Efficient Inclusion for a Class of XML Types with Interleaving and Counting

Giorgio Ghelli[1], Dario Colazzo[2,⋆], and Carlo Sartiani[1]

[1] Dipartimento di Informatica - Università di Pisa - Italy
{ghelli,sartiani}@di.unipi.it
[2] Université Paris Sud, UMR CNRS 8623, Orsay F-91405 - France
dario.colazzo@lri.fr

Abstract. Inclusion between XML types is important but expensive, and is much more expensive when unordered types are considered. We prove here that inclusion for XML types with interleaving and counting can be decided in polynomial time in presence of two important restrictions: no element appears twice in the same content model, and Kleene star is only applied to disjunctions of single elements.
Our approach is based on the transformation of each such type into a set of constraints that completely characterizes the type. We then provide a complete deduction system to verify whether the constraints of one type imply all the constraints of another one.

1 Introduction

XML schemas are an essential tool for the robustness of applications that involve XML data manipulation, transformation, integration, and, crucially, data exchange. To solve any static analysis problem that involves such types one must first be able to reason about their inclusion and equivalence.

XML schema languages are designed to describe ordered data, but they usually offer some (limited) support to deal with cases where the order among some elements is not constrained. These "unordered" mechanisms bring the language out of the well-understood realm of tree-grammars and tree-automata, and have been subject to little foundational study, with the important exception of a recent work by Gelade, Martens, and Neven [1]. Here, the authors study a wide range of schema languages, and show that the addition of interleaving and counting operators raises the complexity of inclusion checking from PSPACE (or EXPTIME, for Extended DTDs) to EXPSPACE. These are completeness results, hence this is really bad news. A previous result in [2] had already shown that the inclusion of Regular Expressions with interleaving alone is complete in EXPSPACE, hence showing that counting is not essential for the high cost. The paper [1] concludes with: "It would therefore be desirable to find robust subclasses for which the basic decision problems are in PTIME". Such subclasses could be used either to design a new schema language, or to design adaptive algorithms, that use the PTIME algorithm when possible, and resort to the full algorithm when needed. To this aim, it is important that (i) the subclass covers large classes of XML types used in practice, (ii) it is easy to verify whether a schema belongs to the subclass.

⋆ Work of this author was partially funded by the French ACI young researcher project "WebStand".

Our Contribution In this paper we define a class of XML types with interleaving and numerical constraints whose inclusion can be checked in polynomial time. These types are based on two restrictions that we impose on the Regular Expressions (REs) used to the define the element content models: each RE is conflict-free (or *single occurrence*) meaning that no symbol appears twice, and Kleene star is only applied to elements or to disjunctions of elements. These restrictions are severe, but, as shown in [3] and [4], they are actually met by most of the schemas that are used in practice.

Our approach is based on the transformation of each type into an equivalent set of constraints. Consider, for instance, the following string type $T = (a\,[1..3] \cdot b\,[2..2]) + c\,[1..2]$, and the following properties for a word w in T:

1. lower-bound: at least one of a, b and c appears in w;
2. cardinality: if a is in w, it appears 1, 2 or 3 times; if b is there, it appears twice; if c is there, it appears once or twice;
3. upper-bound: no symbol out of $\{a, b, c\}$ is in w;
4. exclusion: if one of a, b is in w, then c is not, and if c is in w then neither of a, b is in w;
5. co-occurrence: if a is in w, then b is in w, and vice versa;[3]
6. order: no occurrence of a may follow an occurrence of b.

It is easy to see that every w in T enjoys all of them. We will prove here that the opposite implication is true as well: every word that satisfies the six properties is indeed in T, i.e., that constraint set is *complete* for T.

We will generalize this observation, and will associate a complete set of constraints, in the six categories above, to any conflict-free type (we will actually encode exclusion constraints as order constraints.) We will then define a polynomial algorithm to verify whether, given T and U, the constraints of T imply those for U, so that T is included in U. We will formalize the constraints using a simple ad-hoc logic. We will describe the constraint implication algorithm by first giving a sound and complete constraint deduction system, and then giving an algorithm that exploits the deduction system.

The ability to transform a type into a complete set of constraints expressed in a limited variable-free logic is used here to design an efficient inclusion algorithm. We believe that it can also be exploited for many related tasks, such as PTIME membership checking (which is NP-complete for REs with interleaving), and path containment under a DTD. Quite surprisingly, binary type intersection, which is usually simpler than type inclusion, turns out in this case to be NP-hard; the constraint-based approach was important in our discovery of the proof that we present here.

Paper Outline The paper is structured as follows. Section 2 describes the data model, the type language, and the constraint language we are using. Section 3 shows how types can be characterized in terms of constraints, and how inclusion can be encoded in terms of constraint implication. Section 4 describes a deduction system for type constraints. Section 5, then, sketches a polynomial time algorithm for deciding type inclusion based on the deduction system of Section 4. In Section 6 we show that intersection is NP-hard. In Sections 7 and 8, finally, we briefly revise some related works and draw our conclusions.

[3] The term *co-occurrence constraint* has an unrelated meaning in [5]; we use it as in [6].

2 Type Language and Constraint Language

2.1 The Type Language

Gelade, Martens, and Neven showed that, if inclusion for a given class of regular expressions with interleaving and numerical constraints is in the complexity class $\mathcal{C}$, and $\mathcal{C}$ is *closed under positive reductions* (a property enjoyed by PTIME), then the complexity of inclusion for DTDs and single-type EDTDs that use the same class of regular expressions is in $\mathcal{C}$ too [7, 1]. Hence, we can focus our study on a class of regular expression over strings, and our PTIME result will immediately imply the same complexity for the inclusion problem of the corresponding classes of DTDs and single-type EDTDs. Single-type EDTDs are the theoretical counterpart of XML Schema definitions (see [1]).

We adopt the usual definitions for string concatenation $w_1 \cdot w_2$, and for the concatenation of two languages $L_1 \cdot L_2$. The *shuffle*, or *interleaving*, operator $w_1 \& w_2$ is also standard, and is defined as follows.

Definition 1 **($v\&w,\ L_1\&L_2$).** *The shuffle set of two words $v, w \in \Sigma^*$, or two languages $L_1, L_2 \subseteq \Sigma^*$, is defined as follows; notice that each v_i or w_i may be the empty string ϵ.*

$$\begin{array}{ll} v\&w & =_{def} \{v_1 \cdot w_1 \cdot \ldots \cdot v_n \cdot w_n \\ & \qquad \mid v_1 \cdot \ldots \cdot v_n = v,\ w_1 \cdot \ldots \cdot w_n = w,\ v_i \in \Sigma^*,\ w_i \in \Sigma^*,\ n > 0\} \\ L_1\&L_2 & =_{def} \bigcup_{w_1 \in L_1,\ w_2 \in L_2} w_1\&w_2 \end{array}$$

Example 1. $(ab)\&(XY)$ contains the permutations of $abXY$ where a comes before b and X comes before Y:

$$(ab)\&(XY) = \{abXY, aXbY, aXYb, XabY, XaYb, XYab\}$$

When $v \in w_1\&w_2$, we say that v is a shuffle of w_1 and w_2; for example, $w_1 \cdot w_2$ and $w_2 \cdot w_1$ are shuffles of w_1 and w_2.

We define $\mathbb{N}_* = \mathbb{N} \cup \{*\}$, and extend the standard order among naturals with $n \leq *$ for each $n \in \mathbb{N}_*$. We consider the following type language for strings over an alphabet Σ, where $a \in \Sigma$, $m \in \mathbb{N} \setminus \{0\}$, $n \in \mathbb{N}_* \setminus \{0\}$, and $n \geq m$ (please notice the specific domains for m and n):[4]

$$T ::= \epsilon \ \mid\ a\,[m..n] \ \mid\ T + T \ \mid\ T \cdot T \ \mid\ T\&T$$

Note that expressions like $a\,[0..n]$ are not allowed due to the condition on m; of course, the type $a\,[0..n]$ can be equivalently represented by $a\,[1..n] + \epsilon$.

Our type system generalizes Kleene star to counting, but it only allows symbols to be counted, so that, for example, $(a \cdot b)*$ cannot be expressed. However, it has been found that DTDs and XSD schemas use Kleene star almost exclusively as $a*$ or as $(a + \ldots + z)^*$ (see [3]), which can be easily expressed in our system as: $(a^* \& \ldots \& z^*)$, where a^* abbreviates $(a\,[1..*] + \epsilon)$. The *simple expressions* studied in [3] are a subclass of what can be expressed with our approach, and [3] measured a 97% fraction of XSD schemas with simple expressions only.

[4] We call them "types" because of our background, but they are actually a specific family of REs with interleaving, counting, and some restrictions.

Moreover, most of the non-simple expressions that they present are also easy to express in our system. Chain Regular Expressions [4] can also be expressed with our approach (see Section 7).[5]

Definition 2 $(S(w), S(T), Atoms(T))$**.** *For any string w, $S(w)$ is the set of all symbols appearing in w. For any type T, $Atoms(T)$ is the set of all atoms $a\,[m..n]$ appearing in T, and $S(T)$ is the set of all symbols appearing in T.*

Semantics of types is defined as follows.

$$\begin{aligned}
[\![\epsilon]\!] &= \{\epsilon\} \\
[\![a\,[m..n]]\!] &= \{w \mid S(w) = \{a\},\ |w| \geq m,\ |w| \leq n\} \\
[\![T_1 + T_2]\!] &= [\![T_1]\!] \cup [\![T_2]\!] \\
[\![T_1 \cdot T_2]\!] &= [\![T_1]\!] \cdot [\![T_2]\!] \\
[\![T_1 \& T_2]\!] &= [\![T_1]\!] \& [\![T_2]\!]
\end{aligned}$$

We will use $\otimes$ to range over $\cdot$ and & when we need to specify common properties, such as, for example: $[\![T \otimes \epsilon]\!] = [\![\epsilon \otimes T]\!] = [\![T]\!]$.

In this system, no type is empty. Some types contain the empty string ϵ, and are characterized as follows ($\mathrm{N}(T)$ is read as "T is nullable").

Definition 3. *$\mathrm{N}(T)$ is a predicate on types, defined as follows:*

$$\begin{aligned}
\mathrm{N}(\epsilon) &= \mathit{true} \\
\mathrm{N}(a\,[m..n]) &= \mathit{false} \\
\mathrm{N}(T + T') &= \mathrm{N}(T) \ \mathit{or}\ \mathrm{N}(T') \\
\mathrm{N}(T \otimes T') &= \mathrm{N}(T) \ \mathit{and}\ \mathrm{N}(T')
\end{aligned}$$

Lemma 1. *$\epsilon \in [\![T]\!]$ iff $\mathrm{N}(T)$.*

We can now define the notion of *conflict-free* types.

Definition 4 (Conflict-free types). *Given a type T, T is* conflict-free *if for each subexpression $(U + V)$ or $(U \otimes V)$: $S(U) \cap S(V) = \emptyset$.*

Equivalently, a type T is conflict-free if, for any two distinct subterms $a\,[m..n]$ and $a'\,[m'..n']$ that occur in T, a is different from a'.

Example 2. Consider the following type: $(a\,[1..1]\,\&b\,[1..1]) + (a\,[1..1]\,\&c\,[1..1])$. This type generates the language $\{ab, ba, ac, ca\}$. This type is not conflict-free, since $S(a\,[1..1]\,\&b\,[1..1]) \cap S(a\,[1..1]\,\&c\,[1..1]) = \{a\} \neq \emptyset$.

Consider now $a\,[1..1]\,\&(b\,[1..1] + c\,[1..1])$; it generates the same language, but is conflict-free since $a\,[1..1]$ and $(b\,[1..1] + c\,[1..1])$ have no common symbols.

Conflict-free DTDs have been considered many times before, because of their good properties and because of the high percentage of actual schemas that satisfy this constraint (see Section 7).

Hereafter, we will silently assume that every type is conflict-free, although some of the properties we specify are valid for any type.

[5] We are only discussing here our Kleene-star restriction, ignoring conflict-freedom for a moment.

2.2 The Constraint Language

We verify inclusion between T and U by translating them into constraint sets C_T and C_U and by then verifying that C_T implies C_U. Constraints are expressed using the following logic, where $a, b \in \Sigma$ and $A, B \subseteq \Sigma$, $m \in \mathbb{N}\backslash\{0\}$, $n \in \mathbb{N}_*\backslash\{0\}$, and $n \geq m$:

$$F \ ::= \ A^+ \ \mid \ A^+ \Rightarrow B^+ \ \mid \ a?[m..n] \ \mid \ \text{upper}(A) \ \mid \ a \prec b \ \mid \ F \wedge F' \ \mid \ \mathbf{true}$$

Satisfaction of a constraint F by a word w, written $w \models F$, is defined as follows.[6]

$$\begin{aligned}
w \models A^+ &\Leftrightarrow S(w) \cap A \neq \emptyset, \text{ i.e. some } a \in A \text{ appears in } w \\
w \models A^+ \Rightarrow B^+ &\Leftrightarrow w \not\models A^+ \text{ or } w \models B^+ \\
w \models a?[m..n] \ (n \neq *) &\Leftrightarrow \text{if } a \text{ appears in } w, \\
& \text{ then it appears at least } m \text{ times and at most } n \text{ times} \\
w \models a?[m..*] &\Leftrightarrow \text{if } a \text{ appears in } w, \text{ then it appears at least } m \text{ times} \\
w \models \text{upper}(A) &\Leftrightarrow S(w) \subseteq A \\
w \models a \prec b &\Leftrightarrow \text{there is no occurrence of } a \text{ in } w \text{ that follows} \\
& \text{ an occurrence of } b \text{ in } w \\
w \models F_1 \wedge F_2 &\Leftrightarrow w \models F_1 \text{ and } w \models F_2 \\
w \models \mathbf{true} &\Leftrightarrow \text{always}
\end{aligned}$$

We will also use $A^+ \Rightarrow \mathbf{true}$ as an alternative notation for $\mathbf{true}$. This should not be too confusing, since the two things are logically equivalent, and will simplify the notation for one crucial definition.

The atomic formulas are best understood through some examples.

$dab \models \{a,b,c\}^+$	$ca \models \{a,b,c\}^+$	$\epsilon \not\models A^+$	$w \not\models \emptyset^+$
$dab \not\models \text{upper}(\{a,b,c\})$	$ca \models \text{upper}(\{a,b,c\})$	$\epsilon \models \text{upper}(A)$	$\epsilon \models \text{upper}(\emptyset)$
$ca \models b?[2..*]$	$cba \not\models b?[2..*]$	$cbab \models b?[2..*]$	$bcbab \models b?[2..*]$
$ca \models a \prec b$	$caba \not\models a \prec b$	$aacb \models a \prec b$	$\epsilon \models a \prec b$

Observe that A^+ is monotone, i.e. $w \models A^+$ and w is a subword of w' imply that $w' \models A^+$, while upper(A) and $a \prec b$ are anti-monotone.

We use the following abbreviations:

$$\begin{aligned}
a^+ &=_{def} \ \{a\}^+ \\
a \prec\succ b &=_{def} \ (a \prec b) \wedge (b \prec a) \\
A \prec B &=_{def} \ \bigwedge_{a \in A, b \in B} a \prec b \\
A \prec\succ B &=_{def} \ \bigwedge_{a \in A, b \in B} a \prec\succ b
\end{aligned}$$

[6] Notice that $A^+ \Rightarrow b^+$ differs from the sibling constraint $A \Downarrow b$ of [8], since $A^+ \Rightarrow b^+$ means “if one symbol of A is in w then b is in w”, while $A \Downarrow b$ means “if *all* symbols of A are in w then b is in w”.

The next propositions specify that $A \prec\succ B$ encodes mutual exclusion between sets of symbols.

Proposition 1. *$w \models a \prec\succ b \;\Leftrightarrow\;$ a and b and are not both in $S(w)$*

Proposition 2. $w \models A \prec\succ B \;\Leftrightarrow w \not\models A^+ \wedge B^+$

Definition 5. *$a \in S(F)$ if one of the following is a subterm of F: $a?[m..n]$, $a \prec b$, A^+, $A^+ \Rightarrow B^+$, $\mathrm{upper}(A)$, where, in the last three cases, $a \in A$ or $a \in B$.*

The atomic operators are all mutually independent: only A^+ can force the presence of a symbol independently of any other, only $A^+ \Rightarrow B^+$ induces a positive correlation between the presence of two symbols, only $a?[m..n]$ can count, only $\mathrm{upper}(A)$ is affected by the presence of a symbol that is not in $S(F)$, and only $a \prec b$ is affected by order. However, combinations of the atomic operators can be mutually related (see Proposition 2, for example).

3 Characterization of Types as Constraints

3.1 Constraint Extraction

We first extend satisfaction from words to types, as follows.

Definition 6. $T \models F \;\Leftrightarrow\; \forall w \in [\![T]\!].\ w \models F$

To each type T, we associate a formula $S^+(T)$ that tests for the presence of one of its symbols, as follows.

Definition 7. $S^+(T) \;=\; (S(T))^+$

The $S^+(T)$ formula allows us to express the exclusion constraints associated with the type $T_1 + T_2$: if $S(T_1) \cap S(T_2) = \emptyset$ and $w \in [\![T_1 + T_2]\!]$, then $w \models S^+(T_1)$ is sufficient to deduce that $w \models \neg S^+(T_2)$, i.e. $T_1 + T_2 \models \neg(S^+(T_1) \wedge S^+(T_2))$ (which we actually express as $T_1 + T_2 \models S(T_1) \prec\succ S(T_2)$).

We would like to have a dual constraint for $T_1{\cdot}T_2$, such as $T_1{\cdot}T_2 \models S^+(T_1) \Rightarrow S^+(T_2)$, but this does not hold in case T_2 contains the empty string; we will prove that this weaker constraint holds: $T_1{\cdot}T_2 \models$ if not $\mathrm{N}(T_2)$ then $S^+(T_1) \Rightarrow S^+(T_2)$.

The condition "if not $\mathrm{N}(T)$ then ..." will be expressed using the $\mathit{SIf}(T)$ notation that we define below.

We can now endow a type T with five sets of constraints. We start with the lower-bound, cardinality, and upper-bound constraints (we introduced this terminology in Section 1).

Definition 8 (Flat constraints).

$$\begin{array}{rrll}
\textit{Lower-bound:} & \mathit{SIf}(T) =_{def} & S^+(T) & \textit{if not } \mathrm{N}(T) \\
 & \mathit{SIf}(T) =_{def} & \mathbf{true} & \textit{if } \mathrm{N}(T) \\
\textit{Cardinality:} & \mathrm{ZeroMinMax}(T) =_{def} & \bigwedge_{a[m..n] \in Atoms(T)} a?[m..n] & \\
\textit{Upper-bound:} & \mathrm{upperS}(T) =_{def} & \mathrm{upper}(S(T)) & \\
\textit{Flat constraints:} & \mathcal{FC}(T) =_{def} & \mathit{SIf}(T) \wedge \mathrm{ZeroMinMax}(T) \wedge \mathrm{upperS}(T) &
\end{array}$$

We can now add co-occurrence, order, and exclusion constraints, whose definition is inductive over the type structure. Exclusion constraints are actually encoded as order constraints.

Definition 9 (Nested constraints).

$$
\begin{array}{rl}
\textit{Co-occurrence:} & \\
\mathcal{CC}(T_1 + T_2) =_{def} & \mathcal{CC}(T_1) \wedge \mathcal{CC}(T_2) \\
\mathcal{CC}(T_1 \otimes T_2) =_{def} & (S^+(T_1) \Rightarrow \mathit{SIf}(T_2)) \wedge (S^+(T_2) \Rightarrow \mathit{SIf}(T_1)) \wedge \\
 & \mathcal{CC}(T_1) \wedge \mathcal{CC}(T_2) \\
\mathcal{CC}(\epsilon) =_{def} \mathcal{CC}(a\,[m..n]) =_{def} & \mathbf{true}
\end{array}
$$

$$
\begin{array}{rl}
\textit{Order and exclusion:} & \\
\mathcal{OC}(T_1 + T_2) =_{def} & (S(T_1) \prec\succ S(T_2)) \wedge \mathcal{OC}(T_1) \wedge \mathcal{OC}(T_2) \\
\mathcal{OC}(T_1 \& T_2) =_{def} & \mathcal{OC}(T_1) \wedge \mathcal{OC}(T_2) \\
\mathcal{OC}(T_1 \cdot T_2) =_{def} & (S(T_1) \prec S(T_2)) \wedge \mathcal{OC}(T_1) \wedge \mathcal{OC}(T_2) \\
\mathcal{OC}(\epsilon) =_{def} \mathcal{OC}(a\,[m..n]) =_{def} & \mathbf{true}
\end{array}
$$

$$
\begin{array}{rl}
\textit{Nested constraints:} & \\
\mathcal{NC}(T) =_{def} & \mathcal{CC}(T) \wedge \mathcal{OC}(T)
\end{array}
$$

Notice that, when $\mathrm{N}(T_2)$ is *true*, $S^+(T_1) \Rightarrow \mathit{SIf}(T_2)$ is just **true**, because ($A^+ \Rightarrow$ **true**) is **true**, by definition. This notation is helpful to visualize, for example, the fact that $S^+(T_1)$ and $S^+(T_1) \Rightarrow \mathit{SIf}(T_2)$ imply $\mathit{SIf}(T_2)$.

3.2 Correctness and Completeness of Constraints

We plan to prove the following theorem, that specifies that the constraint system completely captures the semantics of conflict-free types.

Theorem 1. *Given a conflict-free type T, it holds that:*

$$w \in [\![T]\!] \;\Leftrightarrow\; w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$$

We first prove that constraints are complete, i.e., whenever w satisfies all the five groups of constraints associated with T, then $w \in [\![T]\!]$.

Proposition 3 (ZeroMinMax(T)).

$$
\begin{array}{l}
w \models \mathrm{ZeroMinMax}(T_1 + T_2) \Rightarrow w \models \mathrm{ZeroMinMax}(T_1) \wedge \mathrm{ZeroMinMax}(T_2) \\
w \models \mathrm{ZeroMinMax}(T_1 \otimes T_2) \Rightarrow w \models \mathrm{ZeroMinMax}(T_1) \wedge \mathrm{ZeroMinMax}(T_2)
\end{array}
$$

Definition 10. *We define $w|_{S(T)}$ as the string obtained from w by removing all the symbols that are not in $S(T)$.*

We can now prove the crucial completeness theorem.

Theorem 2 (Completeness of constraints).

$$w \models (\mathcal{FC}(T) \wedge \mathcal{NC}(T)) \quad \Rightarrow \quad w \in [\![T]\!]$$

Proof. For the sake of convenience, we will use ZMM-SIf(T) as a shortcut for ZeroMinMax$(T) \wedge \mathit{SIf}(T)$, so that we can rewrite the thesis to prove as

$$w \models (\text{upperS}(T) \wedge \text{ZMM-SIf}(T) \wedge \mathcal{NC}(T)) \quad \Rightarrow \quad w \in [\![T]\!]$$

We prove the following fact, by case inspection and structural induction on T.

$$w \models (\text{ZMM-SIf}(T) \wedge \mathcal{NC}(T)) \quad \Rightarrow \quad w|_{S(T)} \in [\![T]\!]$$

The theorem follows because $w \models \text{upperS}(T)$ implies that $w = w|_{S(T)}$.

We first observe that $w|_{S(T)} = \epsilon$ and $w \models \mathit{SIf}(T)$ imply the thesis $w|_{S(T)} \in [\![T]\!]$. Indeed, $w|_{S(T)} = \epsilon$ implies that $w \not\models S^+(T)$, hence, the hypothesis $w \models \mathit{SIf}(T)$ implies that $\text{N}(T)$ is true, which in turn implies that $\epsilon \in [\![T]\!]$, i.e. $w|_{S(T)} \in [\![T]\!]$.

Having dealt with the $w|_{S(T)} = \epsilon$ case, in the following we assume that $w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$, where $n \neq 0$.

$\mathbf{T = \epsilon}$:

Trivial, as $w|_{S(\epsilon)} = \epsilon$ and $\epsilon \in [\![\epsilon]\!]$.

$\mathbf{T = a\,[m..n]}$:

Since $\text{N}(T)$ is false, $w \models \text{ZMM-SIf}(T)$ implies that $w \models \text{ZeroMinMax}(T) \wedge S^+(T)$, i.e., $w \models \text{ZeroMinMax}(a\,[m..n]) \wedge a^+$, i.e., $w \models a?[m..n] \wedge a^+$, hence $w|_{S(a[m..n])} \in [\![a\,[m..n]]\!]$.

$\mathbf{T = T_1 + T_2}$:

Let $w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$, and assume,without loss of generality, that $a_1 \in S(T_1)$.

By hypothesis we have that $w \models \text{ZMM-SIf}(T_1 + T_2) \wedge (S(T_1) \prec\succ S(T_2)) \wedge \mathcal{NC}(T_1) \wedge \mathcal{NC}(T_2)$. As $w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$ with $a_1 \in S(T_1)$, we also have that $w \models S^+(T_1)$.

This implies that $w \models \mathit{SIf}(T_1)$ (by definition of $\mathit{SIf}()$) and that $w \not\models S^+(T_2)$ (by Proposition 2). This, in turn, implies $w|_{S(T_1+T_2)} = w|_{S(T_1)}$ (*). By Proposition 3 and by $w \models \text{ZMM-SIf}(T_1 + T_2)$ we obtain that $w \models \text{ZeroMinMax}(T_1)$. Putting all together, $w \models \text{ZMM-SIf}(T_1) \wedge \mathcal{NC}(T_1)$.

By induction we have that $w|_{S(T_1)} \in [\![T_1]\!]$; hence, by (*), we get $w|_{S(T_1+T_2)} \in [\![T_1]\!]$, which, in turn, implies that $w|_{S(T_1+T_2)} \in [\![T_1 + T_2]\!]$.

$\mathbf{T = T_1 \cdot T_2}$:

We have two possible cases:

1. $w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$ and $a_1 \in S(T_1)$;
2. $w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$ and $a_1 \in S(T_2)$.

Case 1 ($w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$ and $a_1 \in S(T_1)$).

By hypothesis we have that:

$$\begin{aligned} w \models \text{ZMM-SIf}(T_1 \cdot T_2) &\wedge (S^+(T_1) \Rightarrow \mathit{SIf}(T_2)) \\ &\wedge (S^+(T_2) \Rightarrow \mathit{SIf}(T_1)) \\ &\wedge (S(T_1) \prec S(T_2)) \\ &\wedge \mathcal{NC}(T_1) \wedge \mathcal{NC}(T_2) \end{aligned}$$

Since $w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$ with $a_1 \in S(T_1)$, we have that $w \models S^+(T_1)$, which implies that $w \models \mathit{SIf}(T_1)$ (by definition of $\mathit{SIf}()$) and that $w \models \mathit{SIf}(T_2)$ (by hypothesis). By Proposition 3 we conclude that $w \models \text{ZMM-SIf}(T_1) \wedge \text{ZMM-SIf}(T_2)$.

Let us define $w_1 = w|_{S(T_1)}$ and $w_2 = w|_{S(T_2)}$. As $w \models \mathcal{NC}(T_1) \wedge \mathcal{NC}(T_2)$, by induction we obtain that $w_1 \in [\![T_1]\!]$ and $w_2 \in [\![T_2]\!]$.

By conflict-freedom, w_1 and w_2 do not contain any common symbols, hence, from the constraint $S(T_1) \prec S(T_2)$ we obtain that each symbol of w_1 precedes each symbol of w_2 in w. As a consequence, $w|_{S(T_1 \cdot T_2)} = w|_{S(T_1)} \cdot w|_{S(T_2)} = w_1 \cdot w_2$. Thus, $w|_{S(T_1 \cdot T_2)} \in [\![T_1 \cdot T_2]\!]$.

Case 2 ($w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$ and $a_1 \in S(T_2)$).

By hypothesis we have that:

$$\begin{aligned} w \models \text{ZMM-SIf}(T_1 \cdot T_2) &\wedge (S^+(T_1) \Rightarrow \mathit{SIf}(T_2)) \\ &\wedge (S^+(T_2) \Rightarrow \mathit{SIf}(T_1)) \\ &\wedge (S(T_1) \prec S(T_2)) \\ &\wedge \mathcal{NC}(T_1) \wedge \mathcal{NC}(T_2) \end{aligned}$$

Since $w|_{S(T)} = a_1 \cdot \ldots \cdot a_n$ and $a_1 \in S(T_2)$, we obtain that $w \models S^+(T_2)$, which implies that $w \models \mathit{SIf}(T_1)$ (by hypothesis) and that $w \models \mathit{SIf}(T_2)$ (by definition). By Proposition 3 we conclude that $w \models \text{ZMM-SIf}(T_1) \wedge \text{ZMM-SIf}(T_2)$. As $w \models \mathcal{NC}(T_1) \wedge \mathcal{NC}(T_2)$, by induction we obtain that $w|_{S(T_1)} \in [\![T_1]\!]$ and $w|_{S(T_2)} \in [\![T_2]\!]$.

$w \models (S(T_1) \prec S(T_2))$ and $a_1 \in S(T_2)$ imply that $w \not\models S^+(T_1)$, i.e., $w|_{S(T_1)} = \epsilon$. Hence, $w|_{S(T_1 \cdot T_2)} = w|_{S(T_2)} = \epsilon \cdot w|_{S(T_2)} = w|_{S(T_1)} \cdot w|_{S(T_2)}$. Hence, by $w|_{S(T_1)} \in [\![T_1]\!]$ and $w|_{S(T_2)} \in [\![T_2]\!]$, we conclude that $w|_{S(T_1 \cdot T_2)} \in [\![T_1 \cdot T_2]\!]$.

$\mathbf{T = T_1 \& T_2}$: similar, but simpler. □

In order to prove soundness, we use the following lemma that specifies that the value of any formula F over w does not change if any letter a that is not in $S(F)$ is added or deleted from w, provided that F does not contain the upper(A) operator. Recall that upper(A) is only used to express upper-bound constraints.

Soundness is stated by Theorem 3 below; for reasons of space, we omit the proof and refer the reader to [9] for further details.

Theorem 3 (Soundness).

$$w \in [\![T]\!] \Rightarrow w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$$

Corollary 1. *For any conflict-free type T:*

$$w \in [\![T]\!] \Leftrightarrow w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$$

4 Deduction System

We introduce here a deduction system as a first step for the formalization of a constraint implication algorithm. The system is partitioned into two separate judgements, $\vdash_{cc}$ and $\vdash_{oc}$, for deducing co-occurrence and order constraints, This deduction system is not complete in general, but is powerful enough to decide type inclusion (Theorem 10).

Each judgement $\vdash_x$ will be defined, by a set of deduction rules with shape $F_1 \wedge \ldots \wedge F_n \vdash_x F$; the notation $F_1 \wedge \ldots \wedge F_m \vdash_x F'_1 \wedge \ldots \wedge F'_n$ also means that $F'_1 \ldots F'_n$ can be deduced from $F_1 \ldots F_m$ through the repeated application of the corresponding deduction rules.

From now on, we will often identify a set formula A^+ with the symbol set A; the use will clarify the distinction. Hence, we will use metavariables A and B to range over subsets of Σ and also over set-formulas.

For reasons of space, we omit the proofs of the results of this section and refer the reader to [9] for further details.

4.1 Co-Occurrence Deduction

We start by defining a deduction system that will be used for co-occurrence constraints of the form $A^+ \Rightarrow B^+$. The *R-T-A* rules correspond to the *Armstrong system* used to deduce functional constraints [10], after left-hand-sides are switched with right-hand-sides. We will denote set union as juxtaposition: $AB =_{def} A \cup B$ and $aA =_{def} \{a\} \cup A$. The *False* rule specifies that, if an upper-bound constraint excludes a, then we can deduce any B from the impossible presence of a.

$$\begin{array}{rll}
R: & & \vdash_{cc} A \Rightarrow AB \\
T: & (A \Rightarrow B) \wedge (B \Rightarrow C) & \vdash_{cc} A \Rightarrow C \\
A: & A \Rightarrow B & \vdash_{cc} AC \Rightarrow BC \\
False: a \notin A: & \mathrm{upper}(A) & \vdash_{cc} a \Rightarrow B
\end{array}$$

The backward correspondence between the *R-T-A* rules and Armstrong axioms can be easily explained: a functional dependency $X_1, \ldots, X_n \Rightarrow Y_1, \ldots, Y_m$ over a relation R is an implication of conjunctions $\forall t, u \in R.(P(X_1) \wedge \ldots \wedge P(X_n)) \Rightarrow (P(Y_1) \wedge \ldots \wedge P(Y_m))$, where $P(X)$ is $t.X = u.X$. An implication $\{a_1, \ldots, a_n\}^+ \Rightarrow \{b_1, \ldots, b_m\}^+$ is an implication of disjunctions $\forall w.(a_1 \in S(w) \vee \ldots \vee a_n \in S(w)) \Rightarrow (b_1 \in S(w) \vee \ldots \vee b_m \in S(w))$, that becomes a backward implication of conjunctions by contraposition: $(Q(b_1) \wedge \ldots \wedge Q(b_m)) \Rightarrow (Q(a_1) \wedge \ldots \wedge Q(a_n))$, where $Q(a)$ is $a \notin S(w)$. Hence, co-occurrence constraints can be manipulated as functional dependencies, after the two sides have been switched.

From these rules we can derive some additional rules, shown below.

$$\begin{array}{rll}
Down: A' \subseteq A: & A \Rightarrow B & \vdash_{cc} A' \Rightarrow B \\
Up: B \subseteq B': & A \Rightarrow B & \vdash_{cc} A \Rightarrow B' \\
Union: & (A \Rightarrow C) \wedge (B \Rightarrow C) & \vdash_{cc} AB \Rightarrow C \\
Decomp: & AB \Rightarrow C & \vdash_{cc} A \Rightarrow C
\end{array}$$

These rules are trivially sound.

Theorem 4 (Soundness of co-occurrence deduction). *If $w \models F$ and $F \vdash_{cc} F'$, then $w \models F'$. If $T \models F$ and $F \vdash_{cc} F'$, then $T \models F'$.*

The following lemma contains the core of the completeness proof.

Lemma 2. *For each type T and for each symbol $a \in S^+(T)$, if $T \models a \Rightarrow B$, then $\mathcal{CC}(T) \vdash_{cc} a \Rightarrow B$, using the R-T-A rules only.*

Theorem 5 (Completeness of co-occurrence deduction for subtypes). *If $[\![T_1]\!] \subseteq [\![T_2]\!]$, then* $\mathrm{upperS}(T_1) \wedge \mathcal{CC}(T_1) \vdash_{cc} \mathcal{CC}(T_2)$.

4.2 Order Deduction

Order constraints can be deduced from upper bounds, as follows.

$$\begin{array}{l} FalseL : b \notin A : \ \text{upper}(A) \vdash_{oc} b \prec b' \\ FalseR : b \notin A : \ \text{upper}(A) \vdash_{oc} b' \prec b \end{array}$$

Theorem 6 (Soundness of order deduction). *If $w \models F$ and $F \vdash_{oc} F'$, then $w \models F'$. If $T \models F$ and $F \vdash_{oc} F'$, then $T \models F'$.*

Lemma 3 (Completeness of order deduction). *If $a \neq b$ and $\{a, b\} \subseteq S(T)$ and $T \models a \prec b$, then $\mathcal{OC}(T) \vdash_{oc} a \prec b$.*

Theorem 7 (Completeness of order deduction for subtypes). *If $[\![T_1]\!] \subseteq [\![T_2]\!]$, then $\text{upperS}(T_1) \wedge \mathcal{OC}(T_1) \vdash_{oc} \mathcal{OC}(T_2)$.*

4.3 Flat Constraints Deduction

Flat constraints are manipulated with a different approach. In this case, we check them together, and we directly discuss their soundness and completeness with respect to a pair of types. We first introduce a system to deduce whether the flat constraints of T_1 imply all the flat constraints of T_2.

Definition 11 ($T_1 \vdash_{flat} T_2$).

$$\begin{array}{l} T_1 \vdash_{flat} T_2 \ \Leftrightarrow_{def} \\ \quad (a?[m..n] \in Atoms(T_1) \Rightarrow \exists m' \leq m, n' \geq n.\ a\,[m'..n'] \in Atoms(T_2)) \\ \quad \wedge\ (\text{N}(T_1) \qquad\qquad\quad\ \Rightarrow \text{N}(T_2)) \end{array}$$

Checking all flat constraints together makes sense because the three of them, in a sense, just check inclusion of $Atoms(T_1)$ into $Atoms(T_2)$. But there is another strong reason: the design of a sound and complete deduction system for $SIf(T)$ alone is actually much trickier than expected, while the holistic check is simple, sound, and complete, for the three of them, as formalized below.

Theorem 8 (Soundness of $\vdash_{flat}$). *If $T_1 \vdash_{flat} T_2$, then:*

1. $T_1 \models SIf(T_2)$;
2. $T_1 \models \text{upperS}(T_2)$;
3. $T_1 \models \text{ZeroMinMax}(T_2)$.

Theorem 9 (Completeness of $\vdash_{flat}$). *If $[\![T_1]\!] \subseteq [\![T_2]\!]$, then $T_1 \vdash_{flat} T_2$.*

4.4 Correctness and Completeness of Inclusion Deduction

We can now state the final theorem.

Theorem 10 (Correctness and completeness of inclusion deduction).

$$\begin{array}{lcl} [\![T_1]\!] \subseteq [\![T_2]\!] & \Leftrightarrow & \text{upperS}(T_1) \wedge \mathcal{CC}(T_1) \vdash_{cc} \mathcal{CC}(T_2)\ \wedge \\ & & \text{upperS}(T_1) \wedge \mathcal{OC}(T_1) \vdash_{oc} \mathcal{OC}(T_2)\ \wedge \\ & & T_1 \vdash_{flat} T_2 \end{array}$$

5 Inclusion Checking

Theorem 10 proves that language inclusion among conflict-free string types can be decided through the deduction systems presented in the previous section. From this theorem we can derive an inclusion checking algorithm. The algorithm first verifies whether $T \vdash_{flat} U$, in time $O(n)$ in the size of T and U. The algorithm, then, verifies the deduction of co-occurrence constraints by a simple extension of the Beeri and Bernstein algorithm for functional constraints implication [10] (Section 5.1). The deduction for order constraints is much simpler: we essentially verify that each constraint of $\mathcal{OC}(U)$ either is in $\mathcal{OC}(T)$ or it involves a symbol that is not in $S(T)$ (Section 5.2).

In the following we will only sketch the basic principles of our algorithm; for more details, see [9].

5.1 Co-Occurrence Constraints

We present here an algorithm to verify whether upperS$(T) \wedge \mathcal{CC}(T) \vdash_{cc} \mathcal{CC}(U)$. To this aim, it invokes a "backward closure" algorithm for the U_i argument of each $S^+(U_j) \Rightarrow S^+(U_i)$ constraint generated by the occurrence of an $\otimes$ operator inside U. The "backward closure" of $S(U_i)$ with respect to $F = \mathcal{CC}(T)$ (TBackwardClose$(S(U_i))$) is defined as the maximal $R \subseteq S(T)$ such that $F \vdash_{cc} R \Rightarrow S(U_i)$, and is computed using a reversed version of the standard Beeri-Bernstein algorithm, which is correct and complete for deduction rules R, T, and A [10]. By Lemma 2, and by rules *Union* and *Decomp*, upperS$(T) \wedge \mathcal{CC}(T) \vdash_{cc} S^+(U_j) \Rightarrow S^+(U_i)$ iff $(S(U_j) \cap S(T)) \subseteq$ TBackwardClose$(S(U_i))$.

By a standard argument [10], the backward closure algorithm is linear in the total size of the rules. Since no symbol can appear in more than $2 * d_{\otimes}$ co-occurrence rules, where $d_{\otimes}$ is the nesting level of $\otimes$ operators, each closure invocation is in $O(n * d_{\otimes})$. Backward closure is invoked once, or less, for each argument of each $\otimes$ inside U, which means that the co-occurrence constraint algorithm is in $O(n * n * d_{\otimes})$, i.e. in $O(n^3)$.

In practice, we traverse U bottom up and we compute the T-closure of U subterms that are bigger and bigger. We can easily use dynamic programming in order to reuse the results of closure on the subterms to speed up the closure of a superterm. We do not study this optimization here.

5.2 Order Constraints

Order constraints correspond to the concatenation and union type operators. For each pair of leaves $a\,[m..n]$ and $b\,[m'..n']$ in the syntax tree of T, let $LCA_T[a, b]$ be their common ancestor that is farthest from the root (the *Lowest Common Ancestor*). For each a and b in $S(T)$, $a \prec\succ b \in \mathcal{OC}(T)$ iff $LCA_T[a, b]$ is labeled by $+$: the if direction is clear; for the only if direction, observe that any $+$ that is lower than the LCA is not a common ancestor, and any $+$ that is higher has both a and b below the same child. Similarly, $a \prec b \in \mathcal{OC}(T)$ iff $LCA_T[a, b] = +$ or a precedes b in T and $LCA_T[a, b] = \cdot$. As a consequence, upperS$(T) \wedge \mathcal{OC}(T) \vdash_{oc} \mathcal{OC}(U)$ iff, for each a and b in $S(U)$, such that a precedes b in U:

- if $LCA_U[a, b] = +$ then either $a \notin S(T)$ or $b \notin S(T)$ or $LCA_T[a, b] = +$;

- if $LCA_U[a,b] = \cdot$ then either $a \notin S(T)$ or $b \notin S(T)$ or $LCA_T[a,b] = +$ or ($LCA_T[a,b] = \cdot$ and a precedes b in T).

Hence, we can verify whether upperS$(T) \wedge \mathcal{OC}(T) \vdash_{oc} \mathcal{OC}(U)$ via the following algorithm. We first build an array $LCA_T[a,b]$ which associates each a and b in $S(T)$ with the operator that labels the LCA of a and b in T, and similarly for U; this can be done in linear time [11]. We then scan all the ordered pairs a, b of $S(U)$, checking the condition above, which can be done with $O(n^2)$ constant-time accesses to $LCA_T[\_,\_]$ and $LCA_U[\_,\_]$, which gives a $O(n^2)$ algorithm.

This inclusion-checking algorithm is presented here to prove that inclusion is in PTIME, but we do not expect it to be optimal. Specifically, in the crucial case of co-occurrence constraints, the set $\mathcal{CC}(T)$ has a very regular structure. For example, for any two constraints $L \Rightarrow R$ and $L' \Rightarrow R'$, if $R \cap R' \neq \emptyset$ then either $R \subset R'$ or $R' \subset R$, and similarly for L and L'. It seems plausible that better solutions could be achieved by exploiting this regularity.

6 Complexity of Intersection

Intersection for subclasses of RE corresponds to automata product, while inclusion corresponds to automata complement plus product, hence intersection is in general cheaper than inclusion. We show here that, for conflict-free types, things are quite different: while inclusion is in PTIME, intersection of two confict-free expressions is NP-hard. This result is quite surprising, and it suggests that it makes sense to study such types with an approach that is not based on automata.

Interestingly, NP-hardness does not depend on counting or Kleene star, but our proof depends crucially on the & operator.

Theorem 11. *Emptyness of the intersection of two conflict-free types is NP-hard, even if the types do not use counting and concatenation.*

Proof. (Hint) Consider m boolean variables $x_1, \ldots, x_m$ and a formula $\phi = (a_1^1 \vee a_1^2 \vee a_1^3) \wedge \ldots \wedge (a_n^1 \vee a_n^2 \vee a_n^3)$ where each literal a_j^i is either a variable x_l or a negated variable $\neg x_l$; Satisfiability of ϕ can be encoded as the intersection of two conflict-free types T_1 and T_2 as exemplified below. Both types have one symbol for each occurrence of a literal in ϕ, hence their size is linear in $|\phi|$.

$$\begin{aligned}
\phi\ &= (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)\\
T_1 &= (a_1^1 + a_1^2 + a_1^3)\ \&\ (a_2^1 + a_2^2 + a_2^3)\ \&\ (a_3^1 + a_3^2 + a_3^3)\ \&\ (a_4^1 + a_4^2 + a_4^3)\\
T_2 &= ((a_1^1?) + (a_2^1?\, \&\, a_4^1?))\ \&\ ((a_1^2?) + (a_3^1?))\\
&\qquad \&\ ((a_1^3?\, \&\, a_2^2?) + (\ a_3^2?\, \&\, a_4^2?))\ \&\ ((a_2^3?\, \&\, a_4^3?) + (a_3^3?))
\end{aligned}$$

ϕ is satisfiable iff it has a *witness*, i.e. a choice of literal instances, one from each factor, such that not two instances are contradictory, i.e. if x_i is chosen in a factor then $\neg x_i$ is not chosen in any other factor.

Any element of T_1 corresponds to a choice of literal instances, one from each factor. If the same list also belongs to T_2, then it is not contradictory. Hence, words in $[\![T_1]\!] \cap [\![T_2]\!]$ correspond to witnesses for ϕ.

7 Related Work

The properties of unordered XML types have been studied in several recent papers. In [12], the authors discuss the techniques and heuristics they used in implementing a type-checker, based on sheaves automata with Presburger arithmetic, for unordered XML types. The type language is an extension of the language we are considering here, and shares a similar restriction on the use of repetition types. The main purpose of the paper is to address scalability problems that naturally arise when working on XML types; as a consequence, they describe effective heuristics that improve scalability, but do not affect computational complexity.

Restrictions to RE languages that are similar to ours have been proposed many times. For example, conflict-free REs appear as "conflict-free DTDs" in the context of well-typed XML updates in [13], as "duplicate-free DTDs" in the context of path inclusion in [8], and as "single occurrence REs" in the context of DTD inference in [4]. The same restriction that we pose on Kleene-star can be found, for example, in [12]. Chain Regular Expressions (CHARE's) [4, 1] are also strictly related. They are defined as concatenations of factors, where each factor has a shape $(a_1 + \ldots + a_n)$, $(a_1 + \ldots + a_n)?$, $(a_1 + \ldots + a_n)^*$ or $(a_1 + \ldots + a_n)^+$. As we discussed in Section 2.1, the first three classes of factors can be easily expressed in our language, using counting and interleaving. Factors like $(a_1 + \ldots + a_n)^+$ cannot be expressed in our languages, but we could add them as a third class of base types $\{a_1, \ldots, a_n\}[1..*]$, besides $a\,[m..n]$ and ϵ, with $\mathcal{FC}(A[1..*]) = (A^+ \wedge \text{upper}(A))$ and $\text{N}(A[1..*]) = \textit{false}$. We did not consider these base types just for minimality. Simple expressions [3] have a more general syntax than CHAREs but the same expressive power, hence can still be managed through our approach.

We have cited many times paper [1], where the complexity of type inclusion is studied for many different dialects of REs with interleaving and/or counting, showing that inclusion complexity is almost invariably EXPSPACE-complete. In particular, this is shown to hold for chain-REs with counting, which are concatenations of CHARE factors, as defined above, and counting factors $(a_1 + \ldots + a_k)[m..n]$ (with $n \neq *$ and $m \geq 0$), with no interleaving operator. In a sense, this hints that the conflict-free restriction, rather than the Kleene-star restriction, is crucial for our PTIME result. In the same paper, the authors introduce a sublanguage of CHAREs with PTIME inclusion, but that fragment is quite trivial, since it only includes counting factors $(a_1 + \ldots + a_k)[m..n]$, with the further restriction that $m > 0$ and $n \neq *$, hence cannot express neither optionality nor unbounded repetition (neither * nor $^+$).[7]

8 Conclusions

Inclusion for REs with interleaving, counting, or both, is EXPSPACE-complete, even if we consider the restricted subclass of CHAREs (with counting) [2, 1]. This result easily extends to XML types featuring these operators. We have introduced here a restricted class of REs with interleaving and counting. Our

[7] Observe that our language can express optionality and repetition, but cannot express counting factors $(a_1 + \ldots + a_k)[m..n]$ with $k > 1$, unless $m = 0$ and $n = *$.

restriction is severe, but it seems to match reasonably well the measured features of actual DTDs and XSDs found on the web, and is extremely easy to define and verify. For this class of REs, we have proved that inclusion is in PTIME, a complexity that is surprising low, and trivially extends to DTDs and XSDs that use REs of this class for their content models. We have shown how to use classical algorithms to get a $O(n^3)$ upper bound, but we feel that this could be easy lowered. We also proved that intersection of two conflict-free types has not the same complexity as inclusion (unless P=NP) but is, quite surprisingly, NP-hard.

Our result is based on the transformation of our REs into sets of constraints which completely characterize the expressions and are easy to manipulate. We believe that this constraint-based approach could be fruitfully used for other analysis tasks, such as, for example, type normalization, path minimization under a DTD, or a polynomial membership algorithm.

Acknowledgments We thanks the anonymous referees for their constructive comments and suggestions.

References

1. Gelade, W., Martens, W., Neven, F.: Optimizing schema languages for XML: Numerical constraints and interleaving. In: ICDT. (2007)
2. Mayer, A.J., Stockmeyer, L.J.: Word problems-this time with interleaving. Inf. Comput. **115** (1994) 293–311
3. Bex, G.J., Neven, F., den Bussche, J.V.: DTDs versus XML schema: A practical study. In Amer-Yahia, S., Gravano, L., eds.: WebDB. (2004) 79–84
4. Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of concise DTDs from XML data. In Dayal, U., Whang, K.Y., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y.K., eds.: VLDB, ACM (2006) 115–126
5. Amer-Yahia, S., Cho, S., Lakshmanan, L.V.S., Srivastava, D.: Minimization of tree pattern queries. In: SIGMOD Conference. (2001) 497–508
6. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1: Structures Second Edition. Technical report, World Wide Web Consortium (2004) W3C Recommendation.
7. Martens, W., Neven, F., Schwentick, T.: Complexity of decision problems for simple regular expressions. In: MFCS. Volume 3153 of LNCS, Springer (2004) 889–900
8. Wood, P.T.: Containment for XPath fragments under DTD constraints. In: ICDT (2003) 300–314
9. Ghelli, G., Colazzo, D., Sartiani, C.: Efficient inclusion for a class of XML types with interleaving and counting. Technical report, Dipartimento di Informatica - Università di Pisa (2007)
10. Beeri, C., Bernstein, P.A.: Computational problems related to the design of normal form relational schemas. ACM Trans. Database Syst. **4** (1979) 30–59
11. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In Gonnet, G.H., Panario, D., Viola, A., eds.: LATIN. Volume 1776 of LNCS, Springer (2000) 88–94
12. Foster, J.N., Pierce, B.C., Schmitt, A.: A logic your typechecker can count on: Unordered tree types in practice. In: PLAN-X, informal proceedings. (2007)
13. Barbosa, D., Mendelzon, A.O., Libkin, L., Mignet, L., Arenas, M.: Efficient incremental validation of XML documents. In: ICDE, IEEE Computer Society (2004) 671–682