LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS

UNIVERSITE D'ORLEANS

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Weak Inclusion for XML Types (full version)

Joshua Amavi, Jacques Chabin,
Mirian Halfeld Ferrari, Pierre Réty
LIFO, Université d'Orléans

Rapport n^o **RR-2011-07**

Weak Inclusion for XML Types* (full version)

Joshua Amavi Jacques Chabin Mirian Halfeld Ferrari Pierre Réty

LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France
E-mail: {joshua.amavi, jacques.chabin, mirian, pierre.rety}@univ-orleans.fr

Abstract. Considering that the *unranked* tree languages $L(G)$ and $L(G')$ are those defined by given *non-recursive XML types* G and G', this paper proposes a *simple and intuitive* method to verify whether $L(G)$ is "approximatively" included in $L(G')$. Our approximative criterion consists in weakening the father-children relationships. Experimental results are discussed, showing the efficiency of our method in many situations.

1 Introduction

Today, XML is the *lingua franca* for data exchange on the web. To allow interoperability among systems, one usually needs to obtain partial information from another system file. In the context of tree-modeled data, this operation corresponds to the retrieval of sub-trees according to some given application requests. This retrieval may be approximative, trying to find the XML document that best fit some given constraints. The situation is more complex when the problem consists in comparing (or retrieving) XML types (or schemas) defining approximate sub-trees of the trees generated by a given XML type.

Example 1. Suppose an application where we want to replace an XML type G by a new type G' (*eg.*, a web service composition where a service replaces another, each of them being associated to its own XML message type). We want to analyse whether the XML messages supported by G' contains (in an approximate way) those supported by G. XML types are regular tree grammars where we just consider the structural part of the XML documents, disregarding data attached to leaves. Thus, to define leaves we consider rules of the form $A \to a[\epsilon]$.
Now let us suppose that both of our grammars contain the following rules: F $\to$ firstName[ϵ], L $\to$ lastName[ϵ] , T $\to$ title[ϵ], Y $\to$ year[ϵ] and C $\to$ conference[ϵ]. However, G defines a publication by using the following rule PUB $\to$ publication[$(F.L)^+.T.Y.C$]; while in G' the definition is done by the set of rules: PUB $\to$ publication[$A^*.P$]; A $\to$ authors[$F.L$] and P $\to$ paper[$T.Y.C$]. We want to know whether messages valid with respect to G can be accepted (in an approximate way) by G'. Notice that G accepts trees such as t in Figure 1 that are not valid with respect to schema G' but that represent the same kind of information G' deals with. Indeed, in G', the same information would be organised as the tree t' in Figure 1. □

* A short version has been published at CIAA 2011. Partially supported by: Codex ANR-08-DEFIS-04.

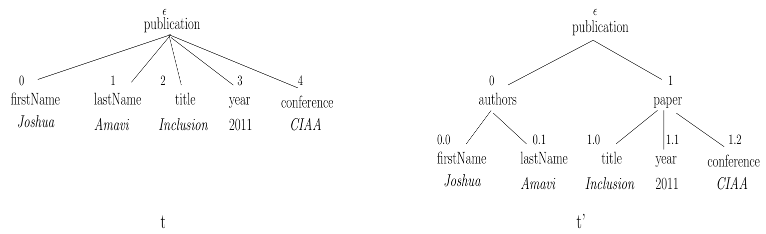


Fig. 1. Examples of trees t and t' valid with respect to G and G', respectively.

The approximative criterion for comparing trees that is commonly used consists in weakening the father-children relationships (*i.e.*, they are implicitly reflected in the data tree as only ancestor-descendant). In this paper, we consider this criterion in the context of tree languages. We denote this relation *weak inclusion* to avoid confusion with the *inclusion* of languages (*i.e.*, the inclusion of a set of trees in another one).

Given two types G and G', we call $L(G)$ and $L(G')$ the set of XML documents valid with respect to G and G', respectively. Our paper proposes a method for deciding whether $L(G)$ is weakly included in $L(G')$, in order to know if the substitution of G by G' can be envisaged. The unranked-tree language $L(G)$ is weakly included in $L(G')$ if for each tree $t \in L(G)$ there is a tree $t' \in L(G')$ such that t is weakly included in t'. Intuitively, t is weakly included in t' (denoted $t \lhd t'$) if we can obtain t by removing nodes from t' (a removed node is replaced by its children, if any). For instance, in Figure 1, t can be obtained by the removal of the nodes *authors* and *paper* from t'.

To decide whether $L(G)$ is weakly included in $L(G')$, we consider the set of trees $WI(L(G')) = \{t \mid \exists t' \in L(G'),\ t \lhd t'\}$. Note that $L(G)$ is weakly included in $L(G')$ iff $L(G) \subseteq WI(L(G'))$.

Assuming that $L(G')$ is bounded in depth (which holds for most XML types), we propose a direct and simple approach that deals with unranked trees, using hedge grammars. The intuition of our method is to change types by allowing the deletion of XML tree levels. Roughly speaking, according to this new type, a given node in an XML tree can have as children those imposed by the original XML type or any of its descendants. With this simple idea we can compute a grammar capable of generating all the weakly included trees of a original non-recursive type G'. We prove that our algorithm is correct and complete.

Example 2. Let us consider G' from Example 1. We start from this tree grammar and use our algorithm to obtain a tree grammar which generates the language containing all the trees weakly-included in $L(G')$. The obtained grammar is:

PUB $\to$ publication$[(A \mid ((F|\epsilon).(L|\epsilon)))^*.\,(P|((T|\epsilon).(Y|\epsilon).(C|\epsilon)))]$
A $\to$ authors$[(F|\epsilon).(L|\epsilon)]$ P $\to$ paper$[(T|\epsilon).(Y|\epsilon).(C|\epsilon)]$
F $\to$ firstName$[\epsilon]$ L $\to$ lastName$[\epsilon]$
T $\to$ title$[\epsilon]$ Y $\to$ year$[\epsilon]$
C $\to$ conference$[\epsilon]$.

Given this new grammar G'' we can verify that $L(G)$ is included in $L(G'')$. □

However, if $L(G')$ is not bounded in depth, computing $WI(L(G'))$ may be difficult as illustrated by the following example.

Example 3. Let G'_1 be a grammar containing the rule $A \to a[B.(A|\epsilon).C]$ where non-terminals B and C generate leaves b and c respectively. In this simple case, it is easy to imagine an extension of our basic algorithm for computing $WI(G'_1)$. This new grammar replaces the first rule by $A \to a[B^*.(A|\epsilon).C^*]$. However, one can take G'_2 with a more complex rule such as $A \to a[B.(A \mid A.A \mid \epsilon).C]$. The solution here should be given by replacing this rule by $A \to a[(A|B|C)^*.(A|\epsilon).(A|B|C)^*]$. Notice, for instance, that in $WI(L(G'_2))$ we can have trees where nodes a, b or c appear on the left of a node labelled a while according to G'_2 this was not possible. We can remark that the method needed to obtain $WI(G'_2)$ is more sophisticated than the one used for $WI(G'_1)$. The situation becomes worse if we suppose G'_3 similar to G'_2 except for the rule concerning B, which is now $B \to b[B|\epsilon]$. In this case, we should guarantee that in $WI(G'_3)$ nodes labelled b will have at most one child. Thus, in $WI(G'_3)$, the rule $B \to b[B|\epsilon]$ stays unchanged. This represents another special case to be treated. □

It seems difficult to define a general and simple algorithm for treating all the recursive cases. To obtain simple methods we believe that different classes of recursivity should be considered. A generic approach may need sophisticated tools.

In this paper, given *non-recursive regular tree grammars*[1] G and G', to check if $L(G)$ is weakly included in $L(G')$, we proceed according to the following steps:

1. Starting from G', we compute a grammar $WI(G')$ that generates $WI(L(G'))$.
2. Then we check whether $L(G) \subseteq WI(L(G'))$, i.e. the inclusion of regular tree languages. The runtime of this step is exponential in the worst case [Sei90]. However, if G' satisfies some deterministic-like restrictions, we show that so does $WI(G')$ and thus the runtime of this step becomes polynomial [MNS04,CGLN08].

Paper organisation: Section 2 gives some theoretical background. Section 3 presents how to compute $WI(G)$ for a given non-recursive grammar G, while Section 4 analyses some experimental results of our method. Section 5 considers the special case of deterministic DTDs. Missing proofs are in the appendix.

Related work: Several works deal with the (weak) tree inclusion problem in the context of ordered trees: different improvements (e.g. [BG05,CSC06,RT97]) have been presented to the initial proposal in [KM95]. Our proposal differs from these approaches because it considers the weak inclusion with respect to *tree languages* (and not with respect to trees only). Given a pattern query, to select

[1] Notice that although Example 2 deals with local tree grammars (DTDs), our algorithm can be applied to any non-recursive regular tree grammar.

the answers, [GKM09] proposes a polynomial algorithm which verifies whether a sub-tree belongs to the language defined by the pattern and by: (i) weakening the father-children relationship and (ii) disregarding the ordering of children. Contrary to us, they do not compare XML types, and, thus, are not concerned by horizontal constraints in general. Testing precise inclusion of XML types is considered in [CGLN08,CGPS09,CGS09,MNS04]. In [MNS04], the authors study the complexity of the inclusion, identifying tractable cases. In [CGLN08] we find a new polynomial algorithm for checking whether $L(A) \subseteq L(D)$, where A is an automaton for unranked trees and D is a *deterministic* DTD.

2 Preliminaries

An XML document is an unranked tree, defined in the usual way as a mapping t from a set of positions $Pos(t)$ to an alphabet Σ. Thus for $v \in Pos(t)$, $t(v)$ is the label of t at the position v, and $t|_v$ denotes the sub-tree of t at position v. Positions are sequences of integers in $\mathbb{N}^*$ and the set $Pos(t)$ satisfies: $j \geq 0, u.j \in Pos(t), 0 \leq i \leq j \Rightarrow u.i \in Pos(t)$. As usual, ϵ denotes the empty sequence of integers, i.e. the root position. In the following definition, let t, t' be unranked trees. The char "." denotes the concatenation of sequences of integers. Figure 1 illustrates trees with positions and labels: we have, for instance, $t(1) = lastName$ and $t'(1) = paper$. The sub-tree $t'|_0$ is the one whose root is *authors*.

Definition 1. Relationships on a tree: Let $p, q \in Pos(t)$. Position p is an *ancestor* of q (denoted $p < q$) if there is a non-empty sequence of integers r such that $q = p.r$. Position p is *to the left* of q (denoted $p \prec q$) if there are sequences of integers u, v, w, and $i, j \in \mathbb{N}$ such that $p = u.i.v$, $q = u.j.w$, and $i < j$. □

Definition 2. Resulting tree after node deletion: For a tree t' and a non-empty position q of t', let us note $Rem_q(t') = t$ the tree obtained after the removal of the node at position q in t' (a removed node is replaced by its children, if any). We have:

1. $t(\epsilon) = t'(\epsilon)$,
2. $\forall p \in Pos(t')$ such that $p < q$: $t(p) = t'(p)$,
3. $\forall p \in Pos(t')$ such that $p \prec q : t|_p = t'|_p$,
4. Let $q.0, q.1..., q.n \in Pos(t')$ be the positions of the children of position q, if q has no child, let $n = -1$. Now suppose $q = s.k$ where $s \in \mathbb{N}^*$ and $k \in \mathbb{N}$. We have:
   - $t|_{s.(k+n+i)} = t'|_{s.(k+i)}$ for all i such that $i > 0$ and $s.(k+i) \in Pos(t')$ (the siblings located to the right of q shift),
   - $t|_{s.(k+i)} = t'|_{s.k.i}$ for all i such that $0 \leq i \leq n$ (the children go up). □

Definition 3. Weak inclusion for unranked trees: The tree t is *weakly included in* t' (denoted $t \lhd t'$) if there exists a series of positions $q_1 \dots q_n$ such that $t = Rem_{q_n}(\cdots Rem_{q_1}(t'))$. □

Example 4. In Figure 1, we have tree $t \lhd t'$. Notice that for each node of t, there is a node in t' with the same label, and this mapping preserves vertical order and left-right order. However a tree t_1 such as $publication(lastName, firstName)$ is not weakly included in t' since the left-right order is not preserved. □

Definition 4. Regular Tree Grammar: A *regular tree grammar* (RTG) (also called hedge grammar) is a 4-tuple $G = (NT, T, S, P)$, where: NT is a finite set of *non-terminal symbols*; T is a finite set of *terminal symbols*; S is a set of *start symbols*, where $S \subseteq NT$ and P is a finite set of *production rules* of the form $X \to a\,[R]$, where $X \in NT$, $a \in T$, and R is a regular expression over NT. We recall that the set of regular expressions over $NT = \{A_1, \ldots, A_n\}$ is inductively defined by: $R ::= \epsilon \mid A_i \mid R|R \mid R.R \mid R^+ \mid R^* \mid R^? \mid (R)$ □

Definition 5. Derivation: For an RTG $G = (NT, T, S, P)$, we say that a tree t built on $NT \cup T$ derives (in one step) into t' iff (i) there exists a position p of t such that $t|_p = A \in NT$ 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 $t \to_{[p, A \to a\,[R]]} t'$. A derivation (in several steps) is a (possibly empty) sequence of one-step derivations. We write $t \to_G^* t'$. Let $Tree_T$ be the set of all trees that contain only terminal symbols. The language $L(G)$ generated by G is defined by : $L(G) = \{t \in Tree_T \mid \exists A \in S,\ A \to_G^* t\}$. □

Remark 1. As usual, in this paper, *we only consider regular tree grammars such that*: (A) every non-terminal generates at least one tree containing only terminal symbols and (B) distinct production rules have distinct left-hand-sides (*i.e.*, tree grammars in the normal form [ML02]). □

Remark 2. Given an RTG $G = (NT, T, S, P)$, for each $A \in NT$, there exists in P a unique rule of the form $A \to a[E]$, i.e. whose left-hand-side is A. □

Example 5. Grammar $G_0 = (NT, T, S, P_0)$, where $NT = \{X, A, B\}$, $T = \{f, a, c\}$, $S = \{X\}$, and $P_0 = \{X \to f\,[A.B],\ A \to a[\epsilon],\ B \to a[\epsilon],\ A \to c[\epsilon]\}$ does not respect the conditions stated in this paper since it is not in the normal form. The conversion of G_0 into normal form gives the set $P_1 = \{X \to f\,[(A|C).B],\ A \to a[\epsilon],\ B \to a[\epsilon],\ C \to c[\epsilon]\}$.

Among regular tree grammars we are particularly interested in local tree grammars which have the same expressive power as DTDs[2]. We recall their definition from [MLMK05]:

Definition 6. Local Tree Grammar: A *local tree grammar* (LTG) is a regular tree grammar that does not have competing non-terminals. Two non-terminals A and B (of the same grammar G) are said to be *competing with each other* if $A \neq B$ and G contains production rules of the form $A \to a[E]$ and $B \to a[E']$ (i.e. A and B generate the same terminal symbol). A *local tree language* (LTL) is a language that can be generated by at least one LTG. □

[2] Note that converting an LTG into normal form produces an LTG as well.

To finish this section we recall some definitions and results concerning the regular expressions that will be important for us in Section 5.

Firstly we recall that, as W3C standard, only 1-unambiguous regular expressions are allowed in DTDs. A regular expression is 1-unambiguous if every symbol in any input string can be uniquely matched to one occurrence of the symbol in the regular expression, without looking ahead in the string. As an example, consider the regular expression $E = (A|B)^*.A.A^*$. and the word $w = BAA$ in $L(E)$. The word w can be parsed in two different ways: (i) the first and the second A in w match the first and the second A in E, respectively; (ii) the first and the second A in w match the second and the third A in E, respectively. The regular expression E is therefore *not* 1-unambiguous. We refer to [BW98] for a formal definition of this concept. It is also known that a regular expression E is 1-unambiguous if and only if its corresponding Glushkov automaton is deterministic [BW98,CZ00,ZPC97].

Definition 7. Monadic and strict regular expression: A regular expression E is *monadic* if each non-terminal of E occurs only once in E. It is *strict* if it does not contain operators + (positive closure) nor ? (optional). A grammar is monadic (resp. strict) if all its regular expressions are monadic (resp. strict). □

The following lemma is an immediate consequence of the previous notions.

Lemma 1. *A monadic regular expression is 1-unambiguous. Consequently, a strict and monadic LTG is deterministic*[3]*.* □

It may happen that algorithm for testing tree language inclusion (second step of our proposal) are built by considering strict regular expressions only. In this case, recall that it is always possible to make a regular expression strict, by replacing each $E^?$ by $E|\epsilon$ and each E^+ by $E.E^*$. Unfortunately, removing operator + does not preserve monadicity. However if $\epsilon \in L(E)$ then $L(E^+) = L(E^*)$ and in this case we can just replace each + by $*$, which preserves monadicity.

3 Weak Inclusion for Regular Tree Grammars

Given a non-recursive regular tree grammar G, in this section we present how to generate a grammar G_1 such that $L(G_1) = WI(L(G))$. To do that, we introduce some definitions and results.

Definition 8. Relation $\rightsquigarrow_G$ over non-terminals: Let $G = (NT, T, S, P)$ be an RTG and A, B be non-terminals. We write $A \rightsquigarrow_G B$ if there exists a rule $A \to a[E]$ in G s.t. $B \in NT(E)$ (where $NT(E)$ denotes the set of non-terminals occurring in E). We say that $A_0, \ldots, A_n$ ($A_i \in NT$) is a chain for $\rightsquigarrow_G$ if $A_0 \rightsquigarrow_G \cdots \rightsquigarrow_G A_n$. The relation $\rightsquigarrow_G$ is *noetherian* if $\rightsquigarrow_G$ does *not* have an infinite chain $A_0 \rightsquigarrow_G \cdots \rightsquigarrow_G A_n \rightsquigarrow_G \cdots$. Grammar G is *recursive* if there exists a non-terminal A s.t. $A \rightsquigarrow_G^+ A$ (where $\rightsquigarrow_G^+$ is the transitive closure of $\rightsquigarrow_G$). □

[3] An LTG or DTD is deterministic if all its regular expressions are 1-unambiguous [BW98].

Lemma 2. *If G is non-recursive then $\rightsquigarrow_G$ is noetherian.* $\square$

To compute $WI(G)$, the idea is: for each non-terminal A that generates terminal a, either we generate a, or a is not generated and we generate its children instead. First, we extend $\rightsquigarrow_G$ to regular expressions. Moreover, to each non-terminal A, we associate a new non-terminal denoted $A^\sharp$ (called *marked non-terminal*).

Definition 9. Relation $\rightsquigarrow_G$ over regular expressions: Let G be a grammar and E be a regular expression appearing in one of its production rules. Suppose that A is a non-terminal appearing at some position in E and that there is a rule $A \to a[E'']$ in G. Let E' be the regular expression defined by $E' = E[A \leftarrow A^\sharp|E'']$ (i.e. this occurrence of A is replaced by $A^\sharp|E''$]). Then we say that $E \rightsquigarrow_G E'$. $\square$

Lemma 3. *If G is non-recursive then $\rightsquigarrow_G$ (over reg. exp.) is noetherian.* $\square$

Definition 10. Substitutions in the context of $\rightsquigarrow_G$: Let G be a grammar. We define a substitution σ over non-terminals as follows. Due to the assumptions, for each non-terminal A there exists in G a unique rule whose left-hand-side is A, say $A \to a[E]$. Then $\sigma(A) = A^\sharp|E$. We extend σ to regular expressions: if E contains at least one non-marked non-terminal, $\sigma(E)$ is the regular expression obtained by replacing each non-marked non-terminal A in E by $\sigma(A)$. Otherwise $\sigma(E)$ is not defined. Note that $E \rightsquigarrow_G^+ \sigma(E)$ (where $\rightsquigarrow_G^+$ is the transitive closure of $\rightsquigarrow_G$). $\square$

Example 6. In grammar G' of Example 1, let us consider the rule $PUB \to$ *publication*$[A^*.P]$. Let $E = A^*.P$ be its regular expression. Then, according to Definition 10, we have $\sigma(E) = (A^\sharp \mid (F.L))^*.(P^\sharp \mid T.Y.C)$. $\square$

In the following definition we present an algorithm to produce grammar $WI(G)$ for a given grammar G. By σ^n we denote n successive applications of σ, i.e. $\sigma^n = \sigma \circ \cdots \circ \sigma$ (n times).

Definition 11. Algorithm for computing $WI(G)$: Let G be a non-recursive grammar. As $\rightsquigarrow_G$ and $\rightsquigarrow_G^+$ are noetherian, for any regular expression E, there exists $n \in \mathbb{N}$ s.t. $\sigma^n(E)$ is defined and $\sigma^{n+1}(E)$ is not, which means that $\sigma^n(E)$ contains only marked non-terminals. We define $E\!\uparrow = \sigma^n(E)$. The grammar $G\!\uparrow$ is the one obtained from G by replacing each regular expression E in G by $E\!\uparrow$. $\square$

Example 2 shows the resulting grammar after applying Definition 11. Notice that the marks inserted by our algorithm are just to follow substitutions already done. The resulting grammar is one where every non terminal is marked, *i.e.*, all substitutions have been applied. We can then rewrite the grammar as usual, disregarding the marks used during the algorithm processing. This is why, when talking about $WI(G)$ we do not consider the marks anymore.

Theorem 1. *Given a non-recursive grammar G, we have $L(G\!\uparrow) = WI(L(G))$ (with common roots).* $\square$

4 Experimental Results

Given a grammar G', the computation of $WI(G')$ (Definition 11) considers each non-terminal of each production rule. Our implementation avoids repeating computation (which may lead to an exponential blow-up in the worst case) by computing each $A\uparrow$ only once. Thus, supposing that G' has n non-terminals (and thus n production rules), the computation of $WI(G')$ can be seen as the traversal of a graph having n nodes and $n \times l$ edges (where l is the max. length of reg. exp.). Notice that $n \times l$ equals the *number of non-terminal occurrences*, denoted by $|G'|$, the size of G'. Thus, the complexity of our algorithm is $O(n + |G'|)$.

Our prototype is implemented in Java and our experiments are done on an Intel Dual Core T2390 with 1.86GHz and 2GB of memory. The first phase of our tests concerns the generation of $WI(G')$. Results shown in Figure 2 correspond to 400 synthetic DTDs whose size ranges from 50 to 10000 non-terminal (NT) occurrences. These experiments concern DTDs with simple regular expressions composed by the concatenation of $A_1 \ldots A_n$; where we vary the number n of non-terminals, allowing as maximal value $n = 9$. Notice that our algorithm does not exceed $100ms$ for DTDs having less than 10000 NT-occurrences. We have also considered 10 real DTDs having about 50 NT-occurrences. The execution time was approximately $10ms$.

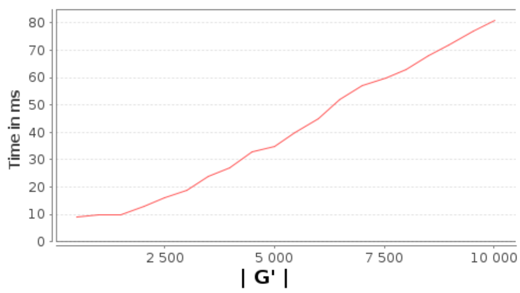


Fig. 2. Runtime for computing $WI(G')$ for grammar G'.

We have run a hundred complete tests and Table 1 shows the results for 21 of them. Here we have considered more complex DTDs with $\star$, $+$, $?$, $|$ and imbrications. In this case, most regular expressions are of the form $E = E_1.E_2.E_3$ where each E_i is a disjunction involving one or more Kleene or positive closure. The DTDs are deterministic or non-deterministic. When a DTD is non-deterministic, some E_i of E are of the form $(A_j.A_{j+1})|(A_j.A_{j+2})$ or $(A_j|(A_{j+3}|A_{j+4}))^+.(A_{j+2}|(A_{j+3}|A_{j+4}))^*$. Results on lines 1 to 9 concern synthetic non-deterministic DTDs, while those on lines 10 to 18 correspond to synthetic deterministic DTDs. On lines 19 to 21 we deal with deterministic real DTDs.

The second phase of our tests analyses the performance of the other steps of our method. Given a grammar G, to decide whether $L(G) \subseteq L(WI(G'))$, we have implemented the algorithm presented in [BHH+08]. Although the complexity of this method is exponential, the authors show that it allows very important performance improvement. Table 1 summarizes our results. Notice that, as the algorithm in [BHH+08] is proposed for ranked trees, to apply this method, we convert $WI(G')$ and G into binary grammars $bin(WI(G'))$ and $bin(G)$, respectively. This conversion gives us grammars having more rules than their unranked counterpart. Given a grammar G, the production rules of $bin(G)$ are generated by considering each regular expression of each rule in G. The number of rules also depends on the format of the regular expressions (*eg.*, the presence of the Kleene closure). For $WI(G')$ this augmentation can be very important since in this grammar regular expressions are more complex than those in G'.

	Unranked grammars					**Ranked grammars**		**Runtime**		**Result**
	$\lvert G\rvert$	$\lvert G'\rvert$	$\lvert WI(G')\rvert$	$\#Rules$ G	$\#Rules$ G'	$\#Rules$ $bin(G)$	$\#Rules$ $bin(WI(G'))$	Phase1 (s)	Phase2 (s)	T/F
1	32	52	123	25	40	113	5622	0	73	T
2	37	68	167	29	50	82	6420	0	139	T
3	42	98	233	33	77	93	19107	0	350	F
4	98	68	167	77	50	314	6420	0	354	F
5	86	98	233	65	77	249	19107	0	918	F
6	19	98	233	14	77	72	19017	0	14	F
7	42	86	222	33	65	93	22762	0	1455	T
8	52	98	233	43	77	168	19107	0	1890	T
9	68	86	222	50	65	200	22762	0	1729	F
10	10	62	125	9	53	30	5728	0	2	T
11	33	62	125	28	53	96	5728	0	61	T
12	42	78	183	34	62	174	7483	0	278	F
13	62	96	249	53	78	166	21808	0	522	F
14	47	96	249	40	78	210	21808	0	90	F
15	42	96	249	34	78	174	21808	0	110	F
16	20	90	224	18	74	22	11299	0	8	F
17	27	96	249	24	78	148	21808	0	18	F
18	48	96	249	40	78	167	21808	0	3217	T
19	31	31	86	25	25	35	3625	0	114	T
20	32	32	68	14	14	190	2254	0	36	T
21	32	31	86	14	25	190	3625	0	1	F

Table 1. Runtime in seconds for Phase1 (computing $WI(G')$) and Phase2 (converting unranked grammars $WI(G')$ and G to their binary counterpart and testing if $L(bin(G)) \subseteq L(bin(WI(G')))$. Result is the boolean value for the inclusion test.

As expected, the first phase is much more faster than the second. In order to have tractable tests in Phase 2, we have chosen small examples having thus insignificant (0s) time for Phase 1 (see also Figure 2). In general, the execution time of Phase 2 is higher when the inclusion is true. However, when languages are very similar, Phase 2 can take a lot of time even for non-included languages (as in line 5, 9). On the contrary, for very different languages the inclusion test is very fast (as in lines 6, 16, 17 and 21). It is interesting to consider the case on line 18 which takes about 2-times longer than for any other examples. Notice that we have DTD with more than 90 non-terminal occurrences, and a positive result for the inclusion test. Indeed, DTD G corresponds to a subset of the rules of DTD G'. To achieve some improvement on Phase 2, we may envisage to apply techniques presented in [MNS04] to find regular expressions for which inclusion verification is tractable or to restrict ourselves to the use of deterministic DTDs which allow us to use a polynomial time algorithm for testing language inclusion. The latter option (that we intend to implement) is discussed in the following section.

5 The Special Case of Deterministic DTDs

We finally discuss a restricted situation where the weak inclusion between XML types can be computed in polynomial time. We first define $Succ(A)$ as the set of non-terminals obtained from A by applying rules of the grammar G (including A itself). Then we consider LTGs respecting some constraints.

Definition 12. Set of successive non terminals: Let $G = (NT, T, S, P)$ be an LTG and $\leadsto_G$ the relation introduced in Definition 8. For any $A \in NT$ we define $Succ(A) = \{B \in NT \mid A \leadsto^*_G B\}$ where $\leadsto^*_G$ is the reflexive-transitive closure of $\leadsto_G$. □

Theorem 2. *Let $G = (NT, T, S, P)$ be a non-recursive monadic LTG such that*

$$\forall\, C \to c[E] \in P,\ \forall A, B \in NT(E),\ (A \neq B \implies Succ(A) \cap Succ(B) = \emptyset)$$

Then $G\!\uparrow$ is a monadic LTG. □

The following example illustrates the need of the condition imposed on non-terminals by Theorem 2. It also introduces the idea that by renaming common terminals and non-terminals one can adapt a given grammar to the condition imposed by Theorem 2.

Example 7. Consider a non-recursive monadic LTG G having the following rules:

$R \to root[PROF^*.STUD^*]$ $\quad PROF \to professor[F.L]$ $\quad STUD \to stud[F.L]$

$F \to firstName[\epsilon]$ $\quad L \to lastName[\epsilon]$

and not respecting the condition in Theorem 2. The resulting $G\!\uparrow$ computed by our algorithm (Definition 11) has a production rule $R \to root[E]$ where $E = (PROF \mid ((F|\epsilon).(L|\epsilon)))^*.(STUD \mid ((F|\epsilon).(L|\epsilon)))^*$. Clearly the regular expression E is not 1-unambiguous and thus the LTG $G\!\uparrow$ is not deterministic □

Now we consider how to compute the weak inclusion of the language generated by a grammar G into the language generated by a grammar G', when G' is a non-recursive monadic (and maybe non-strict) LTG that respects the condition of Theorem 2. Indeed, to decide whether $L(G)$ is weakly included in $L(G')$, we compute $G'\!\uparrow$, which is also a monadic LTG (Theorem 2). Clearly, $G'\!\uparrow$

may be non-strict. However, it is interesting to remark that the construction of $G'\uparrow$ (Definition 11) gives us a grammar where each non terminal of a regular expression in G' can be replaced by ϵ. Indeed, let $E = A_1 \circ A_2 \circ \cdots \circ A_n$ be a part of a regular expression, composed of non-terminals A_i (where $\circ$ is any allowed operator). Each step of our algorithm consists in changing $E = A_1 \circ A_2 \circ \cdots \circ A_n$ into a new regular expression $E' = (A_1 \mid E_1) \circ (A_2 \mid E_2) \circ \cdots \circ (A_n \mid E_n)$ where each E_i is a regular expression in G' (see Definition 11). Then E' is modified by replacing each non terminal B_{i_j} in each expression E_i by $B_{i_j}|E_{i_j}$ and so on, until reaching some $E_{i_{j\ldots k}} = \epsilon$. It follows that all resulting regular expression have the form $E'' = A_1 \,|\, (B_{1_1}|(\cdots|\epsilon)) \circ \cdots \circ A_n \,|\, (B_{n_1}|(\cdots|\epsilon))$. In other words, $\epsilon \in L(E'')$. As explained at the end of Section 2, for a given regular expression E, when $\epsilon \in L(E)$ we have that $L(E^+) = L(E^*)$ and thus we can replace each $+$ by $*$. Based on all these points one can easily see that the obtained $G'\uparrow$ can be transformed into a strict grammar G'_1 by transforming operator ? and by replacing $+$ by $*$. As the LTG G'_1 is strict and monadic, it is also deterministic. Now, to decide whether the language $L(G)$ is *weakly included* into the language $L(G')$, we just need to check whether $L(G) \subseteq L(G'_1)$. Since $L(G'_1)$ is generated by a deterministic LTG, which is equivalent to a deterministic DTD, this can be done in polynomial time by using the method presented in [CGLN08].

6 Conclusion

The main contribution of this paper is a simple algorithm for computing the weak inclusion between two non-recursive XML types. It extends the weak inclusion notion, normally used for trees, to tree languages. Our approach is composed of two steps: the generation of $WI(G')$, which is linear; and precise language inclusion testing, exponential for non-recursive tree grammars (but polynomial for deterministic DTDs). Our tests show a good performance for practical cases. Weak inclusion is important for comparing types by relaxing father-children relationship and can be useful in applications such as the substitution of a web service in a composition.

To process recursive tree grammars, we envisage two directions: by defining restricted classes of recursive grammars, and trying to keep simple the generation of $WI(G')$; or by translating unranked trees into binary trees and using a complex machinery. Another idea could consist in translating the initial regular tree grammars G and G' into context-free word grammars $word(G)$ and $word(G')$ that generate the corresponding XML texts. We refer to [HMU01,Fuj08] as examples of the translation of a DTD or a tree automaton to a context-free word grammar. By using similar techniques it is possible to compute $WI(word(G'))$. Unfortunately, checking that $L(word(G)) \subseteq L(WI(word(G')))$ (phase 2) is undecidable since it amounts to check inclusion between context-free languages.

References

[BG05] Philip Bille and Inge Li Gørtz. The tree inclusion problem: In optimal space and faster. In *Automata, Languages and Programming, 32nd International*

Colloquium, ICALP, pages 66–77, 2005.

[BHH+08] Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar. Antichain-based universality and inclusion testing over non-deterministic finite tree automata. In *Int. Conf. on Implementation and Applications of Automata, CIAA*, pages 57–67. Springer, 2008.

[BW98] A. Brüggeman-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[CGLN08] Jérôme Champavère, Rémi Guilleron, Aurélien Lemay, and Joachim Niehren. Efficient Inclusion Checking for Deterministic Tree Automata and DTDs. In *Int. Conf. Language and Automata Theory and Applications, LATA*, volume 5196 of *LNCS*, pages 184–195. Springer, 2008.

[CGPS09] D. Colazzo, G. Ghelli, L. Pardini, and C. Sartiani. Linear Inclusion for XML Regular Expression Types. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM*, pages 137–146. ACM Digital Library, 2009.

[CGS09] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient Asymmetric Inclusion between Regular Expression Types. In *Proceeding of International Conference of Database Theory, ICDT*, pages 174–182. ACM Digital Library, 2009.

[CSC06] Yangjun Chen, Yong Shi, and Yibin Chen. Tree inclusion algorithm, signatures and evaluation of path-oriented queries. In *Symposium on Applied Computing*, pages 1020–1025, 2006.

[CZ00] Pascal Caron and Djelloul Ziadi. Characterization of Glushkov automata. *Theor. Comput. Sci. (TCS)*, 233(1-2):75–90, 2000.

[Fuj08] Akio Fujiyoshi. Combination of context-free grammars and tree automata for unranked and ranked trees. In *3th International Conference of Implementation and Applications of Automata, CIAA*, pages 283–285, 2008.

[GKM09] Michaela Götz, Christoph Koch, and Wim Martens. Efficient algorithms for descendant-only tree pattern queries. *Inf. Syst.*, 34(7):602–623, 2009.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley Publishing Company, second edition, 2001.

[KM95] Pekka Kilpeläinen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2):340–356, 1995.

[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.

[MNS04] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for simple regular expressions. In *Int. Symp. Mathematical Foundations of Computer Science, MFCS*, pages 889–900, 2004.

[RT97] Richter and Thorsten. A new algorithm for the ordered tree inclusion problem. In Alberto Apostolico and Jotun Hein, editors, *Combinatorial Pattern Matching*, volume 1264 of *LNCS*, pages 150–166. Springer, 1997.

[Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19:424–437, June 1990.

[ZPC97] D Ziadi, J. L. Ponty, and J.M. Champarnaud. Passage d'une expression rationnelle un automate fini non-deterministe. *Bull. Belg. Math. Soc*, 4:177–203, 1997.

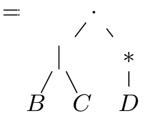
Appendix

7 Proofs of Lemmas 2 and 3

Lemma 2. *If $G = (NT, T, S, P)$ is non-recursive then $\rightsquigarrow_G$ is noetherian.* □

Proof. We prove the contraposition, *i.e.*, if relation $\rightsquigarrow_G$ is not noetherian then G is recursive. Indeed, if $\rightsquigarrow_G$ is not noetherian, there exists an infinite chain $A_0 \rightsquigarrow_G \cdots \rightsquigarrow_G A_n \rightsquigarrow_G \cdots$. However, since NT is finite, $\exists\ i \neq j$ s.t. $A_i = A_j$. Thus, we have $A_i \rightsquigarrow_G^+ A_i$, and G is recursive. □

Now, we see a regular expression E via its 'parsing tree' (denoted t_E). For instance, if $E = (B|C).D^*$ then the corresponding parsing tree is $t_E =$



$Pos_{NT}(t_E)$ denotes the set of positions of non-terminals in t_E. If the non-terminal A appears in t_E at position $p \in Pos_{NT}(t_E)$ (i.e. $t_E(p) = A$), and there is a rule $A \to a[E'']$ in G, and consider $t_{E'} = t_E[p \leftarrow A^\sharp | t_{E''}]$, then we will write (in coherence with Definition 9) $t_E \rightsquigarrow_G t_{E'}$, and when position p matters we will write $t_E \rightsquigarrow_G^p t_{E'}$.

Lemma 3. *If G is non-recursive then $\rightsquigarrow_G$ (over reg. expressions) is noetherian.*

Proof. Suppose that $\rightsquigarrow_G$ (over reg. expressions) is not noetherian; then there exists an infinite chain $t_{E_1} \rightsquigarrow_G^{p_1} t_{E_2} \rightsquigarrow_G^{p_2} \cdots \rightsquigarrow_G t_{E_n} \rightsquigarrow_G^{p_n} \cdots$; and for all i let $A_i = t_{E_i}|_{p_i}$.
Since $t_{E_1}, \cdots, t_{E_n}, \cdots$ are ranked trees (the maximal arity of operators is 2), for each i, the number of replacements in t_{E_i} at the same level is finite; then for having an infinite chain, the replacements of non-terminals must be done in depth. Therefore there exists an infinite sub-chain $t_{E_{i_1}} \rightsquigarrow_G^{p_{i_1}} t_{E_{i_2}} \rightsquigarrow_G^{p_{i_2}} \cdots \rightsquigarrow_G t_{E_{i_k}} \rightsquigarrow_G^{p_{i_k}} \cdots$ s.t. $p_{i_1} < p_{i_2} < \cdots < p_{i_k} < \cdots$. Consequently $A_{i_1} \rightsquigarrow_G A_{i_2} \rightsquigarrow_G \cdots \rightsquigarrow_G A_{i_k} \rightsquigarrow_G \cdots$, and finally $\rightsquigarrow_G$ (over non terminals) is not noetherian. Thus we have a contradiction and consequently we conclude that if G is non-recursive then $\rightsquigarrow_G$ (over regular expressions) is noetherian.

8 Proof of Theorem 1

For proofs, the process of removing marks needs to be explicit. $E_\natural$ (resp. $G_\natural$) denotes the regular expression (resp. the grammar) obtained by removing all marks from E (resp. from G). On the other hand, $L_G(A)$ denotes the language generated by non-terminal A in grammar G.

Lemma 4. *Let* $G = \{A_i \to a_i[E_i] \mid i \in \{1,\ldots,n\}\}$ *and* $G' = \{A_i \to a_i[\sigma^{j_i}(E_i)] \mid i \in \{1,\ldots,n\}, j_i \in I\!N\}$*. Suppose there exists* k *s.t. we can apply* σ *on* $\sigma^{j_k}(E_k)$*. Let* $G'' = \{A_k \to a_k[\sigma(\sigma^{j_k}(E_k)]\} \cup \{A_i \to a_i[\sigma^{j_i}(E_i)] \mid i \in \{1,\ldots,n\}\backslash\{k\}\}$*. Then* $\forall i \in \{1,\ldots,n\}$*,* $L_{G''_\natural}(A_i) \subseteq WI(L_{G'_\natural}(A_i)\}$

Proof. Let $E = \sigma^{j_k}(E_k)$ and $NT(E) \cap NT(G) = \{B_1,\ldots,B_p\}$ (i.e. the non-marked non-terminals of E). We obtain $\sigma(E)$ by remplacing each B_j with $B_j^\sharp|E'_j$ (where $B_j \to b_j[E'_j] \in G$). Therefore in $G''_\natural$, $B_j|E'_j$ can generate either b_j and its children, or directly the children of b_j (without the node b_j), so, by definition, a tree of $WI(L_{G'_\natural}(B_j))$. We conclude that for all i, A_i generates trees in $WI(L_{G'_\natural}(A_i))$.

Corollary 1. Correctness: $L(G\!\uparrow_\natural) \subseteq WI(L(G))$.

Proof. For any tree language S, $WI(WI(S)) = WI(S)$ and for the initial grammar G which does not contain marks, we have $G_\natural = G$.

Lemma 5. $L(G) \subseteq L(G\!\uparrow_\natural)$

Proof. For each regular expression E, words generated by E are also generated by $E\!\uparrow_\natural$. So if we use the rule $A_i \to a_i[E_i]$ to recognize a tree in $L(G)$ we can use the rule $A_i \to a_i[E_i\!\uparrow_\natural]$ which is in $G\!\uparrow_\natural$ to recognize the same tree in $L(G\!\uparrow_\natural)$.

Lemma 6. *Let a tree* $t' \in L(G\!\uparrow_\natural)$ *and a non-empty position* q *in* t'*, then* $Rem_q(t') \in L(G\!\uparrow_\natural)$*. Recall that* $Rem_q(t')$ *is the tree obtained after the removal of the node at position* q *in* t' *(see Definition 2).*

Proof. Suppose $t'(q) = a_i$, to prove that t' is in $L(G\!\uparrow_\natural)$ we use the rule $A_i \to a_i[E_i\!\uparrow_\natural]$ from $G\!\uparrow_\natural$, but we also use another rule $A'_i \to a'_i[E'_i\!\uparrow_\natural]$ from $G\!\uparrow_\natural$ (because $q \neq \epsilon$) where A_i is in $NT(E'_i\!\uparrow_\natural)$. By construction of the rules of $G\!\uparrow_\natural$, we have in $E'_i\!\uparrow_\natural$ the regular expression $(A_i|E_i\!\uparrow_\natural)$. The first part is used to recognize t' in $L(G\!\uparrow_\natural)$, if we use the second part we can recognize $Rem_q(t')$ in $L(G\!\uparrow_\natural)$.

Corollary 2. Completeness: $WI(L(G)) \subseteq L(G\!\uparrow_\natural)$.

Proof. Let $t \in WI(L(G))$, then by Definition 3, there exists $t' \in L(G)$ and positions $q_1,\ldots,q_n$ in $Pos(t')$ such that $t = Rem_{q_n}(\cdots Rem_{q_1}(t'))$. From Lemma 5, $t' \in L(G\!\uparrow_\natural)$. From Lemma 6, $Rem_{q_1}(t') \in L(G\!\uparrow_\natural)$, and by applying Lemma 6 several times, we deduce $t \in L(G\!\uparrow_\natural)$.

9 Proof of Theorem 2

Recall that $E_\natural$ (resp. $G_\natural$) denotes the regular expression (resp. the grammar) obtained by removing all marks from E (resp. from G).

1) Since G is an LTG, there are no competing non-terminals in G. To transform G into $G\!\uparrow_\natural$, only regular expressions are modified. Then there are no competing non-terminals in $G\!\uparrow_\natural$, hence $G\!\uparrow_\natural$ is an LTG.

2) Let E be a regular expression s.t. $E_\natural$ is monadic (E may contain marked non-terminals). Consider the multiset of non-terminals of E, as being $NT(E) = \{A_1, \ldots, A_n\} \cup \{B_1^\sharp, \ldots, B_k^\sharp\}$. Note that $NT(E_\natural)$ does not contain duplicates.

- Let $i \in \{1, \ldots, n\}$. There exists $A_i \to a_i[E_i] \in G$ and $\sigma(A_i) = A_i^\sharp | E_i$. Since G is monadic, E_i is monadic. Since G is non-recursive, $A_i \notin NT(E_i)$. Then $\sigma(A_i)_\natural$ is monadic.
- Let $j \in \{1, \ldots, k\}$. We have $NT(\sigma(A_i)_\natural) \subseteq Succ(A_i)$ and $B_j \in Succ(B_j)$. Therefore, if $\{B_j\} \cap NT(\sigma(A_i)_\natural) \neq \emptyset$ then $Succ(A_i) \cap Succ(B_j) \neq \emptyset$ which is impossible due to the hypothesis.
- Let $j \in \{1, \ldots, n\}$ s.t. $j \neq i$. We have $NT(\sigma(A_j)_\natural) \subseteq Succ(A_j)$. Therefore, if $NT(\sigma(A_j)_\natural) \cap NT(\sigma(A_i)_\natural) \neq \emptyset$ then $Succ(A_j) \cap Succ(A_i) \neq \emptyset$ which is impossible due to the hypothesis.

Therefore $\sigma(E)_\natural$ is monadic.

3) By applying the above result several times, we get $\sigma^n(E)_\natural$ is monadic.