# Weak Inclusion for recursive XML Types

Joshua Amavi Jacques Chabin Pierre Réty

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

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

**Keywords:** XML type, regular unranked-tree grammar, approximative inclusion.

## 1 Introduction

In database area, an important problem is schema evolution, particularly when considering XML types. XML is also used for exchanging data on the web. In this setting, we want to compare XML types in a loose way. To do it, we address the more general problem of approximative comparison of unranked-tree languages defined by regular grammars.

*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 $A \to a[\epsilon]$.
Suppose that $G$ and $G'$ contain the following rules:

$$\mathrm{F} \to \mathrm{firstName}[\epsilon],\ \mathrm{L} \to \mathrm{lastName}[\epsilon]\ ,\ \mathrm{T} \to \mathrm{title}[\epsilon] \text{ and } \mathrm{Y} \to \mathrm{year}[\epsilon].$$

$P$ defines a publication, and $B$ is the bibliography.
In $G$ : $\mathrm{P} \to \mathrm{publi}[(F.L)^+.T.B^?],\ \mathrm{B} \to \mathrm{biblio}[P^+]$.
In $G'$ : $\mathrm{P} \to \mathrm{publi}[A^*.Pa], \mathrm{A} \to \mathrm{author}[F.L], \mathrm{Pa} \to \mathrm{paper}[T.Y.B^?], \mathrm{B} \to \mathrm{biblio}[P^+]$

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 organized as the tree $t'$ in Figure 1. □

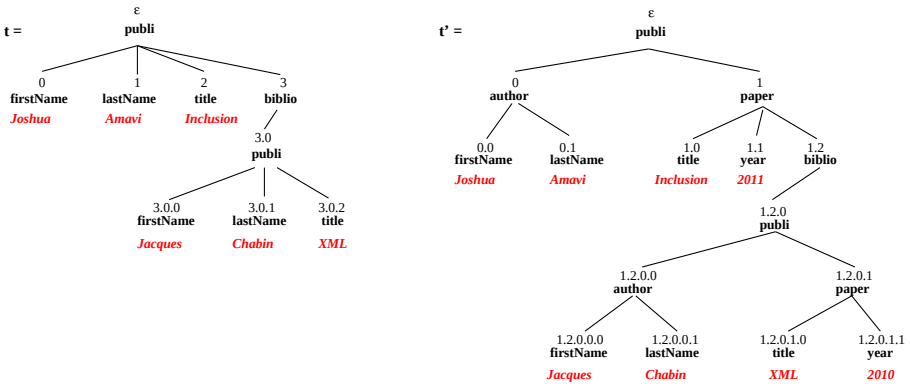



**Fig. 1.** Examples of trees $t$ and $t'$ valid with respect to $G$ and $G'$, respectively.

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 usual 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 sets (also called languages) 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 removing nodes *author*, *paper*, *year* from $t'$, i.e. we have $t \lhd t'$.

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[1] $L(G)$ is weakly included in $L(G')$ iff $L(G) \subseteq WI(L(G'))$.

To compute $WI(L(G'))$, we have already proposed [3] a direct and simple approach using regular unranked-tree grammars (hedge grammars), assuming that $L(G')$ is bounded in depth, i.e. $G'$ is not recursive. Given a hedge grammar $G'$, the idea consists in replacing each non-terminal occurring in a right-hand-side of a production rule, by itself or its children. For example, if $G'$ contains the rules $A \to a[B]$, $B \to b[C]$, $C \to c[\epsilon]$, we get the grammar $G'' = \{C \to c[\epsilon],\ B \to b[C|\epsilon],\ A \to a[B|(C|\epsilon)]\}$. This grammar generates $WI(L(G'))$, because in each regular expression, whenever we have a non-terminal $X$ ($X \in \{A, B, C\}$)

[1] If $L(G)$ is weakly included in $L(G')$, for $t \in L(G)$ there exists $t' \in L(G')$ s.t. $t \lhd t'$. Then $t \in WI(L(G'))$, hence $L(G) \subseteq WI(L(G'))$.
Conversely if $L(G) \subseteq WI(L(G'))$, for $t \in L(G)$ we have $t \in WI(L(G'))$. Thus there exists $t' \in L(G')$ s.t. $t \lhd t'$. Therefore $L(G)$ is weakly included in $L(G')$.

that generates $x$ ($x \in \{a, b, c\}$), we can also directly generate the children of $x$ (instead of $x$), and more generally the successors of $x$.

Unfortunately, it does not work if $G'$ is recursive. Consider $G'_1 = \{A \to a[B.(A|\epsilon).C],\ B \to b[\epsilon],\ C \to c[\epsilon]\}$. If we work as previously, we get the rule $A \to a[B.(A|(B.(A|\epsilon).C)|\epsilon).C]$. However, a new occurrence of $A$ has appeared in the rhs, and if we replace it again, this process does not terminate. If we stop at some step, the resulting grammar does not generate $WI(L(G'))$. In this very simple example, it is easy to see that $G''_1 = \{A \to a[B^*.(A|\epsilon).C^*],\ B \to b[\epsilon],\ C \to c[\epsilon]\}$ generates $WI(L(G'_1))$. But if we now consider $G'_2 = \{A \to a[B.(A.A \mid \epsilon).C],\ B \to b[\epsilon],\ C \to c[\epsilon]\}$, then in $WI(L(G'_2))$, $a$ may be the left-sibling of $b$ (which was not possible in $L(G'_2)$ nor in $WI(L(G'_1))$). Actually, $WI(L(G'_2))$ can be generated by the grammar $G''_2 = \{A \to a[(A|B|C)^*],\ B \to b[\epsilon],\ C \to c[\epsilon]\}$. In other words, the recursive case is much more difficult.

In this paper, we address the general case: some symbols may be recursive, and some others may not. Given an arbitrary regular unranked-tree grammar $G'$, we present a direct approach that computes a regular grammar, denoted $WI(G')$, that generates the language $WI(L(G'))$. To do it, terminal-symbols are divided into 3 categories: non-recursive, 1-recursive, 2-recursive. For $n > 2$, $n$-recursivity is equivalent to 2-recursivity. Surprisingly, the most difficult situation is 1-recursivity. We prove that our algorithm for computing $WI(G')$ is correct and complete. An implementation has been done in Java, and experiments are presented.

Consequently, for arbitrary regular grammars $G$ and $G'$, checking that $L(G)$ is weakly included into $L(G')$, i.e. checking that $L(G) \subseteq WI(L(G'))$, is equivalent to check that $L(G) \subseteq L(WI(G'))$, which is decidable since $G$ and $WI(G')$ are regular grammars.

***Paper organisation:*** Section 2 gives some theoretical background. Section 3 presents how to compute $WI(G)$ for a given possibly-recursive regular grammar $G$, while Section 4 analyses some experimental results. Related work and other possible methods are addressed in Section 5. Due to the lack of space, missing proofs are given in [4].

## 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 $\Sigma$. The set of the trees over $\Sigma$ is denoted by $T_\Sigma$. 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 $Pos(t)$ satisfies: $\forall u, i, j\ (j \geq 0, u.j \in Pos(t), 0 \leq i \leq j) \Rightarrow u.i \in Pos(t)$ (char "." denotes the concatenation). The *size* of $t$ (denoted $|t|$) is the cardinal of $Pos(t)$. As usual, $\epsilon$ denotes the empty sequence of integers, i.e. the root position. $t$, $t'$ will denote trees.
Figure 1 illustrates trees with positions and labels: we have, for instance, $t(1) = lastName$ and $t'(1) = paper$. The sub-tree $t'|_{1.2}$ is the one whose root is *biblio*.

**Definition 1. Position comparison**: 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 $i < j$. Position $p$ is *parallel* to $q$ (denoted $p \parallel q$) if $\neg(p < q) \wedge \neg(q < p)$. □

**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 $t$ obtained from $t'$ by removing the node at position $q$ (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 2.*
In Figure 1, $t = Rem_0(Rem_1(Rem_{1.1}(Rem_{1.2.0.0}(Rem_{1.2.0.1}(Rem_{1.2.0.1.1}(t'))))))$, then $t \lhd t'$. 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 the tree $t_1 = paper(biblio, year)$ is not weakly included in $t'$ since *biblio* should appear to the right of *year*. □

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

**Grammar in Normal Form:** As usual, in this paper, we only consider regular tree grammars such that (i) every non-terminal generates at least one tree containing only terminal symbols and (ii) distinct production rules have distinct left-hand-sides (*i.e.*, tree grammars *in normal form* [13]).
Thus, given an RTG $G = (NT, \Sigma, S, P)$, for each $A \in NT$ there exists in $P$ exactly one rule of the form $A \to a[E]$, i.e. whose left-hand-side is $A$. □

*Example 3.* The grammar $G_0 = (NT_0, \Sigma, S, P_0)$, where $NT_0 = \{X, A, B\}$, $\Sigma = \{f, a, c\}$, $S = \{X\}$, and $P_0 = \{X \to f\,[A.B],\ A \to a[\epsilon],\ B \to a[\epsilon],\ A \to c[\epsilon]\}$, is not in normal form. The conversion of $G_0$ into normal form gives the sets $NT_1 = \{X, A, B, C\}$ and $P_1 = \{X \to f\,[(A|C).B],\ A \to a[\epsilon],\ B \to a[\epsilon],\ C \to c[\epsilon]\}$.

**Definition 5.** Let $G = (NT, \Sigma, S, P)$ be an RTG (in normal form). Consider a non-terminal $A \in NT$, and let $A \to a[E]$ be the unique production of $P$ whose left-hande-side is $A$.
$L^w(E)$ denotes the set of words (over non-terminals) generated by $E$.
The set $L_G(A)$ of trees generated by $A$ is defined recursively by:

$$L_G(A) = \{a(t_1, \ldots, t_n) \mid \exists u \in L^w(E),\ u = A_1 \ldots A_n,\ \forall i, t_i \in L_G(A_i)\}$$

The language $L(G)$ generated by $G$ is: $L(G) = \{t \in T_\Sigma \mid \exists A \in S,\ t \in L_G(A)\}$.
A *nt-tree* is a tree whose labels are non-terminals. The set $L^{nt}_G(A)$ of nt-trees generated by $A$ is defined recursively by:
$L^{nt}_G(A) = \{A(t_1, \ldots, t_n) \mid \exists u \in L^w(E),\ u = A_1 \ldots A_n,\ \forall i\,(t_i \in L^{nt}_G(A_i) \vee t_i = A_i)\}$

## 3 Weak Inclusion for possibly-recursive Tree Grammars

First, we need to compute the recursivity types of non-terminals. Intuitively, the non-terminal $A$ of a grammar $G$ is *2-recursive* if there exists $t \in L^{nt}_G(A)$ and $A$ occurs in $t$ at (at least) two non-empty positions $p, q \in Pos(t)$ s.t. $p \parallel q$. $A$ is *1-recursive* if $A$ is not 2-recursive, and $A$ occurs in some $t \in L^{nt}_G(A)$ at a non-empty position. $A$ is *not recursive*, if $A$ is neither 2-recursive nor 1-recursive.

*Example 4.* Consider the grammar $G$ of Example 1. $P$ is 2-recursive since $P$ may generate $B$, and $B$ may generate the tree $biblio(P, P)$. $B$ is also 2-recursive. On the other hand $F$, $L$, $T$, $Y$ are not recursive. No non-terminal of $G$ is 1-recursive.

**Definition 6.** Let $G = (NT, \Sigma, S, P)$ be an RTG in normal form. For a regular expression $E$, $NT(E)$ denotes the set of non-terminals occurring in $E$.

- We define the relation $>$ over non-terminals by:
  $A > B$ if $\exists A \to a[E] \in P$ s.t. $B \in NT(E)$.
- We define $>$ over multisets[2] of non-terminals, whose size is at most 2, by:
  - $\{A\} > \{B\}$ if $A > B$,
  - $\{A, B\} > \{C, D\}$ if $A = C$ and $B > D$,
  - $\{A\} > \{C, D\}$ if there exists a production $A \to a[E]$ in $G$ and a word $u \in L(E)$ of the form $u = u_1 C u_2 D u_3$.

*Remark 1.* To check whether $\{A\} > \{C, D\}$, i.e. $\exists u \in L(E),\ u = u_1 C u_2 D u_3$, we can use the recursive function "$in$" defined by $in(C, D, E) =$
- if $E = E_1|E_2$, return $in(C, D, E_1) \vee in(C, D, E_2)$,
- if $E = E_1.E_2$, return $(C \in NT(E_1) \wedge D \in NT(E_2)) \vee (C \in NT(E_2) \wedge D \in NT(E_1))$
  $\vee\, in(C, D, E_1) \vee in(C, D, E_2)$,

[2] Since we consider multisets, note that $\{A, B\} = \{B, A\}$ and $\{C, D\} = \{D, C\}$.

- if $E = E_1^*$ or $E = E_1^+$, return $(C \in NT(E_1)) \wedge (D \in NT(E_1))$,
- if $E = E_1^?$, return $in(C, D, E_1)$,
- if $E$ is a non-terminal or $E = \epsilon$, return $false$.

This function terminates since recursive calls are always on regular expressions smaller than $E$. The runtime is $O(|E|)$, where $|E|$ is the size of $E$.

**Definition 7.** Let $>^+$ be the transitive closure of $>$

- The non-terminal $A$ is 2-recursive iff $\{A\} >^+ \{A, A\}$.
- $A$ is 1-recursive iff $A >^+ A$ and $A$ is not 2-recursive.
- $A$ is not recursive iff $A$ is neither 2-recursive nor 1-recursive.

*Remark 2.* The transitive closure of $>$ can be computed using Warshall algorithm. If there are $n$ non-terminals in $G$, there are $p = n + \frac{n.(n+1)}{2}$ multisets of size at most 2. Then a boolean matrix $p \times p$ can represent $>$, consequently the runtime for computing $>^+$ is $O(p^3) = O(n^6)$, which is polynomial.

*Example 5.* Using grammar $G$ of Example 1, we have $\{P\} > \{B\} > \{P, P\}$. Therefore $P$ is 2-recursive.
We have $\neg(\{F\} >^+ \{F, F\})$ and $\neg(F >^+ F)$, therefore $F$ is not recursive.

Now, to define an RTG that generates $WI(L(G))$, we need additional notions.

**Definition 8.** Let $G = (NT, \Sigma, S, P)$ be an RTG in normal form.

- $\equiv$ is the relation over non-terminals defined by $A \equiv B$ if $A >^* B \wedge B >^* A$, where $>^*$ denotes the reflexive-transitive closure of $>$.
  Note that $\equiv$ is an equivalence relation, and if $A \equiv B$ then $A$ and $B$ have the same recursivity type. $\widehat{A}$ will denote the equivalence class of $A$.
- $Succ(A)$ is the set of non-terminals s.t. $Succ(A) = \{X \in NT \mid A >^* X\}$.
- For a set $Q = \{A_1, \ldots, A_n\}$ of non-terminals, $Succ(Q) = Succ(A_1) \cup \cdots \cup Succ(A_n)$.
- $Left(A)$ is the set of non-terminals defined by $Left(A) = \{X \in NT \mid \exists B, C \in \widehat{A}, \exists B \to b[E] \in P, \exists u \in L^w(E), u = u_1 X u_2 C u_3\}$.
- Similarly, $Right(A)$ is the set of non-terminals defined by $Right(A) = \{X \in NT \mid \exists B, C \in \widehat{A}, \exists B \to b[E] \in P, \exists u \in L^w(E), u = u_1 C u_2 X u_3\}$.
- $RE(A)$ is the regular expression $E$, assuming $A \to a[E]$ is the production rule of $G$ whose left-hand-side is $A$.
- $\widehat{RE}(A) = RE(A)|RE(B_1)|\cdots|RE(B_n)$ where $\widehat{A} = \{A, B_1, \ldots, B_n\}$.

*Example 6.* With the grammar $G'$ of Example 1, we have :

- $P \equiv B$, because $P > Pa > B$ and $B > P$.
- $\widehat{P} = \{P, Pa, B\}$.
- $Succ(A) = \{A, F, L\}$.
- $Left(P)$ is defined using non-terminals equivalent ($\equiv$) to $P$, i.e. $B$, $P$, $Pa$, and grammar $G'$, which contains rules (among others):
  B $\to$ biblio$[P^+]$, P $\to$ publi$[A^*.Pa]$, Pa $\to$ paper$[T.Y.B^?]$.
  $P^+$ may generate $P.P$, therefore $Left(P) = \{P\} \cup \{A\} \cup \{T, Y\} = \{P, A, T, Y\}$.

- $RE(P) = A^*.Pa$
- $\widehat{RE}(P) = RE(P)|RE(Pa)|RE(B) = (A^*.Pa)|(T.Y.B^?)|P^+$.

**Lemma 1.** *Let $A$, $B$ be non-terminals.*

- *If $A \equiv B$ then $Succ(A) = Succ(B)$, $Left(A) = Left(B)$, $Right(A) = Right(B)$, $\widehat{RE}(A) = \widehat{RE}(B)$.*
- *If $A$, $B$ are not recursive, then $A \neq B$ implies $A \not\equiv B$, therefore $\widehat{A} = \{A\}$ and $\widehat{B} = \{B\}$, i.e. equivalence classes of non-recursive non-terminals are singletons.*

*Proof.* The first part is obvious.
$(A >^+ A) \implies (A >^+ A \wedge \neg(\{A\} >^+ \{A, A\})) \vee (\{A\} >^+ \{A, A\})$ which implies that $A$ is (1 or 2)-recursive. Consequently, if $A$ is not recursive, then $A \not>^+ A$. Now, if $A$, $B$ are not recursive, $A \neq B$ and $A \equiv B$, then $A >^+ B \wedge B >^+ A$, therefore $A >^+ A$, which is impossible as shown above. □

To take all cases into account, the following definition is a bit intricate. To give intuition, consider the following very simple situations:

- If the initial grammar $G$ contains production rules $A \to a[B]$, $B \to b[\epsilon]$ (here $A$ and $B$ are not recursive), we replace these rules by $A \to a[B|\epsilon]$, $B \to b[\epsilon]$ to generate $WI(L(G))$. Intuitively, $b$ may be generated or removed (replaced by $\epsilon$). See Example 7 below for a more general situation.
- If $G$ contains $A \to a[(A.A.B)|\epsilon]$, $B \to b[\epsilon]$ (here $A$ is 2-recursive and $B$ is not recursive), we replace these rules by $A \to a[(A|B)^*]$, $B \to b[\epsilon]$ to generate $WI(L(G))$. Actually the regular expression $(A|B)^*$ generates all words composed of elements of $Succ(A)$. See Example 8 for more intuition.
- The 1-recursive case is more complicated and is illustrated by Examples 9 and 10.

**Definition 9.** For each non-terminal $A$, we recursively define a regular expression $Ch(A)$ ($Ch$ for children). Here, any set of non-terminals, like $\{A_1, \ldots, A_n\}$, is also considered as being the regular expression $(A_1|\cdots|A_n)$.

- if $A$ is 2-recursive, $Ch(A) = (Succ(A))^*$
- if $A$ is 1-rec, $Ch(A) = (Succ(Left(A)))^*.Ch^{rex}_{\widehat{A}}(\widehat{RE}(A)).(Succ(Right(A)))^*$
- if $A$ is not recursive, $Ch(A) = Ch^{rex}_{\widehat{A}}(RE(A))$

and $Ch^{rex}_{\widehat{A}}(E)$ is the regular expression obtained from $E$ by replacing each non-terminal $B$ occurring in $E$ by $Ch_{\widehat{A}}(B)$, where

$$Ch_{\widehat{A}}(B) = \begin{cases} \widehat{B}|\epsilon & \text{if } B \text{ is 1-recursive and } B \in \widehat{A} \\ Ch(B) & \text{if } (B \text{ is 2 recursive}) \text{ or } (B \text{ is 1-recursive and } B \notin \widehat{A}) \\ B|Ch(B) & \text{if } B \text{ is not recursive} \end{cases}$$

By convention $Ch^{rex}_{\widehat{A}}(\epsilon) = \epsilon$ and $Ch(\epsilon) = \epsilon$.

**Algorithm.** *Input:* let $G = (NT, T, S, P)$ be a regular grammar in normal form. *Output:* grammar $G' = (NT, T, S, P')$ obtained from $G$ by replacing each production $A \to a[E]$ of $G$ by $A \to a[Ch(A)]$.

**Theorem 1.** *The computation of Ch always terminate, and* $L(G') = WI(L(G))$.

The proof is given in [4]. Let us now consider several examples to give more intuition about the algorithm and show various situations.

*Example 7.* Consider grammar $G = \{A \to a[B],\ B \to b[C],\ C \to c[\epsilon]\}$.
$A$ is the start symbol. Note that $A$, $B$, $C$ are not recursive.
$Ch(C) = Ch^{rex}_{\widehat{C}}(RE(C)) = Ch^{rex}_{\widehat{C}}(\epsilon) = \epsilon$.
$Ch(B) = Ch^{rex}_{\widehat{B}}(RE(B)) = Ch^{rex}_{\widehat{B}}(C) = Ch_{\widehat{B}}(C) = C|Ch(C) = C|\epsilon$.
$Ch(A) = Ch^{rex}_{\widehat{A}}(RE(A)) = Ch^{rex}_{\widehat{A}}(B) = Ch_{\widehat{A}}(B) = B|Ch(B) = B|(C|\epsilon)$.
Thus, we get the grammar $G' = \{A \to a[B|(C|\epsilon)],\ B \to b[C|\epsilon],\ C \to c[\epsilon]\}$ that generates $WI(L(G))$ indeed. In this particular case, where no non-terminal is recursive, we get the same grammar as in our previous work [3], though the algorithm was formalized in a different way.

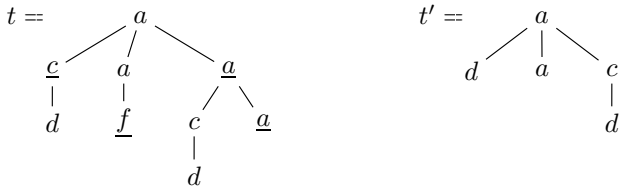
*Example 8.* Consider grammar $G$ that contains the rules:

$$A \to a[(C.A.A^{?})|F|\epsilon],\ C \to c[D],\ D \to d[\epsilon],\ F \to f[\epsilon]$$

$A$ is 2-recursive; $C$, $D$, $F$ are not recursive. $Ch(D) = Ch(F) = \epsilon$. $Ch(C) = D|\epsilon$.
$Succ(A) = \{A, C, D, F\}$. Considered as a regular expression, $Succ(A) = A|C|D|F$.
Therefore $Ch(A) = (A|C|D|F)^*$. We get the grammar $G'$:

$$A \to a[(A|C|D|F)^*],\ C \to c[D|\epsilon],\ D \to d[\epsilon],\ F \to f[\epsilon]$$

The tree $t$ below is generated by $G$. By removing underlined symbols, we get $t' \lhd t$, and $t'$ is generated by $G'$ indeed. Note that $a$ is a left-sibling of $c$ in $t'$, which is impossible in a tree generated by $G$.




*Example 9.* Consider grammar $G$ that contains the rules ($A$ is the start symbol):

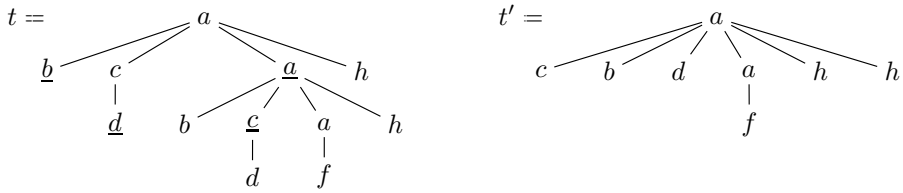
$$A \to a[(B.C.A^{?}.H)|F],\ B \to b[\epsilon],\ C \to c[D],\ D \to d[\epsilon],\ H \to h[\epsilon],\ F \to f[\epsilon]$$

$A$ is 1-recursive; $B$, $C$, $D$, $H$, $F$ are not recursive. $Ch(B) = Ch(D) = Ch(H) = Ch(F) = \epsilon$. $Ch(C) = D|\epsilon$. $\widehat{A} = \{A\}$, then $\widehat{RE}(A) = RE(A) = (B.C.A^{?}.H)|F$.
$Ch^{rex}_{\widehat{A}}(\widehat{RE}(A)) = Ch^{rex}_{\widehat{A}}((B.C.A^{?}.H)|F) = (Ch_{\widehat{A}}(B).Ch_{\widehat{A}}(C).Ch_{\widehat{A}}(A)^{?}.Ch_{\widehat{A}}(H))|Ch_{\widehat{A}}(F)$
$= ((B|Ch(B)).(C|Ch(C)).(A|\epsilon)^{?}.(H|Ch(H)))|(F|Ch(F))$
$= ((B|\epsilon).(C|(D|\epsilon)).(A|\epsilon)^{?}.(H|\epsilon))|(F|\epsilon)$, simplified into $(B^{?}.(C|D|\epsilon).A^{?}.H^{?})|F^{?}$.
$Left(A) = \{B, C\}$, then $Succ(Left(A)) = \{B, C, D\}$.
$Right(A) = \{H\}$, then $Succ(Right(A)) = \{H\}$.
Considered as regular expressions (instead of sets), $Succ(Left(A)) = B|C|D$ and

$Succ(Right(A)) = H$.
Therefore $Ch(A) = (Succ(Left(A)))^*.Ch^{rex}_{\widehat{A}}(\widehat{RE}(A)).(Succ(Right(A)))^* = (B|C|D)^*.[(B^?.(C|D|\epsilon).A^?.H^?)|F^?].H^*$, which could be simplified into $(B|C|D)^*.(A^?|F^?).H^*$; we get the grammar $G'$:

$A \to a[(B|C|D)^*.(A^?|F^?).H^*]$, $B \to b[\epsilon]$, $C \to c[D|\epsilon]$, $D \to d[\epsilon]$, $H \to h[\epsilon]$, $F \to f[\epsilon]$

The tree $t$ below is generated by $G$. By removing underlined symbols, we get $t' \lhd t$, and $t'$ is generated by $G'$ indeed. Note that $c$ is a left-sibling of $b$ in $t'$, which is impossible for a tree generated by the initial grammar $G$. On the other hand, $b$, $c$, $d$ are necessarily to the left of $h$ in $t'$.




*Example 10.* The previous example does not show the role of equivalence classes. Consider $G = \{A \to a[B^?],\ B \to b[A]\}$. $A$ and $B$ are 1-recursive.
$A \equiv B$ then $\widehat{A} = \widehat{B} = \{A, B\}$. $Left(A) = Left(B) = Right(A) = Right(B) = \emptyset$.
Therefore $Ch(A) = Ch^{rex}_{\widehat{A}}(\widehat{RE}(A)) = Ch^{rex}_{\widehat{A}}(B^?|A) = (Ch_{\widehat{A}}(B))^?|(Ch_{\widehat{A}}(A)) = (\widehat{B}|\epsilon)^?|(\widehat{A}|\epsilon) = (A|B|\epsilon)^?|(A|B|\epsilon)$. Note that $\widehat{A}$ and $\widehat{B}$ have been replaced by $A|B$, which is needed as shown by trees $t$ and $t'$ below. $Ch(A)$ can be simplified into $A|B|\epsilon$. Since $A \equiv B$, $Ch(B) = Ch(A)$.
Then we get the grammar $G' = \{A \to a[A|B|\epsilon],\ B \to b[A|B|\epsilon]\}$.
The tree $t = a(b(a))$ is generated by $G$. By removing $b$, we get $t' = a(a)$ which is generated by $G'$ indeed.

## 4 Implementation and Experiments

Our prototype is implemented in Java and the experiments are done on an Intel Quad Core i3-2310M with 2.10GHz and 8GB of memory. The only step that takes time is the computation of recursivity types of non-terminals. The difficulty is for deciding whether a recursive non-terminal is 2- or 1-recursive. To do it, we have implemented two algorithms: one using Warshall algorithm for computing $>^+$, whose runtime is $O(n^6)$ where $n$ is the number of non-terminals[3], and another based on comparison of cycles in a graph representing relation $>$ (over non-terminals, not over multisets). In the worst case, the runtime of the second algorithm is at least exponential, since all cycles should be detected. Actually, the runtime of the first algorithm depends on the number $n$ of non-terminals, whereas the runtime of the second one depends on the number of cycles in the graph.

In Table 1, #1-*rec* denotes the number of 1-recursive non-terminals (idem for #0-*rec* and #2-*rec*), #Cycles is the number of cycles, and $|G|$ (resp. $|WI(G)|$)

[3] Since grammars are in normal form, $n$ is also the number of production rules.

denotes the sum of the sizes of the regular expressions[4] occurring in the initial grammar $G$ (resp. in the resulting grammar $WI(G)$). Results in lines 1 to 4 concern synthetic DTDs, while those in lines 5 to 6 correspond to real DTDs. The experiments show: if $n < 50$, the Warshall-based algorithm takes less than 6 seconds. Most often, the cycle-based algorithm runs faster than the Warshall-based algorithm. An example with $n = 111$ (line 3) took 7 minutes with the first algorithm, and was immediate with the second one. When the number of cycles is less than 100, the second algorithm is immediate, even if the runtime in the worst case is bad.

Now, consider the DTD (line 5) specifying the linguistic annotations of named entities performed within the National Corpus of Polish project [2, page 22]. After transforming this DTD into a grammar, we get 17 rules and some non-terminals are recursive. Both algorithms are immediate (few rules and few cycles). The example of line 6 specifies XHTML DTD[5] (with $n = 85$ and $\#Cycles = 9620$). The Warshall-based algorithm and the cycle-based algorithm respond in 2 minutes.

| | **Unranked grammars** | | | | **Runtime (s)** | | **Sizes** | |
|---|---|---|---|---|---|---|---|---|
| | #0-*rec* | #1-*rec* | #2-*rec* | #Cycles | Warshall | Cycle-compar. | $\|G\|$ | $\|WI(G)\|$ |
| 1 | 9 | 2 | 38 | 410 | 5.48 | 0.82 | 183 | 1900 |
| 2 | 34 | 4 | 12 | 16 | 5.51 | 0.08 | 126 | 1317 |
| 3 | 78 | 12 | 21 | 30 | 445 | 0.2 | 293 | 4590 |
| 4 | 8 | 2 | 16 | 788 | 0.38 | 1.51 | 276 | 397 |
| 5 | 14 | 0 | 2 | 1 | 0.08 | 0.01 | 30 | 76 |
| 6 | 30 | 0 | 55 | 9620 | 136.63 | 113.91 | 1879 | 22963 |

**Table 1.** Runtimes in seconds for the Warshall-based and the Cycle-comparison algorithms.

## 5 Related Work and Discussion

The (weak) tree inclusion problem was first studied in [12], and improved in [5, 7, 15]. Our proposal differs from these approaches because we consider the weak inclusion with respect to *tree languages* (and not only with respect to trees). Testing precise inclusion of XML types is considered in [6, 8, 9, 14]. In [14], the authors study the complexity, identifying efficient cases. In [6] a polynomial algorithm for checking whether $L(A) \subseteq L(D)$ is given, where $A$ is an automaton for unranked trees and $D$ is a deterministic DTD.

In this paper, given a regular unranked-tree grammar $G$ (hedge grammar), we have presented a direct method to compute a grammar $G'$ that generates the set of trees (denoted $WI(L(G))$) weakly included in trees generated by $G$.

In [1], we have computed $G'$ by transforming unranked-tree languages into binary-tree ones, using first-child next-sibling encoding. Then the weak-inclusion relation $\lhd$ is expressed by a context-free synchronized ranked-tree language, and

[4] The size of a regular expression $E$ is the number of non-terminal occurrences in $E$.
[5] http://www.w3.org/TR/xhtml1/dtds.html

using join and projection, we get $G_1$. By transforming $G_1$ into an unranked-tree grammar, we get $G'$. This method is complex, and gives complex grammars.

Another way to compute $G'$ could be the following. For each rule $A \to a[E]$ in $G$ we add the *collapsing rule* $A \to E$. The resulting grammar $G_1$ generates $WI(L(G))$ indeed, but is not a hedge grammar: it is called *extended grammar*[6] in [11], and can be transformed into a context-free hedge grammar $G_2$ (without collapsing rules). Each hedge $H$ of $G_2$ is a context-free word language over non-terminals defined by a word grammar, and if we consider its closure by sub-word[7], we get a regular word language $H'$ defined by a regular expression [10]. Let $G'$ be the grammar obtained from $G_2$ by transforming every hedge in this way. Then $G'$ is a regular hedge grammar, and the language generated by $G'$ satisfies $L(G') = L(G_2) \cup L_2$ where $L_2 \subseteq WI(L(G_2))$ (because of sub-word closure of hedges). Moreover $L(G_2) = L(G_1) = WI(L(G))$. Then $L_2 \subseteq WI(L(G_2)) = WI(WI(L(G))) = WI(L(G))$. Therefore $L(G') = WI(L(G)) \cup L_2 = WI(L(G))$.

## References


1. Amavi, J.: Comparaison des langages d'arbres pour la substitution de services web (in french). Tech. Rep. RR-2010-13, LIFO, Université d'Orléans (2010)
2. Amavi, J., Bouchou, B., Savary, A.: On correcting XML documents with respect to a schema. Tech. Rep. 301, LI, Université de Tours (2012)
3. Amavi, J., Chabin, J., Halfeld Ferrari, M., Réty, P.: Weak inclusion for XML types. In: CIAA. vol. 6807, pp. 30–41. LNCS, Springer (2011)
4. Amavi, J., Chabin, J., Réty, P.: Weak inclusion for recursive XML types (full version). Tech. Rep. RR-2012-02, LIFO, Université d'Orléans, http://www.univ-orleans.fr/lifo/prodsci/rapports/RR/RR2012/RR-2012-02.pdf (2012)
5. Bille, P., Gørtz, I.L.: The tree inclusion problem: In optimal space and faster. In: Automata, Languages and Programming, 32nd ICALP. pp. 66–77 (2005)
6. Champavère, J., Guilleron, R., Lemay, A., Niehren, J.: Efficient inclusion checking for deterministic tree automata and DTDs. In: Int. Conf. Language and Automata Theory and Applications. LNCS, vol. 5196, pp. 184–195. Springer (2008)
7. Chen, Y., Shi, Y., Chen, Y.: Tree inclusion algorithm, signatures and evaluation of path-oriented queries. In: Symp. on Applied Computing. pp. 1020–1025 (2006)
8. Colazzo, D., Ghelli, G., Pardini, L., Sartiani, C.: Linear inclusion for XML regular expression types. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM. pp. 137–146. ACM Digital Library (2009)
9. Colazzo, D., Ghelli, G., Sartiani, C.: Efficient asymmetric inclusion between regular expression types. In: Proceeding of International Conference of Database Theory, ICDT. pp. 174–182. ACM Digital Library (2009)
10. Courcelle, B.: On constructing obstruction sets of words. Bulletin of the EATCS 44, 178–185 (June 1991)
11. Jacquemard, F., Rusinowitch, M.: Closure of hedge-automata languages by hedge rewriting. In: RTA. pp. 157–171 (2008)


[6] In [11], they consider automata, but by reversing arrows, we can get grammars.

[7] A sub-word of a word $w$ is obtained by removing symbols from $w$. For example, *abeg* is a sub-word of *abcdefgh*.

12. Kilpeläinen, P., Mannila, H.: Ordered and unordered tree inclusion. SIAM J. Comput. 24(2), 340–356 (1995)
13. Mani, M., Lee, D.: XML to relational conversion using theory of regular tree grammars. In: In VLDB Workshop on EEXTT. pp. 81–103. Springer (2002)
14. Martens, W., Neven, F., Schwentick, T.: Complexity of decision problems for simple regular expressions. In: Int. Symp. Mathematical Foundations of Computer Science, MFCS. pp. 889–900 (2004)
15. Richter, T.: A new algorithm for the ordered tree inclusion problem. In: Apostolico, A., Hein, J. (eds.) Combinatorial Pattern Matching, LNCS, vol. 1264, pp. 150–166. Springer (1997)