Minimal tree language extensions: a keystone of XML type compatibility and evolution[⋆]

Jacques Chabin[1], Mirian Halfeld-Ferrari, Martin A. Musicante[2], and Pierre Réty[1]

[1] Université d'Orléans, LIFO, Orléans, France
{jacques.chabin, mirian, pierre.rety}@univ-orleans.fr
[2] Universidade Federal do Rio Grande do Norte, DIMAp Natal, Brazil
mam@dimap.ufrn.br

Abstract. In this paper, we propose algorithms that extend a given regular tree grammar G_0 to a new grammar G respecting the following two properties: (*i*) G belongs to the sub-class of local or single-type tree grammars and (*ii*) G is the *least grammar* (in the sense of language inclusion) that contains the language of G_0. Our algorithms give rise to important tools in the context of web service composition or XML schema evolution. We are particularly interested in applying them in order to reconcile different XML type messages among services. The algorithms are proven correct and some of their applications are discussed.

1 Introduction

When dealing with web service composition, one should consider the problem of how to reconcile structural differences among types of XML messages supported by different services. A web service designer may wish to implement a service S that is able to accept XML messages coming from different services A, B or C (*i.e.* services offering the same service, in slightly different formats). To allow the composition of S with any of these services, it would be practical to infer a *general* type from the message types of services A, B and C. Moreover, the initial type accepted by S may evolve if one decides to consider also messages coming from a new service D.

The learning of new types (or *schema*) can be very helpful for the harmonious work of the applications that manipulate these data. Some algorithms for learning XML data have been proposed [GGR⁺00,Chi01,BNV07]. In general, these algorithms consist of learning the schema using sets of (positive or negative) examples. Our work considers another situation, since we aim at integrating different schemas in order to implement a service or to adapt it to a new environment: our goal is to maintain the global behavior of a composition while extending the type of the messages being processed.

We are thus interested in automatically generating a new type which is a conservative extension of some given types. Moreover, we are interested in obtaining

⋆ Supported by: Codex ANR-08-DEFIS-04, e-cloudss (LACCIR), CAPES 3164/08-0

the *least* schema (in the sense of type inclusion) that complies with this condition and that can be specified in current XML schema language standards such as DTD or XMLSchema. The following example illustrates schema evolution as proposed by our method.

Example 1. Consider a web service, built for a library consortium, capable of giving information about publications existing in different libraries. This service should be able to accept messages from different library services, each of them specified over its own message format. Let G_A and G_B be the DTDs (Local Tree Grammars) which define the type of messages coming from library services A and B respectively. As usual we represent non terminals by starting with a capital letter and terminals with a small letter. Besides the production rules presented below (C and Z are the start symbols), we suppose that all the production rules $X \rightarrow x[\epsilon]$ such that (respectively) $X \in \{A, T, D, P, N, V, E, W\}$ and $x \in \{author, title, datePublication, price, number, volume, editor, status\}$ are included in the sets of rules of the schema (grammar):

Productions rules of G_A
$C \rightarrow catalog[B^*]$
$B \rightarrow book[A^+.T.D.W]$

Productions rules of G_B
$Z \rightarrow catalog[(Y \mid L)^*]$
$Y \rightarrow book[A^+.T.D.P?.E]$
$L \rightarrow article[A^+.T.D.J.E]$
$J \rightarrow journal[N.V]$

The simple union of these two grammars ($G_0 = G_A \cup G_B$) is not a solution to the problem, since it is *not* a Local Tree Grammar (LTG). This means that it cannot be described as a well-formed DTD. Our algorithm starts processing G_0 and returns the following grammar G_2 as output (by merging some production rules of G_0), where C_Z is the start symbol.

Productions rules of G_2	
$C_Z \rightarrow catalog[(B_Y \mid L)^* \mid B_Y^*]$	$B_Y \rightarrow book[A^+.T.D.W \mid A^+.T.D.P?.E]$
$L \rightarrow article[A^+.T.D.J.E]$	$J \rightarrow journal[N.V]$

Our algorithms are capable of finding a definition for the *least* (local or single-type) tree language (set of XML documents) that contains the XML documents described by both original types. For instance, in Example 1, grammar G_2 is the least LTG that contains the languages generated by G_A and G_B.

The algorithms proposed in this paper are able to transform a (general) regular tree grammar into a Local or Single-Type tree grammar (the user can choose whether the resulting grammar will be a LTG or a STTG). Our algorithms are proven correct for any regular tree grammar in reduced normal form.

The rest of this paper is organized as follows: in Section 2 we recall the theoretical background needed to the introduction of our method; Section 3 presents our schema evolution algorithms for LTG and STTG and Section 4 discusses the implementation of these methods. The paper finishes by considering some related work and by discussing our perspectives of work.

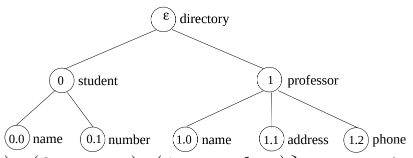
2 Theoretical Background

It is a well known fact that type definitions for XML and regular tree grammars are similar notions and that some schema definition languages can be represented by using specific classes of regular tree grammars. Thus, DTD and XML

Schema, correspond, respectively, to Local Tree Grammars and Single-type Tree Grammars [MLMK05]. Given an XML type T and its corresponding tree grammar G, the set of XML documents described by the type T corresponds to the language (set of trees) generated by G.

In this paper we consider a tree language as a set of *unranked* trees. Tree nodes have a label (from a set Σ) and a position (given as a string of integers). Let U be the set of all finite strings of non-negative integers with the empty string ϵ as the identity. In the following definition we assume that $Pos(t) \subseteq U$ is a nonempty set closed under prefixes (*i.e.* , if $u \preceq v$, $v \in Pos(t)$ implies $u \in Pos(t)$).

Definition 1 (Unranked Σ-valued tree t). A nonempty unranked Σ-valued tree t is a mapping $t : Pos(t) \to \Sigma$ where $Pos(t)$ satisfies: $j \geq 0, uj \in Pos(t), 0 \leq i \leq j \Rightarrow ui \in Pos(t)$. The set $Pos(t)$ is called the set of *positions* of t. We write $t(v) = a$, for $v \in Pos(t)$, to indicate that the Σ-symbol associated to v is a. □
The following figure represents a tree whose alphabet is the set of element names appearing in an XML document.



Given a tree t we denote by $t|_p$ the subtree whose root is at position $p \in Pos(t)$,*i.e.* $Pos(t|_p) = \{s \mid p.s \in Pos(t)\}$ and for each $s \in Pos(t|_p)$ we have $t|_p(s) = t(p.s)$.
For instance, in the figure $t|_0 = \{(\epsilon, student), (0, name), (1, number)\}$, or equivalently, $t|_0 = student(name, number)$.

Given a tree t such that the position $p \in Pos(t)$ and a tree t', we note $t[p \leftarrow t']$ as the tree that results of substituting the subtree of t at position p by t'.

Definition 2 (Sub-tree, Forest). Let L be a set of trees. $ST(L)$ is the set of sub-trees of elements of L, i.e. $ST(L) = \{t \mid \exists u \in L, \exists p \in Pos(u), t = u|_p\}$. A *forest* is a (possibly empty) tuple of trees. For $a \in \Sigma$ and a forest $w = \langle t_1, \ldots, t_n \rangle$, $a(w)$ is the tree defined by $a(w) = a(t_1, \ldots, t_n)$. On the other hand, $w(\epsilon)$ is defined by $w(\epsilon) = \langle t_1(\epsilon), \ldots, t_n(\epsilon) \rangle$, i.e. the tuple of the top symbols of w. □

Definition 3 (Regular Tree Grammar). A *regular tree grammar* (RTG) is a 4-tuple $G = (N, T, S, P)$, where: N is a finite set of *non-terminal symbols*; T is a finite set of *terminal symbols*; S is a set of *start symbols*, where $S \subseteq N$ and P is a finite set of *production rules* of the form $X \to a\,[R]$, where $X \in N$, $a \in T$, and R is a regular expression over N (We say that, for a production rule, X is the left-hand side, $a\,R$ is the right-hand side, and R is the content model.) □

Definition 4 (Derivation). For a RTG $G = (N, T, S, P)$, we say that a tree t built on $N \cup T$ derives (in one step) into t' iff (i) there exists a position p of t such that $t|_p = A \in N$ and a production rule $A \to a\,[R]$ in P, and (ii) $t' = t[p \leftarrow a(w)]$ where $w \in L(R)$ ($L(R)$ is the set of words of non-terminals generated by R). We write $t \to_{[p, A \to a\,[R]]} t'$. More generally, a derivation (in several steps) is a (possibly empty) sequence of one-step derivations. We write $t \to_G^* t'$.
The language $L(G)$ generated by G is the set of trees containing only terminal symbols, defined by : $L(G) = \{t \mid \exists A \in S,\, A \to_G^* t\}$. □

Example 2. Let $G = (N, T, \{X\}, P)$, where $P = \{X \to f\,[A^*.B], A \to a, B \to b\}$. A derivation from the start symbol is $X \to_{[X \to f\,[A^*.B]]} f(A, A, B) \to_G^* f(a, a, b)$. Consequently $f(a, a, b) \in L(G)$. □

To produce grammars that generate least languages, our algorithms need to start from grammars in reduced form and (as in [ML02]) in normal form. A *regular tree grammar* (RTG) is said to be in **reduced form** if (*i*) every non-terminal is reachable from a start symbol, and (*ii*) every non-terminal generates at least one tree containing only terminal symbols. A regular tree grammar (RTG) is said to be in **normal form** if distinct production rules have distinct left-hand-sides.

Example 3. Given the tree grammar $G_0 = (N, T, S, P_0)$, where $N = \{X, A, B\}$, $T = \{f, a, c\}$, $S = \{X\}$, and $P_0 = \{X \to f\,[A.B],\ A \to a,\ B \to a,\ A \to c\}$. Note that G_0 is in reduced form, but it is not in normal form. The conversion of G_0 into the normal form gives the set $P_1 = \{X \to f\,[(A|C).B],\ A \to a,\ B \to a,\ C \to c\}$. Thus $G_1 = (N \cup \{C\}, T, S, P_1)$ is in reduced normal form. □

The following three definitions come from [MLMK05].

Definition 5 (Competing Non-Terminals). Two different non-terminals A and B (of the same grammar G) are said to be *competing with each other* if (*i*) a production rule has A in the left-hand side, (*ii*) another production rule has B in the left-hand side, and (*iii*) these two production rules share the same terminal symbol in the right-hand side. □

Definition 6 (Local Tree Grammar). A *local tree grammar* (LTG) is a regular tree grammar that does not have competing non-terminals. A *local tree language* (LTL) is a language that can be generated by at least one LTG. □

Note that converting a LTG into normal form produces a LTG as well.

Definition 7 (Single-Type Tree Grammar). A *single-type tree grammar* (STTG) is a regular tree grammar in normal form, where (*i*) for each production rule, non terminals in its regular expression do not compete with each other, and (*ii*) starts symbols do not compete with each other. A *single-type tree language* (STTL) is a language that can be generated by at least one STTG. □

In [MLMK05] the expressive power of these classes of languages is discussed. We recall that LTL $\subset$ STTL $\subset$ RTL. Moreover, the LTL and STTL are closed under intersection but not under union; while the RTL are closed under union, intersection and difference.

3 Type Evolution

This section describes our type evolution approach by presenting the main theoretical contributions of our work. The algorithms proposed here take as argument a general regular tree grammar in reduced normal form. They produce a LTG (respectively a STTG) whose language is the least LTL (resp. the least STTL) that contains the language described by the original tree grammar.

The intuitive idea underlying both algorithms is to locate sets of competing non-terminal symbols of the tree grammar, and fix the problem by identifying these non-terminals as the same one.

3.1 Transforming a RTG into a LTG

We consider the problem of obtaining a local tree grammar whose language contains a given tree language. Given a regular tree grammar G_0, we are interested in the definition of the *least* local tree language that contains the language generated by G_0. The new language will be described by a local tree grammar. The algorithm described below obtains a new grammar by transforming G_0. The transformation rules are intuitively simple: every pair of competing non-terminals are transformed into one symbol. We show that this simple transformation of the original grammar yields to a local tree grammar in a finite number of steps.

Now, consider some useful properties of local tree languages. These properties will be used to show the correctness of our grammar-transformation algorithm. Proofs of the properties are omitted here due to the lack of space. They are available in [CHMR09]. The following lemma states that the type of the subtrees of a tree node is determined by the label of its node (i.e. the type of each node is *locally* defined). Recall that $ST(L)$ is the set of sub-trees of elements of L.

Lemma 1. *(see also [PV00, Lemma 2.10]) Let L be a local tree language (LTL). Then, for each $t \in ST(L)$, each $t' \in L$ and each $p' \in Pos(t')$, we have that $t(\epsilon) = t'(p') \implies t'[p' \leftarrow t] \in L$.* □

In the following, we also need a weaker version of the previous lemma:

Corollary 1. *Let L be a local tree language (LTL). Then, for each $t, t' \in ST(L)$, and each $p' \in Pos(t')$, we have that $t(\epsilon) = t'(p') \implies t'[p' \leftarrow t] \in ST(L)$.* □

In practical terms, Corollary 1 gives us a rule of thumb on how to "complete" a regular language in order to obtain a local tree language. For instance, let $L = \{f(a(b), c), f(a(c), b)\}$ be a regular language. According to Corollary 1, we know that L is not LTL and that the least local tree language L' containing L contains all trees where a has c as a child together with all trees where a has b as a child. In other words, $L' = \{f(a(b), c), f(a(c), b), f(a(c), c), f(a(b), b)\}$.

Let us now define our first algorithm for schema (DTD, Local Tree Grammar) evolution. The main intuition behind our algorithm is to merge rules having competing non terminals in their left-hand side. New non terminals are introduced in other to replace competing ones.

Definition 8 (RTG into LTG Transformation). Let $G_0 = (N_0, T_0, S_0, P_0)$ be a regular tree grammar in reduced normal form. We define a new regular tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the following steps:

1. Let $G_2 := G_0$, where G_2 is denoted by (N_2, T_2, S_2, P_2).
2. While there exists a pair of production rules of the form $X_1 \to a\ [R_1]$ and $X_2 \to a\ [R_2]$ in P_2 $(X_1 \neq X_2)$ do:
   (a) Let Y be a new non-terminal symbol and define a substitution $\sigma = [X_1/Y,\ X_2/Y]$.
   (b) Let $G_3 := (N_2 \cup \{Y\} - \{X_1, X_2\},\ T_2,\ \sigma(S_2),\ P_3)$, where $P_3 = \sigma(P_2 \cup \{Y \to a\ [R_1|R_2]\} - \{X_1 \to a\ [R_1], X_2 \to a\ [R_2]\})$.
   (c) Let $G_2 := G_3$, where G_2 is denoted by (N_2, T_2, S_2, P_2).
3. Return G_2. □

Example 4. Consider Example 1 where grammar G_0 is the input for the algorithm of Definition 8. In a first step, rules $Z \to catalog[(Y \mid L)^*]$ and $C \to catalog[B^*]$ are replaced by $C_Z \to catalog[(Y \mid L)^* \mid B^*]$. Following the same idea, rules $B \to book[A^+.T.D.W]$ and $Y \to book[A^+.T.D.P?.E]$ are replaced by $B_Y \to book[A^+.T.D.W \mid A^+.T.D.P?.E]$. This change implies changes on the right-hand side of other rules, consequently $C_Z \to catalog[(Y \mid L)^* \mid B^*]$ is changed into $C_Z \to catalog[(B_Y \mid L)^* \mid B_Y^*]$. In this way we obtain the grammar G_2 as shown in Example 1. □

It can be shown that our algorithm stops, generating a local tree grammar in normal form (see [CHMR09]). The main result of this section is stated below.

Theorem 1. *The grammar returned by the algorithm of Definition 8 generates the least local tree language that contains $L(G_0)$.* □

Example 5. Let the RTG G_0 be the input of the algorithm of Definition 8 and G be the resulting LTG:

G_0:	G:
$S \to a[A.A]$	$Y \to a[Y.Y \mid B]$
$A \to a[B]$	$B \to b[\epsilon]$
$B \to b[\epsilon]$	

Notice that $L(G_0)$ contains just the tree $t = a(a(b), a(b))$ and that $t \in L(G)$. □

3.2 Transforming a RTG into a STTG

Given a regular tree grammar G_0, we are interested in the definition of the *least* single-type tree language that contains the language generated by G_0. The new language will be described by a single-type grammar. The algorithm described below obtains a new grammar by transforming G_0. Roughly speaking, for each production rule $A \to a\,[R]$ in G_0, an equivalence relation is defined on the non-terminals of R, so that all competing non-terminals of R are in the same equivalence class. These equivalence classes form the non-terminals of the new grammar. Let $G_0 = (N_0, T_0, S_0, P_0)$ be a RTG in reduced normal form.

Definition 9 (Grouping competing non-terminals). Let $\parallel$ be the relation on N_0 defined by: for all $A, B \in N_0$, $A \parallel B$ iff $A = B$ or A and B are competing in P_0. For any $\chi \in \mathcal{P}(N_0)$, let $\parallel_\chi$ be the restriction of $\parallel$ to the set χ ($\parallel_\chi$ is defined only for elements of χ).

Lemma 2. *Since G_0 is in normal form, $\parallel_\chi$ is an equivalence relation for any $\chi \in \mathcal{P}(N_0)$.* □

Some notation used in this section:

– $N(R)$ denotes the set of non-terminals occurring in a regular expression R.
– For any $\chi \in \mathcal{P}(N_0)$ and any $A \in \chi$, $\hat{A}^\chi$ denotes the equivalence class of A w.r.t. relation $\parallel_\chi$. In other words, $\hat{A}^\chi$ contains A and the non-terminals of χ that are competing with A in P_0.
– $\sigma_{N(R)}$ is the substitution defined over $N(R)$ by $\forall A \in N(R)$, $\sigma_{N(R)}(A) = \hat{A}^{N(R)}$. By extension, $\sigma_{N(R)}(R)$ is the regular expression obtained from R by replacing each non-terminal A in R by $\sigma_{N(R)}(A)$.

Definition 10 (RTG into STTG Transformation). Let $G_0 = (N_0, T_0, S_0, P_0)$ be a regular tree grammar in reduced normal form. We define a new regular tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the steps:

1. Let $G = (\mathcal{P}(N_0), T_0, S, P)$ where:
   - $S = \{\hat{A}^{S_0} \mid A \in S_0\}$,
   - $P = \{\, \{A_1, \ldots, A_n\} \rightarrow a\,[\sigma_{N(R)}(R)] \mid A_1 \rightarrow a[R_1], \ldots, A_n \rightarrow a[R_n] \in P_0,$
     $R = (R_1| \cdots |R_n)\}$,

   where $\{A_1, \ldots, A_n\}$ denotes *each* possible set of competing non-terminals.
2. Remove all unreachable non-terminals and rules in G, then return it. $\square$

Our generation of STTG from RTG is based on grouping competing non-terminals into equivalence classes. In the new grammar, each non-terminal is formed by a set of non-terminals of N_0. When competing non-terminals which appear in the same regular expression R in G_0 are identified, the sets that contain them form non-terminal symbols. The production rules having these new symbols as left-hand side are obtained from those rules containing the competing symbols in G_0. Although this amounts to an exponential number of non-terminal, we have notice that, in practice, this explosion is not common (notice that unreachable rules are removed at step 2). The algorithm version presented in Definition 10 eases our proofs. An optimized version, where just the needed non-terminals are generated, is given in Section 4.

Example 6. Consider a non-STTG grammar G_0 having the following set P_0 of productions rules (*Image* is the start symbol):

$Image \rightarrow image[Frame1 \mid Frame2 \mid Background.Foreground]$
$Frame1 \rightarrow frame[Frame1.Frame1 \mid \epsilon]$
$Frame2 \rightarrow frame[Frame2.Frame2.Frame2 \mid \epsilon]$
$Background \rightarrow back[Frame1]$ and $Foreground \rightarrow fore[Frame2]$.

Grammar G_0 defines different ways of decomposing an image: recursively into two or three frames or by describing the background and the foreground separately. Moreover, the background (resp. the foreground) is described by binary decompositions (resp. ternary decompositions). In other words, the language of G_0 contains the union of the trees: *image(bin(frame))*; *image(ter(frame))* and *image (back (bin (frame)), fore (ter (frame)))* where *bin* (resp. *ter*) denotes the set of all binary (resp. ternary) trees that contains only the symbol *frame*.

The algorithm returns G, which contains the rules below (the start symbol is $\{Image\}$) :

$\{Image\} \rightarrow image[\{Frame1, Frame2\} \mid \{Frame1, Frame2\}$
$\quad \mid \{Background\}.\{Foreground\}]$
$\{Background\} \rightarrow back[\{Frame1\}]$
$\{Foreground\} \rightarrow fore[\{Frame2\}]$
$\{Frame1, Frame2\} \rightarrow frame[\epsilon \mid \{Frame1, Frame2\}.\{Frame1, Frame2\} \mid \epsilon$
$\quad \mid \{Frame1, Frame2\}.\{Frame1, Frame2\}.\{Frame1, Frame2\}]$
$\{Frame1\} \rightarrow frame[\{Frame1\}.\{Frame1\} \mid \epsilon]$
$\{Frame2\} \rightarrow frame[\{Frame2\}.\{Frame2\}.\{Frame2\} \mid \epsilon]$

Note that some regular expressions could be simplified. G is a STTG that generates the union of *image(tree(frame))* and *image (back (bin (frame)), fore (ter (frame)))* where *tree* denotes the set of all trees that contain only the symbol *frame* and such that each node has 0 or 2 or 3 children. Let $L_G(X)$ denote the language obtained by deriving in G the non-terminal X. Actually, $L_G(\{Frame1, Frame2\})$ is the least STTL that contains $L_{G_0}(Frame1) \cup L_{G_0}(Frame2)$. □

Theorem 2. *The grammar returned by the algorithm of Definition 10 generates the least STTL that contains $L(G_0)$.* □

The rest of this sub-section is a proof sketch of the previous theorem. The notations are those of Definition 10. The proof somehow looks like the proof concerning the transformation of a RTG into a LTG (see [CHMR09] for details). However it is more complicated since in a STTL (and unlike what happens in a LTL), the confusion between $t|_p = a(w)$ and $t'|_{p'} = a(w')$ should be done only if position p in t has been generated by the same production rule as position p' in t', i.e. the symbols occurring in t and t' along the paths going from root to p (resp. p' in t') are the same. This is why, in Definition 11, we introduce notation $path(t, p)$ to denote these symbols. First, we enunciate some properties.

Lemma 3. *Let $\chi \in \mathcal{P}(N_0)$ and $A, B \in \chi$ ($A \neq B$). Then $\hat{A}^\chi$ and $\hat{B}^\chi$ are not competing in P.* □

Example 7. Given the grammar of Example 6, let $\chi = \{\mathit{Frame1}, \mathit{Frame2}, \mathit{Background}\}$. The equivalence classes induced by $\|_\chi$ are $\widehat{\mathit{Frame1}}^\chi = \widehat{\mathit{Frame2}}^\chi = \{\mathit{Frame1}, \mathit{Frame2}\}$; $\widehat{\mathit{Background}}^\chi = \{\mathit{Background}\}$; which are non-competing non-terminals in P.

Lemma 4. *G is a STTG.* □

The next lemma establishes the basis for proving that the language generated by G contains the language generated by G_0. It considers the derivation process over G_0 at any step (supposing that this step is represented by a derivation tree t) and proves that, in this case, at the same derivation step over G, we can obtain a tree t' having all the following properties: (i) the set of positions is the same for both trees ($Pos(t) = Pos(t')$); (ii) positions associated to terminal are identical in both trees; (iii) if position p is associated to a non-terminal A in t then position $p \in Pos(t')$ is associated to the equivalence class $\hat{A}^\chi$ for some $\chi \in \mathcal{P}(N_0)$ such that $A \in \chi$.

Lemma 5. *Let $Y \in S_0$. If G_0 derives:*
$t_0 = Y \rightarrow \cdots \rightarrow t_{n-1} \rightarrow_{[p_n,\, A_n \rightarrow a_n[R_n]]} t_n$ *then G can derive:* $t'_0 = \hat{Y}^{S_0} \rightarrow \cdots \rightarrow t'_{n-1} \rightarrow_{[p_n,\, \hat{A_n}^{\chi_n} \rightarrow a_n[\sigma_{N(R_n|\cdots)}(R_n|\cdots)]]} t'_n$ *s.t.* $\forall i \in \{0, \ldots, n\}, Pos(t'_i) = Pos(t_i) \wedge \forall p \in Pos(t_i)$*:* $(t_i(p) \in T_0 \implies t'_i(p) = t_i(p)) \wedge$

$$(t_i(p) = A \in N_0 \implies \exists \chi \in \mathcal{P}(N_0),\, A \in \chi \wedge t'_i(p) = \hat{A}^\chi)$$

Proof. The proof is by induction in the length of the derivation process.
For $n = 0$, the property holds because $t_0(\epsilon) = Y$ and $t'_0(\epsilon) = \hat{Y}^\chi$ with $\chi = S_0$ and $Y \in \chi$. Induction step: Assume the property for $n - 1 \in \mathbb{N}$. By hypothesis $t_{n-1} \rightarrow_{[p_n,\, A_n \rightarrow a_n[R_n]]} t_n$, then $t_{n-1}(p_n) = A_n \in N_0$. By ind. hyp., $t'_{n-1}(p_n) = \hat{A_n}^{\chi_n}$ for some $\chi_n \in \mathcal{P}(N_0)$, and $A_n \in \chi_n$.
By construction of P, $\hat{A_n}^{\chi_n} \rightarrow a_n[\sigma_{N(R_n|\cdots)}\ (R_n|\cdots)] \in P$.
Thus $t'_{n-1} \rightarrow_{[p_n,\, \hat{A_n}^{\chi_n} \rightarrow a_n[\sigma_{N(R_n|\cdots)}(R_n|\cdots)]]} t'_n = t'_{n-1}[p_n \leftarrow a_n[\sigma_{N(R_n|\cdots)}(w)]]$ whereas $t_n = t_{n-1}[p_n \leftarrow a_n(w)]$ and $w \in L(R_n)$. Consequently $t'_n(p_n) = a_n = t_n(p_n)$ and $\forall i \in \mathbb{N}$, $t_n(p_n.i) = B \in N(R_n) \subseteq N(R_n|\cdots) \wedge t'_n(p_n.i) = \hat{B}^{N(R_n|\cdots)}$.

Example 8. Given the grammar of Example 6, consider trees t, t' and t'' in Figure 1 obtained after three steps in the derivation process: t is a derivation tree for G_0 while t' and t'' are for G. Tree t' is the one that corresponds to t according to Lemma 5. Notice that t'' is a tree that can also be derived from G, but it is not in $L(G_0)$ (indeed, since $Pos(t) \neq Pos(t'')$, tree t'' does not have the properties required in Lemma 5). □

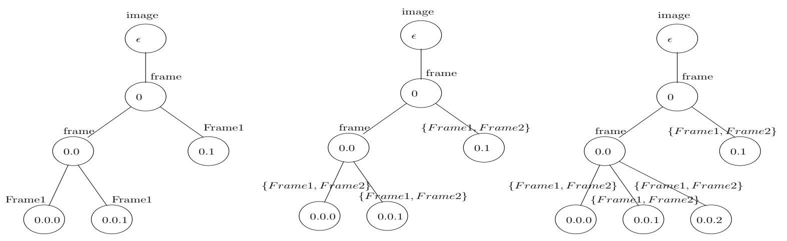


Fig. 1. Derivation trees t, t' and t'' .

The following corollary proves that the language of the new grammar G, proposed by Definition 10, contains the original language of G_0.

Corollary 2. *$L(G_0) \subseteq L(G)$.* □

In the rest of this section we work on proving that $L(G)$ is the least STTL that contains $L(G_0)$. To prove this property, we first need to prove some properties over STTLs. We start by considering paths in a tree. We are interested by paths starting on the root and achieving a given position p in a tree t. Paths are defined as a sequence of labels. For example, $path(a(b, c(d)), 1) = a.c$.

Definition 11 (Path in a tree t to a position p). Let t be a tree and $p \in Pos(t)$. We define $path(t, p)$ as being the word of symbols occurring in t along the branch going from the root to position p. Formally, $path(t, p)$ is recursively defined by : $path(t, \epsilon) = t(\epsilon)$ and $path(t, p.i) = path(t, p).t(p.i)$ where $i \in \mathbb{N}$. □

Given a STTG G, let us consider the derivation process of two trees t and t' belonging to $L(G)$. The following lemma proves that positions (p in t and p' in t') having identical paths are derived by using the same rules. A consequence of this lemma (when $t' = t$ and $p' = p$) is the well known result about the unicity in the way of deriving a given tree with a STTG [ML02].

Lemma 6. *Let G' be a STTG, let $t, t' \in L(G')$.*
Let $X \rightarrow^*_{[p_i, rule_{p_i}]} t$ be a derivation of t and $X' \rightarrow^*_{[p'_i, rule'_{p'_i}]} t'$ be a derivation of t' by G' (X, X' are start symbols). Then $\forall p \in Pos(t)$, $\forall p' \in Pos(t')$,
$(path(t, p) = path(t', p') \implies rule_p = rule'_{p'})$

In a STTL, it is possible to exchange sub-trees that have the same paths.

Lemma 7. *(also in [MNSB06, Prop 6.3 and 6.5])*
Let G' be a STTG. $\forall t, t' \in L(G')$, $\forall p \in Pos(t)$, $\forall p' \in Pos(t')$, $(path(t, p) = path(t', p') \implies t'[p' \leftarrow t|_p] \in L(G'))$

Example 9. Let G be the grammar of Example 6. Consider a tree t as shown in Figure 2. Exchanging subtrees $t|_{0.0}$ and $t|_{0.1}$ gives us a new tree t''. Both t and t'' are in $L(G)$. $\square$

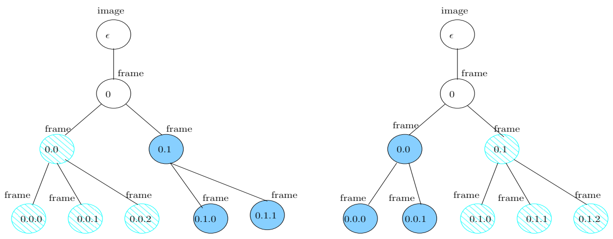


Fig. 2. Trees t and t'' with sub-tree exchange.

The following lemma expresses what the algorithm of Definition 10 does. Given a forest $w = (t_1, \dots, t_n)$, recall that $w(\epsilon) = \langle t_1(\epsilon), \dots, t_n(\epsilon)\rangle$, i.e. $w(\epsilon)$ is the tuple of the top symbols of w.

Lemma 8. $\forall t \in L(G), \forall p \in Pos(t),\ t|_p = a(w) \implies \exists t' \in L(G_0), \exists p' \in pos(t'), t'|_{p'} = a(w') \wedge w'(\epsilon) = w(\epsilon) \wedge path(t', p') = path(t, p)$. $\square$

Example 10. Let G be the grammar of Example 6 and t the tree of Figure 2. Let $p = 0$. Using the notations of Lemma 8, $t|_0 = frame(w)$ where
$w = \langle frame(frame, frame, frame),\ frame(frame, frame)\rangle$. We have $t \notin L(G_0)$. Let $t' = image(frame(frame(frame, frame),\ frame)) \in L(G_0)$ and (with $p' = p = 0$) $t'|_{p'} = frame(w')$ where $w' = \langle frame(frame, frame),\ frame\rangle$. Thus $w'(\epsilon) = w(\epsilon)$. Note that others $t' \in L(G_0)$ suit as well. $\square$

We end this section by proving that the grammar obtained by our algorithm generates the least STTL which contains $L(G_0)$.

Lemma 9. *Let L' be a STTL s.t. $L(G_0) \subseteq L'$. Let $t \in L(G)$. Then, $\forall p \in Pos(t), \exists t' \in L', \exists p' \in pos(t'),\ (t'|_{p'} = t|_p \wedge path(t', p') = path(t, p))$.* $\square$

Proof. We define the relation $\sqsupset$ over $Pos(t)$ by $p \sqsupset q \iff \exists i \in \mathbb{N}, p.i = q$. Since $Pos(t)$ is finite, $\sqsupset$ is noetherian. The proof is by noetherian induction on $\sqsupset$. Let $p \in pos(t)$. Let us write $t|_p = a(w)$. From Lemma 8, we know that:
$\exists t' \in L(G_0), \exists p' \in pos(t'), t'|_{p'} = a(w') \wedge w'(\epsilon) = w(\epsilon) \wedge path(t', p') = path(t, p)$. Thus, $t|_p = a(a_1(w_1), \dots, a_n(w_n))$ and $t'|_{p'} = a(a_1(w'_1), \dots, a_n(w'_n))$. Now let $p \sqsupset p.1$. By induction hypothesis:
$\exists t'_1 \in L', \exists p'_1 \in pos(t'_1), t'_1|_{p'_1} = t|_{p.1} = a_1(w_1) \wedge path(t'_1, p'_1) = path(t, p.1)$. Notice that $t'_1 \in L'$, $t' \in L(G_0) \subseteq L'$, and L' is a STTL. Moreover $path(t'_1, p'_1) = path(t, p.1) = path(t, p).a_1 = path(t', p').a_1 = path(t', p'.1)$.
As $path(t'_1, p'_1) = path(t', p'.1)$, from Lemma 7 applied on t'_1 and t', we get $t'[p'.1 \leftarrow t'_1|_{p'_1}] \in L'$. However $(t'[p'.1 \leftarrow t'_1|_{p'_1}])|_{p'} = a(a_1(w_1), a_2(w'_2), \dots, a_n(w'_n))$ and
$path(t'[p'.1 \leftarrow t'_1|_{p'_1}], p') = path(t', p') = path(t, p)$.
By applying the same reasoning for positions $p.2, \dots, p.n$, we get a tree $t'' \in L'$ s.t. $t''|_{p'} = t|_p$ and $path(t'', p') = path(t, p)$.

Corollary 3. *(when $p = \epsilon$, and then $p' = \epsilon$) Let L' be a STTL s.t. $L' \supseteq L(G_0)$. Then $L(G) \subseteq L'$.* $\square$

4 Implementation

A prototype tool implementing our algorithms can be downloaded from [CHMR]. It is developed using the ASF+SDF Meta-Environment [vdBHdJ+01] and it is about 1000 lines of code.

The algorithm of Definitions 8 is implemented in a straightforward way. However, Definition 12 below gives an improved version of the algorithm of Definition 10. This new version avoids to generate unreachable non-terminals, and is suited for direct implementation. Indeed, it keeps a set of unprocessed non-terminals (denoted by U) which are accessible from the start symbol. We just compute those non-terminal symbols which are accessible from the start symbols of the grammar. More precisely, in Definition 12, we start by computing the equivalence classes of the start symbols of G_0 and we insert them to the set U, containing those non-terminals which are not yet processed. At each iteration of the while loop, an element of U is chosen as the new non-terminal for which a new production rule is going to be created. This non-terminal is added to the set N. At each step, the set U is updated by adding to it those non-terminals appearing on the right-hand side of the new production rule, filtering the non-terminals already processed.

Definition 12 (RTG into STTG Transformation). Let $G_0 = (N_0, T_0, S_0, P_0)$ be a (general) regular tree grammar. We define a new single-type tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the following steps:

```
1. Let $S := \{\hat{A}^{S_0} \mid A \in S_0\}$; $G := (N := \emptyset, T := T_0, S, P := \emptyset)$; $U := S$;
2. While $U \neq \emptyset$ do:
   (a) Choose $\{A_1, \ldots, A_n\} \in U$;
   (b) Let $U := U - \{\{A_1, \ldots, A_n\}\}$; $N := N \cup \{\{A_1, \ldots, A_n\}\}$;
   (c) Let $P := P \cup \{\, \{A_1, \ldots, A_n\} \to a\, \sigma_{N(R)}(R)$
                 $\mid A_1 \to a[R_1], \ldots, A_n \to a[R_n] \in P_0,\ R = (R_1 | \cdots | R_n)\}$;
   (d) Let $U := (U \cup \{\hat{A}^{N(R)} | A \in N(R)\}) - N$;
   End While
3. Return $G$.                                                              □
```

It is straightforward to see that the algorithm of Definition 12 generates the same STTG as that of Definition 10. In the worst case, the number of non-terminals of the STTG returned by both algorithms is exponential in the number of the non-terminals of the initial grammar G_0. However, in many examples, most non-terminals generated by step 1 of Definition 10 are unreachable, and thus are not generated by the implemented algorithm.

The example below represents a usual situation, in which the schemas of two different digital libraries (Grammars G_1 and G_2) are joined into one schema. *Library* and *Lib* are the start symbols. For lack of space, we do not depict the production rules $X \to x[\epsilon]$ such that $X \in$ {Author, Title, ISBN, Publisher, Date, ISSN, Editor, Year, Pages, Dimensions, Scale} and $x \in$ {author, title, isbn, publisher, date, issn, editor, year, pages, dim, scale} respectively. We apply the algorithm on $G_0 = G_1 \cup G_2$. Note that Author, Title, ISBN, Publisher, Date appear both in G_1 and G_2. It is not necessary to rename them before computing the union, since each of them generates only one terminal. G_0 contains 18 rules.

Productions rules of G_1	
Library	→ lib [Book*]
Book	→ book [Author.Title. ISBN.Publisher.Date]

Productions rules of G_2	
Lib	→ lib [(Mag | Record | Book2 | Map)*]
Mag	→ mag [Title.ISSN.Editor.Publisher.Date]
Record	→ rec [Title.Author.Date]
Book2	→ book [ISBN.Title.Author.Publisher.Year.Pages]
Map	→ map [Editor.Dimensions.Scale.Year]

The resulting grammar, after applying the algorithm, is[3]:

Resulting production rules	
Library	→ lib [(Mag | Record | Book | Map)* | Book*]
Book	→ book [ISBN.Title.Author.Publisher.Year.Pages | Author.Title.ISBN.Publisher.Date]
Mag	→ mag [Title.ISSN.Editor.Publisher.Date]
Record	→ rec [Title.Author.Date]
Map	→ map [Editor.Dimensions.Scale.Year]

Notice that the new grammar has only 16 rules (included those with empty regular expressions on their right-hand side). Only 16 new non-terminals were created by our algorithm, that is two less than those from the original grammar. This shows that in a typical situation, our algorithm runs in acceptable time, even if the worst case is exponential.

5 Related Work

As discussed in [Flo05], traditional tools require the data schema to be developed prior to the creation of the data. Unfortunately, in several modern applications the schema often changes as the information grows and different people have inherently different ways of modeling the same information. Complete elimination of the schema does not seem to be a solution since it assigns meaning to the data and thus helps automatic data search, comparison and processing. To find a balance, [Flo05] considers that we need to find how to automatically map schemas and vocabulary to each other and how to rewrite code written for a certain schema into code written for another schema describing the same domain.

Most existing work in the area of XML schema evolution concerns the second proposed solution. For instance, in [GMR05] the idea is to keep track of the updates made to the schema and to identify the portions of the schema that require validation. Our approach aims to be included in the first proposed solution of [Flo05] since it allows the conservative evolution of schemas. Indeed, our method extends the work in [BDH$^+$04,dHM07] which considers the conservative evolution of LTG by proposing the evolution of regular expressions. Contrary to this, the present paper proposes schema evolution in a global perspective, dealing with the tree grammars as a whole. We also consider the evolution of STTG.

Our proposal is inspired in some grammar inference methods (such as those in [BM03,BM06] which deal with ranked tree languages) that return a tree grammar or a tree automaton from a set of positive examples (see [Ang92,Sak97] for surveys). Our method deals with unranked trees, starts from a given RTG

[3] For more readability, the non-terminals have been renamed. *Library* is the start symbol. We have chosen not to simplify the regular expressions, showing them as produced by our algorithm. Our implementation do not implement any simplification yet.

G_0 (representing a set of positive examples) and finds the least LTG or STTG that contains $L(G_0)$. As we consider an initial tree grammar we are not exactly inserted in the learning domain, but their methods inspire us and give us tools to solve our problem, namely, the evolution of a original schema (and not the extraction of a new schema).

Several papers (such as in [GGR$^+$00,Chi01,BNST06,BNV07]) deal with XML schema inference. In [BNST06] DTD inference consists in an inference of regular expressions from positive examples. As the seminal result from Gold [Gol67] shows that the class of all regular expressions cannot be learnt from positive examples, [BNST06] identifies classes of regular expressions that can be efficiently learnt. Their basic method consists in inferring a single occurrence automaton called SOA from a finite set of strings and to transform it to a SORE (regular expressions in which every element name can occur at most once). Their method is extended to deal with XMLSchema (XSD) in [BNV07].

In [AGM09], given a target global type of a distributed XML document, the authors propose a method to provide a subtype for each marked subtree such that (*i*) if each subtree verifies its subtype, the global type is verified and (*ii*) no extra restrictions than those imposed by the global type are introduced. Their approach consists in regarding the problem locally (each node and its children) and to find the FSA which should be associated to the children generated by external sources. Their approach can be seen as the inverse or ours. Let us suppose our library consortium example, with a big distributed XML document where nodes marked by functions are calls to different libraries. Their approach focus on defining the subtypes corresponding to each library supposing that a design is given. Our approach proposes to find the integration of different library subtypes by finding the least library type capable of verifying all library subtypes.

In [MNSB06, Th 10.3], it is shown that deciding whether a regular tree grammar has an equivalent LTG, or an equivalent STTG, is EXPTIME-complete. Using our algorithms, we can also solve these decision problems by computing the LTG (in linear time), or the STTG (in exponential time), that generates the least local (or single-type) language L containing the initial language L_0, and checking (in exponential time) that $L = L_0$.

6 Conclusion

This paper proposes algorithms that compute a *minimal* tree language, by finding the local or single-type grammar which generates it and which extends a given original regular grammar. The paper proves the correctness and the minimality of the generated grammars. A prototype has been implemented in order to show the feasibility of our approach. Our goal is to allow a given type to evolve encompassing the needs of the application using it. Indeed, we aim at developing tools for adapting XML message type of a web service to the needs of a composition.

It is encouraging to us to note that our work complements that of [MNSB06], where the complexity of deciding whether a regular tree grammar has an equivalent LTG or STTG is provided. In that work, the authors are interested in analyzing the actual expressive power of XSD. With some non-trivial amount of work, part of their theorem proofs can be used to produce an algorithm similar to ours. This emphasizes the relevance of our method whose usability is twofold: as a theoretical tool, it can help answering the decision problem announced in [MNSB06]; as an applied tool, it can easily be adapted to the context of digital libraries, web services, etc. Our work complements the proposals in [BDFM09,dHM07], since we consider not only DTD but also XSD, and adopts a global approach where all the tree grammar is taken into account as a whole.

Some aspects of our tool can be improved, in particular the conciseness of the regular expressions appearing in the generated grammars. We are working on improving and extending our approach to solve other questions related to type compatibility and evolution. Indeed, in this context, many other aspects may be taken into account such as integrity constraints (how they evolve when the schema evolves) and semantics of elements (how to deal with identical concepts named differently in each type). We intend not only to extend our work in these new directions but also to build an applied tool capable of comparing types or extracting some relevant parts of a type. Interesting theoretical problems are related to these applications.

References

[AGM09] Serge Abiteboul, Georg Gottlob, and Marco Manna. Distributed xml design. In *PODS '09: Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 247–258. ACM, 2009.

[Ang92] Dana Angluin. Computational learning theory: survey and selected bibliography. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 351–369, New York, NY, USA, 1992. ACM.

[BDFM09] Batrice Bouchou, Denio Duarte, Mirian Halfeld Ferrari, and Martin A. Musicante. Extending XML types using updates. In Dr. Hung, editor, *Services and Business Computing Solutions with XML: Applications for Quality Management and Best Processes*, pages 1–21. IGI Global, 2009.

[BDH$^+$04] B. Bouchou, D. Duarte, M. Halfeld Ferrari, D. Laurent, and M. A. Musicante. Schema evolution for XML: A consistency-preserving approach. In *Mathematical Foundations of Computer Science (MFCS)*, number 3153 in Lecture Notes in Computer Science, pages 876–888. Springer-Verlag, August 2004.

[BM03] Jérôme Besombes and Jean-Yves Marion. Apprentissage des langages réguliers d'arbres et applications. *Traitement automatique de langues*, 44(1):121–153, Jul 2003.

[BM06] Jérôme Besombes and Jean-Yves Marion. Learning tree languages from positive examples and membership queries. *Theoretical Computer Science*, 2006.

[BNST06] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In *VLDB*, pages 115–126, 2006.

[BNV07] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring xml schema definitions from xml data. In *VLDB*, pages 998–1009, 2007.

[Chi01] B. Chidloviskii. Schema extraction from XMLS data: A grammatical inference approach. 2001.

[CHMR] Jacques Chabin, Mirian Halfeld Ferrari, Martin A. Musicante, and Pierre Réty. A software to transform a RTG into a LTG or a STTG. `http://www.univ-orleans.fr/lifo/Members/rety/logiciels/RTGalgorithms.html`.

[CHMR09] Jacques Chabin, Mirian Halfeld Ferrari, Martin A. Musicante, and Pierre Réty. Minimal extensions of tree languages: Application to XML schema evolution. Technical Report RR-2009-06. `http://www.univ-orleans.fr/lifo/Members/rety/RR-2009-06.pdf`, LIFO, 2009.

[dHM07] Robson da Luz, Mrian Halfeld Ferrari, and Martin A. Musicante. Regular expression transformations to extend regular languages (with application to a datalog XML schema validator). *Journal of Algorithms (Special Issue)*, 62(3-4):148–167, 2007.

[Flo05] Daniela Florescu. Managing semi-structured data. *ACM Queue*, 3(8):18–24, 2005.

[GGR$^+$00] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Xtract: A system for extracting document type descriptors from xml documents. In *SIGMOD Conference*, pages 165–176, 2000.

[GMR05] Giovanna Guerrini, Marco Mesiti, and Daniele Rossi. Impact of XML schema evolution on valid documents. In *WIDM'05: Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 39–44, New York, NY, USA, 2005. ACM Press.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[ML02] Murali Mani and Dongwon Lee. Xml to relational conversion using theory of regular tree grammars. In *In VLDB Workshop on EEXTT*, pages 81–103. Springer, 2002.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, 2005.

[MNSB06] Win Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of xml schema. *ACM Trans. Database Syst.*, 31(3):770–813, 2006.

[PV00] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *PODS - Symposium on Principles of Database System*, pages 35–46. ACM Press, 2000.

[Sak97] Yasubumi Sakakibara. Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185(1):15–45, 1997.

[vdBHdJ$^+$01] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The asf+sdf meta-environment: a component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2), 2001.