# Incremental Validation of XML Documents

Y. Papakonstantinou V. Vianu

## Abstract

We investigate the incremental validation of XML documents with respect to DTDs and XML Schemas, under updates consisting of element tag renamings, insertions and deletions. DTDs are modeled as extended context-free grammars and XML Schemas are abstracted as "specialized DTDs", allowing to decouple element types from element tags. For DTDs, we exhibit an $O(m \log n)$ incremental validation algorithm using an auxiliary structure of size $O(n)$, where $n$ is the size of the document and $m$ the number of updates. For specialized DTDs, we provide an $O(m \log^2 n)$ incremental algorithm, again using an auxiliary structure of size $O(n)$. This is a significant improvement over brute-force re-validation from scratch.

## 1 Introduction

The emergence of XML as a standard representation format for data on the Web has led to a proliferation of databases that store, query, and update XML data. Typically, valid XML documents must conform to a specified type that places structural constraints on the document. When an XML document is updated, it has to be verified that the new document still satisfies its type. Doing this efficiently is a challenging problem that is central to many applications. Brute-force validation from scratch is often not practical, because it requires reading and validating the entire database following each update. Instead, it is desirable to develop algorithms for incremental validation. However, this approach has been largely unexplored. In this paper we investigate the efficient incremental validation of updates to XML documents.

An XML document can be viewed abstractly as a tree of nested elements. The basic mechanism for specifying the type of XML documents is provided by Document Type Definitions (DTDs) [W3C98]. DTDs can be abstracted as extended context-free grammars (CFGs). Unlike usual CFGs, the productions of extended CFGs have regular expressions on their right-hand sides. An XML document satisfies a DTD if its abstraction as a tree is a derivation tree of the extended CFG corresponding to the DTD. A more recent XML typing mechanisms uses the XML Schema standard [W3C01], which extends DTDs in several ways. Most notable is the ability to decouple the type of an element from its label. In this paper we abstract XML schemas by *specialized* DTDs, that capture precisely this aspect. It is a well-known and useful fact that specialized DTDs define precisely the regular languages of unranked trees, and so are equivalent to top-down (and bottom-up) non-deterministic tree automata.

Verifying that a word satisfies a regular expression[1] is the starting point in checking that an XML document satisfies a DTD. An obvious way to do this following an update is to verify it from scratch, i.e. run the updated sequence of labels through the non-deterministic finite automaton (NFA) corresponding to the regular expression. However, this requires $O(n)$ steps, under any reasonable set of unit operations, where $n$ is the length of the sequence (note that, in complexity-theoretic terms, membership of a word in a regular language is complete in $NC^1$ under DLOGTIME reductions [Vol99].) We can do better by using incremental validation, relying on an appropriate auxiliary data structure. Indeed, we provide such a structure and corresponding incremental validation algorithm that, given a regular expression $r$, a string $s$ of length $n$ that satisfies $r$, and a sequence of $m$ updates (inserts, deletes, label renamings) on $s$, checks[2] in $O(m \log n)$ whether the updated string satisfies $r$. The auxiliary structure we use materializes in advance relations that describe state transitions resulting from traversing certain substrings in $s$. These are placed in a balanced tree structure that is maintained similarly to B-trees and is well-behaved under insertions and deletions. The size of the auxiliary structure is $O(n)$. In addition, we provide an $O(m \log n)$ time algorithm that maintains the auxiliary structure, so that subsequent updates can also be incrementally validated.

Our approach to incremental validation of trees with respect to specialized DTDs builds upon the incremental validation algorithm for strings. DTDs turn out to be easier to validate than specialized DTDs. Indeed, based on the algorithm for string validation, incremental validation of $m$ updates to a tree $T$ with respect to a DTD can be done in time $O(m \log |T|)$ using an auxiliary structure of size $O(|T|)$ which can also be maintained in time $O(m \log |T|)$. Specialization introduces

[1] A word *satisfies* a regular expression if it belongs to the corresponding language.

[2] For readability, we provide here the complexity with respect to the string and update sequence, for fixed (specialized) DTD or regular expression. The combined complexity is spelled out in the paper.

another degree of complexity. Intuitively, this is due to the fact that an update to a single node may have global repercussions for the typing of the tree. This stands in contrast with DTDs without specialization, where a single update to a node needs to be validated only with respect to the type of its parent and the sequence of its children, so has local impact on type checking.

We first attempt a rather straightforward extension of the incremental validation for DTDs and obtain an algorithm of time complexity $O(m\ depth(T) \log |T|)$ using an auxiliary structure of size $O(|T|)$. However, this is not satisfactory when the tree is narrow and deep. In the worst case, $depth(T) = |T|$. To overcome this, we develop a more subtle approach that has the following main ingredients: First, the unranked tree $T$ representing the XML document is mapped into a binary tree encoding that allows us to unify the horizontal and vertical components of validation. Then the specialized DTD is translated into a bottom-up nondeterministic tree automaton accepting precisely the encodings of valid documents. Finally, an incremental validation algorithm for binary trees with respect to tree automata is developed, based on a divide-and-conquer strategy that focuses on computations along certain paths in the tree chosen to appropriately divide the work. Auxiliary structures are associated to each of these paths. The resulting incremental validation algorithm has time complexity $O(m \log^2 |T|)$ and uses an auxiliary structure of size $O(|T|)$.

**Related Work** As mentioned earlier, XML databases need to efficiently validate updates on their content. Ipedo's XML database [Ipe] validates update commands with respect to XML Schemas; however, to our knowledge no technical information is publicly available on the underlying structures and algorithms. Another application where efficient validation is useful is XML editors (see [XMLa] for a survey of available products). Some XML editors like XMLMind [XMLc] and XMLSpy [XMLb] feature incremental validation of DTDs. Recently, XMLSpy also included validation of XML Schemas [XMLb]. No information is provided on their incremental validation algorithms.

Note that our abstraction of the content models of DTDs [W3C98] by arbitrary regular expressions removes the requirement for 1-unambiguous regular expressions. Furthermore, our abstraction of XML Schemas [W3C01] as specialized DTDs has assumed that the content of a type is described by an arbitrary regular expression over types. XML Schema restricts those regular expressions by requiring that no two different types with the same element name may participate in the same regular expression. The motivation of such restrictions in DTDs and XML Schemas has been the need for efficient validators. However, our efficient incremental validation algorithms have not made use of those restrictions besides stating how 1-unambiguity changes the complexity of the problem.

Closely related to incremental validation is incremental parsing, which is key to incremental program compilation. Research on incremental parsing has focused on LR parsing [GM80, WG98, JG82, Lar95, Pet95] and LL (recursive descent parsing) [MPS90, Li95, Lin93], since programming languages are typically described by LR(0), LR(1), LL(1), LALR(1) and LL(1) grammars. All techniques start by parsing the input text and produce a parse tree, which is typically annotated with auxiliary information. The parse tree is updated as a result of the updates to the input text. A typical theme of the incremental parsing techniques is identifying minimal structural units of the parse tree that are affected by the modifications (see [GM80] for LR(0) parsing and [Lar95] for a generalization to LR($k$).) However, the performance of the incremental parsing algorithms is hard to compare to our validation algorithm because of the differences in settings and goals, which typically involve minimization of the changes on the parse tree. Indeed, the best-case performance of incremental parsers will generally beat the one of our regular expression validation algorithm, which always takes $O(\log n)$ steps for a single update. This is because incremental parsers take advantage of natural "termination points" used in programming languages syntax [Lin93], that typically occur close to the update. Logarithmic complexity in the size of the string is achieved for LALR grammars by [WG98] but only if the grammar is such that its parse trees have depth $O(\log n)$ for a string of length $n$. One can easily see that there are LALR grammars that do not meet this property, and neither do the CFGs corresponding to DTDs. Furthermore, [WG98] require that the interpretation of iterative sequences be independent of the context. See the appendix for an example from [WG98] of a regular expression violating this condition.

The complexity of validation is related to that of membership of a word in a regular language, and of a tree in a regular tree language. The problem of word membership in a regular language is known to be complete in uniform $NC^1$ under DLOGTIME reductions [Loh01] and acceptance of a tree over a ranked alphabet by a tree automaton is complete in uniform $NC^1$ under DLOGTIME reductions if the tree is presented in prefix notation [Vol99], and complete in LOGSPACE if the tree is presented as a list of its edges [Seg02]. To our knowledge, no complexity results exist on the incremental variants of these problems, with the exception of a result of [PI97] discussed below.

Incremental evaluation of queries by first-order means is studied by [DS95] using the notion of first-order incremental evaluation systems (FOIES) A related descriptive complexity approach to incremental computation is developed by Patnaik and Immerman

in [PI97]. They define the dynamic complexity class Dyn-FO (equivalent to FOIES), consisting of properties that can be incrementally verified by first-order means. They exhibit various problems in Dyn-FO, such as multiplication, graph connectivity, and bipartiteness. Most relevant to our work, they show that membership of a word in a regular language is in Dyn-FO. For label renamings, they sketch an approach similar to ours. The incremental algorithm and auxiliary structure for node insertions and deletions that modify the length of the string are not spelled out. Also, no extension to regular tree languages is discussed. The study in [PI97] is pursued in [HI02], where an extension of Dyn-FO is introduced and it is shown that the single-step version of the circuit value problem is complete in Dyn-FO under certain reductions. Complexity models of incremental computation are considered in [MSVT94]. The focus is on the classes *incr*-POLYLOGTIME (*incr*-POLYLOGSPACE) of properties that can be incrementally verified in polylogarithmic time (space). Interesting connections to parallel complexity classes are exhibited, as well as complete problems for classical complexity classes under reductions in the above incremental complexity classes.

**Organization** The paper is organized as follows. Section 2 presents our abstraction of XML documents, DTDs, and XML Schemas, as well as the connection between specialized DTDs and tree automata. We also spell out formally the incremental validation problem and the assumptions made in our complexity analysis. In Section 3 we examine the incremental validation of strings with respect to regular expressions and develop the core divide-and-conquer strategy used later for DTD validation. Section 4 presents the validation algorithm for DTDs and a first attempt to handle specialized DTDs. Finally, Section 5 presents the full algorithm for specialized DTDs yielding $O(m \log^2 |T|)$ incremental validation. Section 6 contains some concluding remarks and future work.

## 2 Basic Framework

We introduce here the basic formalism used throughout the paper, including our abstractions of XML documents, DTDs, and XML Schemas. We also recall basic definitions relating to tree automata.

**Labeled ordered trees** We abstract XML documents as labeled ordered trees. Our abstraction ignores data values present in XML documents, because their validation with respect to an XML Schema is trivial. For example, an XML document holding ads for used cars and new cars is shown in Figure 1 (left), together with its abstraction as a labeled tree.

An *ordered labeled tree* over finite alphabet $\Sigma$ is a pair $T = \langle t, \lambda \rangle$, where $t$ is an ordered tree and $\lambda$ is a mapping associating to each node $n$ of $t$ a label $\lambda(n) \in \Sigma$. Trees are assumed by default to be unranked, i.e. there is no fixed bound on the number of children each node may have. The set of all labeled ordered trees over $\Sigma$ is denoted by $\mathcal{T}_\Sigma$. We sometimes denote a tree consisting of a root $v$ with subtrees $T_1 \dots T_k$ by $v(T_1 \dots T_k)$. We will also consider binary trees, where each node has at most two children. If every internal node has *exactly* two children, the binary tree is called *complete*.

We assume a representation of trees that allows one to find in $O(1)$ (i) the label, (ii) the parent, (iii) the immediate left (right) sibling, and (iv) the first child of a specified node.

**Types and DTDs** As usual, we define XML document types in terms of the document's structure alone, ignoring data values. The basic specification method is (an abstraction of) DTDs. A DTD consists of an extended context-free grammar over alphabet $\Sigma$ (we make no distinction between terminal and non-terminal symbols). In an extended CFG, the right-hand sides of productions are regular expressions over $\Sigma$. An ordered labeled tree $\langle t, \lambda \rangle$ over $\Sigma$ satisfies a DTD $d$ if the tree $\langle t, \lambda \rangle$ is a derivation tree of the grammar. For example, the tree is valid with respect to the DTD in Figure 1.

The start symbol of a DTD $d$ is denoted by $root(d)$. We can assume without loss of generality that for each $a \in \Sigma$ the DTD has a single rule $a \ \rightarrow r_a$ with $a$ on the left-hand side. and we denote by $N_a$ a standard nondeterministic finite-state automaton (NFA) recognizing the language $r_a$. The set of labeled trees satisfying a DTD $d$ is denoted by $sat(d)$.

We use the following notation for NFA. An NFA is a 5-tuple $N = \langle \Sigma, Q, q_0, F, \delta \rangle$ where $\Sigma$ is a finite alphabet, $Q$ is a finite set of *states*, $q_0 \in Q$ is the *start state*, $F \subseteq Q$ is the set of *final states*, and $\delta$ is a mapping from $\Sigma \times Q$ to $\mathcal{P}(Q)$. A string $a_1 \dots a_n$ is accepted by $N$ iff there exists a mapping $\sigma : \{1, \dots, n\} \rightarrow Q$ such that $\sigma(a_1) \in \delta(a_1, q_0)$, $\sigma(a_n) \in F$, and for each $i, 1 \leq i < n$, $\sigma(a_{i+1}) \in \delta(a_{i+1}, \sigma(a_i))$. The set of strings accepted by $N$ is denoted $L(N)$. $N$ is a *deterministic finite-state automaton* (DFA) iff $\delta$ returns singletons on each input. Recall that for each regular expression $r$ there exists an NFA $N$ whose number of states is linear in $r$, such that $N$ accepts the regular language $r$. In general, a DFA accepting $r$ requires exponentially many states wrt $r$. However, for certain classes of regular expressions, the corresponding DFA remains linear in the expression. One such class consists of the 1-unambiguous regular languages [BKW98]. This is relevant in the context of XML types, since DTDs and XML Schemas require the regular expressions used to specify the contents of elements to be 1-unambiguous.

An important limitation of DTDs is the inability to separate the *type* of an element from its *name*. For

```
<dealer>
 <UsedCars>
  <ad>
   <model>Honda</model>
   <year>92</year>
  </ad>
 </UsedCars>
 <NewCars>
  <ad>
   <model>BMW</model>
  </ad>
 </NewCars>
</dealer>
```

dealer
UsedCars NewCars
ad ad
model year model

```
<!DOCTYPE dealer>
<!ELEMENT dealer
 (UsedCars, NewCars)>
<!ELEMENT UsedCars (ad*)>
<!ELEMENT NewCars (ad*)>
<!ELEMENT ad (model, year?)>
<!ELEMENT model PCDATA>
<!ELEMENT year PCDATA>
```

| | | |
|---|---|---|
| $root$ | $:$ | $dealer$ |
| $dealer$ | $\rightarrow$ | $UC\ NC$ |
| $UC$ | $\rightarrow$ | $ad^*$ |
| $NC$ | $\rightarrow$ | $ad^*$ |
| $ad$ | $\rightarrow$ | $model\ (year\vert\epsilon)$ |
| $model$ | $\rightarrow$ | $\epsilon$ |
| $year$ | $\rightarrow$ | $\epsilon$ |

| | |
|---|---|
| $root : d^t$ | |
| $d^t \rightarrow UC^t\ NC^t$ | $\mu(d^t) = dealer$ |
| $UC^t \rightarrow (ad^u)^*$ | $\mu(UC^t) = UC$ |
| $NC^t \rightarrow (ad^n)^*$ | $\mu(NC^t) = NC$ |
| $ad^u \rightarrow m^t\ y^t$ | $\mu(ad^u) = ad$ |
| $ad^n \rightarrow m^t$ | $\mu(ad^n) = ad$ |
| $m^t \rightarrow \epsilon$ | $\mu(m^t) = model$ |
| $y^t \rightarrow \epsilon$ | $\mu(y^t) = year$ |

Figure 1: XML, DTD and specialized DTD ($UC$ and $NC$ stand for `UsedCars` and `NewCars`)

example, consider the dealer document in Figure 1. Used cars have model and year while new cars have model only. There is no mechanism to specify this using DTDs, since rules depend only on the name of elements, and not on its context. To overcome this limitation, XML Schema provides a mechanism to decouple element names from their types and thus allow context-dependent definitions of their structure. We abstract this mechanism using the notion of *specialized DTD* (studied in [PV00] and equivalent to formalisms proposed in [BM99, CDSS98]).

**Definition 2.1** (**Specialized DTD**) A *specialized DTD* is a 4-tuple $\langle \Sigma, \Sigma^t, d, \mu \rangle$ where $\Sigma$ is a finite alphabet of labels, $\Sigma^t$ is a finite alphabet of *types*, $d$ is a DTD over $\Sigma^t$ and $\mu$ is a mapping from $\Sigma^t$ to $\Sigma$. $\diamondsuit$

Intuitively, $\Sigma^t$ provides, for each $a \in \Sigma$, a set of types associated to $a$, namely those $a^t \in \Sigma^t$ for which $\mu(a^t) = a$. In our specialized DTD example (lower right corner of Figure 1) we create two types for the element *ad*: a type $ad^n$ whose content is just a "model" type, and a type $ad^u$ whose content is "model" and "year". Note that $\mu$ induces a homomorphism on words over $\Sigma^t$, and also on trees over $\Sigma^t$ (yielding trees over $\Sigma$). We also denote by $\mu$ the induced homomorphisms.

Let $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$ be a specialized DTD. A tree $t$ over $\Sigma$ satisfies $\tau$ (or is *valid* wrt $\tau$) if $t \in \mu(sat(d))$. Thus, $t$ is a homomorphic image under $\mu$ of a derivation tree in $d$. Equivalently, a labeled tree over $\Sigma$ is valid if it can be "specialized" to a tree that is valid with respect to the DTD over the alphabet of types. The set of all trees over $\Sigma$ that are valid w.r.t. $\tau$ is denoted $sat(\tau)$. When $\tau$ is clear from the context, we simply say that a tree is *valid*.

**Tree automata** There is a powerful connection between specialized DTDs and *tree automata*: they are precisely equivalent, and define the *regular tree languages* [BKMW01]. We will make use of this connection in the paper.

Tree automata are devices whose purpose is to accept or reject an input tree. Classical tree automata are defined on complete binary trees. As in the case of string automata, there are several equivalent variants: top-down nondeterministic automata are equivalent to bottom-up (non)-deterministic ones. In contrast to string automata, top-down deterministic automata are weaker than their non-deterministic counterpart. We next review bottom-up non-deterministic tree automata on complete binary trees. (For technical reasons that will become clear shortly, we assume that all leaves have the same label #.)

**Definition 2.2** (**Bottom-up non-deterministic tree automaton**) A bottom-up non-deterministic tree automaton (BNTA) is a 5-tuple $A = \langle \Sigma, Q, Q_0, q_f, \delta \rangle$ where $\Sigma$ is a finite alphabet, $Q$ is a finite set of *states*, $Q_0$ is the set[3] of *start states* ($Q_0 \subseteq Q$), $q_f$ is the *accept state* ($q_f \in Q$) and $\delta$ is a mapping from $\Sigma \times Q \times Q$ to $\mathcal{P}(Q)$.

A tree $T = \langle t, \lambda, \rangle$ is *accepted* by the automaton $A$ iff there is a mapping $\sigma$ from the nodes of $t$ to $Q$ such that: (i) if $n$ is a leaf then $\sigma(n) \in Q_0$, (ii) if $n$ is an internal node with children $n_1, n_2$ then $\sigma(n) \in \delta(\lambda(n), \sigma(n_1), \sigma(n_2))$, and (iii) if $n$ is the root then $\sigma(n) = q_f$. The set of trees accepted by $A$ is denoted by $\mathcal{T}(A)$. $\diamondsuit$

There is a *prima facie* mismatch between DTDs and tree automata: DTDs describe unranked trees, whereas classical automata describe binary trees. There are two ways around this. First, unranked trees can be encoded in a standard way as binary trees. Alternatively, the machinery and results developed for regular tree languages can be extended to the unranked case, as described in [BKMW01]. For technical reasons, it will be useful to adopt here the first approach.

[3]Some definitions of BNTA require a single start state for each leaf symbol, and allow a set of final states. Having multiple start states and a single final state is a harmless variation, convenient here for technical reasons.

**The incremental validation problem** Given a (specialized) DTD $\tau$, a tree $T \in sat(\tau)$, and a sequence $s$ of updates to $T$ yielding another tree $T'$, we wish to efficiently check if $T' \in sat(\tau)$. In particular, the cost should be less than re-validation of $T'$ from scratch. The individual updates are the following:

(a) replace the current label of a specified node by another label,
(b) insert a new leaf node after a specified node,
(c) insert a new leaf node as the first child of a specified node, and
(d) delete a specified leaf node.

We allow some cost-free one-time pre-processing to initialize incremental validation, such as computing the NFA corresponding to the regular expressions used by the DTDs. We will also initialize and then maintain an auxiliary structure $\mathcal{A}(T)$ to help in the validation. The cost of the incremental validation algorithm is evaluated with respect to:

(a) the time needed to validate $T'$ using $T$ and $\mathcal{A}(T)$, as a function of $|T|$ and $|s|$
(b) the time needed to compute $\mathcal{A}(T')$ from $T, s$, and $\mathcal{A}(T)$,
(c) the size of the auxiliary structure $\mathcal{A}(T)$ as a function of $|T|$.

The analysis will also make explicit the combined complexity taking into account the specialized DTD.

# 3 Warmup: Incremental Validation of Strings

As warmup to the validation problem, we consider in this section the incremental validation of *strings* with respect to a regular language specified by an NFA. We first consider the case when all updates consist of label renamings. We discuss inserts and deletes later.

Consider an NFA $N = \langle \Sigma, Q, q_0, F, \delta \rangle$, and a string $a_1 \ldots a_n \in L(N)$. For compatibility with our tree formalism, we view a string as a sequence of nodes (or elements) each of which has a label. When there is no confusion we sometimes blur the distinction between an element and its label.

Consider a sequence of element renamings $u(a_{i_1}, b_1), \ldots, u(a_{i_m}, b_m)$, where $i_1 < i_2 < \ldots < i_m$. The renaming $u(a_{i_j}, b_j)$ requires that the label of element $a_{i_j}$ be renamed to $b_j$. We would like to efficiently check whether the updated string $a_1 \ldots a_{i_1-1} b_1 a_{i_1+1} \ldots a_{i_m-1} b_m a_{i_m+1} \ldots a_n \in L(N)$. Validating the new string from scratch by running it through $N$ takes $O(n|Q|^2 \log |Q|)$. We can easily do better by maintaining some auxiliary information. For simplicity in the presentation, we assume that we can find the rank of a specified node among its siblings in $O(1)$. This assumption is removed later.

Consider the case of a single renaming $u(i, b)$ for $1 \leq i \leq n$. Suppose that we have pre-computed, for each $i$, $1 < i < n$, the sets $\mathsf{Pre}(i) = \delta(q_0, a_1 \ldots a_{i-1})$ and $\mathsf{Post}(i) = \{s \mid \delta(s, a_{i+1} \ldots a_n) \in F\}$. If we precompute $\mathsf{Pre}$ and $\mathsf{Post}$ in arrays then we can retrieve $\mathsf{Pre}(i)$ or $\mathsf{Post}(i)$ in $O(|Q|)$. An $O(|Q|^2)$ algorithm for checking whether the string is in $L(N)$ following the update $u(i, b)$ is now obvious: If there is a state $s_1 \in \mathsf{Pre}(i)$, a state $s_2 \in \mathsf{Post}(i_1 + 1)$ such that $s_2 \in \delta(b, s_1)$ then the updated string is in $L(N)$.

However, the $\mathsf{Pre}$ and $\mathsf{Post}$ technique does not scale to $m$ updates. Furthermore, maintaining $\mathsf{Pre}$ and $\mathsf{Post}$ is problematic because, following each update $u(i, b)$, we need to recompute all $\mathsf{Pre}(j)$ for $j > i$ and $\mathsf{Post}(j)$ for $j < i$. This requires $O(n|Q|^2 \log |Q|)$ time.

As the next step in the warmup, we can try to keep some additional auxiliary information in order to better handle multiple updates. For each $i, j$, $1 \leq i < j \leq n$, let $T_{ij}$ be the *transition relation* $\{\langle p, q \rangle \mid p, q \in Q, q \in \delta(p, a_i \ldots a_j)\}$. Note that $T_{ij} = T_{ik} \circ T_{kj}$, $i < k < j$, where $\circ$ denotes composition of binary relations. We also denote by $\delta_a$ the relation $\{\langle p, q \rangle \mid q \in \delta(p, a)\}$ for $a \in \Sigma$. If all $T_{ij}$ are available, then checking validity of the updated string $a_1 \ldots a_{i_1-1} b_1 a_{i_1+1} \ldots a_{i_m-1} b_m a_{i_m+1} \ldots a_n$ reduces to verifying that

$$\langle q_0, f \rangle \in T_{0(i_1-1)} \circ \delta_{b_1} \circ T_{(i_1+1)(i_2-1)} \circ \ldots \circ T_{(i_m+1)(n)}$$

for some $f \in F$. This takes time $O(m|Q|^2 \log |Q|)$, if we assume that we have precomputed in a 2-dimensional array all relations $T_{ij}$. In particular, the composition of two relations is a join operation. It can be accomplished in $O(|Q|^2 \log |Q|^2) = O(|Q|^2 \log |Q|)$ by employing sort-merge join. Each relation is sorted in $O(|Q|^2 \log |Q|)$ and then they are merged in $O(|Q|^2)$. The same complexity can be derived if we assume binary tree indices on each attribute of the relations and we employ index-based join [GMUW01]. The size of the array required for the precomputation is $n^2|Q|^2$. However, maintaining the precomputed structure is prohibitively expensive, since we have to recompute every relation $T_{ij}$ if there is an update between the $i$th and $j$th position of the string. We are therefore led to consider a more promising approach, which provides the basis for the solution we adopt.

**Divide-and-conquer validation** We describe a divide-and-conquer approach that allows validating a sequence of $m$ renamings to a string of length $n$, as well as update the auxiliary structure, in $O(m|Q|^2 \log |Q| \log n)$ time. The size of the auxiliary structure is $O(|Q|^2 n)$. Note that the approach below is similar to that briefly sketched in [PI97].

For simplicity, assume first that $n$ is a power of 2, say $n = 2^k$. The main idea is to keep as auxiliary information just the $T_{ij}$ for intervals $[i, j]$ obtained by recursively splitting $[1, n]$ into halves, until $i = j$. More precisely, consider the *transition relation tree* $\mathcal{T}_n$

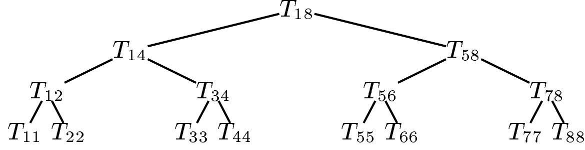


Figure 2: The tree $\mathcal{T}_{18}$

whose nodes are the sets $T_{ij}$, defined inductively as follows:

- the root is $T_{1,2^k}$
- each node $T_{ij}$ for which $j - i > 0$ has children $T_{ik}$ and $T_{(k+1)j}$ where $k = \frac{j-i+1}{2}$,
- $T_{ii}$ are leaves, $1 \leq i \leq n$.

For example, $\mathcal{T}_8$ is shown in Figure 2.

Note that $\mathcal{T}_n$ has $n + (n/2) + \ldots + 2 + 1 = 2n - 1$ nodes and has depth $\log n$. Thus, the size of the auxiliary structure is $O(n|Q|^2)$.

Consider now a string $a_1 \ldots a_n \in L(N)$, and a sequence of renamings $u(i_1, b_1), \ldots, u(i_m, b_m)$, where $i_1 < i_2 < \ldots < i_m$. The updated string is $a_1 \ldots a_{i_1-1} b_1 a_{i_1+1} \ldots a_{i_m-1} b_m a_{i_m+1} \ldots a_n$. Note that the relations $T_{ij}$ that are affected by the updates are those laying on the path from a leaf $T_{i_v i_v}$ $(1 \leq v \leq m)$ to the root of $\mathcal{T}_n$. Let $\mathcal{I}$ be the set of such relations, and note that its size is at most $m \log n$.

The tree $\mathcal{T}_n$ can now be updated by recomputing the $T_{ij}$'s in $\mathcal{I}$ bottom-up as follows: First, the leaves $T_{i_v i_v} \in \mathcal{I}$ are set to $\delta_{b_v}$, $1 \leq v \leq m$. Then each $T_{ij} \in \mathcal{I}$ with children $T_{iv}$ and $T_{vj}$ for which at least one has been recomputed is replaced by $T_{iv} \circ T_{vj}$. Thus, at most $m \log n$ $T_{ij}$'s have been recomputed, each in time $O(|Q|^2 \log |Q|)$, yielding a total time of $O(m|Q|^2 \log |Q| \log n)$.

The validation of the string $a_1 \ldots a_{i_1-1} b_1 a_{i_1+1} \ldots a_{i_m-1} b_m a_{i_m+1} \ldots a_n$ is now trivial: it is enough to check, in the updated auxiliary structure, that $\langle q_0, f \rangle \in T_{1n}$ for some $f \in F$. Thus, validation is also done in time $O(m|Q|^2 \log |Q| \log n)$.

The above approach can easily be adapted to strings whose length is not a power ot 2 (for example, by appropriately truncating $\mathcal{T}_{2^k}$ where $k = \lceil \log n \rceil$).

**Dealing with inserts and deletes** We next extend the divide-and-conquer approach outlined for renamings to the case when node inserts and deletes are also allowed. The above approach no longer works, for two reasons: First, inserts and deletes cause the position of nodes in the string to change. Second, the length $n$ of the string, and therefore the set of relevant intervals used to construct $\mathcal{T}_n$, are now dynamic. Due to these differences, inserts and deletes would require recomputing the entire tree $\mathcal{T}_n$, which is inefficient. Instead, we would like to use a tree structure $\mathcal{T}$ that can be incrementally maintained under inserts and deletes, as well as renamings, while preserving the properties that enabled our divide-and-conquer approach. Most importantly, the tree should continue to be balanced and have depth $O(\log n)$. This suggests adopting an approach based on B-trees, that we describe next. We assume basic familiarity with B-trees (e.g., see [GMUW01]).

The B-tree variant we use, denoted $\mathcal{T}$, has nodes containing 3 cells each. Each cell is either empty or contains a set $T_s$ corresponding to some subsequence $s$ of the string. At most one of the 3 cells in a node can be empty (assuming the current string has length at least two). Each nonempty cell is either at a leaf or has one node (with three cells) as a child.

In the tree $\mathcal{T}_n$, the interval $[i, j]$ associated to a node $T_{ij}$ is made explicit. In the tree $\mathcal{T}$, it is not necessary to compute explicitly the subsequence $s$ associated to each $T_s$. The maintenance algorithm automatically ensures the following:

- the sequence of non-empty leaf cells is $T_{s_1} \ldots T_{s_n}$ where the length of the current string is $n$ and $T_{s_i} = T_{ii}$, $1 \leq i \leq n$;

- if an internal cell contains a relation $T_s$ and its child node contains $T_{s_1}$, $T_{s_2}$ (resp. $T_{s_1}$, $T_{s_2}$, and $T_{s_3}$) then $T_s = T_{s_1} \circ T_{s_2}$ (resp. $T_s = T_{s_1} \circ T_{s_2} \circ T_{s_3}$).

We also maintain pointers providing in $O(1)$, for each element $v$ in the input string, the leaf cell $T_s$ for which the singleton $s$ consists of $v$. Note that the position of the element is never recorded explicitly.

For example, the left part of Figure 3 shows a sequence of seven nodes, several subsequences, and the corresponding tree. Note that the subscript of a node does not necessarily indicate its position in the string. Each sequence $s_i$ is the singleton sequence $n_i$, for $i \in \{1, 2, 3, 5, 6, 7, 9\}$.

The requirement of having 3 cells per node of which at least 2 are non-empty ensures that the tree $\mathcal{T}$ remains balanced and of depth $O(\log n)$ as it is updated. This follows from the standard analysis of B-tree behavior under the maintenance algorithm [GMUW01], which we describe here. In a disk-based implementation one should set the maximum number of cells per node to the number of items that fit in one disk page.

Recall that we wish to validate strings with respect to an NFA $N = \langle \Sigma, Q, q_0, F, \delta \rangle$. We describe below the maintenance algorithm for $\mathcal{T}$. Once $\mathcal{T}$ is computed for the current string, validation is easy: check that for some $f \in F$, $\langle q_0, f \rangle$ belongs to the composition of the sets $T_s$ in the cells of the root node of $\mathcal{T}$, at cost $O(|Q|^2 \log |Q|)$.

The auxiliary structure $\mathcal{T}$ corresponding to a valid string $w$ is initialized by starting from the empty string and constructing $w$ by a sequence of inserts, using the maintenance algorithm. Then $\mathcal{T}$ is maintained incrementally as follows. If the update is a renaming of

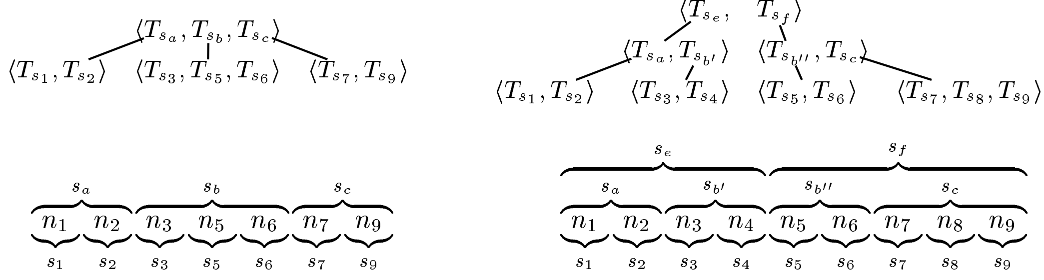

Figure 3: A $\mathcal{T}$ tree before and after the insertion of nodes $n_4$ and $n_8$

element $v$, $\mathcal{T}$ is updated much like $\mathcal{T}_n$: we use the index to find the leaf cell of $T_v$ corresponding to $v$, then update all sets $T_s$ along the path from $T_v$ to the root. This involves $O(\log n)$ updates.

If the update is the insertion or deletion of a new labeled element, the maintenance algorithm mimicks the one for B-trees. In particular, recall that if nodes in a B-tree become too full as the result of an insertion they are split, and if they contain fewer than two non-empty cells as a result of a deletion they are either merged with a sibling node or non-empty cells are transferred from a sibling node. The node splits and merges may propagate all the way to the root. Due to the similarity to classical B-tree maintenance we omit the details but illustrate how to handle the first variant of insertion; deletion and the second variant of insertion are similar. Assume that an element $y$ with label $a$ is inserted after element $x$ in the current string. If there is some empty cell in the leaf node $n$ of $\mathcal{T}$ containing the set $T_x$ corresponding to $x$ we insert the relation $T_y = \delta_a$ in the cell following that for $x$ and we revise the appropriate $T_s$ relations in ancestor nodes. For example, if a new node $n_8$ is inserted in the left string of Figure 3 after $n_7$, we insert $T_{s_8}$ in the node $\langle T_{s_7}, T_{s_9} \rangle$, as shown in the right side of Figure 3, and we revise $T_{s_c}$, which becomes $T_{s_7} \circ T_{s_8} \circ T_{s_9}$.

If the leaf node $n$ for $x$ has no non-empty cells, then we split $n$ into two nodes $n'$ and $n''$ containing two relations each. We delete from the parent the relation $T_s$, where $s$ is the subsequence that corresponds to the node $n$, and we attempt to insert in the parent relations $T_{s'}$ and $T_{s''}$, which correspond to $n'$ and $n''$. If the parent already has three relations, the deletion of $T_s$ and the insertion of $T_{s'}$ and $T_{s''}$ will require splitting the parent into two nodes. As is the case for B-trees, this process may propagate all the way to the root and may end up creating a new root. For example, the insertion of a node $n_4$ following $n_3$ leads to splitting the node $\langle T_{s_3}, T_{s_5}, T_{s_6} \rangle$ into $\langle T_{s_3}, T_{s_4} \rangle$ and $\langle T_{s_5}, T_{s_6} \rangle$. The relation $T_{s_b}$ is deleted and two new relations $T_{s_{b'}}$ and $T_{s_{b''}}$ are inserted into $\langle T_{s_a}, T_{s_b}, T_{s_c} \rangle$, which leads to a new split and a new root. The result tree is shown in the right side of Figure 3. In the worst case, when an insertion in a leaf node results in splits propagating all the way to the root, we need to recompute $2 \log n$ new relations (one at the leaf level, one at the new root, and $2(\log n - 1)$ at the internal nodes). Hence, the worst case complexity is $O(|Q|^2 \log |Q| \log n)$. Deletion proceeds similarly and may lead to node merging or root deletion, with the same complexity. As in the case of B-trees, the maintenance algorithm guarantees that $\mathcal{T}$ always has depth $O(\log n)$ for strings of length $n$. Altogether, maintenance of $\mathcal{T}$ after $m$ updates takes time $O(m|Q|^2 \log |Q| \log n)$.

**1-unambiguous regular expressions** As discussed earlier, XML Schemas require regular expressions used in type definitions to be 1-unambiguous. If $r$ is a 1-unambiguous regular expression, the corresponding DFA is of size linear in $r$. In this case, the relations $T_s$ used in the above auxiliary structure have size $O(|Q|)$ rather than $O(|Q|^2)$. This brings down the size of the auxiliary structure to $O(|Q|n)$ and the complexity of maintenance and validation to $O(m|Q| \log |Q| \log n)$.

# 4 Incremental DTD Validation

The incremental validation of DTDs extends the divide-and-conquer algorithm for incremental validation of strings described in Section 3. Let $d$ be a DTD, $T = \langle t, \lambda \rangle$ a labeled tree satisfying $d$, and consider first updates consisting of a sequence of $m$ label modifications yielding a new tree $T' = \langle t', \lambda' \rangle$. To check that $T'$ satisfies $d$, we must verify that for each node $v$ in $t'$ with children $v_1 \ldots v_n$ for which at least one label was modified, the sequence of labels $\lambda'(v_1) \ldots \lambda'(v_n)$ belongs to $r_{\lambda'(v)}$. If the label of $v$ has not been modified, i.e. $\lambda(v) = \lambda'(v)$, then validation can be done using the divide-and-conquer algorithm described in Section 3 for strings. However, if the label of $v$ has been modified, so that $\lambda(v) \neq \lambda'(v)$, the sequence $\lambda'(v_1) \ldots \lambda'(v_n)$ has to be validated with respect to the new regular language $r_{\lambda'(v)}$ rather than $r_{\lambda(v)}$. Thus, it would seem that, in this case, validation has to start again from scratch. To avoid this, we preemptively maintain information about the validity of each string of siblings with respect to *all* regular languages $r_a$ for $a \in \Sigma$. To this end we maintain some additional auxiliary information. Specifically, for each sequence of siblings in

the tree, we compute the transitions relations $T_s$ of the divide-and-conquer algorithm described in Section 3, for each NFA $N_a$ corresponding to $r_a$, and $a \in \Sigma$. We denote the sets $T_s$ for a particular $a \in \Sigma$ by $T_s^a$. Since the auxiliary structure for each fixed NFA and string of length $n$ has size $O(|Q|^2 n)$ (where $Q$ is the set of states of the NFA), the size of the new auxiliary structure is at most $O(|\Sigma||d|^2|T|)$, where $|T|$ is the size of $T$ and $|d| = max\{|r_a| \mid a \to r_a \in d\}$. The maintenance of the auxiliary structure is done in the same way as in the string case, at a cost of $O(m|\Sigma||d|^2 \log |d| \log |T|)$ for a sequence of $m$ modifications. Finally, the updated tree $T'$ is valid wrt $d$ if for each node $v$ with label $a$ in $T'$ such that either $v$ or one of its children has been updated, $\langle q_0, f\rangle$ is in the relation $T_s^a$ where $s$ is the list of children of $v$, $q_0$ is the start state of $N_a$, and $f$ is one of its final states. Each such test takes $O(|d|^2 \log |d|)$ and the number of tests is $m$ in the worst case. This yields a total validation time of $O(m|\Sigma||d|^2 \log |d| \log |T|)$.

Insertions and deletions of leaves are handled by a straightforward extension of the B-tree approach outlined in Section 3.

**Specialized DTDs: a first attempt** Specialized DTDs add another degree of complexity to the update validation problem. Intuitively, they abstract the ability of XML Schemas to associate different types to each element label. Consider a specialized DTD $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$. Recall that a tree $T$ over $\Sigma$ satisfies $\tau$ iff there exists some tree $T'$ over $\Sigma^t$, satisfying $d$, such that $\mu(T') = T$. Essentially, $T'$ associates a type in $\Sigma^t$ to each node in $T$ so that the DTD $d$ over $\Sigma^t$ is satisfied. The existence of such a type assignment, and therefore the validity of $T$, can be tested in a bottom-up manner as follows. For each leaf $v$ of $T$, let $types(v) = \{\alpha \mid \mu(\alpha) = \lambda(v) \text{ and } \epsilon \in r_\alpha\}$. Thus, $types(v)$ consists of all types in $\Sigma^t$ that may be assigned to the label of $v$ and allow it to be a leaf.

Then apply the following procedure recursively: for each internal node $v$ of $T$ with children $v_1 \ldots v_n$ for which $types(v_i)$ has already been computed, let $types(v)$ consist of the types $\alpha \in \mu^{-1}(\lambda(v))$ for which $types(v_1) \ldots types(v_n) \cap r_\alpha \neq \emptyset$, where $\alpha \to r_\alpha \in d$. In other words, $types(v)$ consists of all types allowed for the label of $v$ for which there is at least one choice of allowed types for its children that is compatible with $d$. Clearly, $T \in sat(\tau)$ iff $types(root(T)) \neq \emptyset$. This procedure closely corresponds to the evaluation on $T$ of a bottom-up unranked tree automaton corresponding to $\tau$.

Consider now a tree $T \in sat(\tau)$. We first consider label modifications. We maintain the following auxiliary structure:

- for each node $v$ in $T$, maintain the set of allowed types $types(v)$. This has size $O(|T||\Sigma^t|)$;
- for each sequence of siblings $v_1 \ldots v_n$ in $T$ and $\alpha \in \Sigma^t$, maintain the sets

$$T_s^\alpha = \{\langle p, q\rangle \mid q \in \delta_\alpha(p, \beta_i \ldots \beta_j),$$
$$\beta_k \in types(v_k), i \leq k \leq j\}$$

where $s$ is a subsequence $v_i \ldots v_j$ used in $\mathcal{T}$, formulated by the usual divide-and-conquer strategy. This has size $O(|\Sigma^t||d|^2|T|)$.

We describe how to maintain the auxiliary structure when a single label is modified. For $m$ modifications, we apply this for each modification. Validity is checked after the auxiliary structure has been updated for all modifications.

Suppose the node whose label is modified is $v$, the old label is $a$, and the new label is $b$. We need to update the sets $types(w)$ for all nodes $w$ on the path from root to $v$ in $T$, as well as the sets $T_s^\alpha$ for the sequences $s$ of siblings where such nodes occur. This is done in a bottom-up fashion as follows. First, if $v$ is a leaf, then $types(v) = \{\beta \mid \mu(\beta) = b, \epsilon \in r_\beta\}$. If $v$ has children $v_1 \ldots v_n$ then $types(v)$ contains all types $\beta \in \mu^{-1}(b)$ such that $\langle q_0, f\rangle \in T_s^\beta$ where $q_0$ is the start state of $N_\beta$, $f$ is one of its accepting states, and $T_s^\beta$ is the root of the auxiliary structure corresponding to $\beta$ and the children of $v$. Note that this step takes $O(|\Sigma^t||d|^2 \log |d|)$. Next, suppose that $w$ is a node in $T$ whose sequence of children $w_1 \ldots w_n$ contains one node $w_k$ for which $types(w_k)$ has been updated. First, the sets $T_s^\alpha$ need to be updated for the $\log n$ affected subsequences $s$ as in the divide-and-conquer string validation algorithm. This takes time $O(|\Sigma^t||d|^2 \log |d| \log n)$. Next, $types(w)$ is updated as in the base case to contain the types $\beta \in \mu^{-1}(\lambda(w))$ for which $\langle q_0, f\rangle \in T_s^\beta$ where $q_0$ is the start state of $N_\beta$, $f$ is one of its accepting states and $s$ is the sequence $w_1 \ldots w_n$. This takes time $O(|\Sigma^t||d|^2 \log |d|)$. Thus, the maintenance time for this step is $O(|\Sigma^t||d|^2 \log |d| \log n)$, and this has to be repeated at most $depth(T)$ times. This yields a total maintenance time of $O(|\Sigma^t||d|^2 \log |d| \, depth(T) \log |T|)$ for a single label modification. For $m$ modifications, the maintenance time is $O(m|\Sigma^t||d|^2 \log |d| \, depth(T) \log |T|)$. Finally the updated tree is valid iff $root(d) \in types(root(T))$. Hence, the total validation time is also $O(m|\Sigma^t||d|^2 \log |d| \, depth(T) \log |T|)$.

Node insertions and deletions can be handled by adapting the B-tree approach used for strings. The resulting complexity is the same as for label renamings.

Note that for fixed specialized DTD and update sequence, the validation algorithm outlined above takes time $O(depth(T) \log |T|)$. Thus, the algorithm works well for shallow trees. However, in the worst case $depth(T)$ could equal $|T|$, in which case the complexity is $O(|T| \log |T|)$. This is not satisfactory. We will see in

the next section how to use a more subtle strategy that reduces the overall maintenance and validation cost to $O(\log^2 |T|)$.

## 5 Incremental Validation via Binary Trees Encodings

In this section we develop a refinement of the incremental validation technique for specialized DTDs described in the previous section. This results in maintenance and validation algorithms of complexity $O(\log^2 |T|)$ for fixed DTD and update sequence, instead of the previous $O(|T| \log |T|)$. Intuitively, the algorithm of Section 4 is based on a divide-and-conquer strategy to split the work of validating sequences of siblings in the tree. However, for trees of small width and large depth, this strategy is defeated. The refinement presented in this section extends the divide-and-conquer strategy to validation of the overall tree, by splitting the work simultaneously with respect to the horizontal and vertical components. To this end, it is useful to adopt a representation of unranked trees as complete binary trees and reduce the problem of validating specialized DTDs on unranked trees to that of acceptance of the binary tree encodings by a corresponding bottom-up tree automaton. The advantage of this approach is that it unifies the horizontal and vertical components of validation and facilitates a natural formulation of the new divide-and-conquer strategy.

**Binary tree encoding of unranked trees** We next describe the encoding of unranked trees as binary trees. We use one of the standard encodings in the literature (e.g, see [Nev02]). To each unranked labeled ordered tree $T = \langle t, \lambda \rangle$ over alphabet $\Sigma$ we associate a binary tree $enc(T)$ over alphabet $\Sigma_{\#} = \Sigma \cup \{\#\}$, where $\# \notin \Sigma$. The input of *enc* is a (possibly empty) sequence of unranked trees over $\Sigma$, and the output is a complete binary tree over $\Sigma_{\#}$. The mapping *enc* is defined recursively as follows (where $\bar{T}_0$ and $\bar{T}$ are sequences of trees, possibly $\epsilon$, and $n_0$ is a single node):

- $enc(\epsilon) \;=\; \#$
- $enc(n_0(\bar{T}_0)\;\bar{T}) \;=\; n_0(enc(\bar{T}_0), enc(\bar{T}))$

For example, a tree $T$ and its encoding $enc(T)$ are shown in Figure 4 (neglect for now the boxes and bold letters).

We would like to reduce the validation of unranked trees $T$ wrt a specialized DTD $\tau$ to the question of whether $enc(T)$ is accepted by a bottom-up nondeterministic tree automaton. To this end, we show the following result (a variant of known results on equivalences of specialized DTDs and unranked tree automata, and of unranked tree automata and automata on binary trees, see [Nev02]):

**Lemma 5.1** *For each specialized DTD* $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$ *there exists a BNTA* $A_\tau$ *over* $\Sigma_{\#}$ *whose number of states is* $O(|\Sigma^t||d|)$*, such that* $\mathcal{T}(A_\tau) = \{enc(T) \mid T \in sat(\tau)\}$. ♢

**Proof:** See Appendix. ♢

Our approach is based on reducing the validation of unranked trees with respect to specialized DTDs to the validation of their binary encodings with respect to the corresponding BNTA, say $A = \langle \Sigma, Q, Q_0, q_f, \delta \rangle$. As before, the problem really amounts to efficiently updating the auxiliary structure associated with the input. In our case, the auxiliary structure will include (among other information to be specified shortly) the binary encoding $enc(T)$ of the input $T$, and will provide, for each node $v$ in $enc(T)$, the set $types(v)$ consisting of the possible states of $A$ at node $v$ after consuming the subtree rooted at $v$. Once the auxiliary structure is updated, validity amounts to checking that $types(root(enc(T)))$ contains the accept state of $A$, where $T$ is the updated tree. The strategy for updating the types associated with nodes applies the divide-and conquer strategy for string validation to certain paths in the tree, chosen to appropriately divide the work. More precisely, we will select, in every subtree $T_0$ of a given tree $enc(T)$, a particular path from the root to a leaf. We call this path the *principal line* of $T_0$, denoted by $line(T_0)$, and defined as follows:

- $root(T_0)$ belongs to $line(T_0)$;
- let $v$ be an internal node of $T_0$ that belongs to $line(T_0)$, and suppose $v$ has children $v_1, v_2$. If $|tree(v_1)| \geq |tree(v_2)|$, then $v_1$ belongs to $line(T_0)$; otherwise, $v_2$ belongs to $line(T_0)$.

Validation of $enc(T)$ can be done by associating to each maximal principal line[4] an NFA that validates that particular line. We make this more precise next.

**From BNTA to NFA on principal lines** Consider the principal line $v_1 \ldots v_n$ of a binary tree encoding $enc(T)$ where $v_1$ is the root and $v_n$ is a leaf. By the definition of binary encodings, each non-leaf node $v_i$ has one child $v'_i$ that does *not* belong to the principal line $v_1 \ldots v_n$, for $1 \leq i < n$. Consider the sets $types(v'_i)$. Note that if these sets are given, we can validate $enc(T)$ by an NFA $N$ that works on the string $v_1 \ldots v_n$. For technical reasons, the constructed NFA recognizes the reverse word $v_n \ldots v_1$. Essentially, the NFA guesses a sequence of state assignments to $v_n \ldots v_1$ that is compatible with the transition function of $A$, given the sets of states $types(v'_i)$.

The above intuition is captured as follows: We define new labels for the nodes $v_i$, which include both

[4] A principal line is maximal if it is not included in another principal line.

$\lambda(v_i)$ and the set $types(v'_i)$. More precisely, let $\Sigma' = \{\#\} \cup (\mathcal{P}(Q) \times \Sigma) \cup (\Sigma \times \mathcal{P}(Q))$ and $\lambda'$ be the labeling function defined as follows:

- $\lambda'(v_i) = \langle \lambda(v_i), types(v'_i) \rangle$, if $v'_i$ is the right child of $v_i$, $1 \leq i < n$,
- $\lambda'(v_i) = \langle types(v'_i), \lambda(v_i) \rangle$, if $v'_i$ is the left child of $v_i$, $1 \leq i < n$.
- $\lambda'(v_n) = \lambda(v_n) = \#$.

The NFA $N$ we construct will accept the string $\lambda'(v_n) \ldots \lambda'(v_1)$ iff $A = \langle \Sigma_\#, Q, Q_0, q_f, \delta \rangle$ accepts $enc(T)$. At any rate, it will compute the type derived by the sequence. More precisely, let $N = \langle \Sigma', Q, q_0, F', \delta' \rangle$, where $\Sigma'$ is as described above, $F' = \{q_f\}$, and $\delta'$ is defined by the following (and is empty everywhere else)

- $\delta'(\#, q_0) = Q_0$;
- $\delta'(\langle a, S \rangle, q) = \bigcup_{q' \in S} \delta(a, q, q')$ for $a \in \Sigma$
- $\delta'(\langle S, a \rangle, q) = \bigcup_{q' \in S} \delta(a, q', q)$ for $a \in \Sigma$

Intuitively, the NFA simulates $A$ by allowing only state transitions compatible with the transition function of $A$ and the sets of states associated to siblings. It is easy to verify that $N$ works as desired.

Note that the number of states of $N$ is $O(|Q|)$. Recall that $|Q|$ is itself $O(|\Sigma^t||d|)$ where $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$ is the specialized DTD to which the BNTA $A$ corresponds. The size of its alphabet $\Sigma'$ is $O(|\Sigma| 2^{|Q|})$ which is $O(|\Sigma| 2^{|\Sigma^t||d|})$. Hence, each symbol in $\Sigma'$ can be represented in space $O(|\Sigma^t||d| + \log |\Sigma|)$. Notice however that our auxiliary structure never represents the alphabet or the transition mapping of $N$ explicitly.

**The auxiliary structure** The auxiliary structure used for incremental validation includes (i) the binary tree $enc(T)$, (ii) for each subtree of $enc(T)$ its principal line and (iii) for each maximal principal line in $enc(T)$, the auxiliary transition relation tree for the NFA corresponding to that line.

Note that the principal lines can be specified concisely by annotating each node in $enc(T)$ with 0 or 1 by a labeling $\mu$ as follows: $\mu(root(enc(T))) = 0$, and for every pair of siblings $v_1, v_2$, $\mu(v_1) = 1$ and $\mu(v_2) = 0$ if $|tree(v_1)| \geq |tree(v_2)|$; otherwise, $\mu(v_1) = 0$ and $\mu(v_2) = 1$. Clearly, the principal line of a subtree $T_0$ is the unique path from $root(T_0)$ to a leaf where all non-root nodes are labeled 1. Note that the principal line of $T_0$ is maximal iff $\mu(root(T_0)) = 0$.

For example, consider the unranked tree represented in Figure 4 (top), and its binary encoding in the same figure (bottom). In the binary encoding in the figure, the nodes $w$ for which $\mu(w) = 0$ are those inside a box. Note that this identifies all maximal principal lines. The bold and underlined nodes participate in the principal line of $enc(T)$. The nodes of one of the secondary principal lines (line $j, k$) are in italics.

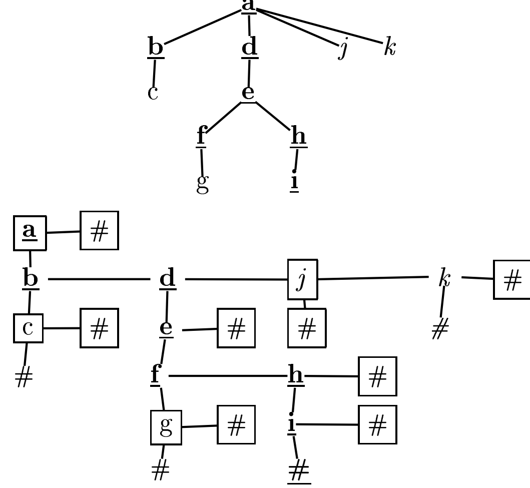

Figure 4: A tree $T$ (top) and its encoding $enc(T)$

Part (iii) of the auxiliary structure provides the transition relation trees for the NFAs associated with the maximal principal lines. The size of each transition relation tree for an NFA $N$ is $O(|enc(T)||Q|^2)$ where $Q$ is the number of states of $N$.

In summary, consider an input tree $T$ and a specialized DTD $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$. In view of our construction of the BNTA $A$ from $\tau$ (Lemma 5.1), of the NFA $N$ from $A$ (above), and of the tree of transition relations for each NFA $N$ (Section 3) it follows that the size of the auxiliary structure associated with $T$ and $\tau$ is $O((|\Sigma^t|^2|d|^2|T|)$.

**Validation and maintenance for label renamings** Let us consider first the validation and maintenance of updates consisting of label renamings. Note that label renamings in $T$ translate straightforwardly to label renamings in $enc(T)$. To validate a sequence of label renamings, it is sufficient to show how the auxiliary structure is maintained for a single renaming. For a sequence of renamings this is iterated one update at a time and validity is checked at the end using the updated auxiliary structure. So, suppose the label of some node $v$ in $enc(T)$ is modified from $a$ to $b$. Suppose first that $v$ belongs to the maximal principal line $l = v_1 \ldots v_n$ of $enc(T)$, say $v = v_k$. In the string $\lambda'(v_1) \ldots \lambda'(v_n)$ the label renaming corresponds to modifying the label of $v_k$ from $a$ to $b$ if $k = n$ and from $\langle a, types(v'_k) \rangle$ to $\langle b, types(v'_k) \rangle$ if $k < n$ and $v'_k$ is the right child of $v_k$ (left is analogous). Then the transition relation tree associated to $l$ is updated as in the string case in time $O(|Q|^2 \log |Q| \log |l|)$, that is $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log |l|)$. Since $\log |l|$ is $O(|enc(T)|)$ and $|enc(T)|$ is $O(|T|)$, the update takes time $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log |T|)$.

Now suppose that $v$ does not belong to the principal line $l$ of $enc(T)$. Then there is some $k > 0$ such that $v$ belongs to $tree(v'_k)$ where $v'_k$ is the child of some

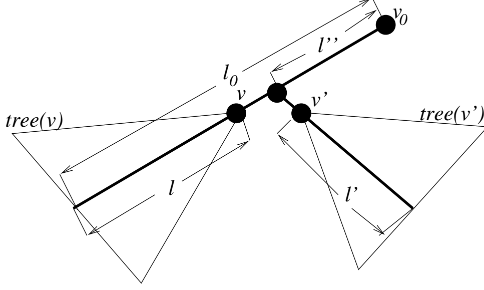


Figure 5: Scenario of Line Rearrangement

$v_k$ belonging to $l$. Note that the update to the label of $v$ may cause a change in the value of $types(v'_k)$. In order to update $l$, we now have to first compute the new value for $types(v'_k)$, then apply the update procedure for the corresponding modification in the label $\langle\lambda(v_k), types(v'_k)\rangle$ of $v_k$. If $v$ belongs to the principal line $l'$ of $tree(v'_k)$ then the transition relation tree associated with the NFA for $l'$ can be updated as before in time $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|)\log|T|)$. This provides, in particular, the new value for $types(v'_k)$. Continuing inductively, it is clear that renaming the label of a node $v$ affects precisely the maximal principal lines encountered in the path from root to $v$. Let $M$ be the number of such maximal principal lines. Clearly, $M$ is precisely the number of nodes $w$ along the path from root to $v$ for which $\mu(w) = 0$. We next provide a bound on this number, using the notion of *line diameter* of a tree.

**Definition 5.1** (**Line diameter**) The *line diameter* of $enc(T)$ is the maximum number of distinct maximal principal lines crossed by any path from root to leaf in $enc(T)$. Equivalently, the line diameter of $enc(T)$ is the maximum number of nodes $w$ for which $\mu(w) = 0$, occurring along a path from root to leaf in $enc(T)$, where $\mu$ is defined as above. ◇

For example, the line diameter of $enc(T)$ in Figure 4 is 3. The following is proven in the Appendix:

**Lemma 5.2** *The line diameter of $enc(T)$ is no larger than $1 + \log|enc(T)|$.* ◇

From the bound on the line diameter of $enc(T)$, it follows that a label renaming can cause at most $O(\log|enc(T)|)$ updates to distinct transition relation trees of maximal principal lines in $enc(T)$. Since each update takes time $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|)\log|T|)$, the entire auxiliary structure can be updated in time $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|)\log^2|T|)$. For a sequence of $m$ label renamings, updating the auxiliary structure and validating the new tree therefore takes time $O(m|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|)\log^2|T|)$.

**Insertions and deletions** We next describe how to extend the maintenance and validation algorithm described above to updates that include insertions and deletions of leaf nodes.

For a maximal principal line $l$ in $enc(T)$, we denote by $N_l$ the NFA corresponding to $l$ and by $\mathcal{T}_l$ the transition relation tree corresponding to $l$ and $N_l$.

Note that each insertion or deletion of a leaf node in $T$ translates into up to four node insertions and deletions into $enc(T)$ (for example, deleting a node in $T$ may require deleting in $enc(T)$, besides the node itself, up to two leaves labeled #, and may require inserting another such leaf). This constant factor blow-up in the number of updates does not affect our analysis.

Insertions and deletions are handled by an extension of the technique used to maintain the transition relation trees for maximal principal lines in the case of label renamings. Insertions and deletions that do not cause a change in the set of maximal principal lines existing prior to the update are handled straightforwardly. More precisely, let us call an insertion or deletion *line preserving* if the restriction of $\mu$ to the nodes of $enc(T)$ that are not affected by the update is the same before and after the update. Note that an insertion may be line preserving but nonethelss introduce a new singleton maximal principal line consisting of the new node. Also observe that line-preserving updates affect precisely the maximal principal lines intersected by the path from the root of $enc(T)$ to the newly inserted node or to the parent of the deleted node. The transition relation trees for these maximal principal lines are updated as in the case of label renamings, at the same cost. If a new singleton maximal line $l$ consisting of an inserted node needs to be added, computing its auxiliary transition relation tree takes additional time $O(|Q|^2)$ where $Q$ is the set of states of the NFA $N_l$. This is dominated by the rest of the cost.

Handling inserts and deletes that are not line preserving requires more care. In this case, the set of maximal principal lines in $enc(T)$ changes as the result of updates. To illustrate the problem, consider the situation depicted in Figure 5. The maximal principal line $l_0 = line(tree(v_0))$ contains a node $v$, which has a sibling $v'$. Initially, $|tree(v)| \geq |tree(v')|$. However, a deletion in $tree(v)$ or an insertion in $tree(v')$ may make $tree(v')$ larger than $tree(v)$. In this case a new line structure is needed, where the line $l = line(tree(v))$ becomes a maximal principal line and the new principal line $line(tree(v_0))$ is the concatenation of $l''$ and $l' = line(tree(v'))$. This requires updating the auxiliary structure in two steps: First we compute the transition relation tree $\mathcal{T}_l$ for the new maximal principal line $l$ obtained by truncating $l_0$. Then we compute the transition relation tree $\mathcal{T}_{l''l'}$ for the new maximal principal line obtained by concatenating $l''$ and $l'$.

Fortunately, the new transition relation trees can be computed efficiently from the old ones. Specifically, $\mathcal{T}_l$ is obtained by truncating $\mathcal{T}_{l_0}$, and $\mathcal{T}_{l''l'}$ is obtained by merging a subtree of $\mathcal{T}_{l_0}$ corresponding to $l''$ with the tree $\mathcal{T}_{l'}$. This is done by adapting usual B-tree techniques. We next provide more details.

Given the balanced tree $\mathcal{T}_{l_0}$ of the line $l_0$, we compute the balanced tree $\mathcal{T}_l$ by traversing bottom-up the path in $\mathcal{T}_{l_0}$ from the leaf that contains $v$ to the root. Note that the path has maximum length $\lceil \log |l_0| \rceil$. At each cell $n$ along the path we delete the relations $T_{s_1}$ where $s_1 \cap l = \emptyset$ and we recompute the relation $T_s$, where the segment $s$ contains $v$. Recall that each cell in $\mathcal{T}_{l_0}$ has between two and three relations, so it is not possible for any cell to become empty after these deletions. In addition, if the deletions have left only the relation $T_s$ at cell $n$ then we do the following, assuming $n$ is not the root (the case where $n$ is root is simple):

- if the right sibling $n'$ of $n$ has two relations we delete $n$ and we transfer $T_s$ (and the corresponding child node) to $n'$. In the parent of $n$ and $n'$ we delete the relation that corresponds to $n$ and we continue our processing at the parent of $n$ and $n'$.
- if the right sibling $n'$ of $n$ has three relations we move its leftmost relation (and the corresponding child node) to the cell $n$, so that $n$ also has two relations. We recompute the entry of $n'$ at the parent of $n$ and $n'$.
- if $n$ has no right sibling the we delete $n$ and we copy $T_s$ at the parent cell. Notice that this case reduces the depth of the balanced tree.

In all cases we continue recursively with the parent cell. The complexity of this procedure is $O(|Q|^2 \log |Q| \log |l_0|)$ where $Q$ is the set of states of $N_{l_0}$, since the size of the traversed path is at most $\lceil \log |l_0| \rceil$ and in each step we recompute at most two relations.

Next, consider the computation of the balanced tree $\mathcal{T}_{l''l'}$ of the new main line $l''\ l'$. First, we compute a balanced tree $\mathcal{T}_{l''}$ for the segment $l''$ in $O(|Q|^2 \log |Q| \log |l''|)$. Then we merge $\mathcal{T}_{l''}$ and $\mathcal{T}_{l'}$ as follows. Assume that the depth of $\mathcal{T}_{l''}$ is equal or less to the depth of $\mathcal{T}_{l'}$ - the other case is symmetrical. Locate a node $n_2$ on the leftmost path of $\mathcal{T}_{l'}$ such that the depth of the tree rooted at $n_2$ is $depth(\mathcal{T}_{l''})$. Then insert each segment (and corresponding child node) of the root of $\mathcal{T}_{l''}$ into $n_2$. The insertions are handled as usual: if there is not enough space in $n_2$ then $n_2$ will be split, and so on. It is easy to see that the merge takes $O(|Q|^2 \log |Q| \log |l'|)$ since we have to recompute one or two relations at each level on the path from $n_2$ to the root of $\mathcal{T}_{l'}$. Overall, the rearrangement of these lines requires $O(|Q|^2 \log |Q| (\log |l_0| + \log |l'|))$, which is $O(|Q|^2 \log |Q| \log |enc(T)|)$. Also, note that a single insertion or deletion may cause at most $O(\log |enc(T)|)$ line rearrangements (one for each maximal principal line intersected by the path from root to the affected node). Thus, all line rearrangements can be done in time $O(|Q|^2 \log |Q| \log^2 |enc(T)|)$. In terms of the original specialized DTD and input tree $T$, this is $O((|\Sigma^t|^2 |d|^2 \log(|\Sigma^t||d|) \log^2 |T|)$.

Once the line rearrangements have been computed, additional updates to the transition relation trees of maximal principal lines may have to be computed, as in the case of label renamings. This takes again time $O((|\Sigma^t|^2 |d|^2 \log(|\Sigma^t||d|) \log^2 |T|)$.

In summary, the size of the auxiliary structure used for incremental validation is $O((|\Sigma^t|^2 |d|^2 |T|)$. Maintaining the auxiliary structure and validating the updated tree following a sequence of $m$ updates (label renamings, insertions, or deletions) is done in time $O(m|\Sigma^t|^2 |d|^2 \log(|\Sigma^t||d|) \log^2 |T|)$.

# 6 Conclusions and Future Work

The incremental validation algorithms we exhibited are significant improvements over brute-force validation from scratch. However, several issues need further investigation:

**Lower bounds** To understand how close our algorithms are from optimal, it would be of interest to exhibit lower bounds on incremental maintenance of strings, DTDs, and specialized DTDs. There are known results that yield lower bounds for validation from scratch: acceptance of a tree by a tree automaton is complete for uniform $\mathrm{NC}^1$ under DLOGTIME reductions [Loh01]. However, this does not seem to yield any non-trivial lower bound on the incremental validation problem. We are not aware of any work providing such lower bounds applicable to our framework.

**Optimizing over multiple updates** For a sequence of $m$ updates, our incremental validation algorithm modifies the auxiliary structure one update at a time, then checks validity of the final updated tree. Clearly, it is sometimes more efficient to consider groups of updates at a time. For example, this may avoid performing unnecessary intermediate line rearrangements in the incremental algorithm for specialized DTDs. Also, if the number of updates is large compared to the size of the resulting tree, it may be more efficient to re-validate from scratch.

**More complex updates on trees** We only considered here elementary updates affecting one node at a time. Some scenarios, such as XML editors, require more complex updates arising from manipulation of entire subtrees (deletion, insertion, cut-and-paste, etc). Our approach can still be applied by reducing each of these updates to a sequence of elementary updates. However, in this case it may be more efficient to consider updates of coarser granularity.

# References


[BKMW01] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over non-ranked alphabets. Technical Report HKUST-TCSC-2001-05, Hong Kong Univ. of Science and Technology Computer Science Center, 2001. Available at http://www.cs.ust.hk/tcsc/RR/2001-05.ps.gz.

[BKW98] A. Bruggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[BM99] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Int'l. Conf. on Database Theory*, pages 296–313, 1999.

[CDSS98] Sophie Cluet, Claude Delobel, Jerome Simeon, and Katarzyna Smaga. Your mediators need data conversion! In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 177–188, 1998.

[DS95] G. Dong and J. Su. Space-bounded foies. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 139–150. ACM Press, 1995.

[GM80] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *JACM*, 27(3), 1980.

[GMUW01] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[HI02] B. Hesse and N. Immerman. Complete problems for dynamic complexity classes. In *Proc.IEEE LICS*, 2002. To appear.

[Ipe] Ipedo XML Database: Technical overview. Available at `http://www.ipedo.com/downloads/products_ixd_technical_overview.pdf`.

[JG82] F. Jalili and J. Gallier. Building friendly parsers. In *ACM Symposium on Principles of Programming Languages*, 1982.

[Lar95] J. Larcheveque. Optimal incremental parsing. *ACM Transactions on Programming Languages and Systems*, 17(1), 1995.

[Li95] W. Li. A simple and efficient incremental LL(1) parsing. In *Theory and Practice of Informatics*, 1995.

[Lin93] G. Linden. Incremental updates in structured documents, 1993. Licentiate Thesis, Report C-1993-19, Department of Computer Science, University of Helsinki.

[Loh01] M. Lohrey. On the parallel complexity of tree automata. In *Proceedings of the 12th RTA, LNCS 2051*, 2001.

[MPS90] A. Murching, Y. Prasant, and Y. Srikant. Incremental recursive descent parsing. *Computer Languages*, 15(4), 1990.

[MSVT94] P.B. Miltersen, S. Subramanian, J.S. Vitter, and R. Tamassia. Complexity models for incremental computation. *TCS*, 130(1):203–236, 1994.

[Nev02] F. Neven. Automata, logic and XML. In *Computer Science Logic*, 2002. Available at `http://alpha.luc.ac.be/ lucg5503/publs.html`.

[Pet95] L. Petrone. Reusing batch parsers as incremental parsers. In *Foundations of Software Technology and Theoretical Computer Science*, 1995.

[PI97] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *JCSS*, 55(2), 1997.

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 35–46. ACM, 2000.

[Seg02] L. Segoufin. Personal communication, 2002.

[Vol99] H. Vollmer. *Introduction to Circuit Complexity*. Springer Verlag, 1999.

[W3C98] W3C. The extensible markup language (XML), 1998. W3C Recomendation available at http://www.w3c.org/XML.

[W3C01] W3C. XML schema definition, 2001. W3C Recomendation available at http://www.w3c.org/XML/Schema.

[WG98] T. Wagner and S. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(2), 1998.

[XMLa] XML editor products. Available at `http://www.perfectxml.com/soft.asp?cat=6`.

[XMLb] XML Spy document editor. Available at `http://www.xmlspy.com/products_doc.html`.

[XMLc] XMLmind XML Editor. Available at `http://www.xmlmind.com/xmleditor/`.


# Appendix

**A comment on the worst-case complexity of the [WG98] algorithms** The $O(\log n)$ performance guarantee provided by [WG98], where $n$ is the size of the string, does not apply to the case when the interpretation of the yield of a sequence of symbols of unbounded length depends on its context. In particular, [WG98] provide the following "bad grammar", which recognizes the regular expression $(a|b)x^*$

$$
\begin{aligned}
&S \to aC^+|bD^+\\
&C \to x\\
&D \to x
\end{aligned}
$$

This grammar is problematic for their algorithm because the reduction of an $x$ to either a $C$ or a $D$ is determined by the initial symbol in the sentence, which is arbitrarily

distant. In this case their algorithm needs $O(n)$ recomputation, where $n$ is the size of the string. Notice that our divide-and-conquer algorithm for the incremental validation of regular expressions does not pose any restriction on the regular expression.

**Proof of Lemma 5.1** For each $\alpha \in \Sigma^t$, let $N_\alpha = \langle \Sigma^t, Q_\alpha, q_0^\alpha, F_\alpha, \delta_\alpha \rangle$ be a standard NFA that accepts the language $r'_\alpha = \{w^r \mid w \in r_\alpha\}$ where $w^r$ is the reverse of $w$. Distinct $N_\alpha$ have disjoint sets of states. Let $\mathcal{Q}_d = \bigcup_{\alpha \in \Sigma^t} Q_\alpha$. Let $A_\tau$ be the BNTA $\langle \Sigma_\#, Q, Q_0, q_f, \delta \rangle$ where $Q = \{q_f\} \cup \mathcal{Q}_d$, $Q_0 = \{q_0^\alpha | \alpha \in \Sigma^t\}$, $q_f$ is the accept state ($q_f \notin \mathcal{Q}_d$), and $\delta$ is defined as follows ($\delta$ is empty whenever not specified):

- If $a \in \Sigma, \alpha \in \Sigma^t, \alpha \neq root(d), \mu(\alpha) = a, \beta \in \Sigma^t, q^\beta \in Q_\beta$ and $q_f^\alpha \in F_\alpha$ then

$$\delta(a, q_f^\alpha, q^\beta) = \delta_\beta(\alpha, q^\beta)$$

- If $\rho = root(d), r = \mu(\rho), q_f^\rho \in F_\rho, \beta \in \Sigma^t$ and $q^\beta \in Q_\beta$ then

$$\delta(r, q_f^\rho, q^\beta) = \delta_\beta(\rho, q^\beta) \cup \{q_f\}$$

It is easily seen that $\mathcal{T}(A_\tau) = \{enc(T) \mid T \in sat(\tau)\}$.

**Proof of Lemma 5.2** Consider a path from root to leaf in $enc(T)$ and let $w_1 \ldots w_M$ be the sequence of nodes $w$ along the path for which $\mu(w) = 0$. Note that, by the definition of $\mu$, $w_1$ is the root of $enc(T)$, and each node $w_i$ other than $w_M$ has two children $w'_i$ and $w''_i$ where $w'_i$ is on the path from $w_i$ to $w_{i+1}$ and $|tree(w'_i)| \geq |tree(w''_i)|$. It easily follows that $|enc(T)| \geq 2^{M-1}$ so $M \leq 1 + \log |enc(T)|$.