Typing and querying XML documents: some complexity bounds

Luc Segoufin
INRIA,France

Abstract

We study the complexity bound of validating XML documents, viewed as labeled unranked ordered trees, against various typing systems like DTDs, XML schemas, tree automata... We also consider query evaluation complexities for various fragments of XPath. For both problems, validation and query evaluation, we consider data and combined complexity bounds.

1 Introduction

The Extended Markup Language (XML) is emerging as the standard for data exchange on the Web. Many applications require processing large amounts of data in XML format using limited resources: CPU constraints or memory constraints. Such processing includes validating XML documents against a given type system (DTD, XML Schema ...) and querying XML documents. Efficient practical algorithms are therefore crucial (see for instance [21]). In order to evaluate the efficiency of such algorithms it is often useful to know the best possible theoretical complexity bounds. The goal of this paper is to provide such bounds. In particular we give tight upper bounds for the complexity of typing XML documents against various typing systems and upper bounds for evaluating XPath expressions.

We model XML documents by labeled unranked ordered trees. This is classical in the literature (see for instance the survey [29, 36, 4]) because it permits to formalize XML typing specifications and XML query languages using unranked tree automata and use all the known results on tree automata. Indeed, tree automata for trees over ranked alphabets were systematically studied because of their application in term rewriting systems [15]. Generalization of these automata to unranked trees was done in the context of Program Verification and of XML. It turns out that most of the properties of tree automata for trees over ranked alphabets extends to unranked trees even though the proofs are sometime more involved [11].

The first family of problems studied in this paper is type checking of labeled trees. We consider various kind of typing systems. The first two ones, called DTD and *extended* DTD (which extends DTDs with a specialization mechanism [31]), are motivated by XML standards as they roughly correspond to XML DTD and XML Schemas. Both those typing systems are subsumed by tree automata. We therefore generalize our study to all possible variants of tree automata: top-down, bottom-up, deterministic, nondeterministic, walking.

We consider two variants for the type checking problem. In the first one, the type is fixed and the input consists only of the data tree. In the second one the type is also part of the input. The first case is called the *data complexity* for the validation problem and measure the complexity as a function of the size of the data. It gives a good approximation of the behavior of the problem when the size of the type is assumed to be unimportant compared to the size of the data. The second one is the *combined complexity* and gives a more accurate measure of the difficulty of the problem. For instance our study shows that the data complexity is independent of the typing system while the typing system used is crucial for the combined complexity bound.

The data complexity of the validation problem depends crucially on the coding of the input tree because the complexity classes involved are low (below LOGSPACE). While the membership of a tree into a language defined by a tree automata over a *ranked alphabet* was already studied in [27], the extension to unranked trees is not immediate because the encoding used for rank trees (a string based on a prefix notation of the tree) does not work anymore for unranked trees. We consider two codings that reflects the two most used models for XML: the first one is a pointer structure as used in the DOM model [1], the second is a string corresponding to the succession of opening and closing tags encountered during the depth-first traversal of the tree, as used in the SAX data model [2].

For the DOM-like coding we show that, for all typing systems considered here, the data complexity of the validation problem is in LOGSPACE and that this upper bound is tight.
For the SAX-like coding we show that, for all typing systems considered here, the data complexity problem is in uNC^1 (uNC^1 and all the complexity classes used in this introduction will be formally defined in the next section). This therefore extends the results of [27] to unranked trees with the SAX-like coding. Again this upper bound is tight. The combined complexity of the validation problem obviously depends on the syntax of the typing system and we obtain complexities ranging from LOGSPACE-complete to LOGCFL-complete.
Again the results for tree automata over ranked alphabet can be found in [27]. For nondeterministic tree automata over ranked alphabet [27] showed that it is LOGCFL-complete using a non trivial reduction from membership into a context-free language. We give a new, short and elementary, proof of this result using a circuit characterization of LOGCFL, and then extend it to unranked tree automata. Note again that the extension is not straightforward as the coding of an automata over unranked trees is more complex than for trees over ranked alphabet. Indeed each transition now consists of a regular expression instead of a finite set of words. As a matter of fact for deterministic unranked tree automata we obtain complexities that are slightly more complex than for the ranked case.
Some of the algorithms given in this paper will reduce the unranked case to the ranked one. This is done by encoding an unranked tree into a ranked one. It is a well known fact that there exists a mapping from unranked trees to ranked trees which preserve the recognition by regular tree automata [36, 32, 29]. As a side result of independent interest we show that for this transformation, the coding of the output (ranked) tree can be computed in uTC^0 (resp. LOGSPACE) from the coding of the input (unranked) tree if the coding is SAX (resp. DOM), and that the output automata can be computed in LOGSPACE from the input automata.
In the last part of the paper we initiate the theoretical complexity study of query languages for XML by considering XPath. XPath (XML Path Language) is a language for addressing parts of an XML document recommended by the W3C [3]. It is the starting point of several more powerful query languages such as XSLT and XQuery. Because of its central position efficient algorithms are crucial (see for instance [21]).
We consider the query evaluation complexity of a fragment of XPath query. It correspond to the navigational power of XPath: all what can be done without looking at the data values. It is the Core XPath as defined in [21]. Again we consider the *data complexity* for this problem where the query is fixed and the input consists only of the data tree and the *combined complexity* where the query is also part of the input.
In [21] it is shown that the full XPath could be evaluated in polynomial time, combined complexity, and an algorithm is given that evaluates the Core XPath in time $|q| \times |t|$ where $|q|$ is the size of the query and $|t|$ the size of the XML document. We show that data complexity of evaluating the Core XPath fragment is in uTC^0 and that the combined complexity of its positive fragment is in LOGCFL. Notice that this upper bound does not improve the sequential time complexity of [21]. But it is interesting for two reasons, first it shows the high parallelizability of the evaluation of XPath expressions and, second, it shows that it is possible to perform the evaluation with limited memory resources as LOGCFL is included in $\mathrm{DSPACE}(\log^2)$. Because being able to evaluate XML queries with little memory resources is important in practice we give non trivial fragments of the Core XPath that can be evaluated with even less memory: LOGSPACE combined complexity.

related work When trees are labeled over a ranked alphabet, [27] show that, with a prefix encoding, the membership problem in a regular tree language could be solved in uNC^1. [27] also show that the combined membership problem is LOGCFL-complete. The starting point of this work was to extend those results to unranked trees and to other typing systems. We also give a simple version of the LOGCFL hardness proof.
The combined complexity for XPath query evaluation is studied in [21] which give a PTIME algorithm for the general case and a $O(|t| * |q|)$ algorithm for a powerful fragment named Core XPath. From the LOGCFL upper bound we learn that the query evaluation of XPath expressions is highly parallelizable and can be done with limited ($\mathrm{DSPACE}(\log^2)$) memory resources.
In the same proceeding, independently from this work, [22] also provide complexity bounds around XPath. In particular they prove that the combined complexity of evaluating the (positive fragment of) Core Xpath is PTIME-complete (LOGCFL-complete).

The paper is organized as follows. In Section 2 we give all notations, definitions and preliminary results that will be needed through all the paper. The membership problem for unranked tree automata is studied in Section 3 while Section 4 is devoted to the combined complexity problem for validation. Data complexity and combined complexity for evaluating XPath queries will be found in Section 5.

2 Preliminaries

All over this paper, Σ denotes a finite alphabet.

2.1 Complexity and Logic

We assume some familiarity with the following complexity classes uAC^0, uTC^0, uNC^1, LOGSPACE, LOGDCFL, LOGCFL. The "u" in the above class names stands for "uniform" and the uniformity measure used in this paper is DLOGTIME-uniformity as in [8]. uAC^0 is the class of languages that can be recognized by a family $(G_n)_{n\in\mathbb{N}}$ (each input of size n is evaluated by G_n) of boolean circuits consisting of AND and OR gates of unbounded fan-in such that G_n has a constant depth and a size polynomial in n and can be constructed in DLOGTIME from the string 1^n. uTC^0 additionally has MAJORITY gates of unbounded fan-in. In uNC^1, there are no MAJORITY gates, the depth of G_n is allowed to be logarithmic in n, but fan-in is now bounded. LOGCFL (LOGDCFL, LOGUCFL) is the class of languages that can be reduced in LOGSPACE to the membership problem of a (deterministic, unambiguous) context-free language. More precise definitions can be found in [40].
We have uAC^0 $\subsetneq$ uTC^0 $\subseteq$ uNC^1 $\subseteq$ LOGSPACE $\subseteq$ NLOGSPACE $\subseteq$ LOGCFL $\subseteq$ PTIME and all inclusions are suspected to be strict.
In this paper, completeness for complexity classes above PTIME is under PTIME reductions. The reductions used for other complexity classes are explicitly stated in the results. Roughly it is LOGSPACE for complexity classes above LOGCFL and uAC^0 or DLOGTIME for complexity classes below LOGSPACE. In all classes completeness is relative to many-one reductions. We don't consider lower-bounds within uTC^0 because there is no known complete problem for uTC^0 under many-one reductions [6].
We will use characterizations of some complexities classes in term of logic. For this we consider strings as first order structures which universe consists of an initial segment of the integers, $[1\ n]$, where n is the length of the string and each integer correspond to a position in the string. First-order logic over these structures have access to a unary predicate $Q_a(i)$ for each letter a in Σ saying that the position i of the string is an a, a binary predicate $\leq$ denoting the usual order over the integers and a binary predicate BIT(i,j) saying that the i^{th} bit of j in the binary representation of j is 1. This will be denoted by FO. We also consider FOM, an extension of FO with a majority quantifier M such that $Mx\ \varphi(x)$ is true when at least half of the positions x validate φ. The following lemmas are well known:

Lemma 1 [8] FO=uAC^0 and FOM=uTC^0

Lemma 2 [8] The following predicates are expressible in FOM: $x + y = z$ and $x = \sharp y\ \varphi(y)$, the number of positions y such that $\varphi(y)$ holds.

A function $f : \Sigma^* \to \Gamma^*$ for which there exists a constant k such that $|f(w)| \leq k|w|$ is a FOM reduction if there exists FOM formulas $\varphi(x)$ and $\varphi_a(x)$ for each symbol $a \in \Gamma$ such that, for each first order structure of a word w and each $i \in \{1, \cdots, k.|w|\}$, $\varphi(i)$ holds iff $i = |f(w)|$ and $\varphi_a(i)$ holds iff the i-th position of $f(w)$ is an a.
The following is straightforward.

Lemma 3 [27] uTC^0 and uNC^1 are closed under FOM reductions.

In the string encoding a forest is thus a well balanced strings over $\Sigma \cup \bar{\Sigma}$. This can be checked efficiently as shown by the following lemma.

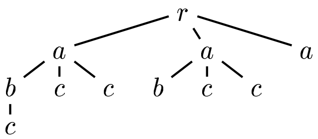
Lemma 4 [7] Given a word w over the alphabet $\Sigma \cup \bar{\Sigma}$ it is decidable in uTC^0 whether w is well balanced.

The above lemma together with Lemma 1 gives:

Corollary 1 *: There exists* FOM *formulas θ and θ^* such that, for any word w and any positions i, j of the word, $w \models \theta^*(i,j)$ iff $i < j$ and the subword of w from position i to position j is well balanced and $w \models \theta(i,j)$ iff $w \models \theta^*(i,j)$ and the closing parenthesis matching the i^{th} character is the j^{th} one.*

2.2 Encodings

We view XML documents as unranked ordered trees labeled with elements in Σ :



Given a tree t, its number of nodes is denoted by $|t|$. The *rank* of a node n is the number of children of n. The *rank* of a tree t is the maximum, over all nodes n of t, of the rank of n.

Trees as pointer structures This is roughly the DOM encoding [1]: each node is an object containing its label and pointers to (i) its parent (ii) its first child (iii) its previous sibling (iv) its next sibling.
If t is a tree, the size of the pointer structure associated to t is roughly $4.|t|$ and is thus also denoted by $|t|$.

Trees as strings The second encoding is close to the model used the SAX interface [2]. It is the actual text representation of an XML document, with no data values, which is a succession of opening and closing tags. It correspond to a depth-first left-first traversal of the tree. We reuse the notations of [35].
For each $a \in \Sigma$ let a itself represent the opening tag and $\bar{a}$ represent the closing tag for a. Let $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$. With this notation, the string associated to the labeled tree above is

$$rabc\bar{c}\bar{b}c\bar{c}c\bar{c}\bar{a}ab\bar{b}c\bar{c}c\bar{c}\bar{a}a\bar{a}\bar{r}$$

More generally, we associate to each labeled tree t a string representation denoted $[t]$ and defined inductively as follows: if t is a single root labeled a, then $[t] = a\bar{a}$; if t consists of a root labeled a and subtrees $t_1 \dots t_k$ then $[t] = a[t_1]\dots[t_k]\bar{a}$. If T is a set of trees, we denote by $\mathcal{L}(T)$ the language consisting of the string representations of the trees in T.
If t is a tree, the size of $[t]$ is twice the number of nodes in t. Thus we also use the notation $|t|$ to denote the size of the string encoding of t.
The following lemma gives the complexities of switching between the two representations.

Lemma 5 (1) Given a tree t as a pointer structure, $[t]$ can be computed in LOGSPACE from t.
(2) Given t as a string structure, the pointer encoding can be computed in uTC^0 from $[t]$.

proof : In order to switch from the string representation to the pointer representation we need to compute for each node (i) its first child (ii) its left and right siblings (iii) its parent. This is trivially done in LOGSPACE from the pointer representation of t. Using Lemma 1, we show that this can be done in uTC^0 from the string representation of t by exhibiting the FOM corresponding formulas. In the following we identify a node of t with the position of the corresponding opening parenthesis in its word encoding. Recall the definitions of θ and θ^* as FOM formulas in Corollary 1. Let Open(i) be the FOM formula that checks that the position i is in Σ (as opposed to $\bar{\Sigma}$).

Next-Sibling$(i,j) \equiv$ Open$(i) \wedge$ Open$(j) \wedge i \leq j \;\wedge\; \exists u\ (\theta(i,u) \wedge u+1=j)$
Previous-Sibling$(i,j) \equiv$ Open$(i) \;\wedge\;$ Open$(j) \;\wedge\; j \leq i \;\wedge$
$\exists u\ (\theta(j,u) \wedge u+1=i)$
First-Child$(i,j) \equiv$ Open$(i) \wedge$ Open$(j) \wedge j = i+1$
Parent$(i,j) \equiv$ Open$(i) \wedge$ Open$(j) \wedge j \leq i \wedge (i = j+1 \vee \theta^*(j+1, i-1))$ $\square$
Remark : Note that the complexity of (1) of Lemma 5 is optimal as it was shown in [17] that computing the depth-first traversal of a tree is FLOGSPACE-hard under uNC^1 reductions, where FLOGSPACE is the functional variant of LOGSPACE.

2.3 DTDs and Tree automata

We assume familiarity with regular languages (REG), regular expressions and finite state automata, NSA and DSA (SA for String Automata). In this paper NSA and DSA are ϵ-free automata. If A is a NSA, e a regular expression, $\mathcal{L}(A)$ and $\mathcal{L}(e)$ denotes the associated regular languages. Context-free grammars (CFG) and context-free language (CFL) are defined as usual. If G is a CFG, $\mathcal{L}(G)$ denotes the associated CFL. See [33] for precise definitions.

Tree types and DTDs DTDs and their variants provide a typing mechanism for XML documents. We will use two notions of types for trees. The first one corresponds closely to the DTD proposed for XML documents, and we therefore (by slight abuse) continue to use the same term. A DTD consists of an extended context-free grammar[1] over alphabet Σ (we make no distinction between terminal and non-terminal symbols at this point). Since regular expressions are closed under union, we can assume w.l.o.g. that each DTD has a unique rule $a \rightarrow L$ for each symbol $a \in \Sigma$. We then denote the regular expression L for a by R_a. An XML document satisfies a DTD τ (or is valid w.r.t. τ) if, for each node labeled a, the ordered list of its children form a word contained in R_a. For example, the labeled tree above is valid w.r.t. the DTD: $\begin{array}{l} r \rightarrow a^* \\ a \rightarrow bc^*|\epsilon \\ b \rightarrow c|\epsilon \\ c \rightarrow \epsilon \end{array}$.

The second type system correspond closely to XML-Schema. It is an extension of DTDs with *specialization* (also called decoupled types) which, intuitively, allows defining the type of a tag by several "cases" depending on the context. Extended DTDs (EDTD) have been studied in [31] and are equivalent to formalisms proposed in [9, 14]. They are present in a restricted form in XML-Schema.

Definition 1 *An EDTD over Σ is a tuple $\tau = (\Sigma, \Sigma', \tau', \mu)$ where (i) Σ and Σ' are finite alphabets; (ii) τ' is a DTD over Σ'; and (iii) μ is a mapping from Σ' to Σ.*
A tree document t over Σ satisfies an EDTD τ, if $t \in \mu(SAT(\tau'))$.

Intuitively, Σ' provides for some a's in Σ a set of specializations of a, namely those $a' \in \Sigma'$ for which $\mu(a') = a$. We also denote by μ the homomorphism induced on strings and trees by μ, extended whenever needed to symbols in $\bar{\Sigma}'$ by $\mu(\bar{a}') = \overline{\mu(a')}$.
When considering only string representations it may be easier to see DTD and EDTD as extended context-free grammars with terminal alphabet $\Sigma \cup \bar{\Sigma}$, non-terminal alphabet $V = \Sigma'$, containing rules of the form $S \rightarrow aR_S\bar{a}$, where $S \in V, a \in \Sigma$, R_S is a regular relation over V. DTDs have the restriction that for each $a \in \Sigma$ there is a unique rule involving a. EDTDs do not have this restriction and have one rule per element in Σ'.
If τ is a (E)DTD, SAT(τ) is the (regular) set of trees satisfying τ and $\mathcal{L}(\tau)$ is the language over $\Sigma \cup \bar{\Sigma}$ consisting of the string representations of all tree documents in $SAT(\tau)$, that is $\{[t] \mid t \in SAT(\tau)\}$.

Tree automata We also consider tree automata (TA). We recall quickly their definitions for unranked trees. The reader is refered to [11] for a more detailed presentation. As usual they are either bottom-up (BU) or top-down (TD) and either deterministic (D) or non deterministic (N).
A non deterministic top-down tree automata (NTDTA) is a tuple (Σ, Q, q_0, δ) where Q is a finite set of states, q_0 is the initial state, and $\delta \subseteq \Sigma \times Q \times Q^*$ is such that $\delta(a, q)$ is a regular language over Q for each $a \in \Sigma$ and each $q \in Q$ (we assume that for $a \in \Sigma$ and each $q \in Q$, $\delta(a, q)$ is given as a NSA). The computation starts at the root of the tree in the initial state and proceeds towards the leaves of the tree by assigning states to the children of a node according to its label, α, the current state, q, and $\delta(\alpha, q)$ (see [11] for more details). The tree is accepted if, for each leaf n of the tree, the automata reach n in a state q such that ϵ is in $\delta(\text{label}(n), q)$. A deterministic TDTA (DTDTA) is a NTDTA such that for each $a \in \Sigma$ and each $q \in Q$ and each $n \in \mathbb{N}$ there exists a unique $w \in \delta(a, q)$ of length n.

A non deterministic bottom-up tree automata (NBUTA) is a tuple (Σ, Q, F, δ) where Q is a finite set of states, F is a set of accepting states and $\delta \subseteq \Sigma \times Q^* \times Q$ is such that $\{w \in \Sigma^* \mid \delta(a, w, q)\}$ is a regular language over Q for each $a \in \Sigma$ and each $q \in Q$. The computation starts at the leaves by assigning to each leaf of label α a state according to $\delta(\alpha, \epsilon)$ and proceeds towards the root of the tree by assigning states to a node according to its label, a, the sequence of states of its children, w, and $\delta(a, w)$ (see [11] for more details). The tree is accepted if the automaton reach the root in an accepting state. A deterministic BUTA (DBUTA) is a NBUTA such that for each $a \in \Sigma$ and each $w \in Q^*$ there is a unique q such that $\delta(a, w) = q$.

If A is a tree automata (bottom-up or top-down) then we denote by SAT(A) the set of trees accepted by A, and by $\mathcal{L}(A)$ the set of string encodings of trees accepted by A.
Fact (folklore, see also [11]): Let T be a set of labeled unranked trees. T is accepted by an EDTD iff T is recognized by a NBUTA iff T is recognized by a NTDTA iff T is recognized by a DBUTA. In this case T is said to be a regular tree language. It is also known that DTD and DTDTA fail to capture all regular languages.

[1] In an extended CFG, the right-hand sides of productions are regular expressions over the terminals and non-terminals.

Finally we consider tree-walking automata (TWA). TWA automata we initially introduced in [5], see also [19, 18]. A non deterministic TWA (NTWA) is a tuple $(\Sigma, Q, q_0, F, \delta)$ where Q is a finite set of states, q_0 is the initial state, F is a set of accepting states and $\delta \subseteq \Sigma \times Q \times \{\text{root, leaf, leftchild, rightchild, other}\} \times Q \times \{\text{up, down, left, right, stay}\}$. The computation starts at the root in the initial state and proceeds by walking in the tree as follows: when reaching at node n of label a in state q, the automata checks whether the status of the current node is a root or a leaf or a leftmost (rightmost) child or just a regular inner node, and, according to $\delta(a, q, \text{status})$ changes its state to q' and move up, down to the leftmost child, to the next sibling, to the previous sibling or stay in the current node of the tree. A deterministic TWA (DTWA) is a NTWA such that $\delta(a, q, \text{status})$ is unique for each $a \in \Sigma$ and each $q \in Q$. It can be shown that NTWA and DTWA recognize only regular tree languages. Whether all regular tree languages can be recognized by a NTWA is still an open problem.

3 Data complexity for validation

In this section we study the data complexity of the validation problem for all the typing systems mentioned in the Section 2. As mentioned earlier they all define regular tree language. We thus start with the most general case and consider membership of a tree in a regular tree language. This will give upper-bounds (depending on the coding chosen for the input tree) for all of the considered typing systems. In the last part of this section we will see that these upper-bounds are actually matched by all them. Therefore the data complexity of the validation problem does not depend on the typing used.

Let T be a regular tree language. Consider the following decision problem:

[Val$_T$] INPUT: a labeled tree t
OUTPUT: true iff $t \in T$.

The complexity of VAL$_T$ depends on the coding used for t. We consider two encodings for trees: the DOM-like and the SAX-like as described in the previous section. We start with the case where t is coded as pointer (tree) structure as it better illustrates the difference between ranked and unranked trees.

3.1 t is a pointer structure

In this section t is given as a pointer structure. We want to check whether $t \in \text{SAT}(A)$ for some unranked tree automata A. We will proceed as follows: first we will consider the case where all trees have a fixed rank and then we will reduce the unranked case to the ranked case.

Assume first that all trees have a rank bounded by k. Let T be an arbitrary regular (rank k) tree language and let A be a DBUTA recognizing T. Let Q be the set of states of A and Γ be $(Q \cup \{\sqcup\})^k$, where $\sqcup$ is a new symbol.

In what follows, i is a pointer over t and S is a stack over the alphabet Γ. If $H \in \Gamma$ and $q \in Q$, UPDATE(H, q) is the function that replace the leftmost $\sqcup$ symbol of H by q. Let $<$ be a total order on the nodes of t and NEXT-SIBLING$_<(i)$ and FIRST-CHILD$_<(i)$ be the functions returning the pointers to the nodes of t corresponding to the next-sibling (first-child) of the node pointed by i depending on $<$. Initially i points to the root of t and S is empty.

Consider the following sequential algorithm running a tree automata over a tree.

```
While true do {
  if i is a leaf do {
/* If i is the first of the siblings that is
evaluated we initiate a new computation.
otherwise we recall the previous computations
from the stack */
   if i is a first-child< do H=⊔^k else H = pop;
   q = δ(label(i),ϵ); H = update(H,q);
   while i has no next-sibling< do {
/* If all siblings of i have been evaluated we
can compute the state of father(i) using H
and proceed upward */
      if i is the root return q;
```

```
      i = father(i); q = δ(i,H);
      if i is a first-child< do H=⊔^k
                              else H = pop;
      H = update(H,q);
   }/*end while*/
/* i has a sibling which is not yet evaluated,
we store the current evaluation on the
stack and proceed*/
   push(H); i = next-sibling<(i);
  }/*end of i is a leaf*/
/* Depth-first <-first scan of the tree */
  else i = first-child<(i)
}
```

This algorithm simply parse the tree in a depth-first $<$-first fashion, computing for each node, the states reached by the automata at that node. That state can only be obtained when all its children have been processed and the stack S is used to store partial computations. The analysis of the resources needed by this algorithm depends on $<$. The naive approach would be to use the depth-first left-first order of t (recall that the siblings of t are ordered) for $<$. Then it is easy to see that the functions NEXT-SIBLING$_<$ and FIRST-CHILD$_<$ are computable in LOGSPACE but notice that an element is pushed in the stack S, each time a first-child of a next-sibling is visited. Thus the depth of the stack S can be linear in the size of t. Thus the overall space used is $O(k.|t|)$. A better approach is to order the siblings according to the size of their corresponding subtrees, taking the biggest first and the left-most sibling first in case of equality (this strategy of considering biggest subtrees first is common for trees, see [26] for instance) In that case each time a first-child of a next-sibling is visited the size of the corresponding subtree is at least divided by 2. Thus the depth of the stack S is now bounded by $\log |t|$. Notice that the size of a subtree can be computed in LOGSPACE and therefore the functions NEXT-SIBLING$_<$ and FIRST-CHILD$_<$ remain computable in LOGSPACE. The overall space complexity is thus $O(k. \log |t|)$ which is LOGSPACE when k is fixed. Thus we have:

Lemma 6 If T is a regular ranked tree language then Val_T is in LOGSPACE.

This algorithm does not generalize to unranked tree because it is linear in the rank of the tree which, in general, could be linear in $|t|$. For that reason, when the trees are unranked, any depth-first $<$-first evaluation is doomed to fail. In order to maintain the LOGSPACE complexity, the idea is to traverse the tree using the following order: the successor j of a node i is either the left-most child of i or its next-sibling depending on whether the size of the subtree rooted in i is bigger than the total size of the subtrees rooted at its right siblings. It can be shown that the algorithm for the rank case adapted for such a traversal is in LOGSPACE.
Another way to get the LOGSPACE complexity is to preprocessed the unranked tree in order to encode it as a ranked tree. This reduction is fairly standard in the literature [36, 32, 29]. Let $\sharp$ be a new symbol and $\Sigma^\sharp$ be $\Sigma \cup \{\sharp\}$. The transformation takes an unranked labeled tree t over Σ and returns a binary labeled tree t' over $\Sigma^\sharp$. The transformation is denoted by RANK and is defined inductively as follows (see Figure 1): If t is the empty tree then RANK(t) is a single node labeled $\sharp$. Let $\bar{T}$ and $\bar{T}'$ be possibly empty forests and t be a tree which root is labeled by $a \in \Sigma$ and its children is the forest $\bar{T}$, then RANK$(t\bar{T}')$ is the binary tree rooted by a node labeled by a which left child is RANK$(\bar{T})$ and which right child is RANK$(\bar{T}')$.
Notice that RANK is 1-1 and, importantly, it preserve regularity:

Lemma 7 [folklore] Given a regular tree language T the set $T' = \{t' | t' = \text{RANK}(t)\}$ is also a regular tree language.

Moreover it can be checked that this encoding can be done in LOGSPACE. The algorithm just copy the input nodes, sets the link appropriately (this can be done in LOGSPACE because this concern only local nodes), and finally adds as many $\sharp$ nodes as required.

Lemma 8 Let t be a tree coded by its pointer structure. Then the pointer structure coding RANK(t) can be computed in LOGSPACE from t.

Lemma 7, Lemma 8 and Lemma 6 immediately gives the main result of this section:

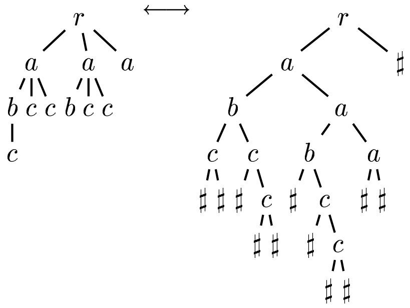


Figure 1: from unranked to ranked tress

Theorem 1 *: For all regular tree language T, if t is encoded as a pointer structure, Val$_T$ is in* LOGSPACE.

The complexity of the above theorem is optimal as shown below.

Theorem 2 *: There exists T such that, assuming that t is coded as a pointer structure, Val$_T$ is complete for* LOGSPACE *under uAC^0 reductions.*

proof : Let T be the regular family of trees consisting of a single branch forming a string in the regular language $a^*ba^*ca^*$.
We prove the hardness of Val$_T$ by reducing it to the following LOGSPACE-complete problem under uAC^0 reductions [20].

ORD:	INPUT	a directed graph $G = (V, E)$ which is a line and two nodes v_i, v_j
	OUTPUT	true iff $v_i < v_j$ in the order induced by the graph.

Let G be a directed graph that is a line and v_i, v_j two nodes of G. G can be seen as a tree t and a pointer structure for t can be constructed from G in the obvious way. The labels of the nodes of t are set such that v_i is labeled b, v_j is labeled c and all other nodes are labeled a. This reduction is First-Order and thus computable in uAC^0.
It can be checked that ORD is true for G, v_i, v_j iff $t \in T$.
This concludes the proof. □

3.2 t is encoded as a string

In this section we assume that the input t of Val$_T$ is encoded as a string. Recall that if t is a tree, $[t]$ denotes its string encoding. The main result of this section is:

Theorem 3 *: For all T, if t is coded as a string, Val$_T$ is in uNC^1.*

The proof of the theorem follows the strategy of the previous section: it first reduce the unranked case to the ranked one. The reduction is based on the following Lemma:

Lemma 9 Let t be a tree $[\text{RANK}(t)]$ is computable in uTC^0 from $[t]$.

proof : By Lemma 1 it suffices to give a logical FOM formula corresponding to the reduction. If S is a subset of Σ we denote by $Q_S(i)$ the predicate $\bigvee_{a\in S} Q_a(i)$. Recall Corollary 1. Given a position i which contains a letter of Σ the unique position j containing the corresponding closing parenthesis is the unique j verifying the FOM formula $\theta(i, j)$. In the following we identify a node of the tree by the position of the corresponding opening parenthesis in its word encoding. The following FOM formula thus checks that a node i is a leaf of t.
Leaf(i) $\equiv$ $Q_\Sigma(i) \wedge \theta(i, i+1)$
The following FOM formula checks that a node i has no right sibling.
NoRightSibling(i) $\equiv$ $\exists j\ (\theta(i, j) \wedge (Q_{\bar{\Sigma}}(j+1) \vee Last(j)))$
The following FOM formula checks that two nodes i and j are siblings.

Sibling$(i,j) \equiv (i \leq j \wedge \exists u,v\ (\theta(i,u) \wedge \theta(j,v) \wedge (u+1=j \vee \theta^*(u+1,j-1)))) \quad \vee \quad (j \leq i \wedge \exists u,v\ (\theta(i,u) \wedge \theta(j,v) \wedge (v+1=i \vee \theta^*(v+1,i-1))))$

Finally the following FOM formula checks that j is an ancestor of i.

Ancestor$(i,j) \equiv \exists u\ (j \leq i \leq u \wedge \theta(j,u))$

Let t' =RANK(t). By definition t' contains at most two extra nodes per node of t, thus $|\text{RANK}(t)| \leq 3.|t|$. More precisely t' contains the same nodes as t plus nodes labeled $\sharp$, one for each leaf of t and one for each node of t having no right sibling. Because each node of the tree correspond to 2 characters in its string representation the size z of $[t']$ can be expressed by the following FOM formula (recall Lemma 2):

$\exists x,y\ \ (z = \max + 2x + 2y \wedge x = \sharp i$ Leaf$(i) \wedge$
$\qquad y = \sharp i$ NoRightSibling$(i))$

Let n be a node labeled by $a \in \Sigma$ in t. Let i be the position in $w' = [t']$ of the opening tag of n and j be its position in $w = [t]$. We are looking for a FOM formula defining i from j and w. i is computed by counting the number of tags of each type occurring before i in w'. Notice first that the depth-first left-first traversal of t and t' will go through the same nodes in the same order (assuming we skip the nodes labeled by $\sharp$ in t'). Thus before i (in w') there are as many opening tags as there are before j (in w). This number is denoted by x in the formula below. Before$i$ we find as many $\sharp$ as there are nodes before j which are either a leaf or with no right sibling, this number is denoted by y in the formula below, notice that each $\sharp$ is immediately followed by a $\bar{\sharp}$ and thus count for two characters. Therefore in order to get i from j it remains to compute the number of closing tags that must occurs before i. To do this notice that it suffices to compute the number of closing tags before j and remove all those that have a sibling which is an ancestor of n and which is thus no yet completely processed by RANK. This number is denoted by z in the formula below. In conclusion the following FOM formula $\varphi_a(i)$ gives the positions in t' labeled by a:

$\exists j \quad (Q_a(j) \wedge$
$\qquad x = \sharp u\ u \leq j \wedge Q_\Sigma(u) \quad \wedge$
$\qquad y = \sharp u\ \exists v\ (u \leq v \leq j \wedge \theta(u,v)) \wedge ($Leaf$(u) \vee$
$\qquad\quad$ NoRightSibling$(u)) \quad \wedge$
$\qquad z = \sharp u\ \exists v\ (u \leq j \wedge Q_{\bar{\Sigma}}(u) \wedge \theta(v,u) \wedge$
$\qquad\quad \neg(\exists w$ (Sibling$(v,w) \wedge$ Ancestor$(w,j)))) \quad \wedge$
$\qquad i = x + 2y + z)$

Next we let i be the position in w' of the closing tag of n and j be its position in w. Again we want to compute i from j and w. Counting opening tags and $\sharp$ symbols is done as above. For counting closing tags, notice that in w' the closing tag of n occurs when the entire forest of the siblings of n is processed except for the closing tags of the left siblings of n. Therefore the following FOM formula $\varphi_{\bar{a}}(i)$ gives the positions in w' labeled by $\bar{a}$ (in the formula below, j' represents the position of the opening tag of n and k the position of the closing tag of the rightmost sibling of n. This is enforced by the first line of the formula):

$\exists j, j', k \quad (Q_{\bar{a}}(j) \wedge \theta(j',j) \wedge \theta^*(j',k) \wedge$
$\qquad\quad (\forall k'\ k < k' \rightarrow \neg\theta^*(j',k')) \wedge$
$\qquad x = \sharp u\ u \leq k \wedge Q_\Sigma(u) \wedge$
$\qquad y = \sharp u\ \exists v\ (u \leq v \leq k \wedge \theta(u,v)) \wedge ($Leaf$(u) \vee$
$\qquad\quad$ NoRightSibling$(u)) \wedge$
$\qquad z = \sharp u\ \exists v\ (u \leq k \wedge Q_{\bar{\Sigma}}(u) \wedge \theta(v,u) \wedge$
$\qquad\quad (\neg$Sibling$(v,j) \vee j \leq u)) \wedge$
$\qquad i = x + 2y + z)$

The formula for $\sharp$ and $\bar{\sharp}$ are obtained similarly. It suffices to guess the node n in t which either is a leaf or has no right sibling, then, for a leaf, the current position of the opening tag of n in w' is computed as above and 1 is added for $\sharp$, 2 for $\bar{\sharp}$. If n has no right sibling the position of the closing tag of n in w' is computed as above and 1 is removed for $\bar{\sharp}$, 2 for $\sharp$.

This concludes the proof of the lemma. □

The ranked case is proved by adapting techniques of [27] for this new coding. Intuitively it reduces, in uTC^0, the problem in checking membership into a parenthesis CFL which is known to be in uNC^1 [13].

proof : of Theorem 3. Let T be a regular tree language, let T' be the regular tree language accepting the image of T by RANK (see Lemma 7). Let $A = (\Sigma^\sharp, Q, \delta, q_0, F)$ be a BUTA recognizing T'. Let $G = (Q, \Sigma^\sharp, R, q_0)$ be the CFG where R contains $q \rightarrow aq'q''\bar{a}$ iff $\delta(a,q',q'') = q$. It is easy to see that $t \in T$ iff $[\text{RANK}(t)] \in L(G)$. Therefore by Lemma 9 and Lemma 3 it suffice to show that $[\text{RANK}(t)] \in L(G)$ can checked done in uNC^1. This can be achieved by adapting the proof of [27] to our coding of a tree: We further transform $t' = \textsc{rank}(t)$. Let β be the string operation defined inductively on string representation of trees as follows (note that the trees are now binary and complete):

- $\beta(\epsilon) = \epsilon$
- $\beta(auv\bar{a}) = (a\beta(u)\beta(v))$ where u and v are well balanced strings.

Consider now the CFG $G' = (Q, \Sigma^\sharp \cup \{(,)\}, R', q_0)$ where R' contains $q \rightarrow (aq'q'')$ iff $q \rightarrow aq'q''\bar{a}$ is in R. It is easy to check that $[t] \in L(G)$ iff $\beta([t]) \in L(G')$. We show that β is a FOM reduction.
Let $w = [t]$ for a complete binary tree t. The FOM formula giving the size i of $w' = \beta(w)$ is given by:
$i = 3.\sharp j \ Q_{\Sigma^\sharp}(j)$
The FOM formula giving the positions i of w' containing the symbol $a \in \Sigma^\sharp$ is given by the formula below. In this formula j guess the position in w of the node which image by β is i, x computes the number of nodes which have already been completely processed by β before reaching j (notice that each such node is encoded with 3 characters in w' and all of them are before i) and y computes the nodes that are not yet completely processed by β before reaching j, namely the ancestors of j (notice that each such node has exactly 2 characters that occurs before i and one after - the closing parenthesis).

$$\begin{array}{lll} \varphi_a(i) \equiv & \exists j & Q_a(j) \wedge \\ & & x = \sharp u \ \exists v \ Q_\Sigma(u) \wedge u \leq v \leq j \wedge \theta(u,v) \wedge \\ & & y = \sharp u \ \text{Ancestor}(u,j) \wedge \\ & & i = 3x + 2y \end{array}$$

The formulas for (and) are obtained similarly.
Therefore, by Lemma 3 it suffice to show that $w \in \mathcal{L}(G')$ can be checked in uNC^1. In general checking that a word belongs to a CFL is in LogCFL which is believed to be much more powerful than uNC^1. But G' is a special CFG, it is a parenthesis CFG as defined in [28]. It turns out that membership in a parenthesis CFL can be done in uNC^1 [13]. This concludes the proof of the theorem. ☐
The upper-bound of Theorem 3 is optimal because there exists regular string languages L such that the problem of checking whether a word w is in L is complete for uNC^1 under DLogTime reductions. Such a language can be found in [8]. This regular language can easily be turned into a regular tree language, thus we have:

Theorem 4 *: There exists T such that Val$_T$ is complete for u*NC^1 *under* DLogTime *reductions.*

3.3 Other typing systems

All the typing systems mentioned in the introduction, TA, TWA, DTD, EDTD, define regular tree languages. Moreover, all of them can encode any string regular language and there exists regular languages for which testing membership is uNC^1-complete (see [40]) when the input is given as a string, and LogSpace-complete (see Theorem 2) when the input is given as a pointer structure. From this Theorem 1 and Theorem 3 we obtain the following corollary:

Corollary 2 *: Let τ be a DTD, EDTD, TWA, or a TA. If the input tree is coded as a pointer structure the complexity of Val$_\tau$ is in* LogSpace*, if it is coded with a string, the complexity of Val$_\tau$ is in u*NC^1. *Both upper bound are optimal for each of the typing system.*

Remark : The actual definition of XML DTDs and XML Schemas requires a special normal form for the regular expressions they contains, called Unique Particle Attribution Constraints. This normal form correspond to 1-unambiguous languages as defined by [12]. Not all regular languages have an expression which is 1-unambiguous. It is therefore natural to wonder whether such languages enjoy a lower validation complexity. This is not the case. Indeed it is easy to come out with an example of a 1-unambiguous regular expression which syntactic group is non-solvable and thus having a membership problem complete for uNC^1.

4 Combined complexity

In the previous section the input was simply the encoding of a tree. In this section we consider the problem of validation when the type is also part of the input. Let C be a class of tree types. For instance C could be the class of all DTDs, or all EDTDs or all tree automata etc. More formally we will study the following problems:

[TreeM(C)] INPUT: a tree t, a type $\tau \in C$
OUTPUT: true iff $t \in \mathrm{SAT}(\tau)$

As all types considered in this paper are a generalization of string regular languages, we recall the complexities for strings.

[StringM(C)] INPUT: a string w, a type $\tau \in C$
OUTPUT: true iff $w \in \mathcal{L}(\tau)$

Lemma 10 [24, 25]

1. StringM(DSA) is LOGSPACE-complete under uAC^0 reductions.
2. StringM(NSA) is NLOGSPACE-complete under uAC^0 reductions[2].
3. StringM(Regular-expression) is NLOGSPACE-complete under uAC^0 reductions[3].

All family of automata that we study are extensions of finite automata over strings thus Lemma 10 implies that all the complexities are going to be above LOGSPACE. This Lemma together with Lemma 5 implies that the encoding of the input tree is now irrelevant and we will arbitrarily use the string encoding or the pointer encoding depending on what we need. On the other hand the complexity of TreeM(C) will now depend on C.

4.1 DTDs

Recall that a DTD associates a (unique) regular expression to each symbol of Σ. We have:

Theorem 5 *: TreeM(DTD) is* NLOGSPACE*-complete under u*AC^0 *reductions.*

proof : The upper bound is shown as follows. We will use a pointer p_1 in order to do the depth-first walk of the input tree. Another pointer p_2 will be used to parse the DTD. An extra pointer p_3 on the tree will be used for local computations. For each node pointed by p_1 the word formed by the labels of the list of its children can be parsed in LOGSPACE using p_3. The pointer p_2 is set to the regular expression of the DTD corresponding to the label pointed by p_1. It is now possible to check that the word pointed by p_3 is indeed in the regular expression pointed by p_2 in NLOGSPACE (this is Lemma 10).
The lower bound comes from the fact that the problem is obviously harder than StringM(Regular-expression) which is NLOGSPACE-complete by Lemma 10. ☐
For real XML DTDs, where regular expression are required to be unambiguous (recall the remark of Section 3.3), or for DTDs where a deterministic automata is given instead of a regular expression the situation becomes simpler:

Theorem 6 *: TreeM(DTD with DSA) and*
TreeM(XML DTD) are LOGSPACE*-complete under u*AC^0 *reductions.*

4.2 TWA

For TWA the complexity depends on whether the automata is deterministic or not. The following extends the result for two-way string automata:

Theorem 7 *: TreeM(DTWA) is* LOGSPACE*-complete under u*AC^0 *reductions*
TreeM(NTWA) is NLOGSPACE*-complete under u*AC^0 *reductions*

proof : We assume the tree is given by is string encoding. The lower bound follows directly from Lemma 10. For the upper bound the Turing machine needs only two pointers, one to record the current position of the head on the input string w, and one to point to the current state. This can be done in LOGSPACE. To get to the next step, it browses deterministically (resp. non deterministically) the list of transitions of the automata until it gets one corresponding to the current label and state, and the move the head pointer and the state pointer accordingly. All this can be done in LOGSPACE (resp. NLOGSPACE). ☐

---

[2] [25] actually prove completeness under so called Rudimentary-Log reductions which are equivalent to uAC^0 as mentioned in [40]
[3] This can be derived from 2) because there is a LOGSPACE reduction from regular expression to ϵ-free NSA [24]

4.3 TA

For tree automata over ranked alphabet TreeM was solved in [27]: TreeM(Ranked-NBUTA) and TreeM(Ranked-NTDTA) are LOGCFL-complete, TreeM(Ranked-DBUTA) is in LOGDCFL and
TreeM(Ranked-DTDTA) is LOGSPACE-complete. For TreeM(Ranked-NBUTA), the hardness proof of [27] uses a reduction from membership into a context-free language. The data tree then roughly corresponds to the skeleton of a derivation tree of the input word and from the CFG a tree automata is constructed that checks that indeed a derivation tree of the input word can be obtained from the data tree. The difficulty is to achieve this within a bounded rank because there is *a priori* no reason for the derivation tree of the input word to have a bounded rank. A combinatorial result of Ruzzo [34] is used to show that this is indeed possible.
We give a very elementary short new proof of [27] result based on the uSAC^1 characterization of LOGCFL due to Venkateswaran [39]. As noticed by many authors, see for instance [23], checking that an uSAC^1 circuit evaluates to true on input word w reduce in finding a *proof tree* for the circuit on w. This *proof tree* can be seen as a certificate that the circuit evaluates to 1. It turns out that, for circuit in *normal form*, the skeleton of all *proof trees* depends only on n and can be computed easily in LOGSPACE. The reduction now works as follows: given an input word w of size n, the data tree will be the skeleton of all *proof trees* corresponding to n, and the tree automata will be isomorphic to the circuit C_n and will therefore assign a state to each node of the tree transforming the data tree into a *proof tree* iff the circuit evaluates to 1 on input w.

Lemma 11 [27] TreeM(Ranked-NBUTA) and TreeM(Ranked-NTDTA) are LOGCFL-complete.

proof : The upper-bound is immediate we only consider the lower bound. We show that membership in a uSAC^1 language can be reduced in LOGSPACE to TreeM(Ranked-NTDTA) (TreeM(Ranked-NBUTA) is treated similarly). We follow the lines of [23] for the definition of normal form and *proof tree* for uSAC^1 circuits.
A uSAC^1 family of Boolean circuits is in *normal form* (NF) if the following conditions are satisfied: (i) the fan-in of all AND gates is 2, (ii) a level can be assigned to the nodes of each circuit such that inputs are at level 0 and each gate of level i receives all its inputs from nodes of level $i - 1$, (iii) the circuits are strictly alternating between OR gates and AND gates (odd-level gates are OR gates and even-level gates are AND gates), and, (iv) each circuit has an odd number of levels, thus the output gates are AND gates. Lemma 4.6 of [23] shows that we can assume WLOG that uSAC^1 circuits are in NF.
A *proof tree* T for a circuit G of a uSAC^1 family in NF on input word w is a rooted tree t together with a labeling function λ from nodes N of T to gates of G such that: (i) the root of T is labeled with the output gate of G, (ii) if $\lambda(N) = g$ and g is an AND gate then N has exactly 2 children N_1 and N_2 such that $\lambda(N_1)$ and $\lambda(N_2)$ are the input gates of g, (iii) if $\lambda(N) = g$ and g is an OR gate then N has a unique child N_1 such that $\lambda(N_1)$ is one of the input gate of g, and, (iv) each leaf N of T is labeled with an input gate of G such that the corresponding bit is 1. Intuitively a *proof tree* is a proof that shows that the circuit indeed evaluates to 1.
Notice that for each NF family of uSAC^1, each n and each w of length n the tree t of the *proof tree* T of G_n on input w only depends on n (the labeling function λ does depends on w and G_n). We note by t_n this tree, denoted as the *skeleton* of T for size n in [23].
Example of circuits in NF, together with their corresponding proof trees and skeletons can be found in [23]. We now turn to the proof of the theorem. Let $(G_n)_{n\in\mathbb{N}}$ be a family of NF circuits in uSAC^1 and $w \in \{0,1\}^*$. Let n be the length of w. We have to compute in LOGSPACE a ranked tree t and an ranked NTDTA A such that $t \in$SAT(A) iff w is accepted by G_n.
Let t be t_n, the skeleton of the *proof tree* for G_n on input w, where each node is assign the same label a. Let $A = (\{a\}, Q, q_0, \delta)$ where $Q = V$, the set of gates of G_n, q_0 is the output gate of G_n. δ is defined as follows. $\delta(a, q) = \{q_1 q_2\}$ if q is a AND gate with input gates q_1 and q_2. If q is an OR gate then $\delta(a, q) = \{q' \mid q'$ is an input gate of $q\}$. If q is an input gate of G_n then $\delta(a, q) = \{\epsilon\}$ if it evaluates to 1 according to w, otherwise we set $\delta(a, q) = \emptyset$.
It is now straightforward to verify that $t \in$ SAT(A) iff t, together with the labeling that assign to each of its node the state of A corresponding to an accepting run, form a *proof tree* of G_n on input w iff w is accepted by G_n. It remains to show that the reduction is LOGSPACE: the fact that t_n can be computed in LOGSPACE from n is shown in [23], and the LOGSPACE computation of A is immediate from the LOGSPACE uniformity of uSAC^1 circuits. □
The general result for unranked trees is obtained by reduction to the ranked case. Recall Lemma 7. Given a regular tree language T the tree language RANK(T) is regular. In what follows we overload the notation of RANK, and denote

by $\textsc{rank}(A)$ a tree automata recognizing the regular tree language $\textsc{rank}(\mathrm{SAT}(A))$. The following shows that the reduction can be computed in LOGSPACE.

Lemma 12 Given a NTDTA A, the NTDTA $\textsc{rank}(A)$ can be computed in LOGSPACE.
Given a NBUTA A, the NBUTA $\textsc{rank}(A)$ can be computed in LOGSPACE.

proof : We first prove the NTDTA case by giving the LOGSPACE construction of $A' = \textsc{rank}(A)$. Let $A = (\Sigma, Q, q_0, \delta)$ be a NTDTA. For each letter $a \in \Sigma$ and each state $q \in Q$ let $A^{a,q} = (Q, \Delta^{a,q}, p_0^{a,q}, F^{a,q}, \delta^{a,q})$ be nondeterministic automata recognizing the regular string language $\delta(a,q)$, the alphabet is Q and the set of states $\Delta^{a,q}$. Let Δ be $\bigcup_{a,q} \Delta^{a,q}$. Each state of A' will be a quadruple of $\Sigma \times Q \times Q \times \Delta$ in order to record the current language $A^{a,q}$ that it is processing, the current letter, and the current state in this language.
Let $A' = (\Sigma^\sharp, Q', Q_0, \delta')$ be the NTDTA defined by :

- $Q' = \{\langle a, q, q', p\rangle \mid a \in \Sigma, q \in Q, q' \in Q, p \in \Delta\} \cup q_\sharp$.
- $Q_0 = \{\langle a, q_0, q, p_0^{a,q_0}\rangle \mid a \in \Sigma, q \in Q\}$
- $\langle a, q_i, q_j, p_0^{a,q_i}\rangle\langle b, q, q_k, p_k\rangle \in \delta'(a, \langle b, q, q_i, p_i\rangle)$ if $\delta^{b,q}(q_i, p_i) = p_k$.
- $\langle a, q_i, q_j, p_0^{a,q_i}\rangle q_\sharp \in \delta'(a, \langle b, q, q_i, p_i\rangle)$ if $p_i \in F^{b,q}$.
- $q_\sharp\langle b, q, q_k, p_k\rangle \in \delta'(a, \langle b, q, q_i, p_i\rangle)$ if $\delta^{b,q}(q_i, p_i) = p_k$ and $p_0^{a,q_i} \in F^{a,q_i}$.
- $q_\sharp q_\sharp \in \delta'(a, \langle b, q, q_i, p_i\rangle)$ if $p_i \in F^{b,q}$ and $p_0^{a,q_i} \in F^{a,q_i}$.
- $\delta'(\sharp, q_\sharp) = \epsilon$.
- nothing else is in δ'.

It can be checked that the above computation is correct (A' is indeed $\textsc{rank}(A)$) and that it can be done within LOGSPACE: 4 pointers are needed in order to define a state of the output, thus 12 pointers suffice to compute the transition function δ'.

The proof for NBUTA is similar. Let $A = (\Sigma, Q, F, \delta)$ be a NBUTA. For each letter $a \in \Sigma$ and each state $q \in Q$ let $A^{a,q} = (Q, \Delta^{a,q}, p_0^{a,q}, F^{a,q}, \delta^{a,q})$ be nondeterministic automata recognizing the regular string language $\{\bar{w} \mid \delta(a, w, q)\}$. This can be easily computed in LOGSPACE from A. Let Δ be $\bigcup_{a,q} \Delta^{a,q}$. Each state of A' will be a triple of $\Sigma \times Q \times \Delta$ in order to record the current language $A^{a,q}$ that it is processing and the current state in this language.
Let $A' = (\Sigma^\sharp, Q', F', \delta')$ be the NTDTA defined by :

- $Q' = \{\langle a, q, p\rangle \mid a \in \Sigma, q \in Q, p \in \Delta\}$
- $F' = \{\langle a, q, p\rangle \mid q \in F$
- $\langle a', q', p''\rangle \in \delta'(b, \langle a, q, p\rangle\langle a', q', p'\rangle)$ if $a = b, p \in F^{a,q}$ and $\delta^{a',q'}(q, p') = p''$.
- $\langle a, q, p_0^{a,q}\rangle \in \delta'(\sharp, \epsilon)$.
- nothing else is in δ'.

It can be checked that the above computation is correct (A' is indeed $\textsc{rank}(A)$) and that it can be done within LOGSPACE: 3 pointers are needed in order to define a state of the output, thus 9 pointers suffice to compute the transition function δ'. □
From Lemma 12 and Lemma 11 we immediately have:

Theorem 8 *: TreeM(NBUTA) and TreeM(NTDTA) are* LOGCFL*-complete under* LOGSPACE *reductions.*

4.4 Others

Given an EDTD, the NTDTA equivalent to it is immediately obtained in LOGSPACE. Thus Theorem 8 implies:

Corollary 3 *: TreeM(EDTD) is* LOGCFL*-complete under* LOGSPACE *reductions.*

For deterministic tree automata the situation is not as simple. It is easy to see that the RANK transformation does not preserve determinism. For the ranked case [27] gives an algorithm which works in LOGDCFL. By [38] LOGDCFL coincide with the class of languages recognizable by deterministic auxiliary pushdown automata in logarithmic space and polynomial time. The algorithm used in [27] is similar to the one given in Section 3.1 with depth-first left-first order for $<$. The difference is that the set Q of states of the automata is now part of the input and therefore Γ is an alphabet which letters are of size $k.\log|Q|$ where k is the rank of the tree. This is why an auxiliary stack of logarithmic size is required. In order to compute the state of a node from the states of its children it uses an intermediate storage of size $k.\log|Q|$ and thus cannot be extended to unranked trees. When the tree is unranked, thisstep can be done by guessing the (unique) state q of the target node of label a. Then the automata deterministically checks that the guess is correct by simulating the automata $\delta(a,q)$ on the word consisting of the already computed states of its children using the stack. Because there is a unique accepting computation, the overall complexity is in LOGUCFL. The precise complexity remains open.
For TreeM(DTDTA) the situation is slightly more complex for unranked trees than for ranked ones. Indeed it is LOGSPACE-complete under uAC^0 reductions for the ranked case. The algorithm given in [27] works as follows. It scans through all the leaves of the tree and for each of them checks that the path going from the root to that leaf does verify the transitions given by the automata. Because the automata is deterministic, this is sufficient for validation. The navigation in the tree is indeed in LOGSPACE, it remains to compute, given a node n of label a and a state q, the state of the i^{th} child of n. This can easily be done in LOGSPACE in the ranked case by scanning $\delta(a,q)$. When the input tree is unranked one as to extract from the automata of $\delta(a,q)$, the i^{th} character of the (unique) word of length p accepted by this automata, where p is the number of children of n. This can be done in NLOGSPACE (actually in ULOGSPACE). The precise complexity remains open.

Theorem 9 *: TreeM(DBUTA) is in* LOGUCFL.
TreeM(DTDTA) is in ULOGSPACE.

5 Query evaluation: XPath

We initiate the study of query evaluation complexity by considering XPath. XPath (XML Path Language) is a language for addressing parts of an XML document recommended by the W3C [3]. It is the starting point of several more powerful query languages such as XSLT and XQuery.
As we don't consider data value in this paper we will only deal with the corresponding subset of XPath. It contains all the navigational power of XPath but lack the possibility of concatenating string data value or performing arithmetical computation on number data value. It roughly correspond to the Core XPath as defined in [21] and can be evaluated in time $O(n.q)$ where n is the size of the XML document and q is the size of the Core XPath query. More formally we will use the following grammar for XPath expressions :

$$\begin{array}{rcl} \text{exp} & := & \text{path} \mid /\text{path} \\ \text{path} & := & \text{step}(/\text{step})^* \\ \text{step} & := & \text{sort} :: \Sigma \mid \text{sort} :: \Sigma[\text{pred}] \\ \text{pred} & := & \text{pred} \cup \text{pred} \mid \text{pred} \cap \text{pred} \mid \neg\text{pred} \mid \text{exp} \end{array}$$

where *sort* is an element of $\{$ *parent, ancestor, ancestor-or-self, preceding, preceding-sibling, self, child, descendant, descendant-or-self, following, following-sibling* $\}$. An XPath expression returns a set of nodes satisfying its path constraints. We refer to [3] for a precise semantics. Intuitively, $e[e']$, when evaluated at a node p, returns the nodes n such that the path from p to n verifies e and such that there exists a node m for which the path from n to m satisfies e'. The meaning of each *step* is intuitive.
Given a labeled tree t and an XPath expression e, we denote by $e(t)$ the set of nodes satisfying the path constraints of e. Again we divide the analysis into two parts. The first one will deal with the *data complexity* of evaluating XPath expressions where the query is fixed and only the data vary. The second one will consider its *combined complexity* where the query is also part of the input.

5.1 Data Complexity

Let e be a core XPath expression. We consider the following decision problem:

[Eval(e)] INPUT: a tree t and a node n
OUTPUT: true iff $n \in e(t)$.

Again the coding of the input tree is crucial for the data complexity. We will consider the two encodings presented in Section 2: DOM-like and SAX-like.

Theorem 10 *: Let e be an XPath expression,*
If t is given as a string, Eval(e) is in $u\mathrm{TC}^0$.
If t is given as a pointer structure, Eval(e) is in LOGSPACE.

proof: Consider first the case where t is coded as a string. We show that Eval(e) can be expressed by a FOM formula $\varphi_e(\text{root}, y)$ such that $t, i, j \models \varphi_e(x, y)$ iff the node pointed by j in t satisfy $e(t)$ when evaluated starting at node i. Then Theorem 10 follows from Lemma 1 and Lemma 5.
This is done by induction on e. We only give a few examples. If e is $e'/\mathit{step} :: a$, then $\varphi_e(x, y)$ is $\exists z\ \varphi_{e'}(x, z) \wedge \mathit{step}(z, y) \wedge Q_a(y)$, where $\mathit{step}(x, y)$ is the FOM formula checking that the node x and y are related as described by *step*. We gave examples of such formulas in Lemma 5 and Lemma 9, the others can be obtained similarly. If e is $e_1[e_2]$, then $\varphi_e(x, y)$ is $\varphi_{e_1}(x, y) \wedge \exists z \varphi_{e_2}(y, z)$.
When t is given by a pointer structure, the upper bound is straightforward from Lemma 5 and the above. □
The upper-bounds of Theorem 10 are tight. Indeed recall the problem ORD from the proof of Theorem 2 which checks in a branch that a node labeled a occurs before a node labeled b. It is LOGSPACE-complete under uAC^0 reductions. The result follows as this property can be expressed in XPath.

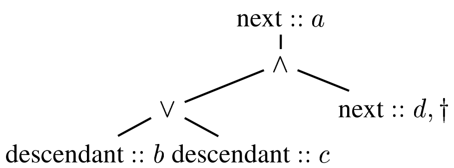
Remark : The full XPath also have the ability of counting the number of nodes that satisfy a given expression and perform tests on the result. Because FOM also have this capability, extending the above core fragment with such tests, assuming that the arithmetic stays within uTC^0 (recall that uTC^0 can perform comparisons, addition and multiplication), does not affect the results of Theorem 10.

5.2 Combined complexity

For the combined complexity e is part of the input. We also assume that the tree is given as a pointer structure. Formally the problem studied in this section is:

[Eval(XPath)] INPUT: an XPath expression e,
a tree t and a node n of t
OUTPUT: true iff $n \in e(t)$.

In all the algorithms below, the XPath query is viewed as a tree where each node is labeled either by a boolean operation ($\vee, \wedge, \neg$) or by a step-expression of the form pred :: a where $a \in \Sigma$. The tree t_e corresponding to a core XPath expression e is constructed inductively as follows. If $e = e_1/e_2$ where e_1 is a step-expression and e_2 a path-expression, then t_e is constructed by adding a new (root) node of label e_1 to t_{e_2} with an edge to the root of t_{e_2}. If $e = e_1[e_2]/e_3$ where e_1 is a step expression, e_3 a path-expression and e_2 a pred-expression, then t_e is constructed by adding two new nodes n and m to t_{e_2} and t_{e_3}. Their label are respectively e_1 and $\wedge$. n is the root of t_e and has an edge to m. m has an edge to each of the root of t_{e_2} and t_{e_3}. If $e = e_1 \cup e_2$ then t_e is constructed from t_{e_1} and t_{e_2} by adding a new root node n of label $\vee$ and connecting it to the roots of t_{e_1} and t_{e_2}. The others boolean operators are treated similarly. A special tag † is added for the leaf of the tree corresponding to the last step-expression of the path-expression which do not occur as a predicate. For instance the tree associated to the XPath expression next :: a[descendant :: $b \cup$ descendant :: c]/next :: d is



This query tree will never be materialized but all queries will be parsed by doing a depth-first left-first traversal of the corresponding tree. A pointer to an XPath expression we will always point to the first character of a step-expression

or a boolean operator and will thus correspond to a node of the query tree. Notice that, given an XPath expression as a string, the pointer to the next node in this depth-first traversal can be computed in LOGSPACE from the current position.
We first consider the Core XPath expressions containing no negation. Let Eval(XPath-) be the special case of Eval(XPath) where the XPath expression contains no negation. In that case evaluation can be achieved by guessing the data node that correspond to each node of the query tree, using a stack for storing partial computation.

Theorem 11 *: Eval(XPath-) is in* LOGCFL.

proof : Given a pointer i in a query e its *depth* is defined as the number of nodes of type step-expressions occurring in the path from the node n (corresponding to i) to the root of the tree corresponding to e, including n. In the example above, the depth of the node labeled "descendant :: c" is 2.
We use Sudborough characterization of LOGCFL ([38, 37]) and exhibit a nondeterministic auxiliary pushdown automata which recognize Eval(XPath) in logarithmic space and polynomial time. The automata will maintain a pointer p on the input XPath expression that will perform a preorder traversal of the query tree. Recall that this preorder traversal can be done in LOGSPACE. The automata will use another pointer q on the input tree that will point to the node of the input tree that will make true the query at p. The automata starts by setting p and q to the respective roots of the input query and input tree and works as follows.
Let p' be the successor of p in the preorder traversal of the query and n be the node pointed by q (if there is no successor of p the automata returns accepts). Assume first that p' is a descendant of p. The automata has to check that the corresponding path-expression is true. If the label of p' is boolean, p' is skipped and the automata move to the next node. If the label of p' is a step-expression, the automata guesses a node n' of the input tree that such that the path from n to n' satisfies the step-expression (if no such node exists then the automata rejects). If p' contains the special tag † then the automata check that n' correspond to the input node, otherwise it rejects. It then pushes the pointer of n' on the stack, reset p and q to their new value p' and q' and proceed the main loop with those new values.
Assume now that p' is an *uncle* of p. Let d be current depth of p and let p'' be the sibling of p' which is an ancestor of p. At this point the automata knows that the subtree rooted at p'' evaluates to true. Thus the automata now sets p to the first ancestor of p' which label is not a $\vee$. Let d' be the new depth of p, the automata pops the stack $d - d'$ time and sets q to the value in the top of the stack. It then proceed as above with these values for p, p', q, q'.
It can be verified that this algorithm is correct: it accepts iff $n \in e(t)$. It uses only pointer to its input and thus works in LOGSPACE. For each node in the query tree, it makes exactly one guess and one check for a step-expression. The later check can be done in time $O(|t|)$. Thus the automaton works in time $O(|e|.|t|)$. This proves the theorem. □
Note first that the straightforward deterministic variant of the above algorithm works in time exponential in the maximal depth of the query tree of the XPath expression. Therefore it is not clear whether Eval(XPath-) is in LOGDCFL.
In [10] counting techniques are used in order to show the closure of LOGCFL under complement. It could be also tempting to use this technique in order to generalize Theorem 11 with XPath expressions using negation and counting. Unfortunately, as shown in [10], each counting operation requires a time polynomial in the size of the data. Therefore the total running time of the nondeterministic auxiliary PDA would be exponential in the number of nested negation and counting, and thus would go beyond LOGCFL(unless the nesting is bounded). Indeed it is shown in [16] that nondeterministic auxiliary PDA running in logarithmic space and exponential time characterizes PTIME.
Theorem 11 shows that XPath- is highly parallelizable and can be done with limited memory resources as LOGCFL $\subseteq$ DSPACE($\log^2$). As evaluating XPath with limited resources is important (for streaming XML for instance, see [35]), we illustrate this further by giving two non trivial special cases of Eval(XPath) that can be evaluated in LOGSPACE.
Let Eval(XPath^1) (resp. Eval(XPath^{+*})) be the restriction of Eval(XPath) where each step-expression have a sort in { *previous-sibling, self, child, parent, next-sibling* } (resp. in { *self, descendant, descendant-or-self* }). XPath^1 allows only step-expression which can go to a node at distance at most one from the current node. XPath^{+*} allows only long distance steps in the future.
We have:

Theorem 12 *: Eval(XPath^1) is in* LOGSPACE.
Eval(XPath^{+*}) is LOGSPACE*-complete.*

proof : In the case of XPath^1 the first step of the algorithm is to remove the step-expressions that looks *backward*: *parent* and *previous sibling*. This can be done in LOGSPACE using the rewriting rules of [30]. [30] gave a more general algorithm that removes all *backward* step-expressions in any XPath expression. The algorithm described in

[30] requires exponential time in general but it can be checked that the same algorithm, when restricted to expressions of XPath^1 is in LOGSPACE. Therefore we can assume that XPath^1 expressions contains only *forward* step-expressions. In both cases the algorithm perform a depth-first traversal of the tree of the XPath expression, and, for each node parse the data using a depth-first left-first traversal in order to find a matching node. At each step, it only stores the current pointer to the tree and the pointer to the matching node in the data tree. This requires only a space logarithmic in the input size. Some extra logarithmic space is used for local computations. The trick is that when no matching is found, backtracking can be performed in LOGSPACE. In the case of XPath^1 this is straightforward as each step can be executed in reverse order. In the case of XPath^{+*} this is due to the fact that, for each branch, it is sufficient to consider the first possible match, the $*$ recursion taking care of all the others.
Recall also that a boolean expressions can be validated in LOGSPACE by skipping the disjunctions as soon as one of its subtree evaluates to 1 and successively evaluating all the subtrees of an conjunction. Negation are handled similarly. In order to simplify the presentation, in the following we ignore the nodes of the tree labeled with a boolean predicate and always evaluates all its subtrees. Using the boolean expression evaluation described above it is straightforward to extend it in order to includes boolean gates while maintaining the LOGSPACE constraints.
The algorithm works similarly in both cases. Let `next` be a subroutine that computes the depth-first traversal of the tree of the XPath expression. This can be done in LOGSPACE. The cases of Eval(XPath^1) and Eval(XPath^{+*}) only differs in the subroutines `backtrack` and `fetch`.
`fetch`(β, i, j) takes as input two pointers i and j to the data tree and a pointer β to the query tree. It returns a pointer k to a node such that, k strictly follows j in the depth-first left-first order of the data tree, and the path from i to k satisfies the step predicate of the label of β. If no such node exists, it returns false. This subroutine can easily be done in LOGSPACE in both cases.
`backtrack`(α, i, β) takes as input two pointers α and β to the query tree such that the father δ of β is an ancestor of α, and a pointer i to the data tree. It then computes a pointer to a data node k such that: the path from k to i satisfies the path-expression of the subset of the query going from δ to α. In the case of XPath^1 this k is unique and can be computed in LOGSPACE. In the case of XPath^{+*} we return the ancestor of i which is closest to the root such that the path from the root to k satisfies the path constraints given by the subquery query given by the path from the root to δ. It can be shown that this k can be computed in LOGSPACE and that this strategy works with the following algorithm:

```
   /* initialization */
   α = root(query tree)
   i = root(data tree)
   if label(α) ≠ label(i) return FALSE

   /* main loop */
1  if done(α) return TRUE
   /* process the next node of the query */
   β = next(α)
   /* i was the data node corresponding to α */
   i = backtrack(α,i,β)
   /* it is now the data node corresponding
        to the father of β */
   α = father(β)
   j = i

   /* find a matching data node in
        a depth-first way */
2  if (fetch(β,i,j)) /* there is a matching j */
      then i = j; α = β; goto (1)
   else /* backtrack */
   if i=root return FALSE
   else i = backtrack(α,i,α)
   /* i is now the data node corresponding
        to the father of α */
   β = α; α = father(β); j = i
   goto (2)
```

This proves the upper-bounds.

The lower bounds for Eval(XPath^{+*}) is proved as in the paragraph following Theorem 10 by a reduction to ORD. □

6 Discussion

We have seen that checking the membership of an unranked labeled tree in a regular tree language was not more difficult than checking the membership of a string in a regular string language. Thus the data complexity of checking that an XML document conform a type is independent of the typing system as soon as this typing system is an extension of regular string languages and a restriction of regular tree languages. On the other hand we have seen that the coding used for the data tree is crucial and that two of the most used models of XML data give two different complexity bounds: uNC^1 vs. LOGSPACE.
The combined complexity for type validation does not depend on the coding of the input data tree but on the type language. It was shown that, depending on the typing language, complexities could vary from LOGSPACE-complete to LOGCFL-complete.
The combined complexity of Core XPath is actually PTIME-complete. This was shown in [22]. The same paper also show that the complexity given in Theorem 11 is tight by proving that Eval(XPath-) is LOGCFL-complete.

References

[1] Document Object Model (DOM). Available at http://www.w3c.org/dom.

[2] Simple API for XML (SAX). Available at http://www.saxproject.org/.

[3] XML Path Language (XPath). Available at http://www.w3.org/TR/2002/WD-xpath20-20020816.

[4] S. Abiteboul. Semistructured data: from practice to theory. In *Proc. IEEE Conf. on Logic in Computer Science*, 2001.

[5] A. V. Aho and J. D. Ullman. Translations on a context-free grammar. *Inform. and Control*, 19:439–475, 1971.

[6] E. Allender. The division breakthroughs. *The Computational Complexity Column, EATCS Bulletin*, 74:61–77, 2001.

[7] D. A. M. Barrington and J. Corbett. On the relative complexity of some languages in NC1. *Information Processing Letter*, 32:251–256, 1989.

[8] D. A. M. Barrington, N. Immerman, and H. Straubing. On Uniformity within NC1. *Journal of Computer and System Sciences*, 41(3):274–306, 1990.

[9] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proc. of Intl. Conf. on Database Theory*, 1999.

[10] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. Inductive Counting for Complementation Problems. *SIAM J. Comput.*, 18(3):559–578, 1989.

[11] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets. Available at ftp://ftp11.informatik.tu-muenchen.de/pub/misc/caterpillars/, 2001.

[12] A. Brüggemann-Klein and D. Wood. One-Unambiguous Regular Languages. *Information and Computation*, 142(2), 1998.

[13] S. R. Buss. The Boolean formula value problem. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, 1987.

[14] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1998.

[15] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available at http://www.grappa.univ-lille3.fr/tata, 1999.

[16] S. A. Cook. Characterizations of Pushdown Machines in Terms of Time-Bounded Computers. *Journal of ACM*, 18(1):4–18, 1971.

[17] S. A. Cook and P. McKenzie. Problems Complete for Deterministic Logarithmic Space. *Journal of Algorithms*, 8:385–394, 1987.

[18] J. Engelfriet, H. Hoogeboom, and J.-P. van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

[19] J. Engelfriet and H. J. Hoogeboom. Tree-walking pebble automata. In J. Karhumäki, H. Maurer, G. Paun, and G.Rozenberg, editors, *Jewels are forever, contributions to Theoretical Computer Science in honor of Arto Salomaa*, pages 72–83. Springer-Verlag, 1999.

[20] K. Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54(3):400–411, 1997.

[21] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. of Intl. Conf. on Very Large Data Bases*, 2002.

[22] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. ACM Symp. on Principles of Database Systems*, 2003.

[23] G. Gottlob, N. Leone, and F. Scarcello. The Complexity of Acyclic Conjunctive Queries. *Journal of ACM*, 43(3):431–498, 2001.

[24] T. Jiang and B. Ravikumar. A note on the space complexity of some decision problems for finite automata. *Information Processing Letters*, 40:25–31, 1991.

[25] N. D. Jones. Space-Bounded Reducibility among Combinatorial Problems. *Journal of Computer and System Sciences*, 11:68–85, 1975.

[26] S. Lindell. A Logpsace Algorithm for Tree Canonization. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, 1992.

[27] M. Lohrey. On the Parallel Complexity of Tree Automata. In *Rewriting Techniques and Applications (RTA'01)*, 2001.

[28] R. McNaughton. Parenthesis Grammars. *Journal of Computer and System Sciences*, 1967.

[29] F. Neven. Automata, Logic, and XML. In *Proc. of Computer Science Logic*, 2002.

[30] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management, XMLDM 2002*, 2002.

[31] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. ACM Symp. on Principles of Database Systems*, 2000.

[32] Y. Papakonstantinou and V. Vianu. Incremental Validation of XML Documents. In *Proc. of Intl. Conf. on Database Theory*, 2003.

[33] G. Rozenberg and A. Salomaa, editors. *Handbook of formal language*. Springer, 1997.

[34] W. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21:218–235, 1980.

[35] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proc. ACM Symp. on Principles of Database Systems*, 2002.

[36] D. Suciu. Typechecking for Semistructured Data. In *Proc. workshop on Database Programming Language*, 2001.

[37] I. Sudborough. Time and tape bounded auxiliary pushdown automata. In *Mathematical Foundations of Computer Science (MFCS)*, volume 53 of *LNCS*, pages 493–503. Springer-Verlag, 1977.

[38] I. Sudborough. On the tape complexity of deterministic context-free languages. *Journal of ACM*, 25(3):405–414, 1978.

[39] H. Venkateswaran. Properties that characterize LOGCFL. *Journal of Computer and System Sciences*, 43:380–404, 1991.

[40] H. Vollmer. *Introduction to circuit complexity*. Springer, 1999.