nl} and s_{nr} (to be passed to GREC), we scan w' using M_E. If the inserted symbol already belongs to the alphabet of M_E, a simple backtracking technique may be used to identify s_{nl} and s_{nr} (see [2] for details). Notice that both s_{nl} and s_{nr} exist and, when M_E is deterministic (as usually recommended in XML domain), they are unique.

Without loss of generality, we assume that an insertion operation always corresponds to the insertion of a new position in E. Thus, to accept the new word, we should insert a new state in M_E. This new state (s_{new}) should be added to M_E and there should exist a transition from s_{nl} to s_{new}. However this is not the only change to be performed on M_E. Other changes are needed in order to keep the graph associated to the automaton as a Glushkov graph. These changes depend on the situation of s_{nl} and s_{nr} in the Glushkov graph.

For a general regular expression E, the task of finding the places where the new symbol may be added is not trivial. There is a great variety of possible solutions and it is hard to find those that fit the best in a given context. As shown in Example 1, we want that the candidates respect the nesting of sub-expressions of the original regular expression. The reduction process of [6] is well adapted to our goal of proposing solutions that preserve the general structure of the original regular expression E, since it follows the syntactic nesting of E

using the orbits. Moreover, inserting a new state in M_E means inserting just one new position in the corresponding $\overline{E}$. In other words, our approach proposes only new regular expressions E' such that $\mathcal{D}(E, E') = 1$.

Each reduction step in [6] consists in replacing part of the automaton graph by nodes containing more complex regular expressions. The reduction finishes when the automaton graph is formed by just one node containing one regular expression, which corresponds to the original regular expression. Our goal is to build a new graph G' from a Glushkov graph G by preserving the Glushkov properties (Theorem 1).

Fig. 4 presents a high level algorithm for the procedure `GREC` (**G**enerate **R**egular **E**xpression **C**hoices), which is an extension of the method of [6] (explained in Sect. 2). `GREC` generates a list of regular expressions, each of which corresponds to a solution obtained by the insertion (in different places) of s_{new} in the original graph. The generated list contains the options we give to the administrator. `GREC` takes five input parameters: a graph without orbits G_A, a hierarchy of orbits O_A, two nodes of the graph, corresponding to s_{nl} and s_{nr}, and the new node s_{new} to be inserted.

```
(1) procedure GREC($G_A$, $O_A$, $s_{nl}$, $s_{nr}$, $s_{new}$) {
(2)  if graph $G_A$ has only one node
(3)    then stop
(4)    else{
(5)     $R_i$ := ChooseRule($G_A$, $O_A$);
(6)     for each $(G_B, O_B)$:=LookForGraphAlternative($G_A, O_A, R_i, s_{nl}, s_{nr}, s_{new}$) do
(7)         GraphToRegExp($G_B$, $O_B$);
(8)     $G_C$ := ApplyRule($R_i$, $G_A$);
(9)     GREC($G_C, O_A, s_{nl}, s_{nr}, s_{new}$);
(10)   }      }
```

Fig. 4. Algorithm to generate regular expression choices from a Glushkov graph

The Procedure `ChooseRule` uses the information concerning orbits to select a rule to be applied in the reduction of the graph. The Procedure `ApplyRule` computes a new graph resulting from the application of the selected rule, and the Procedure `GraphToRegExp` computes a regular expression from a given graph.

At each step of the reduction, `GREC` checks whether the chosen rule affects nodes s_{nl} or s_{nr} and, in this case, it modifies the graph to take into account the insertion of the new node s_{new}. The modifications to the graph being reduced are driven by $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3$ and by the information concerning the orbits of the original graph. Each of these modifications is performed by the iterator `LookForGraphAlternative` (line `(6)` of Fig. 4). The iterations stop when no more alternatives are found. The role of iterator `LookForGraphAlternative` is two-fold: (i) it verifies whether nodes s_{nl} and s_{nr} satisfy the conditions stated in $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3$ and (ii) it generates new graphs (G_B in the algorithm of Fig. 4) over which the algorithm `GraphToRegExp` is applied to generate new regular expressions. The following definitions formalize how `LookForGraphAlternative` builds new graphs from the original one.

Definition 2. *Let $G_A = (X_A, U_A)$ be a Glushkov graph and $x, y \in X_A$ be two nodes on which $\boldsymbol{R}_1$ can be applied. Define each $G_i = (X_i, U_i)$ from G_A as follows:*

<u>Case 1</u> ($x = s_{nl}$ and $y = s_{nr}$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(x, s_{new})\} \cup \{(s_{new}, y)\}$.
<u>Case 2</u> ($x = s_{nr}$ and $y = s_{nl}$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(y, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(y)\}$.
 $\boldsymbol{G}_2$: $X_2 = X_A \cup \{s_{new}\}$; $U_2 = U_A \cup \{(s_{new}, x)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$.

Definition 3. *Let $G_A = (X_A, U_A)$ be a Glushkov graph and $x, y \in X_A$ be two nodes on which $\boldsymbol{R}_2$ can be applied. Define each $G_i = (X_i, U_i)$ from G_A as follows:*

<u>Case 1</u> ($x = s_{nl}$ and $s_{nr} \in Q^+(x)$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(x, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
<u>Case 2</u> ($x = s_{nr}$ and $s_{nl} \in Q^-(x)$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(s_{new}, x)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$.
<u>Case 3</u> ($s_{nl} \in Q^-(x)$ and $s_{nr} \in Q^+(x)$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(z, s_{new}) \mid z \in Q^-(x)\} \cup \{(s_{new}, v) \mid v \in Q^+(x)\}$.

Definition 4. *Let $G_A = (X_A, U_A)$ be a Glushkov graph and $x \in X_A$ be a node on which $\boldsymbol{R}_3$ can be applied. Define each $G_i = (X_i, U_i)$ from G_A as follows:*

<u>Case 1</u> ($s_{nl} \in Q^-(x)$ and $s_{nr} \in Q^+(x)$) or ($s_{nl} \in Q^-(x)$ and $s_{nr} \notin Q^+(x)$) or ($s_{nl} \notin Q^-(x)$ and $s_{nr} \in Q^+(x)$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(s_{new}, x)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$.
 $\boldsymbol{G}_2$: $X_2 = X_A \cup \{s_{new}\}$; $U_2 = U_A \cup \{(x, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
 $\boldsymbol{G}_3$: $X_3 = X_A \cup \{s_{new}\}$; $U_3 = U_A \cup \{(z, s_{new}) \mid z \in Q^-(x)\} \cup \{(s_{new}, v) \mid v \in Q^+(x)\}$.

Rules $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3$ are first applied inside each orbit [6]. During the reduction process, each orbit $\mathcal{O}$ of the original graph is reduced to just one node containing a regular expression. This regular expression is then decorated by $^+$. Before applying this decoration we have to consider the insertion of s_{new} in the orbit $\mathcal{O}$. The next definition summarizes the situations in which we perform a modification on an orbit. It gives the conditions and modifications concerning the cases in which a whole orbit is represented by one node of the graph.

Definition 5. *Let $G_A = (X_A, U_A)$ be a Glushkov graph. Let $\mathcal{O}$ be an orbit reduced to one node $x \in X_A$. We define each $G_i = (X_i, U_i)$ from G_A as follows:*

<u>Case 1</u> ($x = s_{nl}$ and $x = s_{nr}$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(s_{new}, x)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$.
 $\boldsymbol{G}_2$: $X_2 = X_A \cup \{s_{new}\}$; $U_2 = U_A \cup \{(x, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
<u>Case 2</u> ($s_{nl} \in Q^-(x)$ and $s_{nr} \in \mathcal{O}$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(s_{new}, x)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$.
 $\boldsymbol{G}_2$: $X_2 = X_A \cup \{s_{new}\}$; $U_2 = U_A \cup \{(v, s_{new}) \mid v \in Q^-(x)\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
<u>Case 3</u> ($s_{nl} \in \mathcal{O}$ and $s_{nr} \in Q^+(x)$):
 $\boldsymbol{G}_1$: $X_1 = X_A \cup \{s_{new}\}$; $U_1 = U_A \cup \{(x, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
 $\boldsymbol{G}_2$: $X_2 = X_A \cup \{s_{new}\}$; $U_2 = U_A \cup \{(v, s_{new}) \mid v \in Q^-(x)\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
Moreover, in all cases, s_{new} is added to the orbit $\mathcal{O}$.

Notice that for each graph built using the definitions above a regular expression is obtained.

Example 3. Consider Example 1 and the Glushkov automaton (Fig. 5(a)) corresponding to $E{=}q_{Sub}(q_{Year}\ q_{JPaper}{}^{+})^{*}\#$ (and $\overline{E}{=}\ 1\ (2\ 3^{+})^{*}4$). In this case, `GREC` takes as input a hierarchy of orbits containing $\mathcal{O}_1 = \{3\}$ and $\mathcal{O}_2 = \{2, 3\}$. It starts the reduction by $\mathcal{O}_1$. At this step, Definition 5 (third case) applies since $\mathcal{O}_1$ is represented by just one node, $s_{nl} = 3$ and $s_{nr} = 4$. Graphs (without orbits) G_1 and G_2 (Fig. 5(b)-(c)) are built, giving rise to $(q_{Sub}(q_{Year}(q_{JPaper}\ q_{ConfPaper}!)^{+})^{*})$ and $(q_{Sub}(q_{Year}(q_{JPaper}\ \mid\ q_{ConfPaper})^{+})^{*})$, respectively (options 3 and 4 from Example 1).

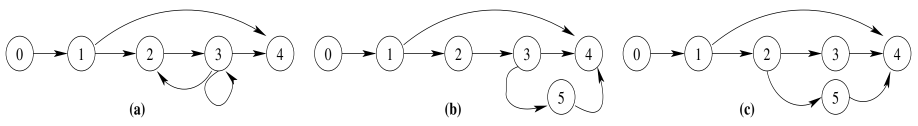

Fig. 5. (a)Automaton. (b)-(c) Graphs updated by `LookForGraphAlternative`

`GREC` solutions respect the properties stated by the following theorems.

Theorem 2. *Let G be Glushkov graph and $G_A = SO(G)$. Let O_A be the hierarchy of orbits obtained during the construction of G_A. Let R_i be one of the reduction rules $\boldsymbol{R}_1$, $\boldsymbol{R}_2$ or $\boldsymbol{R}_3$. For any nodes s_{nl}, s_{nr} and s_{new}, each pair (G_B, O_B) resulting from the execution of* `LookForGraphAlternative(`G_A`,` O_A`,` R_i`,` s_{nl}`,` s_{nr}`,` s_{new}`)` *is a representation of a Glushkov graph G', where G_B is a graph without orbits, and O_B is the hierarchy of orbits obtained when constructing G_B from G'.*

Theorem 3. *Let E be a regular expression and $L(E)$ be the language associated to E. Given $w \in L(E)$ such that $w = \alpha\beta$, let $w' = \alpha n \beta$ where n is a symbol and $w' \notin L(E)$. Let M_E be a deterministic Glushkov automaton corresponding to E, let G_A be the graph without orbits obtained from M_E and let O_A be the hierarchy of orbits obtained during the construction of G_A. Let s_{nl} be a state in M_E reached after reading α and let s_{nr} be a state that succeeds s_{nl} in M_E when reading w. Let s_{new} be a new node not in G_A. The execution of* `GREC (`G_A`,` O_A`,` s_{nl}`,` s_{nr}`,` s_{new}`)` *returns a finite, nonempty set of regular expressions $\{E_1, \ldots, E_m\}$. For each E_i, we have $L(E) \cup \{w'\} \subset L(E_i)$ and $\mathcal{D}(E, E_i) = 1$.*

If unambiguous expressions are required as a result, `GREC` signalizes the ambiguity of a candidate regular expression - we can then transform the chosen candidate into an equivalent unambiguous regular expression along the lines of [**?**]. Notice that if E is unambiguous and n is not in E then each candidate regular expression E_i given by `GREC` is unambiguous.

4 Conclusion

In this paper we propose a method to dynamically change the schema for XML databases based on an update of just one document. Our approach is original in the sense that it does not impose changes on documents, but rather, computes a set of new schema that preserve the consistency of the document and that has minimal changes in relation to original regular expression. `GREC` solutions are ordered according to the hierarchy of orbits (given as input) which can define different context of updates (see [2] for details). We have implemented a prototype of `GREC` using the ASF+SDF [4] meta-environment under Linux.

We are currently considering the following research directions: (*i*) The generalization of the schema evolution process discussed here, to consider a sequence (or a set) of administrator's updates and (*ii*) the implementation of an XML update language such that UpdateX [**?**] in which incremental schema evolution will be integrated.

Acknowledgements: We would like to thank the anonymous referees for their useful comments and corrections.

References

1. B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, and D. Laurent. Extending tree automata to model XML validation under element and attribute constraints. In *ICEIS*, 2003.
2. B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, D. Laurent, and M. Musicante. Evolving schemas for XML: An incremental approach. Technical Report (To appear), Université de François Rabelais, LI, 2004.
3. B. Bouchou and M. Halfeld Ferrari Alves. Updates and incremental validation of XML documents. In *The 9th DBPL*, number 2921 in LNCS, 2003.
4. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling rewrite systems: The ASF+SDF compiler. *ACM, Transactions on Programming Languages and Systems*, 24, 2002.
5. A. Bruggeman-Klein and D. Wood. Deterministic regular languages. In *STACS*, 1992.
6. P. Caron and D. Ziadi. Characterization of glushkov automata. *TCS: Theorical Computer Science*, 233:75–90, 2000.
7. E. Kuikka, P. Leinonen, and M. Penttonen. An approach to document structure transformations. In *Proceedings of Conference on Software: Theory and Practice, pp. 906-913.*, 2000.
8. J. Roddick, L. Al-Jadir, L. Bertossi, M. Dumas, F. Estrella, H. Gregersen, K. Hornsby, J. Lufter, F. Mandreoli, T. Männistö, E. Mayol, and L. Wedemeijer. Evolution and change in data management - issues and directions. *SIGMOD Record*, 29(1):21–25, 2000.
9. M. de Rougemont. The correction of XML data. In *The First Franco-Japanese Workshop on Information, Search, Integration and Personalization - ISIP*, 2003.
10. H. Su, D. Kramer, L. Chen, K. T. Claypool, and E. A. Rundensteiner. XEM: Managing the evolution of XML documents. In *RIDE-DM*, pages 103–110, 2001.
11. H. Su, H. Kuno, and E. A. Rundensteiner. Automating the transformation of XML documents. In *3rd WIDM.* ACM, 2001.