Incremental Constraint Validation of XML Documents under Multiple Updates

Béatrice Bouchou[1] Ahmed Cheriat[1] Mírian Halfeld Ferrari Alves[1]
Dominique Laurent[2] Martin A. Musicante[3]

[1] Université François-Rabelais de Tours-LI/Campus de Blois, France
{bouchou, cheriat, mirian}@univ-tours.fr
[2] Université de Cergy-Pontoise - Departement Informatique, France
dominique.laurent@dept-info.u-cergy.fr
[3] Universidade Federal do Paraná - Departamento de Informática, Brazil
mam@inf.ufpr.br

Abstract. This paper addresses the problem of incremental validation of XML documents, wrt schema under multiple updates. A given sequence of updates is composed of insertions, deletions and replacements of any subtree in an XML unranked tree. DTD, XDS and specialized DTD schemas are represented by tree automata.

In a complete XML tree traversal (*à la* SAX), where a sequence of updates is treated, only the cost of validation tests is relevant. Our approach is incremental since only some nodes, defined from update positions, trigger validation tests. In this context, let m be the number of updates over an XML tree whose size is $|T|$. Our incremental schema validation, for DTD and XDS, is done in time $O(m.\log_c |T|.c)$ where c is the maximal fan out of T. Experimental results show that our algorithms behave efficiently in practice.

1 Introduction

We consider a data-exchange environment where an XML document should respect schema constraints. Schema constraints establish the structural form of XML documents. We address the problem of *incremental* validation of updates performed on a valid XML document (*i.e.*, one that respects a given set of constraints).

An XML document is an unranked labeled tree T. The labeled tree representing part of the XML document used in our examples is shown in Fig.1. In this figure, a tree node is identified by a position and it is associated to a label. Labels are XML element or attribute names. Notice that the data values found in an XML document appear on the tree leaves.

In our approach, update operations correspond to the insertion, the deletion and the replacement of subtrees of T. Only updates that do not violate constraints are accepted. Thus, before applying updates on a valid tree, one have to test whether these updates do not violate the validity of the tree. The goal of an incremental validation method is to perform these tests without testing the entire tree, but just the part of it which is concerned by the update.

Operations capable of updating and incrementally validating an XML tree can be proposed for two different situations:
• *Single updates:* One uses an update language to perform one update at a time, *i.e.*, an incremental validation is performed for each single update operation. In this case, document validity is assured wrt each update operation.
• *Multiple updates:* A sequence of update operation is treated as one unique transaction. Document validity is assured just after considering the *whole* sequence of updates - and not after each update of the sequence, independently. A single update is a special case of a multiple update.

The main contribution of this paper is the introduction of an incremental schema validation method which deals with multiple updates. This paper extends our previous work [8, 7] by dealing with unranked trees, usually much shorter than the binary ones, the complexity of our incremental schema validation method happens to be similar to the one proposed in [6, 14]. However, contrary to [6, 14], our update operations can be applied at any node of the XML tree and auxiliary structures are not necessary when using DTDs and XSD.

In the following, we roughly explain how the incremental validation tests are performed in our approach.

Fig. 1. Tree representation of an XML document.

Overview of our method - Let *UpdateTable* be a sequence of updates to be performed on an XML tree T. To visit T we proceed in a depth-first visit of the XML document, triggering tests and actions according to the required updates (in *UpdateTable*). To simplify our explanation, schema constraint routine can be summarized as follows.

- When an update node is reached, the required update is taken into account (considered as done). Nodes which are descendant of update nodes are skipped *i.e.*, they are not treated.
- For each node p which is an ascendant of an update position a *validation step* is activated when reaching the close tag corresponding to p. A validation step at position p verifies whether p's children (in the updated tree version) respect schema constraints. For instance, if a sequence requires updates on positions 0.1.2 and 0.3 of Fig. 1 then a validation step is activated on nodes 0.1, 0 and ϵ.
- All other nodes (those that are not on the path between the root and an update position) are skipped, *i.e.*, no action is triggered on them.

In this paper, XML trees are treated *à la* SAX [2]. This choice implies a complete XML tree traversal but it avoids the use of an auxiliary structure for incrementally verifying some kinds of schema constraints. It is also possible to consider our approach in a DOM context. In this case, space requirements are bigger, but time complexity can be smaller. It is worth remarking that the use of SAX allows the treatment of much bigger XML documents [13].

Roughly, our algorithm visits all nodes in an XML tree. However a more detailed analysis shows that only some nodes trigger validation actions while others are just "skipped" (*i.e.*, no action is activated when reaching them). The cost of skipping nodes is not relevant when compared to the cost of validation actions. Thus, in Sections 5 we consider the complexity of our method only wrt the total number of validation actions that should be performed.

This paper is organized as follows. Section 2 introduces some concepts necessary in the paper. In Section 3 we consider the validation wrt schema constraints. We present In Section 4 algorithms that incrementally verify the validity of an XML document wrt multiple updates and we show some experimental results. Our framework takes into account XML attributes as well as elements. Section 7 concludes the paper.

2 Background

An XML document is an unranked labeled tree where: (*i*) the XML outermost element is the tree root and (*ii*) every XML element has its sub-elements and attributes as children. Elements and attributes associated with arbitrary text have a *data* child. Fig. 1 shows an XML tree.

To define our trees formally, let U be the set of all finite strings of positive integers (which usually we separate by dots) with the empty string ϵ as the identity. The *prefix relation* in U, denoted by $\preceq$ is defined by: $u \preceq v$ iff $u.w = v$ for some $w \in U$.

Now, if Σ is an alphabet then a Σ-valued tree T (or just a tree) is a mapping $T : dom(T) \rightarrow \Sigma$, where $dom(T)$ is a tree domain. A finite subset $dom(T) \subseteq U$ is a (finite) *tree domain* if: (1) $u \preceq v$, $v \in dom(T)$ implies $u \in dom(T)$ and (2) $j \geq 0, u.j \in dom(T), 0 \leq i \leq j \Rightarrow u.i \in dom(T)$.

Each tree domain may be regarded as an unlabeled tree, *i.e.*, a set of tree positions. We write $T(p) = a$ for $p \in dom(T)$ to indicate that the symbol a is the label in Σ associated with the node at position p. For XML trees, Σ is composed by element and

attribute labels together with the label *data*. We consider the existence of a function *value* that returns the value associated with a given *data* node.

Let T be an XML tree, valid wrt some integrity and schema constraints, and consider updates on it. We assume three update operations (*insert*, *delete* and *replace*) whose goal is to perform changes on XML trees. Fig. 2 illustrates the individual effect of each update operation over T.

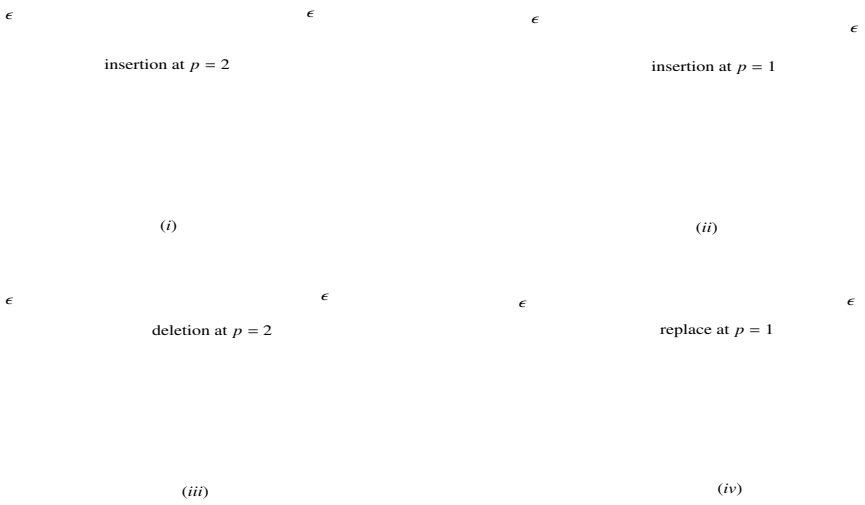

Fig. 2. Update operations over a tree T. (*i*) Insertion at a frontier position. (*ii*) Insertion at a position of $dom(T)$. Right siblings are shifted right. (*iii*) Deletion. Right siblings are shifted left. (*iv*) Replace.

In this paper we are interested in multiple updates, *i.e.*, we suppose an input file containing a sequence of update operations that we want to apply over an XML tree. The effective application of this sequence of updates depends on its capability of preserving document validity. In other words, a valid XML tree is updated (according to a given update sequence) only if its updated version remains valid. The acceptance of updates relies on *incremental validation*, *i.e.*, only the validity of the part of the original document directly affected by the updates is checked.

A sequence of updates is treated as one unique transaction, *i.e.*, we assure validity just after considering the whole sequence of updates - and not after each update of the sequence, independently. In other words, as a valid document is transformed by using a sequence of primitive operations, the document can be temporarily invalid but in the end validity is restored. This extends our previous work in [5, 8, 9].

Let $UpdateTable$ be the relation that contains updates to be performed on an XML tree. Each tuple in $UpdateTable$ contains the information concerning the update position p and the update operation op. In this paper we assume that $UpdateTable$ is the result of a pre-processing over a sequence of updates required by a user. In the resulting

$UpdateTable$ the following properties hold:
P1 - An update on a position p excludes updates on descendants of p. In other words, there are not in $UpdateTable$ two update positions p and p' such that $p \preceq p'$.
P2 - If a position p appears more than once in $UpdateTable$ then the operation op involving p is an insertion.
P3 - Updates in an $UpdateTable$ are ordered by position, according to the document order.
P4 - An update position in an $UpdateTable$ always refers to the original tree. Consider for instance the tree of Fig. 1. In an $UpdateTable$ an insertion operation refers to position 0.3 even if a deletion at position 0.1 precedes it.

3 Schema Verification

It is a well known fact that XML schemas can be represented by tree automata. A tree automaton can be built from a schema specified using schema languages such as DTD, XDS, specialized DTD or RELAX NG [1]. In our approach, we use a bottom-up unranked tree automaton capable of dealing with both (unordered) attributes and (ordered) elements in XML trees [8].

Definition 1. - Non-deterministic bottom-up finite tree automaton: A tree automaton over Σ is a tuple $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where Q is a set of states, $Q_f \subseteq Q$ is a set of final states and Δ is a set of transition rules of the form $a, S, E \rightarrow q$ where *(i)* $a \in \Sigma$; *(ii)* S is a pair of disjoint sets of states, *i.e.*, $S = (S_{compulsory}, S_{optional})$ (with $S_{compulsory} \subseteq Q$ and $S_{optional} \subseteq Q$); *(iii)* E is a regular expression over Q and *(iv)* $q \in Q$. □

The tree automaton $\mathcal{A}$ obtained from a schema specification D may have different characteristics according to the schema language used for D. These characteristics, discussed below, match the taxonomy of regular tree grammars introduced in [13].

Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a tree automaton. Two different states q_1 and q_2 in Q are *competing* if Δ contains different transition rules ($a, S_1, E_1 \rightarrow q_1$ and $a, S_2, E_2 \rightarrow q_2$) which share the same label a. Notice that we assume that no two transition rules have the same state in the right-hand side and the same label in the left-hand side, since two rules of this kind can be written as a single one. A regular expression E in a transition rule *restrains competition* of two competing states q_1 and q_2 if for any sequence of states α_U, α_V, and α_W, either $\alpha_U q_1 \alpha_V$ or $\alpha_U q_2 \alpha_W$ fails to match E.

Based on the concepts of competing states and competition-restrictive regular expressions, regular tree languages are classified as follows:
$C1-$ *Regular tree languages* (RTL): A regular tree language is a language accepted by any tree automaton specified by Definition 1. The automaton obtained from a specialized DTD recognizes languages in this class.
$C2-$ *Local tree languages* (LTL): A local tree language is a regular tree language accepted by a tree automaton that does not have competing states. This means that, in this case, each label is associated to only one transition rule. The automaton obtained from a DTD recognizes languages in this class.
$C3-$ *Single-type tree languages* (STTL): A single-type tree language is a regular tree language accepted by a tree automaton having the following characteristics: (i) For each

transition rule, the states in its regular expression do not compete with each other and (ii) the set Q_f is a singleton. The combination of these two characteristics implies that although it is possible to have competing states, the result of a successful[4] execution of such an automaton can consider just a single type (state) for each node of the tree. The automaton obtained from a schema written in XDS recognizes languages in this class.
$C4-$ *Restrained-competition tree languages* (RCTL): A restrained-competition tree language is a regular tree language accepted by a tree automaton having the following characteristics: (i) For each transition rule, its regular expression restraints competition of states and (ii) the set Q_f is a singleton. No schema language proposed for XML is classified as a RCTL.

Notice that, as shown in [13], the expressiveness of the above classes of languages can be expressed by the hierarchy LTL $\subset$ STTL $\subset$ RCTL $\subset$ RTL (where $L_1 \subset L_2$ means that L_2 is strictly more expressive than L_1).

The execution of a tree automaton $\mathcal{A}$ over an XML tree corresponds to the validation of the XML document wrt to the schema constraints represented by $\mathcal{A}$. In its general form, a *run* r of $\mathcal{A}$ over an XML tree T is a tree such that: (i) $dom(r) = dom(T)$ and (ii) each position p is assigned a set of states Q_p. The assignment $r(p) = Q_p$ is done by verifying whether the attribute and element constraints imposed to p's children are respected. The set Q_p is composed by all the states q such that:

1. There exist a transition rule $a, (S_{compulsory}, S_{optional}), E \rightarrow q$ in $\mathcal{A}$.
2. $T(p) = a$.
3. $S_{compulsory} \subseteq Q_{att}$ and $Q_{att} \backslash S_{compulsory} \subseteq S_{optional}$ where Q_{att} is the set containing the states associated to each attribute child of p.
4. There is a word $w = q_1, \ldots, q_n$ in $L(E)$ such that $q_1 \in Q_1, \ldots, q_n \in Q_n$, where $Q_1 \ldots Q_n$ are the set of states associated to each element child of p.

A run r is *successful* if $r(\epsilon)$ is a set containing at least one final state. A tree T is *valid* if a successful run exists on it. A tree is *locally valid* if the set $r(\epsilon)$ contains only states that belong to $\mathcal{A}$ but that are not final states. This notion is very useful in an update context [8].

Restricted forms of schema languages permit simplified versions of *run*. For instance, in a run of a tree automaton for (simple) DTD, the sets Q_p are always singleton. This situation considerably simplifies the implementation of item (4) above [8]. Moreover, implementations can be enhanced by considering the fact that XML documents should be read sequentially to be validated. While reading an XML document, we can store information useful to avoid the possible ambiguity of state assignment, expressed by the transition rules. For instance, in the implementation of the run of a tree automaton for XDS, each tree node can always be associated to one single state. This state is obtained by intersecting a set of "expected" states (computed during the sequential reading of the document so far) and the set of states obtained by the bottom-up application of the rules of the automaton (Proposition 1).

[4] See below the definition of the *run* of a tree automaton over a tree.

4 Incremental Schema Verification

Given a tree T and a set of updates over T, the incremental validation problem consists in checking whether the updated tree complies with the schema, by validating only the part of the tree involved by the updates. We propose a method to perform the incremental validation of an XML tree T by triggering a local validation method only on positions that are a prefix of an update position p (including both the root position ϵ and p itself).

Remark: When considering the most general classes of schema languages, during an incremental validation, we need to know which states were assigned by a previous validation to each node of the tree being updated. A data structure containing the result of the run over the original document should be kept. However, schema verification for languages in STTL (*i.e.*, XSD) and LTL (*i.e.*, DTD), does not impose the need for auxiliary, permanent data structures [13]. □

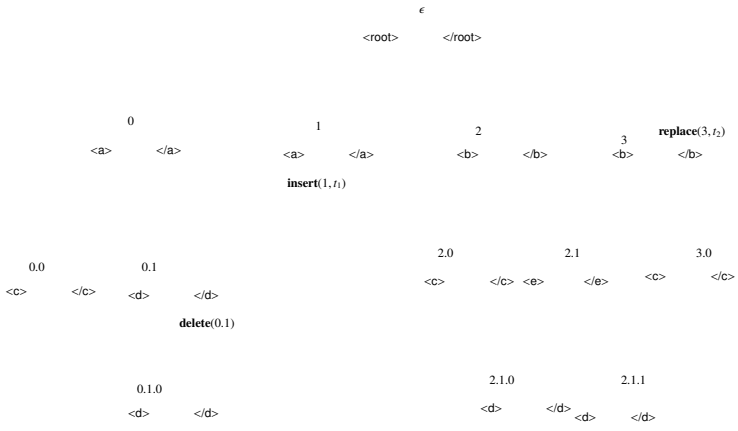

Fig. 3. XML tree and update operations.

The following example illustrates our method in an intuitive way.

Example 1. Consider the XML tree of Fig. 3, where update positions are marked. Bold arcs represent the necessary tree transversal for our method, to check the validity of the updates we want to perform.

Let us suppose that there is a tree automaton $\mathcal{A}$, representing the schema to be verified. We also suppose that the original tree (Fig. 3) is valid w.r.t. $\mathcal{A}$, and that the subtrees being inserted are *locally valid* w.r.t. $\mathcal{A}$. It is interesting to remark that:

– When the open tag <d> (at position 0.1) is reached, the deletion operation is taken into account and the subtree rooted at this position is skipped.
To verify whether this deletion can be accepted, we should consider the transition rules in $\mathcal{A}$ associated to the parent of the update position. This test is performed

when the close tag (position 0) is found. Notice that to perform this test we need to know the state assigned to position 0.0, but we do not need to go below this position (those nodes, when they exist, are skipped).

- When the second open tag <a> (position 1) is reached, the insertion operation is taken into account and the new, locally valid sub-tree t_1 (rooted at this position) is inserted. This implies that if all the updates are accepted, right-hand side siblings of position 1 are shifted to the right.
  We proceed by reading nodes at (original) positions 1 and 2. Notice that we can skip all nodes below position 2, since there is no update position below this point.
- The replace operation at position 3 combines the effects of a deletion and an insertion.
- The close tag </root> activates a validity test that takes into account the root's children. This test follows the definition of the run of the tree automaton. □

The implementation of this approach is done by Algorithm 1. This algorithm takes a tree automaton representing the schema, an XML document and a set of updates to be performed on the document. The algorithm checks whether the updates should be accepted or not. It proceeds by treating the XML tree in document order. During the execution, the path from the root to the *current* position p defines a borderline between nodes already treated and those not already considered.

Our algorithm keeps two structures in order to perform the validation. The first one stores the states *allowed* at the current position by the tree automaton. The second one contains, for each position p' on the borderline path, the states *really assigned* to the left-hand side children of p'. In the following, we formally define these structures.

Definition 2. ***- Permissible states for children of a position*** *p: Let $PSC(p)$ be inductively defined on positions p as follows:*

$$PSC(\epsilon) = \{q \mid \text{there exists } a, S, E \to q_a \in \Delta \text{ such that } t(\epsilon) = a, \\ q \text{ is a state appearing in } E \text{ and } q_a \in Q_f\}$$

$$PSC(pi) = \{q \mid \text{there exists } a, S, E \to q_a \in \Delta \text{ such that } t(pi) = a, \\ q \text{ is a state appearing in } E \text{ and } q_a \in PSC(p)\} \quad \square$$

Roughly speaking, for each position pi (labeled a, child of node p), the set $PSC(pi)$ contains the states that *can be* associated to pi's children. To this end, we find those states appearing in the regular expression E, for each rule associated to the label a. Notice however that we consider only transition rules that can be applied at the current node, according to the label of pi's father (*i.e.*, only those rules having a head that belongs to the set of states $PSC(p)$). For instance, suppose two rules $a, S_1, E_1 \to q_{a1}$ and $a, S_2, E_2 \to q_{a2}$. Consider now that an element $struct_1$ has a child that should conform to rule $a, S_1, E_1 \to q_{a1}$ while an element $struct_2$ has a child that should conform to rule $a, S_2, E_2 \to q_{a2}$. When computing the state associated to an element labeled a that is a child of $struct_1$, state q_{a2} is not considered. This is possible because $PSC(p)$ for the element being treated contains just q_{a1}.

The following definition shows how each position p is associated to a list composed by the set of states assigned by the tree automaton to p's children. This list is built taking into account the updates to be performed on the XML document.

Definition 3. ***- State attribution for the children of a position*** *p: Given a position p, the list SAC(p) is composed by the set of states (associated by the schema verification) to the children of position p as follows:*

$$SAC(p) = [Q_{att}, Q_1, \ldots, Q_n]$$
where for $1 \leq i \leq n$:

$$Q_i = \begin{cases} \{\ \} & \textbf{\textit{If}}\ pi\ \textbf{\textit{is a delete position}} \\ \Phi \cap PSC(p) & \textbf{\textit{If}}\ pi\ \textbf{\textit{is an insert or a replace position}} \\ \{q \mid q \in PSC(p), t(pi) = a, \textit{there is a rule } a, S, E \rightarrow q\} & \\ & \textbf{\textit{If}}\ pi\ \textbf{\textit{has no descendant update positions}} \\ \{q \mid \textit{there is a rule } a, (S_{compulsory}, S_{optional}), E \rightarrow q,\ \textit{such that} & \\ \quad t(pi) = a,\ \textit{for } Q_{att}\ \textit{of } SAC(pi)\ \textit{we have } S_{compulsory} \subseteq Q_{att} & \\ \quad \textit{and } Q_{att} \setminus S_{compulsory} \subseteq S_{optional}\ \textit{and} & \\ \quad L(E) \cap L(SAC(pi)) \neq \emptyset\} & \\ & \textbf{\textit{If}}\ pi\ \textbf{\textit{is an ascendant of an update position}} \end{cases}$$

Moreover, the set Q_{att} contains the states associated to p's children that are attributes. The set Φ contains the states associated to the root of the sub-tree being inserted at position p, i.e., the result of a successful local validation. □

The construction of each set Q_i in the list $SAC(p)$ depends on the situation of p with respect to the update positions. When pi is an update position the set Q_i takes into account the type of the update. If pi is an ancestor of an update position then it represents a position where a validity test may be necessary. In this case, Q_i is the set of states associated to pi when the validation at this position succeeds. If there is no update over a descendant of pi, then Q_i contains all possible states for pi.

Now we present Algorithm 1 that is responsible for the construction of both $PSC(p)$ and $SAC(p)$, for each position p.

Algorithm 1 - Incremental Validation of Multiple Updates
Input:
(*i*) *doc*: An XML document
(*ii*) $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$: A tree automaton
(*iii*) $UpdateTable$: A relation that contains updates to be performed on *doc*.
Each tuple in $UpdateTable$ has the form $\langle pos, op, T_{pos}, \Phi \rangle$ where pos is an update position (considering the tree representation of *doc*), op is an update operation, T_{pos} is the subtree to be inserted at pos (when op is an insertion or a replace operation) and Φ is the set of states associated to the root of T_{pos} by the execution of $\mathcal{A}$ over T_{pos} (*i.e.*, the result of the local validation). All inserted sub-trees are considered to be locally valid.
Output: If the XML document remains valid after all operations in $UpdateTable$ the algorithm returns the boolean value *true*, otherwise $false$.

(1) **for** *each event v in the document*
(2) *skip*:= **false**;
(3) **switch** *v* **do**
(4) **case** *start of element a at position p*:
(5) **if** *a* ≠"<root>"{

```
(6)          if ∃ u = (p, delete, T_p, Φ) ∈ UpdateTable then skip:= true;
(7)          if ∃ u = (p, replace, T_p, Φ) ∈ UpdateTable then {
(8)              Compute Q_p (Definition 3);
(9)              if (Q_p = ∅) then report "invalid" and halt;
(10)             SAC(father(p)) = SAC(father(p))@Q_p;
                 //Append Q_p to SAC(father(p))
(11)             skip:= true;
(12)         }
(13)         for each u = (p, insert, T_p, Φ) ∈ UpdateTable do {
(14)             Compute Q_p (Definition 3);
(15)             if (Q_p = ∅) then report "invalid" and halt;
(16)             SAC(father(p)) = SAC(father(p))@Q_p;
(17)         }
(18)         if ∄u' = (p', op', T', Φ') ∈ UpdateTable such that p ≺ p' {
                 //If there is no update over a descendant of p
(19)             Compute Q_p (Definition 3);
(20)             SAC(father(p)) = SAC(father(p))@Q_p;
(21)             skip:= true;
(22)         }
(23)     }
(24)     if a ="<root>" or ¬skip then {
                 //If p is an ascendant of an update position
(25)         Compute PSC(p) (Definition 2);
(26)         SAC(p) = SAC(p)@Q_att;
                 //Starting the construction of the list SAC(p)
(27)     }
(28)     if skip then skipSubTree(doc, a, p);
(29) case end of element a at position p:
(30)     if ∃ u = (p.i, insert, T_p.i, Φ) ∈ UpdateTable
             and p.i is a frontier position then {
(31)         Compute Q_p.i (Definition 3);
(32)         if (Q_p.i = ∅) then report "invalid" and halt;
(33)         SAC(p) = SAC(p)@Q_p.i;
(34)     }
(35)     Compute Q_p (Definition 3);
(36)     if (Q_p = ∅) then report "invalid" and halt;
(37)     if a ≠"</root>" then SAC(father(p)) = SAC(father(p))@Q_p;
(38) report "valid"                                                    □
```

Algorithm 1 processes the XML document *à la* SAX [2]. While reading the XML document, the algorithm uses the information in *UpdateTable* to decide which nodes should be treated. When arriving to an open tag representing a position p concerned by an update, different actions are performed according to the update operation:

- delete: The subtree rooted at p is skipped. This subtree will not appear in the result and thus should not be considered in the validation process (line 6).
- replace: The subtree rooted at p is changed to a new one (indicated by T_p in the *UpdateTable*). The set of states Q_p indicates whether the locally valid subtree T_p is allowed at this position. The set Q_p is appended to the list $SAC(father(p))$ to form the list that should contain the states associated to each sibling of p. The (original) sub-tree rooted at p is skipped (lines 7-12).

– insert: The validation process is similar to the previous case for each insertion at p (lines 13-17), but the (original) sub-tree rooted at p is *not* skipped since it will appear in the updated document on the right of the inserted sub-trees.

When we are in a position p (labeled a) where there is no update over a descendant (lines 18-22) we can skip the sub-tree rooted at p. The list $SAC(father(p))$ is appended with the set $\{q \mid$ there is a rule $a, S, E \rightarrow q$ in Δ such that $q \in PSC(father(p))$ and $T(p) = a\}$. In other words, in $SAC(father(p))$, the set Q_p contains the permissible states for the child at position p. We use *skipSubTree* (line 28) to "skip nodes" until reaching a position important for the incremental validation process. Notice that when such a position is reached, *skipSubTree* changes the value of the variable *skip* accordingly.

When reaching an open tag representing a position p that is an ascendant of an update position, the structures $PSC(p)$ and $SAC(p)$ should be initialized (lines 24-27) . The set $PSC(p)$ contains the states that can be associated to the children of the element at position p (labeled a). To this end, we find the states appearing in the regular expression E of rules associated to label a. Only rules that can apply to the current element are considered, *i.e.*, only those having a head that belongs to $PSC(father(p))$ (see Definition 2).

When reaching a close tag representing a position p we verify firstly if there is an insert operation on the frontier position (*i.e.*, on a position $pi \notin dom(T)$ such that $p \in dom(T)$). In this case, the insertion is performed (lines 30-34) .

Next (lines 35-37), we should test whether the p's children respect the schema. In fact, reaching a (not skipped) close tag (representing position p), means that updates were performed over p's descendants.

Schema constraints for the current node p (labeled a) are verified by taking into account the list $SAC(p)$ (*i.e.*, $[Q_{att}, Q_1, \ldots, Q_n]$) which, at this point, is completely built. Recall that the set Q_{att} contains the states associated to the attributes of p while the sets $Q_1, \ldots, Q_n$ contain the states associated to each element child of p (in the document order). In fact, at this point of the algorithm, our goal is to find the set Q_p to be appended to $SAC(father(p))$. This computation corresponds to the last case of Definition 3.

More precisely, we consider the language $L(SAC(p))$ which is defined by the regular expression $(q_1^0 \mid q_1^1 \mid \ldots \mid q_1^{k_1}) \ldots (q_n^0 \mid q_n^1 \mid \ldots \mid q_n^{k_m})$ where each $k_i = |Q_i|$ and each $q_i^j \in Q_i$ (with $1 \leq i \leq n$). The resulting set of states Q_p is composed by all states q for which we can find transition rules in Δ of the form $a, (S_{compulsory}, S_{optional}), E \rightarrow q$, that respect all the following properties: (1) q is a state in $PSC(father(p))$; (2) $S_{compulsory} \subseteq Q_{att}$; (3) $Q_{att} \setminus S_{compulsory} \subseteq S_{optional}$ and (4) $L(E) \cap L(SAC(p)) \neq \emptyset$.

Notice that Algorithm 1 considers all the updates over the children of a node p before performing the validity test on p.

Our algorithm is general, however, as shown in the following proposition, for the case of single-type tree languages, $SAC(p)$ (for each position p) is a list of singleton sets. In this case, the operation $L(E) \cap L(SAC(p)) \neq \emptyset$ is reduced to the verification of a word to belong to a regular language.

Proposition 1. *Given a schema defining languages in STTL, for each position p in Algorithm 1, we have that the set of states assigned to element children that are not delete positions is always a singleton set, that is: if $SAC(p) = [Q_{att}, Q_1, \ldots, Q_n]$ then $| Q_i |= 1$ for all $1 \leq i \leq n$.* □

PROOF: By cases, on the definition of each Q_i:
If *pi* is an insert or a replace position. In this case, we have $Q_i = \Phi \cap PSC(p)$. If we suppose that $\{q_1, q_2\} \subseteq Q_i$, then (i) Both states q_1 and q_2 are in competition (since they belong to Φ) and (ii) they belong to the same regular expression in a transition rule (since they belong to $PSC(p)$). The conjunction of the two conditions above contradicts the definition of a single-typed language, to which the schema belongs.
If *pi* has no descendant update positions. In this case the set $Q_i = \{q \mid q \in PSC(p), T(pi) = a,$ there is a rule $a, S, E \rightarrow q\}$.
If we suppose that $\{q_1, q_2\} \subseteq Q_i$, then, by the definition of Q_i, the states q_1 and q_2 compete to each other (since there is only one label associated to the position *pi* of the tree), leading to a contradiction.
If *pi* is an ascendant of an update position. In this case the set Q_i is also defined as being composed by states which are in the right-hand side of a rule for a given label a. The existence of more than one state in this set contradicts the definition of a single-typed language, to which the schema belongs. □

Proposition 1 allows us to define a simplification on Algorithm 1, to use single states instead of sets of states, in such a way that, for these classes of tree languages, we can represent $SAC(p) = [Q_{att}, Q_1, \ldots, Q_n]$ as a set of states Q_{att} and a word of states.

Notice that while performing validation tests, a new updated XML tree is being built (as a modified copy of the original one). If the incremental validation succeeds, a *commit* is performed and this updated version is established as our current version. Otherwise, the *commit* is not performed and the original XML document stays as our current version. The next section considers the implementation of our method, comparing it to a validation from scratch.

5 Complexity and Experimental Results

As said in Section 1, the complexity of our method is presented taking into account that the cost of "skipping" nodes is not relevant when compared to the cost of validation actions. In this context, notice that in Algorithm 1 (lines (35)-(37)), for each node $p \in dom(T)$ which is an ascendant of an update position, a *validation step* is executed.

Let E be the regular expression defining the structure of p's children. When a DTD or XSD schema is used, each validation step corresponds to checking whether a word w is in $L(E)$, where w results from the concatenation of the states associated to p's children. Thus, for DTD or XSD schema each validation step is $O(|w|)$.

When a specialized DTD schema is used, each validation step corresponds to checking if there is a word $w = q_i, \ldots, q_n$ in $L(E)$ such that $q_i \in Q_i, \ldots, q_n \in Q_n$ (see the run definition on Section 3). In other words, we should test if $L(E_{aux}) \cap L(E) \neq \emptyset$, where E_{aux} is the regular expression representing all the words we can build from the concatenation of the states associated to p's children, *i.e.*, $E_{aux} = (q_1^0 \mid q_1^1 \mid \ldots \mid q_1^{k_1}) \ldots (q_n^0 \mid q_n^1 \mid \ldots \mid q_n^{k_n})$. This test is done by the intersection of the two automata $M_{E_{aux}}$ and M_E. The solution of this problem runs in time $O(n^2)$ where n is the size of the automata [11, 12].

Thus, if we assume that n is the maximum number of children of a node (fan out) in an XML tree T, then a *validation step* runs in time $O(n)$ (for DTD or XML schema) or in time $O(n^2)$ (for specialized DTD).

Let m be the number of updates to be performed on a tree t of size $|T|$. Given an update position p, a validation step should be performed for each node on the path between p and the root. If all m updates are on the leaves, the number of validation steps equals the depth of the tree ($O(\log_n |T|)$). However, in practice, updates can happen in any level $l \leq (\log_n |T|)$. Thus, we can conclude that when dealing with DTD or XSD schema Algorithm 1 runs in time $O(m.n.\log_n |T|)$. However for specialized DTD schema it runs in time $O(m.n^2.\log_n |T|)$. Notice that in some cases, we can introduce optimizations that avoid going up to the root to finish validation. For instance (see [8]), when dealing with a DTD, we just need to verify whether the state associated to the father of an update position changes. This is easily done by using transition rules. Thus, for multiple updates with DTD, we just need to perform verifications until reaching the father of the update position which is the nearest to the root.

Table 51 - Experimental results.

Attributes	*Attributes + Elements*	*Xerces (ms)*	*our Application (ms)*
675	*2 300*	*828*	*1 127 (50 updates)*
6.4K	*722K*	*1 047*	*1 219 (50 updates)*
12K	*47K*	*1 266*	*1 500 (50 updates)*
50.5K	*180K*	*2 125*	*2 406 (50 updates)*
156K	*540K*	*4 890*	*3 657 (50 updates)*
473K	*1.6M*	*11 922*	*5 843 (50 updates)*
1.5M	*5.1M*	*35 969*	*13 297 (10 updates)* *17 891 ms (50 updates)*
2.3M	*10.2M*	*71 031*	*23 578 ms (10 updates)* *35 500 ms (50 updates)*
6M	*20.5M*	*142 266*	*45 344 ms (10 updates)* *59 890 ms (50 updates)*
18M	*61.1M*	*422 859 ms*	*136 735 ms (10 updates)* *149 453 ms (50 updates)*

The worst-case time complexity analysis presented above uses the maximum fan out of children for nodes as well as the depth of the tree for a balanced XML tree. This consideration leads us to a theoretical (worst case) time complexity that is not representative of the practical situations in which the algorithm will be used. Experimental results (Table 51) show that our algorithm behaves very efficiently in practice.

Table 51 compares our incremental validation method (Algorithm 1) to a validation from scratch approach which requires reading the entire document after each update. We use Xerces [**?**] to perform validation from scratch. In the current implementation of Algorithm 1, the structures *PSC*() and *SAC*() are implemented as stacks.

In order to compare these approaches we use ten XML documents of different sizes (from 2 500 to 61 000 000 nodes). These documents are part of a STTL and describe different car suppliers. They are valid wrt an XDS whose principal schema constraints are expressed by the following transition rules.

$$
\begin{array}{ll}
supplier, (\emptyset, \emptyset), q^{+}_{shop} q^{*}_{garage} & \rightarrow q_{supplier} \\
shop, (\emptyset, \emptyset), q^{*}_{newVeh} & \rightarrow q_{shop} \\
garage, (\emptyset, \emptyset), q^{+}_{oldVeh} & \rightarrow q_{garage} \\
vehicle, (\{id\}, \{type\}), q_{name} q_{cv} q_{cat}? & \rightarrow q_{newVeh} \\
vehicle, (\{id\}, \emptyset), q_{name} q_{cv} q_{km}? & \rightarrow q_{oldVeh}
\end{array}
$$

Table 51 shows the results obtained when comparing our incremental validation method and the validation from scratch of Xerces. Table 51 provides the size of each XML document by indicating, for each of them, the number of attributes and elements. Given an XML document, we consider a sequence of 10 or 50 updates over it. The algorithm was implemented in Java, and experiments were performed on a 3.0 GHz Pentium system with 512MB of memory and a 30GB, 4200rpm hard drive.

Table 51 and Fig. 4 show that our incremental validation method is very efficient for large documents. Indeed, it takes almost a third of the time needed for Xerces to validate (50 or 10) updates on a document having 61 000 000 nodes. Similarly, it takes about half of the time needed for Xerces to validate documents having 10 000 000 nodes.

Fig. 4. Xerces (validation from scratch) × Our incremental validation method.

6 Related Work

The importance of algorithms for the efficient validation of XML documents grows together with the use of schema languages. Algorithms for the incremental validation are very useful when dealing with very large XML documents, since they can substantially reduce the amount of time of the verification process. Several XML document editors such as XML Mind [3] and XMLSpy [4] include features for the validation of documents (wrt one or more schema languages). However, the documentation of most of these tools include little or no information on their validation algorithms.

In [13] validation algorithms are presented but incremental validation is not considered. One of the most referenced work dealing with incremental validation of XML documents wrt schema constraint is [6, 14]. In those papers incremental validation methods wrt DTD, XSD and specialized DTDs are presented. Their approach is based on the use of bottom-up, binary tree automata; *i.e.*, schema descriptions are translated to binary tree automata and XML documents are seen as binary trees. They propose two main incremental algorithms to validate a number m of updates on the leaves of a given tree T. The first algorithm, for DTD and XSD, has time complexity $O(m.log|T|)$, and uses an auxiliary structure of size $O(|T|)$. The second algorithm, for specialized DTDs, has time complexity $O(m.log^2|T|)$ and also uses an auxiliary structure of size $O(|T|)$.

Our approach deals with multiple updates and incremental validation over an unranked tree. As we have already said, our algorithm visits all the nodes of an XML tree but only some nodes trigger the validation actions that represent the relevant cost of the approach. Thus, we can say that our time complexity is similar to the one found in [6, 14]. Due to our use of unranked trees instead of the binary trees of [6, 14], each validation step on our method can be more expensive, but our XML tree is usually much shorter.

Our work differs from that in [14] in four main aspects:

1. Our update operations can be applied at any node of the tree, and not just on the leaves. This feature permits us to change large areas of the XML tree on one single operation.
2. In [14], testing whether a word belongs to a language is done in an incremental way by storing an auxiliary structure. This optimization can be easily integrated in our algorithm, but it seems to be interesting only in case of very large fan-out.
3. When dealing with DTD and XSD we do not use auxiliary structures to store the result of a previous validation process.

7 Conclusions

In this paper we present a method to incrementally verify schema constraints dealing with multiple updates in XML documents. The update operations correspond to the insertion, deletion and replacement of *any* subtree of an XML document. Only after analyzing all update operations in a given update sequence, the validity of the XML document is determined.

The incremental verification of schema constraints is performed by using a bottom-up tree automaton to re-validate the parts affected by the sequence of updates. We have analyzed the different characteristics of tree automata according to the schema language, showing that these characteristics match the taxonomy of regular tree grammars introduced in [13].

The algorithms presented here have been implemented in Java, and the experimental results show that the incremental schema verification with multiple updates for large XML documents takes almost a third of the time needed for the validation from scratch.

Our time complexity and our experimental results show the efficiency of our approach. Our approach is as efficient as those proposed by [10, 14] and it has some ad-

vantages such as space requirements, multiple updates on any tree node and flexibility of integrating schema and constraint validation.

References

1. Apache xml editor. Available at http://www.apache.org/xerces-j/.
2. Official website for SAX. Available at http://www.saxproject.org.
3. Xmlmind. Available at http://www.xmlmind.com/xmleditor/.
4. Xmlspy. Available at http://www.xmlspy.com/products_doc.html.
5. M. A. Abrao, B. Bouchou, M. Halfeld-Ferrari, D. Laurent, and M. A. Musicante. Incremental constraing checking for XML documents. In *Proceedings of the Second International XML Database Symposium*, number 3186 in Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2004.
6. Andrey Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst*, 29(4):710–751, 2004.
7. B. Bouchou, A. Cheriat, M. Halfeld Ferrari Alves, D. Laurent, and M. Musicante. Tree automata in the validation of xml under different schemas languages. Technical Report 281, Université François-Rabelais de Tours, LI/ Campus de Blois, 2005.
8. B. Bouchou and M. Halfeld Ferrari Alves. Updates and incremental validation of XML documents. In Springer, editor, *The 9th International Workshop on Database Programming Languages (DBPL)*, number 2921 in LNCS, 2003.
9. B. Bouchou, M. Halfeld Ferrari Alves, and M. A. Musicante. Tree automata to verify key constraints. In *Web and Databases (WebDB)*, San Diego, CA, USA, June 2003.
10. Y. Chen, S. Davidson, and Y. Zheng. Validating constraints in XML. Technical Report MS-CIS-02-03, Department of Computer and Information Science, University of Pennsylvania, 2002.
11. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley Publishing Company, second edition, 2001.
12. George Karakostas, Richard J. Lipton, and Anastasios Viglas. On the complexity of intersecting finite state automata. In *IEEE Conference on Computational Complexity*, pages 229–234, 2000.
13. M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema language using formal language theory. In *Extreme Markup Language, Montreal, Canada*, 2001.
14. Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2003.