

Conservative Extensions of Regular Languages

Béatrice Bouchou Denio Duarte Mírian Halfeld Ferrari Alves
Université François Rabelais - LI/Antenne de Blois - France
{bouchou,mirian}@univ-tours.fr, denio.duarte@etu.univ-tours.fr

Dominique Laurent
Université de Cergy-Pontoise - LICP - France
dominique.laurent@dept-info.u-cergy.fr

Martin A. Musicante
Universidade Federal do Paraná - Departamento de Informática - Curitiba - Brazil
mam@inf.ufpr.br

Abstract

Given a regular expression E and a word w belonging to the language associated to E (i.e. $w \in L(E)$), we consider the following problem: Let the word w' be obtained from w by adding or removing one symbol. In this case, we are interested in building new regular expressions E' such are similar to E and such that $L(E) \cup \{w'\} \subseteq L(E')$.

The new regular expressions are computed by an algorithm called GREC that performs changes on a finite state automaton accepting $L(E)$, in order to derive the new regular expressions E' .

Our method consists in proposing different choices, permitting the evolution of the data belonging to an application. As the final choice depends on the intended semantics for the data, it is left to an advised user.

1. Introduction

We consider a word w belonging to a regular language L , represented by the regular expression E . In this language, each symbol has a precise semantics. For instance, suppose that E serves as the content model (schema) of an XML element. In this context, a word $w \in L$ represents the current content (i.e., the sub-elements) of the considered element. We assume that updates to w are possible (i.e., the insertion or deletion of one symbol in a position p of w). Clearly, an update over w results in a new word w' not necessarily belonging to the language L . Indeed, if we consider a finite state automaton M_E associated to E and if we assume that $w' \notin L$, then we can say that M_E fails recognizing w' . We want this failure to trigger the creation of new

regular expressions E' which extend L to new regular languages L' . In our context, the extension should be *conservative* (i.e., $L \subseteq L'$) and should include not only w' but also other words intended to be semantically close to w' . Our goal is to propose several choices of regular expressions, trying to foresee the needs of an application, and this choices must be as close as possible to the original regular expression. An authoritative user is then responsible for choosing one of them.

To build new regular expressions, we propose an algorithm that works on the automaton M_E associated to E . It performs changes on M_E in order to obtain new automata M'_E . For each M'_E , a regular expression E' is generated, representing a choice for the user. These new regular expressions are produced by respecting the syntactic nesting of E . Indeed our algorithm, called GREC, is an extension of the reduction process proposed in [1] to transform a Glushkov automaton into a regular expression. This reduction procedure is well adapted to our goal since it respects the syntactic nesting of the starred sub-expressions of the regular expression E . We can verify the possibility of inserting the new state at each step of the reduction process, depending on the new word we want to accept.

Research work on learning automata (such as [2, 3]) deal with the construction of automata from scratch, based on examples and counterexamples. Our approach is different from theirs, in the sense that we need to *extend* a previously defined regular language, as well as proposing regular expressions for the extended languages. These regular expressions should be as similar as possible to the one that defines the original regular language. Similarity between regular expressions is characterized by a simple notion of distance.

Our main contributions can be summarized as follows:

- An algorithm to compute new regular expressions E' in order to extend a given language L (represented by the regular expression E) in a conservative way. This computation is activated by the failure of the automaton M_E in recognizing w' , a word resulting from an update performed over a word $w \in L$. Along with the fact that the extension must be conservative, the proposed new languages L' are, usually, more general than $L \cup \{w'\}$, trying to foresee the needs of an application. Moreover, each new regular expression E' is as close as possible to E , according to a distance to be defined in the paper.
- Different choices of new regular expressions are proposed to an authoritative user, capable of deciding the best way an application should evolve. This method has been applied in an XML-based data-exchange environment as a decision support system for schema evolution. In this context, it helps administrators of information systems who are experts in the domain of an application, but who are not experts in computer science.

The paper is organized as follows: Section 2 recalls the reduction principles presented in [1]. Section 3 presents GREC and an example of its execution is shown in Section 4. Proofs are omitted due to lack of space (see [4]).

2. Glushkov Automata and Regular Expressions

This section explores the transformation of regular expressions into finite state automata. The algorithm for building a finite state automaton $M = (\Sigma, Q, \Delta, q_0, F)$ from a regular expression is straightforward and can be found in the literature [1, 5]. In particular, the algorithm of Glushkov [1] obtains a homogeneous¹ finite state automaton, called Glushkov automaton.

Given a regular expression E , a Glushkov automaton is built by subscribing each alphabet symbol in E 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 $\overline{E} = (a_1(b_2|c_3)^*)^*d_4$.

Given a homogeneous finite state automaton M , we consider the corresponding *Glushkov graph* $G = (X, U)$ where X is the set of vertices (isomorphic to the set of states of the automaton) and U is the set of edges (corresponding to the transition relation Δ). As we are dealing with homogeneous

automata, we drop the superfluous labels on edges and work with an unlabeled directed graph.

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

Given a Glushkov graph $G = (X, U)$, an *orbit* is a set $\mathcal{O} \subseteq X$ such that for all x and x' in \mathcal{O} there exists a non trivial path from x to x' . A *maximal orbit* \mathcal{O} is an orbit such that for each node x of \mathcal{O} and for each node x' not in \mathcal{O} , there does not exist at the same time a path from x to x' and a path from x' to x .

The *input* and *output nodes* of an orbit are respectively defined as follows:

$$In(\mathcal{O}) = \{x \in \mathcal{O} \mid \exists x' \in (X \setminus \mathcal{O}), (x', x) \in U\}$$

and

$$Out(\mathcal{O}) = \{x \in \mathcal{O} \mid \exists x' \in (X \setminus \mathcal{O}), (x, x') \in U\}.$$

An orbit \mathcal{O} is said to be *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 sub-orbit is strongly stable (resp. strongly transverse).

Given a Glushkov graph G , a *graph without orbits* G_{wo} is defined 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.

Example 2.1 Consider the Glushkov automaton from Figure 1(a) that represents² $E = (a(b|c)^*)^*d$. Figure 1(b) shows the corresponding Glushkov graph G which has one maximal orbit: $\mathcal{O}_1 = \{1, 2, 3\}$ (with $In(\mathcal{O}_1) = \{1\}$ and $Out(\mathcal{O}_1) = \{1, 2, 3\}$). Orbit \mathcal{O}_1 is both transverse and stable. We can build a graph without orbit from G as follows: (i) Remove all the arcs $Out(\mathcal{O}_1) \times In(\mathcal{O}_1)$ of G . (ii) The resulting graph G' also has one maximal orbit: $\mathcal{O}_2 = \{2, 3\}$ (with $In(\mathcal{O}_2) = Out(\mathcal{O}_2) = \{2, 3\}$). Delete the arcs $Out(\mathcal{O}_2) \times In(\mathcal{O}_2)$ to obtain a new graph G'' without orbits. Both maximal orbits \mathcal{O}_1 and \mathcal{O}_2 are strongly stable and strongly transverse. \square

Given G_{wo} , we consider the algorithm presented in [1] to obtain a regular expression. G_{wo} is said to be *reducible* if

¹ A finite state automaton is said to be *homogeneous* [1] if one always enters a given state by the same symbol.

² In fact, the Glushkov automaton is built from the subscribed expression $\overline{E} = (a_1(b_2|c_3)^*)^*d_4$.

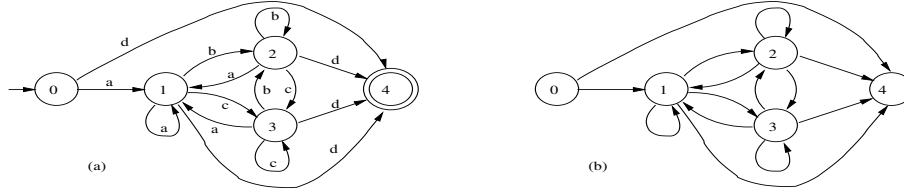


Figure 1. (a) Pictorial representation of a FSA for $(a(b|c)^*)^*d$. (b) Its Glushkov graph.

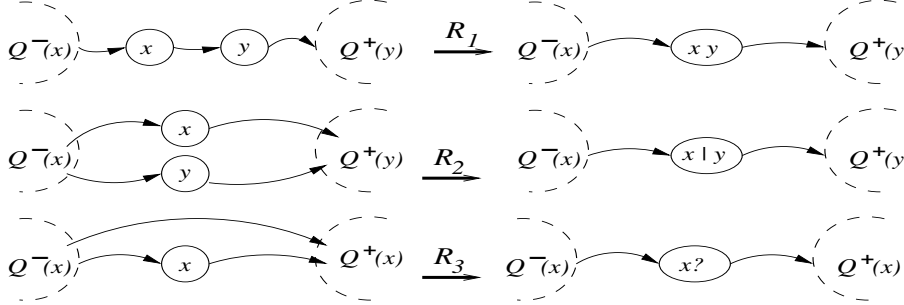


Figure 2. Reduction rules.

it is possible to reduce it to one state by successive applications of any of the three rules \mathbf{R}_1 , \mathbf{R}_2 and \mathbf{R}_3 below (illustrated by Figure 2).

Let x be a node in $G_{wo} = (X, U)$. We note $Q^-(x) = \{y \in X \mid (y, x) \in U\}$ the set of immediate predecessors of x and $Q^+(x) = \{y \in X \mid (x, y) \in U\}$ the set of immediate successors of x . The reduction rules are defined as follows:

Rule \mathbf{R}_1 : If two nodes x and y are such that $Q^-(y) = \{x\}$ and $Q^+(x) = \{y\}$, i.e., node x is the only predecessor of node y and node y is the only successor of x , then concatenate the regular expressions associated to x and y , assign this new regular expression to x , and delete y .

Rule \mathbf{R}_2 : If two nodes x and y are such that $Q^-(x) = Q^-(y)$ and $Q^+(x) = Q^+(y)$, i.e., the nodes x and y have the same predecessors and successors, then build a regular expression that corresponds to the union of those expressions associated to x and y , assign this new regular expression to x , and delete y .

Rule \mathbf{R}_3 : If a node x is such that $y \in Q^-(x) \Rightarrow Q^+(x) \subset Q^+(y)$, i.e., each predecessor of node x is also a predecessor of any successor of node x , then delete the edges going from $Q^-(x)$ to $Q^+(x)$. In this case 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 for $E?$ or E^* .

It is important to remark that the reduction process starts at the lower level of the hierarchy of orbits and works bottom-up, from the smaller orbits to the maximal ones (set inclusion). Indeed, during the construction of G_{wo} ,

the orbits are hierarchically ordered, according to the set-inclusion relation. The information concerning the orbits of the original graph is used to add the transitive closure operator (“+”) to the regular expression being constructed. Thus, during the reduction process when a single node representing a whole orbit is obtained, its content is decorated with a “+”.

Theorem 2.1 [1] $G = (X, U)$ is a Glushkov graph iff the following conditions are satisfied: (1) G is a hammock³, (2) each maximal orbit in G is strongly stable and strongly transverse and (3) the graph without orbit of G is reducible.

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 2.1 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 $\mathcal{D}(E, E') = |\text{card}(S^E) - \text{card}(S^{E'})|$. (Where $\text{card}(S)$ represents the number of elements of the finite set S .)

3. Generation of New Regular Expressions

We are interested in *changes* to a regular expression E representing a language L , provoked by the failure of the

³ We establish the convention that all regular expressions are finished by an end mark #.

automaton M_E associated to E . Neither the changes to the regular expression nor the words that cause them are random ones. The regular expression E has an intuitive meaning for a human user, serving as a *schema* or *type* for some data. The word w' , that triggers the process of changing E , is the result of an update over a word $w \in L$.

We consider two update operations on strings. The operation $Ins(w, \sigma, p) = \alpha\sigma\beta$, where $w = \alpha\beta$ and $|\alpha| = p$, inserts a symbol σ at the position⁴ p of a given word w . The operation $Del(w, p) = \alpha\beta$, where $w = \alpha\sigma\beta$ and $|\alpha| = p$, returns the word obtained by removing the p th symbol from w .

As w' provokes a failure of M_E , we propose changes on E . However, we are interested in changes to the regular expression that are *intuitive* to the user. The aim is not just to add the new word to the language, but to have a richer extension based on the structures of the regular expression and the new word. Thus, 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 updates⁵. Our interest concerns candidates E' whose distance from E such that they are as similar to E as possible. In fact, as the change from the word w to w' consist of the insertion of a new symbol, the number of positions in the regular expressions proposed by our algorithm must be larger than the number of positions in the original one. Indeed, our interest concerns candidates E' whose distance from E is minimal, this why we consider only candidates E' such that $\mathcal{D}(E, E') = 1$.

We can summarize our problem as follows: let E be a regular expression, let w be a word in $L(E)$ and let w' be the word obtained after inserting or deleting a symbol at position p in w . We want to obtain new regular expressions E' such that

$$L(E) \cup \{w'\} \subseteq L(E')$$

and

$$\mathcal{D}(E, E') = 1.$$

The case for deletions is simple. For instance, given a regular expression $E = ab^+c$ and a word $w = abc$, if the symbol b in w is deleted, *i.e.*, $w' = ac$, we build a new regular expression E' by making the symbol b in E optional, *i.e.*, $E' = ab^*c$.

The process above can be generalized as follows: Given the regular expression E , we use the algorithm in [1] to obtain a Glushkov FSA M_E accepting $L(E)$. As M_E is a Glushkov FSA, it has one state for each position of E . Let us s be the state in M_E corresponding to the position that will become optional. In order to obtain a new FSA $M_{E'}$, accepting $L(E')$, we modify M_E by adding new transitions from all the predecessors of s to all the successors of s . It

⁴ Position 0 is the first position of a word.

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

can be verified that the addition of the new transitions preserves the property of the automaton to be a Glushkov FSA. The new regular expression E' is then obtained by reducing $M_{E'}$ using the algorithm given in the previous section.

In the rest of this paper we deal only with insertions. A regular expression for the language obtained after the deletions of a symbol in a word is straightforward to define: it consist in rendering optional the corresponding deleted symbol of the original regular expression.

In this context, we propose an algorithm, called GREC (**Generate Regular Expression Choices**), which is an extension of the reduction method in [1] (see Section 2). GREC computes several regular expressions which are presented to the user who should choose one of them.

Firstly, we consider the execution of the (Glushkov) finite state automaton M_E over the word w' . Let p be the position of w' where the new symbol is inserted. Let the *nearest left state* (s_{nl}) be a state in M_E reached after reading the first $p - 1$ symbols in w' (or in w). Let the *nearest right state* (s_{nr}) be a state in M_E that is the successor of s_{nl} when reading the p -th symbol of w . Notice that we scan w' using M_E and, when w' is not accepted by M_E , the scanning process help us finding where to place the new state (s_{new}) in M_E . In fact, s_{new} is the state that reflects in M_E the insertion operation over w (which gives rise to w'). Notice that, looking for the place of s_{new} in M_E , means looking for states s_{nl} and s_{nr} , which should be passed to GREC. In some cases, a simple backtracking maybe necessary. We remark that both s_{nl} and s_{nr} exist. When M_E is deterministic they are unique, otherwise we can find more than one pair (s_{nl}, s_{nr}). We apply our algorithm to each of these pairs.

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. Our proposal is to deal with these solutions in a hierarchical way, obtaining first those solutions that involve inner starred subexpressions of E . The reduction procedure of [1] is well adapted to this purpose, since it respects the syntactic nesting of the starred subexpressions of the regular expression (it works using the hierarchy of maximal orbits of the graph).

Figure 3 presents a high level algorithm for the procedure GREC. This procedure takes five parameters: a graph without orbits G_1 , a hierarchy of orbits O , two nodes of the graph, corresponding to s_{nl} and s_{nr} , and the new node s_{new} to be inserted.

```

(1) procedure GREC( $G_1, O_1, s_{nl}, s_{nr}, s_{new}$ ) {
(2)   if graph  $G_1$  has only one node
(3)     then stop
(4)   else{
(5)      $R_i :=$  ChooseRule( $G_1, O_1$ );
(6)     for each ( $G_2, O_2$ ):= LookForGraphAlternative( $G_1, O_1, R_i, s_{nl}, s_{nr}, s_{new}$ ) do
(7)       GraphToRegExp( $G_2, O_2$ );
(8)      $G_3 :=$  ApplyRule( $R_i, G_1$ );
(9)     GREC( $G_3, O_1, s_{nl}, s_{nr}, s_{new}$ );
(10)  } }

```

Figure 3. Algorithm to generate regular expression choices from a Glushkov graph.

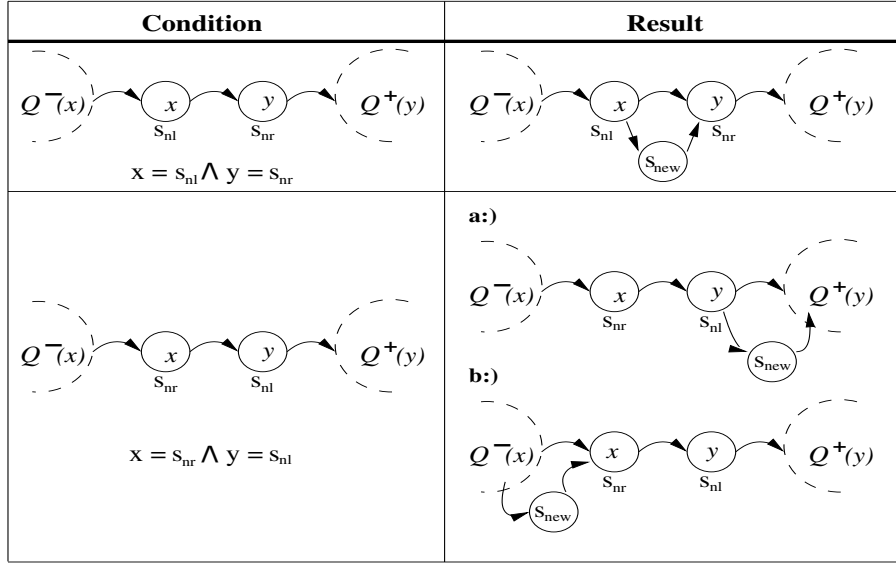


Figure 4. Graph modifications and conditions tested before the application of Rule R_1 .

The procedures `ChooseRule` and `ApplyRule` are implemented according to [1]: `ChooseRule` uses the information concerning orbits to select a rule to be applied in the reduction of the graph, `ApplyRule` builds a new graph resulting from the application of the selected rule.

At each step of the reduction, the iterator `LookForGraphAlternative` checks whether the chosen rule affects nodes s_{nl} or s_{nr} and, in each of these cases, it modifies the graph to take into account the insertion of the new node s_{new} . In the following, we explain how the tests and modifications are performed, according to the type of the rule being considered. Tables presented in Figures 4 to 7 summarize the behavior of `LookForGraphAlternative`. These tables show the tests performed by `LookForGraphAlternative`, as well as the modifications to be performed when the tested conditions are met. In these tables, the column **Condition** shows the situation of the graph being reduced and the conditions to be verified. The column **Result** repre-

sents the new graph (G_2 in the algorithm of Figure 3).

Figure 4 deals with the conditions and modifications when applying R_1 . Notice that, in this figure, the conditions for the first case are: (i) the node x corresponds to s_{nl} and (ii) the node y corresponds to s_{nr} . The modification corresponds to the insertion of s_{new} as an option. For instance, suppose the regular expression $E = ab$, the original word $w = ab$, and the new word $w' = an^*b$. The generated regular expressions in this case are an^*b and $an^?b$. In the second case of Figure 4, s_{nl} is an output of an orbit and s_{nr} belongs to the input of the same orbit (this will be the only situation in which the precondition of the transformation holds⁶). Two different modifications can be proposed in this case. They correspond to the inclusion of the new node in this orbit. Moreover, the inserted node inherits the features of the node to which it is attached, *i.e.*, it

⁶ Notice that as we work on a graph without orbits, edges from the output to the input of the orbits are not represented in Figures 4-7.

Condition	Result
<p>$x = s_{nl} \wedge s_{nr} \in Q^+(x)$</p>	
<p>$x = s_{nr} \wedge s_{nl} \in Q^-(x)$</p>	
<p>$s_{nl} \in Q^-(x) \wedge s_{nr} \in Q^+(x)$</p>	

Figure 5. Graph modifications and conditions tested before the application of Rule R_2 .

Condition	Result
<p>$s_{nl} \in Q^-(x) \wedge s_{nr} \in Q^+(x)$</p> <p>OR</p>	<p>a:)</p>
<p>$s_{nl} \in Q^-(x) \wedge s_{nr} \notin Q^+(x)$</p> <p>OR</p>	<p>b:)</p>
<p>$s_{nl} \notin Q^-(x) \wedge s_{nr} \in Q^+(x)$</p> <p>OR</p>	<p>c:)</p>

Figure 6. Graph modifications and conditions tested before the application of Rule R_3 .

will be an input or an output of the orbit. For instance, suppose the regular expression $E = (ab)^*c$, the original word $w = ababc$ and the new word $w' = abnabc$. The modified graphs (column **Result**) lead us to the regular expressions $(abn!)^*c$ and $(n!ab)^*c$ (from now on we use the symbol $!$ to represent both $?$ and $*$, thus, the expression $(abn!)^*c$ means $(abn?)^*c$ and $(abn^*)^*c$).

Figure 5 deals with the conditions and modifications when applying \mathbf{R}_2 . In the first case $x = s_{nl}$ and $s_{nr} \in Q^+(x)$. The new state s_{new} is introduced as part of the option s_{nl} . For example, if $E = a(b|c)d$, $w = abd$ and $w' = abnd$, then the modified graph leads to the regular expression $a(b\ n!|c)d$. The second case of Figure 5 is symmetric to the first one. The third case introduces a new alternative in the regular expression. For example, if $E = a(b|c)?d$, $w = ad$ and $w' = and$ then the modified graph leads to the regular expression $a(b|c|n!)?d$.

Figure 6 deals with the conditions and modifications when applying \mathbf{R}_3 . The conditions to be verified are: (i) $s_{nl} \in Q^-(x)$ and $s_{nr} \in Q^+(x)$ or (ii) $s_{nl} \in Q^-(x)$ and $s_{nr} \notin Q^+(x)$ or (iii) $s_{nl} \notin Q^-(x)$ and $s_{nr} \in Q^+(x)$. The conditions (ii) and (iii) deal with cases similar to the second condition in Figure 4. Three different solutions are proposed. They consist in inserting the new node before, after and as an alternative of the optional node x . For instance, given $E = ab?c$, $w = ac$ and $w' = anc$ we obtain modified graphs resulting in the regular expressions $an!b?c$, $ab?n!c$ and $a(n!|b?)c$.

During the reduction process, each orbit of the original graph is reduced to just one node containing a regular expression which is then decorated by $+$. Before applying this decoration we should consider the insertion of s_{new} in the reduced orbit. Figure 7 summarizes the situations in which we perform a modification on an orbit. In the first case, we have to test if the node z containing the orbit coincides with s_{nl} and s_{nr} . In this case two solutions are possible. In both of them we insert s_{new} in the orbit represented by z : at the beginning or at the end of it. For example, given $E = a^*$, $w = aa$ and $w' = ana$, we obtain modified graphs resulting in the regular expressions $(n!a)^*$ and $(an!)^*$. The second case tests if $s_{nl} \in Q^-(z)$ and $s_{nr} = z$. Here, two solutions are also proposed. The first one is similar to the solution proposed for the first case. The second one proposes an orbit built over an option between z and s_{new} . For instance, given $E = ab^*$, $w = abb$ and $w' = anbb$, we obtain new regular expressions $E = a(n!b)^*$ and $E = a(n|b)^*$. The third case is symmetric to the second one.

GREC solutions respect the properties stated by the following theorems.

Theorem 3.1 *Let G_1 be the graph without orbits obtained from a Glushkov graph G . Let O_1 be the hierarchy of orbits obtained during the construction of G_1 . Let R_i be one of the reduction rules \mathbf{R}_1 , \mathbf{R}_2 or \mathbf{R}_3 . For any nodes s_{nl} ,*

s_{nr} and s_{new} , each pair (G_2, O_2) resulting from the execution of `LookForGraphAlternative` ($G_1, O_1, R_i, s_{nl}, s_{nr}, s_{new}$) is a representation of a Glushkov graph G' , where G_2 is a graph without orbits, and O_2 is the hierarchy of orbits obtained when constructing G_2 from G' .

Theorem 3.2 *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 the state in M_E reached after reading α and let s_{nr} be the 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 $\{E_1, \dots, E_m\}$. 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 [6]. Notice that if E is unambiguous and n is not in E then each candidate regular expression E_i given by GREC is unambiguous.

We have implemented a prototype of GREC using the ASF+SDF [7] meta-environment under Linux. We have chosen the meta environment for its simplicity and abstractness when implementing formal (set theoretical) concepts.

4. A Detailed Example

In this section we present a detailed example of the execution of GREC, showing how regular expressions are generated. Consider $E = a(bc^+)^*\#$ and $\bar{E} = a_1(b_2c_3^+)^*\#_4$. From now on, we drop the symbols and we work with $\bar{E} = 1(2\ 3^+)^*4$.

Let us suppose that the word $w = abbc\#$ is modified by inserting a new symbol n after the symbol c , yielding the new word⁷ $w' = abbcn\#$. The new symbol will be represented by a new position (“5”) of the regular expressions proposed by GREC.

Let G be a Glushkov graph that represents \bar{E} and G_1 be its corresponding graph without orbits (Figure 8).

We execute GREC over this graph. According to the algorithm of Figure 3, we notice that we reduce the graph (represented by G_1 and G_3) as proposed in [1]. At each step of this reduction, before the application of each rule (\mathbf{R}_1 , \mathbf{R}_2 or \mathbf{R}_3), we construct new graphs, represented by G_2 . The

⁷ Let us enforce the update $Ins(w, n, 4)$.

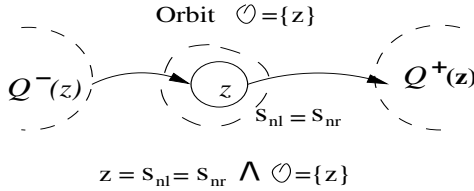
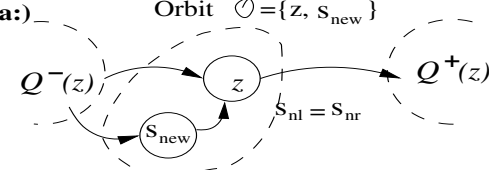
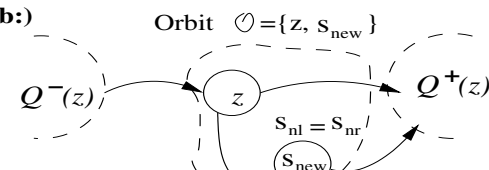
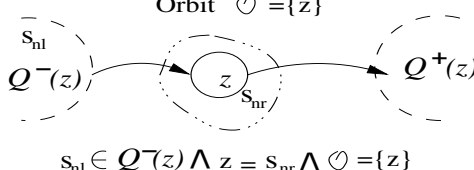
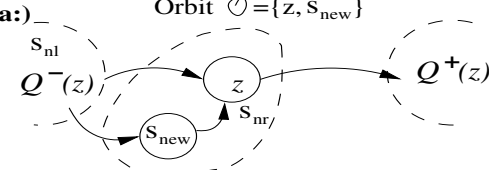
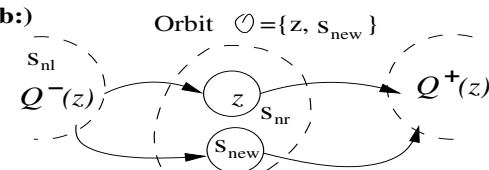
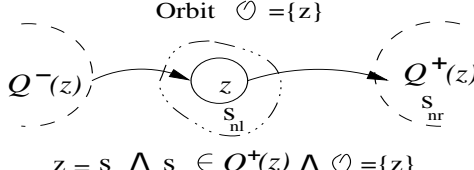
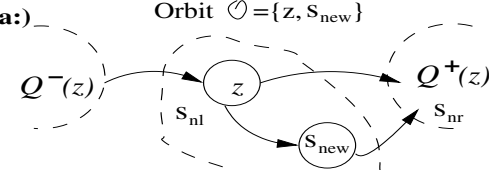
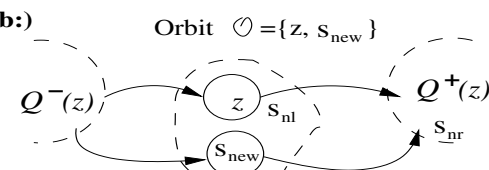
Condition	Result
 <p style="text-align: center;">$z = s_{nl} = s_{nr} \wedge \mathcal{O} = \{z\}$</p>	<p>a) </p> <p>b) </p>
 <p style="text-align: center;">$s_{nl} \in Q^-(z) \wedge z = s_{nr} \wedge \mathcal{O} = \{z\}$</p>	<p>a) </p> <p>b) </p>
 <p style="text-align: center;">$z = s_{nl} \wedge s_{nr} \in Q^+(z) \wedge \mathcal{O} = \{z\}$</p>	<p>a) </p> <p>b) </p>

Figure 7. Graph modifications performed after reducing an orbit.

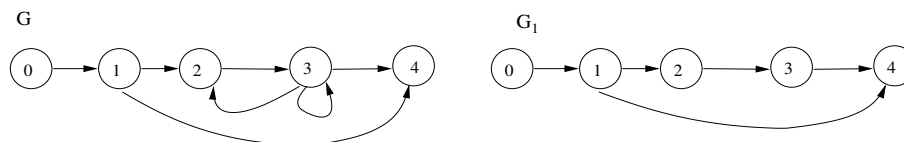


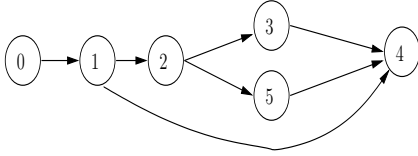
Figure 8. Glushkov graph G for expression $1(23^+)^*4$ and the corresponding $G_{wo}(G_1)$.

new graphs correspond to the cases shown in Figures 4 to 7, adding the new node s_{new} to G_1 .

Firstly, we consider each input parameter of the procedure: G_1 (the graph without orbits of Figure 8), \mathcal{O} (the hierarchy of orbits containing $\mathcal{O}_1 = \{3\}$ and $\mathcal{O}_2 = \{2, 3\}$), s_{nl} (node 3 in G_1), s_{nr} (node 4 in G_1 corresponding to the end mark since we are at the end of the original word) and s_{new} (new node 5 that does not appear in G_1).

Now we run GREC (Figure 3) step by step:

1. As G_1 has more than one node, after the `if` test at line (2), GREC continues at line (4).
2. We choose \mathcal{O}_1 as the orbit to be reduced. No rule is chosen at line (5) since \mathcal{O}_1 is a singleton.
3. As there exists an orbit with one single node, `LookForGraphAlternative` tries to apply the conditions of Figure 7. The third condition of this figure is verified. Applying the first modification (**a:**) of this case, the regular expressions $1(2(3\ 5^?)^+)^*4$ and $1(2(3\ 5^*)^+)^*4$ are obtained. The second modification (**b:**) allows the construction of the graph below:



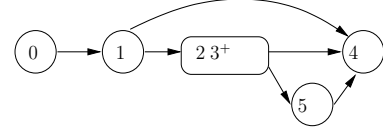
Notice that the modification to the graph is accompanied by the changes to the orbits due to the insertion of the new node. The new orbits are: $\mathcal{O}_1 = \{3, 5\}$, $\mathcal{O}_2 = \{2, 3, 5\}$, $In(\mathcal{O}_1) = \{3, 5\}$, $Out(\mathcal{O}_1) = \{3, 5\}$, $In(\mathcal{O}_2) = \{2\}$ and $Out(\mathcal{O}_2) = \{3, 5\}$.

After getting the new graph, the procedure `GraphToRegExp` is called. This procedure transforms the graph into the regular expression $1(2(3|5)^+)^*4$.

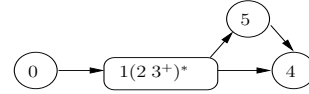
4. Following the reduction process of G_1 , the procedure `ApplyRule` (line (8)) adds the operator “+” to the node that represents an orbit in G_1 , as mentioned in Section 3. Thus, this node corresponds to the regular expression 3^+ .
5. The procedure GREC is then recursively called (line (9)).
6. We choose \mathcal{O}_2 as the orbit to be reduced. Rule \mathbf{R}_1 is the rule chosen at line (5), since it is the only one applicable over the nodes in \mathcal{O}_2 , *i.e.*, nodes 2 and 3.
7. As the application of this rule affects neither s_{nl} nor s_{nr} , `LookForGraphAlternative` does not modify G_1 . `ApplyRule` applies Rule \mathbf{R}_1 on G_1 and we have G_3 with a node with the regular expression $2\ 3^+$.

8. The procedure GREC is then recursively called (line (9)).
9. Rule \mathbf{R}_3 is the chosen rule at line (5).
10. As there exists an orbit with one single node, the procedure `LookForGraphAlternative` builds modified graphs following the explanation given in Figures 6 and 7. We check all conditions of both tables. The only successful condition is the third one of Figure 7.

The first modification (**a:**) of this case allows the construction of the graph presented opposite. For this graph, we obtain $1(2\ 3^+5^*)^*4$ and $1(2\ 3^+5^+)^*4$. The second modification (**b:**) of this case will also be proposed, obtaining the regular expressions $1(2\ 3^+|5)^*4$.



11. The algorithm now continues by reducing the graph of step 7. Before applying rule \mathbf{R}_3 over G_1 , the node that represents one orbit (\mathcal{O}_2) is decorated by “+”. Thus, we have the expression $(2\ 3^+)^+$. By applying the rules \mathbf{R}_3 and \mathbf{R}_1 and we have a node with the regular expression $1(2\ 3^+)^*$.
12. Rule \mathbf{R}_1 is now chosen at line (5), since it is the only one applicable. The node x in Figure 4 corresponds to the node containing $1(2\ 3^+)^*$. The node y is 4.
13. In this context, we can apply the first case of Figure 4, thus obtaining the graph opposite. This graph will produce the regular expressions $1(2\ 3^+)^*5^*4$ and $1(2\ 3^+)^*5^+4$.



14. The algorithm continues reducing the graph of Step 11, until it has just one node. No other solutions are constructed.

The number of candidates built by GREC depends on the number of orbits where the new symbol can be inserted and the matched cases from Figures 4 to 7. We present them to the user classified according to the context of insertion of the new symbol (*i.e.*, according to the starred subexpression in which the new symbol is inserted). It is straightforward to do it since for each maximal orbits in G of E there exists a starred subexpression in E [8]. In our example ($E = a(bc^+)^*\#$ and $w' = abbcn\#$), the proposed solutions are the following:

1. In the same context as c (*i.e.*, inside (c^+)): $a(bc\ n^?)^+)^*$, $a(bc\ n^*)^+)^*$, and $a(b(c|n)^+)^*$.

2. In the same context as b (i.e., inside $(bC)^*$, with $C = (c^+)$): $a(bc^+n?)^*$, $a(bc^+n^*)^*$, and $a(bc^+|n)^*$.

3. In the same context as a (i.e., outside any starred subexpression): $a(bc^+)^*n?$, $a(bc^+)^*n^*$.

Notice that we are not interested in making important changes in the original regular expression to accept the new word. For instance, in this example, we are not interested in options like $a(bc^+)^*n?(bc^+)^*$. Thus, the distance between each above candidate E' and E is 1. For instance, for $E_1 = a(bc^+n^+)^*$, $S^{E_1} = \{1, 2, 3, 4, 5\}$ and, as $S^E = \{1, 2, 3, 4\}$, we have $\mathcal{D}(E, E') = |4 - 5| = 1$.

We finish this section by illustrating the use of GREC in the context of an XML [9] application, for the evolution of schema. Suppose that part of the schema is given by a regular expression similar to $a(bc^+)^*$; for instance, the DTD (Document Type Definition) [10] syntax for regular expressions, we can have: `(Subject, (Year, Article+)*)`. Suppose that n stands for `TechReport`. Now, let `<!ELEMENT Publications (Subject, (Year, Article+)*)>` be the definition of the element `Publications` in a schema given by a DTD. All the previous proposed solutions are given in response of an update performed by an authoritative user, over an XML document that initially conforms to the DTD. They represent different ways of changing the DTD (i.e., of replacing `<!ELEMENT Publications (Subject, (Year, Article+)*)>`).

5. Conclusion

In this paper, we propose a method to perform a conservative generalization of a given regular language. A regular expression E , defining the regular language L , serves as a schema for some data. An update operation, performed over a previously recognized word w , gives rise to a new word w' that may not match E . The failure in recognizing w' triggers the process of creating new regular expressions (based on the old one). We define (and implement) an algorithm that computes several regular expressions E' such that $L(E) \cup \{w'\} \subseteq L(E')$ and $\mathcal{D}(E, E') = 1$. An authoritative user can then choose one of them.

We intend to extend our approach in order to deal with updates of a set of symbols. In this new context, we can assume that given a regular expression E and a set of new words $W = \{w | w \notin L(E)\}$, we want to modify E in order to obtain E' such that $W \subset L(E')$. Note that in this new context, we shall investigate the similarities of our approach and those in the area of regular grammar inference. The regular grammar inference problem is defined as follows [11]:

“Given a finite set of positive examples (sentences belonging to the language of the target grammar) and a finite (possibly empty) set of negative examples (sentences that do not belong to the language of the target grammar), identify a regular grammar G^ that is equivalent to the target grammar G (two grammars G_1 and G_2 are equivalent if their languages are exactly the same).”*

Angluin in [2] has described the use of a minimally adequate teacher to guide the learner in the identification of the target DFA. A minimally adequate teacher is capable of answering membership queries of the form “Does this sentence belong to the target language?” and equivalence queries of the form “Is this DFA equivalent to the target?” Using labeled examples together with membership and equivalence queries it is possible to correctly identify the target DFA.

Our problem is different from the regular grammar inference since we have just one negative example, which is built from just one positive example by inserting or deleting a symbol in w and the built regular grammar G' is a generalization of the known grammar G , such that $L(G')$ includes $L(G)$ as well as a set of words W such that $w' \in W$.

We have applied a simplified version of our algorithm to the schema evolution for XML documents [12]. In this context, regular expressions are unambiguous (the corresponding automata are deterministic). Based on the update of one document, we allow an authoritative user to dynamically change the schema without interfering with other documents in the database. Our system proposes several options for schema evolution. All of them are consistency-preserving and represent extensions of the original schema that intend to foresee the needs of the user. In the case of our XML applications, we have that both the original and the derived regular expressions are unambiguous. This condition simplifies the problem of looking for the place in the Glushkov automaton where the new node will be inserted.

Acknowledgments

Thanks to the anonymous referees for their insightful comments and suggestions. Denio Duarte is supported by CAPES (Brazil) BEX0353/01-9. Part of this work was done while Martin A. Musicante was visiting Université François Rabelais, partly supported by CAPES (Brazil) BEX1851/02-0.

References

- [1] P. Caron and D. Ziadi, “Characterization of Glushkov automata,” *TCS: Theoretical Computer Science*, vol. 233, pp. 75–

90, 2000.

- [2] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.
- [3] R. Parekh and V. Honavar, "Learning DFA from simple examples," *Machine Learning*, vol. 44, no. 1/2, pp. 9–35, 2001.
- [4] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, D. Laurent, and M. Musicante, "Evolving schemas for XML: An incremental approach," Université de François Rabelais, LI, Tech. Rep. (To appear), 2004.
- [5] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory Languages and Computation*, 2nd ed. Addison-Wesley Publishing Company, 2001.
- [6] H. Ahonen, "Disambiguation of SGML content models," in *PODP*, ser. Lecture Notes in Computer Science, vol. 1293. Springer, 1997.
- [7] M. Brand, J. Heering, P. Klint, and P. A. Olivier, "Compiling rewrite systems: The ASF+SDF compiler," *ACM, Transactions on Programming Languages and Systems*, vol. 24, 2002.
- [8] A. Bruggeman-Klein and D. Wood, "Deterministic regular languages," in *STACS*, 1992.
- [9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, *Extensible markup language (XMLTM)*. <http://www.w3.org/XML>, 2000.
- [10] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C. M. Sperberg-McQueen, L. Wood, and J. Clark, "W3C XML specification DTD," Available from <http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm>, Tech. Rep., Feb/1998.
- [11] R. Parekh and V. Honavar, "Automata induction, grammar inference, and language acquisition," in *Handbook of Natural Language Processing*, R. Dale, H. L. Somers, and H. Moisl, Eds. Marcel Dekker, Inc., 2000.
- [12] B. Bouchou, D. Duarte, M. H. F. Alves, D. Laurent, and M. A. Musicante, "Schema evolution for xml: A consistency-preserving approach," in *Mathematical Foundations of Computer Science 2004: 29th International Symposium*, ser. Lecture Notes in Computer Science, no. 3153, Prague, Czech Republic, August 2004, pp. 876 – 888.