Efficient Constraint Validation for Updated XML Databases

Béatrice Bouchou, Ahmed Cheriat and Mírian Halfeld Ferrari
Université François-Rabelais de Tours/Campus Blois - LI - France
E-mail: beatrice.bouchou@univ-tours.fr, ahmed.cheriat@univ-tours.fr, mirian.halfeld@univ-tours.fr

Dominique Laurent
Université de Cergy-Pontoise - ETIS - UMR CNRS 8051 - France
E-mail: dominique.laurent@dept-info.u-cergy.fr

Maria Adriana Lima
Pontifícia Universidade Católica de Minas Gerais - Poços de Caldas - Brazil
E-mail: adriana@pucpcaldas.br

Martin A. Musicante
Universidade Federal do Rio Grande do Norte - DIMAp - Natal - Brazil
E-mail: mam@dimap.ufrn.br

Keywords: XML, schema, constraints

Received: May 12, 2006

XML constraints are either schema constraints *representing rules about document structure (*e.g.* a DTD, an XML Schema definition or a specification in Relax-NG), or* integrity constraints*, which are rules about the values contained in documents (*e.g.* primary keys, foreign keys, etc.).*
We address the problem of incrementally verifying these constraints when documents are modified by updates. The structure of an XML document is a tree, whose nodes are element (or attribute) names and whose leaves are associated to values contained in the document. Considered updates are insertion, deletion or replacement of any subtree in the XML tree. Schema constraints are represented by tree automata and tree grammars. Key and foreign key constraints are represented by attribute grammars, adding semantic rules to schema grammars, to carry key and foreign key values (to verify their properties).
Our incremental validation tests both schema and integrity constraints while treating the sequence of updates, in only one pass over the document. Only nodes involved in updates trigger validation tests. An analysis of complexity shows that worst cases are determined by the shape of the XML tree being processed (asymptotic upper bounds are presented). Experimental results show that our algorithms behave efficiently in practice.

Povzetek: Opisana je inkrementalna verifikacija podatkovnih baz XML.

1 Introduction

XML is now a standard for exchanging data and, by extension, for representing information. For reliable exchange as well as for information system design, it is necessary to define rules that data must conform to.

XML documents can be represented as unranked trees: nodes are identified by a position and associated to a label, which is the name of an element or an attribute. Our tree representation of an XML document is exemplified in Fig.1. This figure shows the representation of part of an XML document, which will be used in some of our examples. Notice that the data values in the XML document appear as leaves of its tree representation.

XML documents can be constrained either by *structural rules* or *semantic rules*. Rules about the structure of documents are called a *schema*: there are several formalisms to express these constraints, *e.g.* DTD, XML Schema (XSD) [4] or Relax-NG [41]. It is a well known fact that a schema can be represented by a tree grammar. From this grammar, it is easy to derive a tree automaton [20, 43]: the validation of the document wrt the schema is the run of this tree automaton over the XML tree.

Integrity constraints impose restrictions to the values that may appear in XML documents. Integrity constraints (*e.g.*, primary and foreign keys) are devised to improve the semantic expressiveness of XML, in the same way as their counterpart in relational databases. In this paper we propose to represent integrity constraints by attribute grammars [8, 40], adding semantic rules to schema grammars, to carry key and foreign key values (to verify their properties).

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), represented by the XML tree T. In our approach, update operations correspond to the insertion, the deletion and the replacement of subtrees of T. They are results of a user request's pre-processing. A user can modify an XML document either by using a text editor such as *XmlSpy* [1] or *Stylus studio* [46], or by means of a program, written in an update language such as *XSLT* [26] or *UpdateX* [47], which is embedded in XQuery [16].

In the case of a text editor, different scenarios are possible: validity is tested while the user changes his document (this method is cumbersome and not realistic since it implies verification after each change) or validity tests are explicitly activated (by the user) after the performance of several changes. In the latter context, some updates can be "subsumed" by others (*e.g.*, the user can build or change a part of the document and finally delete it). Thus, before applying an *incremental* validation method one should define which updates are to be taken into account. To this end, one can either use pre-processing over the set of changes performed by the user or compute the difference between the last verified version and the current (modified) one. In both cases, the overhead caused by this pre-processing may be greater than incremental validation profit, this is why most of the existing XML editors apply a complete re-validation.

The use of a language to specify updates open different possibilities of work. Considering the integration of such an approach to a text editor framework, we may imagine that an update language editor would be capable of specifying update positions and then our algorithms would directly apply. More generally, an update language allows to specify updates such as, for example: *increment by 10 percents all accounts of a given consumer*. Thus, it allows to manage XML documents from a database point of view. To this end, update languages integrated in XQuery are the most promising solutions. The W3C has edited a first public working draft on *XQuery Update Facility* [22]: it recommends that the evaluation of any expression produces either a new XML document or a pending update list, which is a set of update primitives. An update primitive has a target node and consists in insertions, deletions or replacements to be operated at this node (or just before, or just after or just under this node, in case of insertions). The update list can be held in wait until an operation (upd:applyUpdate) is performed. An operation of validation (upd:revalidate) must exist.

Update operations considered in this paper can be seen as updates in such an update pending list.

Only updates that do not violate constraints are accepted. Thus, before applying updates on a valid tree, we need 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 updates. In accordance with the *snapshot semantics* [14, 47], document validity is assured just after considering the *whole* sequence of updates. The snapshot semantics consists in delaying update application to the end of query evaluation. In this way, all updates always refer to the original document[1].

In the following, we roughly explain how the incremental validation tests are performed. Let *UpdateTable* be the 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 and integrity constraints are treated as separated sub-routines. A system that makes these routines work together is a simple generalization of the one presented in this paper. The sub-routines can be summarized as follows.

Schema constraint routine:

• 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.

Integrity constraint routine:

• Similarly to the schema constraint routine, when an update node is reached, the required update is taken into account (considered as done). However, contrary to the schema constraint routine, in the integrity constraint routine the removal of a subtree (rooted at an update position p) triggers the subtree traversal for searching key and foreign key values involved in the update. For instance, consider a key constraint on the document of Fig. 1 establishing that in a *collection*, a *recipe* is uniquely identified by its *name* and *author*. We assume that a deletion is required at position 0.1. The deletion "treatment" is activated (in order to find key values) when the open tag `<recipe>` is reached. After the traversal of the subtree rooted at position 0.1, *i.e.*, when reaching the close tag `</recipe>` a verification test is performed. This test consists in checking whether the key value $\langle Shrimp\ Soup, J.Fox\rangle$ (involved in the deletion) is referenced by a non deleted foreign key. To perform this test the routine uses an auxiliary structure (called *keyTree*) created during the first key validation (from scratch). During the tree traversal necessary to analyze an update sequence, some marks can be inserted into the *keyTree* to indicate that the document is temporarily invalid. A subsequent update in the same transaction can re-establish validity and remove the mark.

[1]The language XQuery! (*"XQueryBang"*) [33] extends XQuery 1.0 with compositional updates, offering user-level control over update application. The user controls update semantics via an operator called *"snap"* (for snapshot). However, it can also implements snapshot semantics.

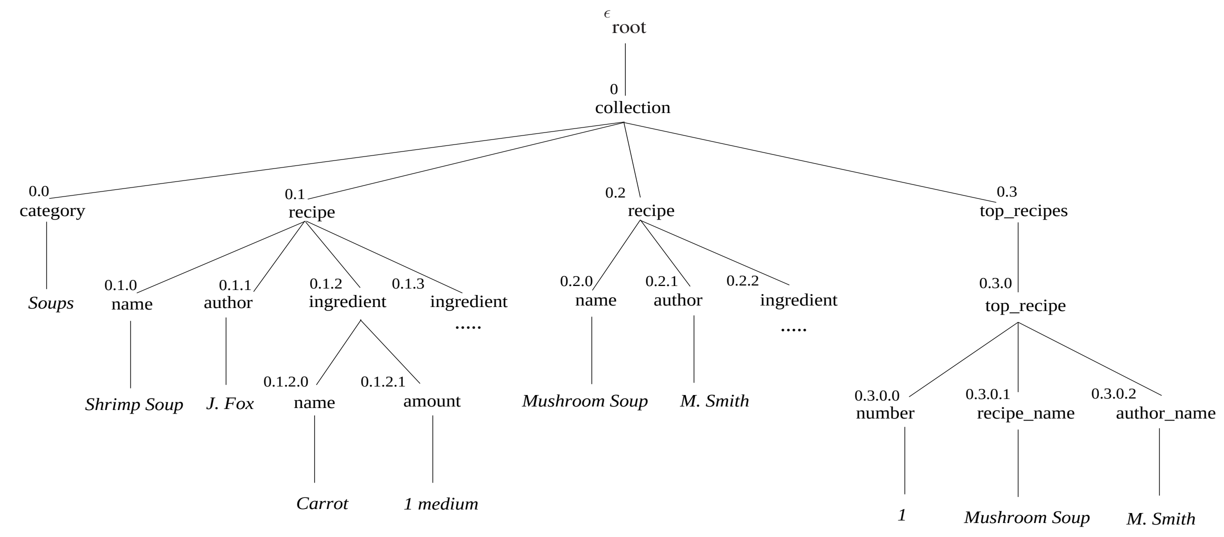

Figure 1: Tree representation of an XML document.

• When reaching the root, a final test verifies if any violation mark exists in *keyTree* (*i.e.*, a constraint violation not corrected until the end of the traversal).

• All other nodes are skipped. □

XML trees are treated in the style of SAX [3], in order to allow the treatment of much bigger XML documents [42][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 worth noting that, although our algorithm visits all nodes in the XML tree, 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 3.2 and 4.3 we consider the complexity of our method only wrt the total number of validation actions that should be performed.

The algorithms presented in this paper have been implemented in Java, and experimental results demonstrate advantages of the incremental schema verification over the verification from scratch, for multiple updates over large XML documents. Experimental results obtained with our key and foreign key validation routines are also good, despite their theoretical complexity: curves grow almost linearly with the size of the processed document. Tests for the incremental key verification programs give even better results than those for the verification from scratch.

The main contribution of this paper is the integration of incremental schema, key and foreign key validation, while dealing with multiple updates. It extends our previous work [7, 17, 18] not only in this aspect but also in the use of attribute grammars to deal with integrity constraint verification.

[2]It is also possible to consider our routines in a DOM context. In this case space requirements are bigger, but time complexity can be smaller.

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 [11, 45]. However, contrary to [11, 45], our update operations can be applied at any node of the XML tree and auxiliary structures are not necessary when using DTD and XSD. In terms of key validation, our proposition is close to [23], but contrary to them, we treat foreign keys and multiple updates. Moreover, our routines for schema validation and integrity constraint verification can work simultaneously.

This paper is organized as follows. Section 2 introduces some necessary concepts. In Section 3 we consider the validation wrt schema constraints while in Section 4 we present our validation method wrt key and foreign key constraints. In section 5 we conclude and discuss some perspectives.

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.

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 [7, 17, 18] and follows the ideas in [14, 47].

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 set of updates required by a user. In the resulting $UpdateTable$ the following properties hold:

P1 - 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.

P2 -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 $p \preceq p'$.

P3 - $UpdateTable$ then one of the operations involving p can be *replace* or *delete*, but all others are *insert*.

P4 - Updates in an $UpdateTable$ are ordered by position, according to the document order.

3 Schema verification

A tree automaton can be built from a schema specified using schema languages such as DTD, XSD or RELAX NG. In our approach, we use a bottom-up unranked tree automaton capable of dealing with both (unordered) attributes and (ordered) elements in XML trees [17].

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 [42].

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[3] 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 XSD 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.

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

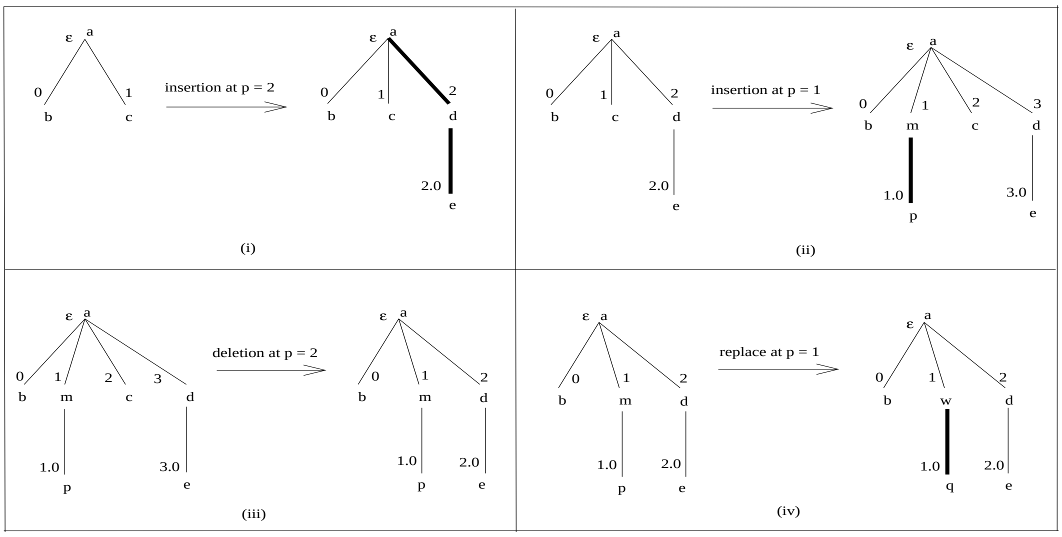

Figure 2: Update operations over a tree T. (i) Insertion at a frontier position. (ii) Insertion at a position of $dom(T)$. Right siblings are shifted right. (iii) Deletion. Right siblings are shifted left. (iv) Replace.

Notice that, as shown in [42], 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).

Example 1 Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a tree automaton where $Q_f = \{q_C\}$. Consider the following transition rules with q_{A1} and q_{A2} as competing states. Assume that states q_1, q_2 and q_3 are defined by simple transition rules that do not introduce competition and do not have regular expressions involving q_{A1} and q_{A2}.

$$\begin{array}{lll} (1) & a, (\emptyset, \emptyset), q_1 q_2? & \to q_{A1} \\ (2) & a, (\emptyset, \emptyset), q_1 q_3? & \to q_{A2} \\ (3) & b, (\emptyset, \emptyset), (q_{A1} \mid q_{A2})^* & \to q_B \\ (3') & b, (\emptyset, \emptyset), (q_{A1})^* & \to q_B \\ (4) & c, (\emptyset, \emptyset), (q_B)^* & \to q_C \end{array}$$

In this context, consider different sets Δ containing subsets of the above rules: Firstly, consider a set Δ with rules (1), (2), (3) and (4). The language recognized by $\mathcal{A}$ is a RTL which is not a STTL. Secondly, assume that Δ contains rules (1), (2), $(3')$ and (4). Then the language recognized by $\mathcal{A}$ is a STTL which is not a LTL. Finally, assume Δ has no competing states (*e.g.*, from the above rules, only rules (1), $(3')$ and (4) are in Δ). In this case, the language recognized by $\mathcal{A}$ is a LTL. □

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 $\mathcal{Q}_p$. The assignment $r(p) = \mathcal{Q}_p$ is done by verifying whether the attribute and element constraints imposed to p's children are respected. The set $\mathcal{Q}_p$ is composed by all the states q such that:

1. There exists a transition rule $a, (S_{compulsory}, S_{optional}), E \to q$ in $\mathcal{A}$.
2. $T(p) = a$.
3. $S_{compulsory} \subseteq \mathcal{Q}_{att}$ and $\mathcal{Q}_{att} \backslash S_{compulsory} \subseteq S_{optional}$ where $\mathcal{Q}_{att}$ 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 $q_1 \in \mathcal{Q}_1, \ldots, q_n \in \mathcal{Q}_n$, where $\mathcal{Q}_1 \ldots \mathcal{Q}_n$ 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 [17].

Restricted forms of schema languages permit simplified versions of *run*. For instance, in a run of a tree automaton for (simple) DTD, the sets $\mathcal{Q}_p$ are always singleton. This situation considerably simplifies the implementation of item (4) above [17]. 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 XSD, 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).

3.1 Incremental schema verification

Given a tree T and a sequence 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 [42]. □

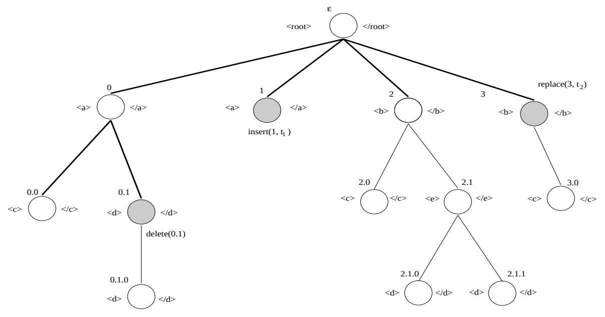

Figure 3: XML tree and update operations.

The following example illustrates our method in an intuitive way.

Example 2 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 </a> (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 subtree, 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 sequence 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:*

$$\begin{aligned} PSC(\epsilon) = \{q \mid \exists\, a, S, E \to q_a \in \Delta \\ \textit{such that } t(\epsilon) = a,\ q \textit{ is a state} \\ \textit{appearing in } E \textit{ and } q_a \in \mathcal{Q}_f\} \\ PSC(pi) = \{q \mid \exists\, a, S, E \to q_a \in \Delta \\ \textit{such that } t(pi) = a,\ q \textit{ is a state} \\ \textit{appearing in } E \textit{ and } q_a \in PSC(p)\} \end{aligned}$$ □

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 sets of states associated (by the schema verification process) to the children of position* p*, i.e.,* $SAC(p) = [\mathcal{Q}_{att}, \mathcal{Q}_1, \ldots, \mathcal{Q}_n]$*, where each set* $\mathcal{Q}_i$*, for* $1 \leq i \leq n$ *is calculated as described in Fig. 4. Moreover, the set* $\mathcal{Q}_{att}$ *contains the states associated to* p*'s children that are attributes. The set* Φ *contains the states associated to the root of the subtree being inserted at position* p*, i.e., the result of a successful local validation.* □

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

Figure 4: Calculation of the sets $\mathcal{Q}_i$.

The construction of each set $\mathcal{Q}_i$ in the list $SAC(p)$ depends on the situation of p wrt the update positions. When pi is an update position the set $\mathcal{Q}_i$ 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, $\mathcal{Q}_i$ 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 $\mathcal{Q}_i$ 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 subtrees 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 $a \neq$“<root>”{
(6)        if $\exists\, u = (p, \mathsf{delete}, T_p, \Phi) \in UpdateTable$
           then skip:= true;
(7)        if $\exists\, u = (p, \mathsf{replace}, T_p, \Phi) \in UpdateTable$
            then {
(8)          Compute $\mathcal{Q}_p$ (Definition 3);
(9)          if ($\mathcal{Q}_p = \emptyset$) then report “invalid” and halt;
(10)         $SAC(father(p)) = SAC(father(p))@\mathcal{Q}_p$;
                 //Append $\mathcal{Q}_p$ to $SAC(father(p))$
(11)         skip:= true;
(12)       }
(13)       for each $u = (p, \mathsf{insert}, T_p, \Phi) \in UpdateTable$
           do {
(14)         Compute $\mathcal{Q}_p$ (Definition 3);
(15)         if ($\mathcal{Q}_p = \emptyset$) then report “invalid” and halt;
(16)         $SAC(father(p)) = SAC(father(p))@\mathcal{Q}_p$;
(17)       }
(18)       if $\nexists u' = (p', op', T', \Phi') \in UpdateTable$
           such that $p \prec p'$ {
           //If there is no update over a descendant of $p$
(19)         Compute $\mathcal{Q}_p$ (Definition 3);
(20)         $SAC(father(p)) = SAC(father(p))@\mathcal{Q}_p$;
(21)         skip:= true;
(22)       }
(23)     }
(24)     if $a =$“<root>” or $\neg skip$ then {
         //If $p$ is an ascendant of an update position
(25)       Compute $PSC(p)$ (Definition 2);
(26)       $SAC(p) = SAC(p)@\mathcal{Q}_{att}$;
         //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)     foreach $u = (p.i, \mathsf{insrt}, T_{p.i}, \Phi) \in UpdateTable$
         where $p.i$ is a frontier position do {
(31)       Compute $\mathcal{Q}_{p.i}$ (Definition 3);
(32)       if ($\mathcal{Q}_{p.i} = \emptyset$) then report “invalid” and halt;
(33)       $SAC(p) = SAC(p)@\mathcal{Q}_{p.i}$;
(34)     }
(35)     Compute $\mathcal{Q}_p$ (Definition 3);
(36)     if ($\mathcal{Q}_p = \emptyset$) then report “invalid” and halt;
(37)     if $a \neq$“</root>”
         then $SAC(father(p)) = SAC(father(p))@\mathcal{Q}_p$;
(38) report “valid”                                          □
```

Algorithm 1 processes the XML document as it is done by SAX [3]. 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 $\mathcal{Q}_p$ indicates whether the locally valid subtree T_p is allowed at this position. The set $\mathcal{Q}_p$ 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) subtree 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) subtree rooted at p is *not* skipped since it will appear in the updated document on the right of the inserted subtrees.

When we are in a position p (labeled a) where there is no update over a descendant (lines 18-22) we can skip the subtree rooted at p. The list $SAC(father(p))$ is appended with the set $\{q \mid$ there is a rule $a, S, E \to q$ in Δ such that $q \in PSC(father(p))$ and $T(p) = a\}$. In other words, in $SAC(father(p))$, the set $\mathcal{Q}_p$ 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.*, $[\mathcal{Q}_{att}, \mathcal{Q}_1, \ldots, \mathcal{Q}_n]$) which, at this point, is completely built. Recall that the set $\mathcal{Q}_{att}$ contains the states associated to the attributes of p while the sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ 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 $\mathcal{Q}_p$ 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 $k_i = |\mathcal{Q}_i|$ and each $q_i^j \in \mathcal{Q}_i$ (with $1 \leq i \leq n$). The resulting set of states $\mathcal{Q}_p$ is composed by all states q for which we can find transition rules in Δ of the form $a, (S_{compulsory}, S_{optional}), E \to q$, that respect all the following properties: (1) q is a state in $PSC(father(p))$; (2) $S_{compulsory} \subseteq \mathcal{Q}_{att}$; (3) $\mathcal{Q}_{att} \setminus S_{compulsory} \subseteq S_{optional}$ 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.

Example 3 Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a tree automaton where $Q_f = \{q_r\}$. The set Δ contains all the transition rules below together with rules $data, (\emptyset, \emptyset), \epsilon \to q_{data}$, and $\alpha, (\emptyset, \emptyset), q_{data} \to q_\alpha$ (for $\alpha \in \{c, d, m, n\}$).

$(1) r, (\emptyset, \emptyset), q_3 \mid q_4.q_c \to q_r$ $(2) x, (\emptyset, \emptyset), q_2 \to q_4$

$(3) x, (\emptyset, \emptyset), q_1 \to q_3$ $(4) a, (\emptyset, \emptyset), q.q_d^* \to q_1$

$(5) a, (\emptyset, \emptyset), q'.q_d^* \to q_2$ $(6) b, (\emptyset, \emptyset), q_c \to q$

$(7) b, (\emptyset, \emptyset), q_c.q_m.q_n \to q'$

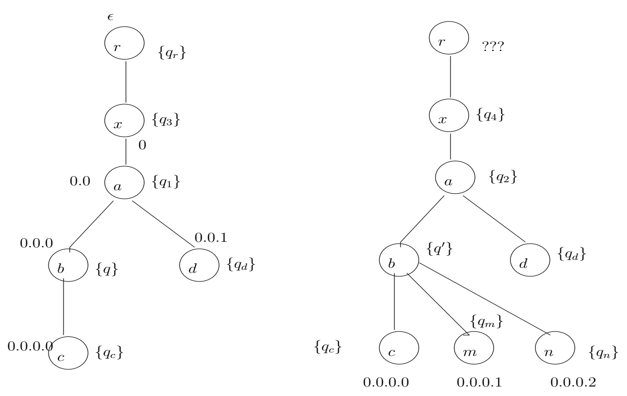

Figure 5: XML tree and its running tree: (a) before updates (b) after insertions.

Fig. 5(a) shows a valid XML tree wrt the schema constraints expressed by $\mathcal{A}$. Fig. 5(b) shows the tree we shall obtain after two insertions on the frontier position 0.0.0.1. In both cases, the result of the running is showed by the set of states associated to each position. Notice that the two insertions on 0.0.0.1 generate changes not only on the set of states associated to its father (0.0.0) but also on the sets of states associated its *ancestors*. Note that even the set of states associated to the root changes and the tree is not valid anymore! This example illustrates why, when dealing with non LTL , the validator tests the ancestors of the update position (and not just its parent). We propose an algorithm that performs tests until reaching the root, but optimizations are possible: one should perform tests until reaching a node whose associated set of states does not need to be changed.

To illustrate the execution of Algorithm 1, consider, in the table below, the situation of structures $PSC()$ and $SAC()$ at some specific instants, during the incremental validation process that take into account the updates illustrated by Fig. 5(b).

Position	Situation of $PSC()$ when we reach `<b>`	Situation of $SAC()$ when we reach `</b>`	Situation of $SAC()$ when we reach `<d>`
0.0.0	$\{q_c, q_m, q_n\}$	$[\{q_c\}, \{q_m\}, \{q_n\}]$	
0.0	$\{q, q', q_d\}$	$[\,]$	$[\{q'\}]$
0	$\{q_1, q_2\}$	$[\,]$	$[\,]$
ϵ	$\{q_3, q_4, q_c\}$	$[\,]$	$[\,]$

According to Definition 2, when `<b>` is reached, states q_c, q_m, q_n are put in $PSC(0.0.0)$ since they appear in rules (6) and (7) (which have label b and whose right-hand side are in $PSC(0.0)$).

When we reach `</b>` the list $SAC(0.0.0)$ is complete since it now contains the set of states assigned to all the children of position 0.0.0. Notice that $SAC(0.0)$ is an empty list since we still do not know the states that are effectively associated to its children. As `</b>` is found, $SAC(0.0.0)$ is popped and the state associated to the leftmost child of position 0.0 can be computed.

Indeed, when we reach <d>, the list $SAC(0.0)$ is not empty anymore - it contains the set of the states we effectively associate to position 0.0.0 (chosen among those in $PSC(0.0)$ and taking into account the transition rules that apply). □

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 $SAC(p) = [\mathcal{Q}_{att}, \mathcal{Q}_1, \ldots, \mathcal{Q}_n]$ *then* $\mid \mathcal{Q}_i \mid = 1$, *for all* $1 \leq i \leq n$. □

PROOF: By cases, on the definition of each $\mathcal{Q}_i$:
If pi is an insert or a replace position. In this case, we have $\mathcal{Q}_i = \Phi \cap PSC(p)$. If we suppose that $\{q_1, q_2\} \subseteq \mathcal{Q}_i$, 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 $\mathcal{Q}_i = \{q \mid q \in PSC(p), T(pi) = a,$ there is a rule $a, S, E \rightarrow q\}$.
If we suppose that $\{q_1, q_2\} \subseteq \mathcal{Q}_i$, then, by the definition of $\mathcal{Q}_i$, 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 $\mathcal{Q}_i$ 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 $SAC(p) = [\mathcal{Q}_{att}, \mathcal{Q}_1, \ldots, \mathcal{Q}_n]$ as a set of states $\mathcal{Q}_{att}$ 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.

3.2 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, validation steps are performed only for those nodes $p \in dom(T)$ which are ascendants of update positions.

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 $L(E)$. The word w is 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 $q_i \in \mathcal{Q}_i, \ldots, q_n \in \mathcal{Q}_n$ (see the definition of a run, in 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 [35, 36].

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 depth h). Given an update position p, in the worst-case, a validation step should be performed for each node on the path between p and the root. For a worst-case analysis, we can also consider that all m updates are performed on the leaves. We also suppose that all the paths from nodes p to the root are disjoint. In this case, the complexity of our algorithm can be stated as $O(m.n.h)$ (DTD or XML Schema) or $O(m.n^2.h)$ (when specialized DTD is considered).

Notice that, in a general XML setting, updates can happen at any level of the tree (so that the limit imposed by h is seldom reached). Moreover, when dealing with multiple updates, part of the paths between the update nodes and the root are common to two or more of these updates. In this case, only one validation action is performed for the shared nodes.

The worst-case of our algorithm is reached for a very unusual configuration of the tree being processed (the configuration maximizing the product $n.h$). In this configuration, one half of the nodes are leafs, children of the root, while the others form a list (pending from the root).

Other singular configurations are:

- A flat tree, where all the nodes (except the root) are leaves, and children of the root node. In this case, the depth of the tree is one and the complexity expressions are reduced to $O(m.n)$ (DTD or XML Schema) or $O(m.n^2)$ (specialized DTD).

- A list, where there is exactly one leaf node and the maximum fan-out of the tree nodes is one. In this case, both complexity expressions are reduced to $O(m.h)$ (DTD, XML Schema or specialized DTD).

• If we consider an n-ranked, balanced tree t, its depth is given by $h = \log_n |t|$, where $|t|$ is the size of the tree. In this case, the complexity expressions are reduced to $O(m.n.\log_n |t|)$ (DTD or XML Schema) or $O(m.n^2.\log_n |t|)$ (specialized DTD).

Notice that when dealing with a DTD, we just need to verify whether the state associated to the father of an update position changes (see [17]). 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. Example 3 illustrates that this optimization cannot be applied to non-LTL schemas.

Experimental results (Table 1) show that our incremental algorithm behaves very efficiently in practice. In order to compare these approaches we use ten XML documents of different sizes (from $3,000$ to $61,000,000$ nodes). These documents are in the STTL class of languages and describe different car suppliers. They are valid wrt an XSD whose principal schema constraints are expressed by the following transition rules:

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

Given an XML document, we consider a sequence of 50 updates over it. Our implementation is just a prototype in Java. Experiments were performed on a 1.5 GHz Pentium M system with 512MB of memory and a 40GB, 5400rpm hard drive.

Table 1 and Fig. 6 show the superiority of Xerces [2] when only validation from scratch is considered. This is a natural result when comparing a prototype with a commercial product. However, when we compare our incremental validation method (Algorithm 1) to a validation from scratch approach Table 1 and Fig. 7 show that our incremental validation method is very efficient for large documents[4]. Indeed, it takes almost a third of the time needed for Xerces to validate 50 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.

A sufficient condition for *commutative* update lists is given in [34]. In this paper, our *UpdateTable* respects this condition and thus, single updates can be performed in any order, without changing the result of the global update on the XML tree. However, as our validation process is done by using SAX, a left-right computation of updates is more efficient than one that considers tree nodes at random. If we do not care about loading all the XML file, a bottom up computation might be proposed, since all update positions

[4]For small documents, the number of updates represents changes over a high percentage of the document. In this case, incremental validation cost is close to our validation from scratch cost and thus it is worse than Xerces, a commercial product.

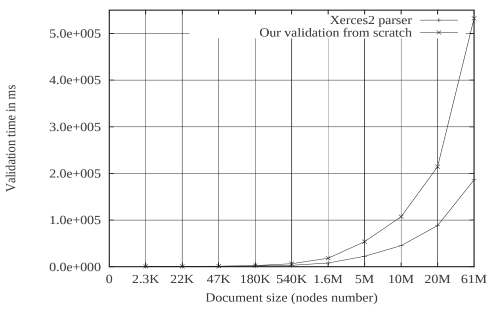

Figure 6: Comparing validation from scratch performed by Xerces and our prototype.

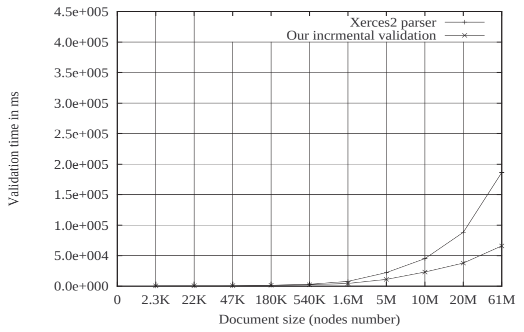

Figure 7: Xerces (validation from scratch) $\times$ Our incremental validation method.

activate validation verification on their ancestor nodes. Indeed, schema validation performance depends on the depth of update nodes as discussed before.

3.3 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 [5] and XMLSpy [1] 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 [42] 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 [11, 45]. In those papers incremental validation methods wrt DTD, XSD and specialized DTDs are presented. Their approach is based on word automata, built from schema descriptions. The words considered are *lines* (*i.e.*, paths) computed from a *binary version* of the document tree. Not only the binary version of the document tree is stored as auxiliary data, but also

Document characteristics		**Existing product**	**Our method**	
Attributes	Attributes + Elements	Xerces (ms)	From scratch (ms)	Incremental (ms)
700	3K	451	683	658 (50 updates)
6.5K	25K	727	814	773 (50 updates)
13K	50K	929	1 156	991 (50 updates)
52K	200K	1 509	2 435	1 520 (50 updates)
170K	600K	3 095	6 357	2 258 (50 updates)
500K	1.7M	7 855	18 100	4 526 (50 updates)
1.5M	5.2M	22 208	53 625	11 160 (50 updates)
2.4M	10.5M	45 065	107 195	23 200 (50 updates)
6M	21M	88 270	214 280	37 883 (50 updates)
18M	61.5M	186 144	532 505	66 009 (50 updates)

Table 1: Experimental results.

trees of transition relations, one for each *line*, representing the potentially legal evolutions of the line. This is important to notice because this is the reason why the method can hardly generalize to updates on whole subtrees: such an update would require to compute again ALL auxiliary data, *i.e.*, (i) the binary tree, (ii) the set of *lines* and (iii) trees of transition relations. Indeed, the method is applied only for updates on nodes: insertion or deletion of *leaves*, and renaming of one node. In [11, 45], 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 [11, 45]. For updates on leaves and for node renaming, their method may be more efficient than ours because the use of trees of transition relations is quite efficient for incremental word verification. However these updates are very special ones: for more general updates, as considered in our paper (involving whole subtrees), our method performs a minimum of tests without maintaining huge auxiliary structures.

Due to our use of unranked trees instead of the binary trees of [11, 45], each validation step on our method can be more expensive, but our XML tree is usually much shorter. Moreover, our work differs from that in [45] 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 with one single operation.

2. In [45], testing whether a word belongs to a language is done in an incremental way by storing an auxiliary structure. This optimization might be easily integrated in our algorithm for verifying child state word, 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 any auxiliary structures to store the result of a previous validation process.

4. Integrity constraint verification for keys and foreign keys is naturally integrated to our algorithm (Section 4).

4 Integrity constraint verification

In this section we present our constraint language, called CTK (for Context-Target-Key), and we show that constraints in CTK can be compiled to an attribute grammar. CTK is used to specify absolute and relative keys, together with foreign keys. As in [21], CTK uses path expressions built from a fragment of XPath. This XPath fragment includes (a) the empty path ϵ, (b) an element or attribute name, (c) a wildcard matching any single node name (_), (d) an arbitrary downward path ($//$) and (e) the concatenation of paths (P/Q where P and Q are paths, as defined by these rules). Indeed, this fragment is the same used by XML Schema to specify integrity constraints ([15]). Notice that a given path defines a language whose symbols are XML labels. Path expressions are, in fact, regular expressions over XML labels.

The following example illustrates how keys and foreign keys are specified. Next we define their syntax and semantics.

Example 4 The XML document represented by the tree T in Fig. 1 describes a collection of cooking recipes. Each collection maintains a categorized list of recipes, and a list of the top recipes. We want to validate this document wrt the following keys and foreign key, defined in CTK:

- K_1 : (*/*, (*./collection*, {*./category*}))
It indicates that, for the whole document, every collection must be uniquely identified by its category.
- K_2: (*/collection*, (*.//recipe*, {*./name*, *./author*}))
It means that, in the context of a collection, a recipe is uniquely identified by its name and author.
- K_3: (*//recipe*, (*./ingredient*, {*./name*}))
Similarly to K_2, it indicates that, in the context of a recipe, each ingredient is uniquely identified by its name.

- FK_4: $(/collection, (./top\_recipes/top\_recipe, \{./recipe\_name, ./author\_name\}) \subseteq (.//recipe, \{./name, ./author\}))$ where $(/collection, (.//recipe, \{./name, ./author\}))$ is the key K_2.

The foreign key constraint indicates that, in a collection, the name and the author of a top recipe must already exist as the name and the author of a recipe (in the same collection, but disregarding the order). □

Definition 4 - CTK key and foreign key syntax [21]: A key is represented by an expression $(P, (P', \{P^1, \dots, P^m\}))$ in which the path P is called the *context path*; P' is the *target path* and $P^1, \dots, P^m$ are the *key paths*.

A foreign key is represented by $(P, (P'_0, \{P^1_0, \dots, P^m_0\}) \subseteq (P', \{P^1, \dots, P^m\}))$ where $(P, (P', \{P^1, \dots, P^m\}))$ is a key K and $P^1_0, \dots, P^m_0$ are called *foreign key paths*. □

In this paper, key and foreign key specifications respect the following constraints: (a) Context and target paths should reach element nodes; (b) Key paths always exist and are unique and (c) Key (or foreign key) paths are required to end at a node associated to a value, *i.e.*, attribute nodes or elements having just one child of type *data*. Notice that our key specification corresponds to a special case of strong keys defined in [21] and thus it imposes the *uniqueness* of a key and *equality* of key values (*i.e.*, keys values cannot be null). This concept is similar to the concept of key in relational databases.

In the following definition we use the notion of tuple in a *named perspective* as described in [6]. Thus, tuples are functions that associate a non-null value to each component (name). The order of values appearing in the tuple is not important since each component value is associated to its name (in our case, the name of the tuple component is an XML label, *i.e.*, the name of the element or attribute whose value we want to consider). Thus, wrt the textual representation of an XML element, the definition below states that the order of elements (or attributes) is unimportant in defining equality.

Definition 5 - Semantics of CTK keys and foreign keys: An XML tree T satisfies a key $(P, (P', \{P^1, \dots, P^m\}))$ if for each context position p reached by following path P from the root, the following two conditions hold:

(i) For each target position p' reachable from p via P' there exists a unique position p^h from p', for each $P^h (1 \leq h \leq m)$, and

(ii) For any target positions p' and p'', reachable from p via P', whenever $\tau' = \tau''$ (where tuples[5] τ' and τ'' are built following $P^1 \dots P^m$ from p' and p'', respectively) then p' and p'' must be the same position.

Similarly, an XML tree T satisfies a foreign key $(P, (P'_0, \{P1_0, \dots, P^m_0\}) \subseteq (P', \{P^1, \dots, P^m\}))$ if:

[5]A tuple τ has the general format $\tau = [\tau(P^1) : v_1, \dots, \tau(P^m) : v_m]$. We use $\tau(P^h)$ to denote the name (label) of the component of τ corresponding to the node reached via P^h from a given target position.

(i) It satisfies its associated key $K = (P, (P', \{P1, \dots, P^m\}))$, and

(ii) For each target position p'_0 reachable from the context position p via P' there exists a unique position p^h from p', for each $P^h (1 \leq h \leq m)$, and

(iii) Each tuple $\tau_0 = [\tau_0(P^1_0) : v_1, \dots, \tau_0(P^m_0) : v_m]$, that was built following the paths $P/P'_0/P^1_0, \dots, P/P'_0/P^m_0$ is equivalent in value to a tuple $\tau = [\tau(P^1) : v_1, \dots, \tau(P^m) : v_m]$, built following paths $P/P'/P^1, \dots, P/P'/P^m$. In other words, for each $1 \leq h \leq m$, the τ_0-component name P^h_0 corresponds to the τ-component name P^h and their values are equal. □

Example 5 In Example 4, if we assume an inversion of the two rightmost children of position 0.3.0 of Fig. 1 (*author_name* on position 0.3.0.1 and *recipe_name* on position 0.3.0.2) then we obtain tuples [*recipe_name*: *Mushroom Soup*, *author_name*: *M. Smith*] and [*name*: *Mushroom Soup*, *author*: *M. Smith*] which are equivalent in value. This is because foreign key specification relates *recipe_name* to *name* and *author_name* to *author*. Thus, according to Definition 5, the foreign key is satisfied. This remark is also valid when comparing key values. □

In order to perform the validation of key constraints, we represent the paths in key definitions by finite state automata: for a context path P, we have the automaton $M = \langle \Theta, \Sigma, \delta, e, F \rangle$. This automaton will be referred to as the context automaton. It is defined to recognize the language generated by the path (regular expression) P.

Similar automata are defined for target, key and foreign key paths: For a target path P', its corresponding target automaton is defined as $M' = \langle \Theta', \Sigma, \delta', e', F' \rangle$; and for key or foreign key paths $P^1, \dots, P^m$, their key or foreign key automata are $M'' = \langle \Theta'', \Sigma, \delta'', e'', F'' \rangle$.

We denote by $M.e$ the current state e of the finite state automaton M. $M.e$ is the *configuration* of the automaton, representing a snapshot of it during its run. We illustrate the above definitions with an example.

Example 6 Fig. 8 illustrates finite state automata that correspond to the (context, target and key) paths in K_1, K_2, K_3 and FK_4 of Example 4. Given the XML tree of Fig. 1, those finite state automata are used in the tree traversal to perform constraints validation. □

4.1 Key and foreign key validation: attribute grammar approach

We consider a context-free grammar G = (V_N, V_T, P, B) where V_N is the set of non-terminals, V_T is the set of terminals, P is the list of productions, and B is the start symbol. In order to add "extra" information to a non-terminal symbol we can attach a set of attributes to it. An attribute can represent anything: a string, a number, a type, a memory location or whatever [8]. An *attribute grammar* is G augmented by semantic rules, which are declarative specifications describing how the attached attributes are computed.

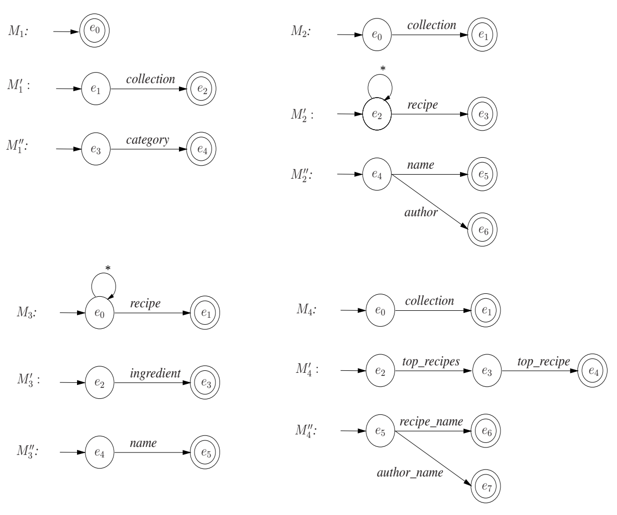

Figure 8: Context, target and key automata corresponding to keys K_1, K_2, K_3 and to the foreign key FK_4.

Production	Attributes
$R \rightarrow \alpha_1 \dots \alpha_m$	$R.conf$:= { $M.q_0$ }
	/* *Inherited Attributes* */
	for each α_i $(1 \leq i \leq m)$ do
	$\alpha_i.conf$:= { $M.q'$ $\mid \delta_M\ (q_0, \alpha_i) = q'$ }
	if $(q_0 \in F_M)$ then
	$\alpha_i.conf$:= $\alpha_i.conf \cup$
	{ $M'.q_1'$ $\mid \delta_{M'}(q_0', \alpha_i) = q_1'$ }
	end for
	/* *Synthesized Attributes* */
	if $(q_0 \in F_M)$ then
	$R.c^K := \langle \forall w, z\!: w \neq z \Rightarrow \alpha_w.t \cap \alpha_z.t = \emptyset \rangle$
	if $(\exists\ FK \subseteq K) \wedge (q_0 \in F_{M_{FK}}) \wedge$
	$(R.t^{FK} \subseteq R.t^K)$ then
	$R.c^{FK} := \langle true \rangle$
	if $(q_0 \notin F_M)$ then
	$R.c := \langle \forall x_w : \alpha_w.c = \langle x_w \rangle \Rightarrow \bigwedge_{w=1}^{m} x_w \rangle$

Table 2: Attribute Grammar for Keys and Foreign Keys (Root block).

Thus, the value of an attribute at a parse tree node is defined by a semantic rule associated to the production used at that node [8].

We recall that a schema can be seen as an extended context free grammar G (regular tree grammar)[37]. Thus, in the context of integrity constraint verification one may augment G, whose production rules are those defining the schema, by semantics rules [44], using attributes which represent information about integrity constraints. In our case, as we assume that integrity constraint validation is independent from schema verification, we consider G as a simpler grammar just describing any XML tree. We write $A \rightarrow \alpha_1 \dots \alpha_m$ to indicate that the semantic rule applies to the node labeled a and to its children (labeled α_i). The semantic rules provide a mechanism for annotating the nodes of a tree with attributes, which can work either bottom-up for *synthesized attributes* or top-down for *inherited attributes*.

Tables 2, 3 and 4 present the attribute grammar for keys and foreign keys. In these tables the definition of semantic rules for a key K (or a foreign key FK) is given according to the kind of production rules. Indeed, we classify our production rules into three kinds, according to the XML node over which they can be applied:

1: Rules applied at the root node (root block in Table 2).

2: Rules applied at the leaves (or data) nodes (data block in Table 4).

3: General rules, applied at all other nodes (general block in Table 3).

We use the top-down direction, *i.e.*, *inherited attributes*, in order to determine the role of each node wrt keys and foreign keys, defined according to language CTK. We define just one inherited attribute, called *conf*. For each node, *conf* is computed by executing the finite state automata that recognize the paths in the definition of K (or FK).

In this way, Table 2 defines, for each key or foreign key, how the attributes *conf* are computed for a rule in which the left-hand side is a symbol that represents the root node. To this end, we firstly assign to the root node the set of values $\{M.q_0\}$ where $M.q_0$ is the initial configuration of the context automaton M. Then, we compute the value of *conf* for each child of the root (by executing M), without forgetting to verify whether we need to change from the context to the target automaton.

In Table 3 the computation of *conf* is similar to the one performed for the root's children in Table 2. Notice that to compute the value of *conf* for a node α, we start by considering each configuration $\overline{M}.q$ in the set of values associated to the attribute *conf* of its parent A. We should also verify if it is necessary to change from one automaton to another.

Example 7 - Consider the XML document, key K_2 and foreign key FK_4 of Example 6 with their corresponding finite state automata (respectively M_2, M_2', M_2'' and M_4, M_4', M_4''). We have one inherited attribute *conf* for each key or foreign key, as illustrated in Fig. 9. The attributes *conf*^{K_2} and *conf*^{FK_4} are computed top-down by the execution of their finite state automata. For instance, at the root node, we assign to *conf*^{K_2} and *conf*^{FK_4} their corresponding initial configuration, respectively the sets $\{M_2.e_0\}$ and $\{M_4.e_0\}$. In order to compute *conf*^{K_2} for node *collection* we execute a first transition in M_2 using the label *collection* as input. The result is the set $\{M_2.e_1\}$. Similarly, the computation of *conf*^{FK_4} for node *collection* results in $\{M_4.e_1\}$. □

We use the bottom-up direction, *i.e.*, *synthesized attributes*, to carry the values that are part of a key or foreign key up to their context node. At this level, we verify if the integrity constraints are respected. For each key (or foreign key) definition K (according to language CTK), we use three attributes, called c, t and k, for (respectively) *context values*, *target values* and *key values*. At each node, these attributes receive values depending on the role of the node for K (*i.e.*, the value of attribute *conf*), and depending also on values of c, t and k from children nodes.

In this context, Table 4 shows that an attribute k obtains the data value of a leaf whose parent is a key node for some K. Then, Table 3 defines the values of the synthesized attributes concerning a node p (labeled A) in the following way:

Production	**Attributes**
$A \rightarrow \alpha_1 \ldots \alpha_m$	/* *Inherited Attributes* */
	for each α_i $(1 \leq i \leq m)$ do
	for each $\overline{M}.q \in A.conf$ do
	$\alpha_i.conf := \{\overline{M}.q' \mid \delta_{\overline{M}}(q, \alpha_i) = q'\}$
	if $(\overline{M} = M) \wedge (q \in F_M)$ then $\alpha_i.conf := \alpha_i.conf \cup \{M'.q'_1 \mid \delta_{M'}(q'_0, \alpha_i) = q'_1\}$
	if $(\overline{M} = M') \wedge (q \in F_{M'})$ then $\alpha_i.conf := \alpha_i.conf \cup \{M''.q''_1 \mid \delta_{M''}(q''_0, \alpha_i) = q''_1\}$
	end for
	end for
	/* *Synthesized Attributes* */
	for each configuration $\overline{M}.q$ in $A.conf$ do
	if $(\overline{M} = M'') \wedge (q \notin F_{M''})$ then $A.k := < \alpha_1.k \ \ldots \ \alpha_m.k >$
	if $(\overline{M} = M') \wedge (q \in F_{M'}) \wedge (\lvert< \alpha_1.k \ \ldots \ \alpha_m.k >\rvert = n)$ then $A.t := A.t \cup \{ < \alpha_1.k \ldots \alpha_m.k > \}$
	if $(\overline{M} = M') \wedge (q \notin F_{M'}) \wedge (\forall w, z : w \neq z \Rightarrow \alpha_w.t \cap \alpha_z.t = \emptyset)$ then $A.t := \bigcup_{w=1}^{m} \alpha_w.t$
	if $(\overline{M} = M) \wedge (q \in F_M)$ then $A.c^K := \langle \forall w, z : w \neq z \Rightarrow \alpha_w.t \cap \alpha_z.t = \emptyset \rangle$
	if $(\exists\, FK \subseteq K) \wedge (\overline{M} = M^{FK}) \wedge (q \in F_{M^{FK}}) \wedge (A.t^{FK} \subseteq A.t^K)$ then $A.c^{FK} := \langle true \rangle$
	if $(\overline{M} = M) \wedge (q \notin F_M)$ then $A.c := \langle \forall x_w : \alpha_w.c = \langle x_w \rangle \Rightarrow \bigwedge_{w=1}^{m} x_w \rangle$
	end for

Table 3: Attribute Grammar for Keys and Foreign Keys (General block).

Production	**Attributes**
$A \rightarrow data$	/* *Synthesized Attributes* */
	for each configuration $\overline{M}.q$ in $A.conf$ do
	if $(\overline{M} = M'') \wedge (q \in F_{M''})$
	then $A.k := < value(data) >$
	end for

Table 4: Attribute Grammar for Keys and Foreign Keys (Data block).

```
<!DOCTYPE keyTree[
<!ELEMENT keyTree (context*)>
<!ATTLIST keyTree nameKey CDATA #REQUIRED>
<!ELEMENT context (target+)>
<!ATTLIST context pos CDATA #REQUIRED>
<!ELEMENT target (key+)>
<!ATTLIST target pos CDATA #REQUIRED
                 refCount CDATA #REQUIRED>
<!ELEMENT key #PCDATA>]
```

Figure 11: DTD specifying the structure *keyTree*

1. If p is in a key path, then its attribute k is the tuple composed by the key values (those associated to the attribute k of each child of p).

2. If p is a target node, then its attribute t is a set containing the tuple composed by the key values carried up from p's children. This tuple is computed from the values of attributes k, as explained in item 1. Notice that the assignment of a value to the attribute t depends on the verification of the key size (*i.e.*, the size of the key tuple must respect the key definition).

3. If p is in a target path and if all the tuple values carried up by p's children are distinct, then the attribute t for node p is assigned with the union of the sets containing these tuples.

4. If p is a context node for a key and the tuple values carried up by p's children are distinct, then the attribute c (wrt the key K) is assigned with a tuple containing the value *true*. Otherwise the tuple contains the value *false*.

5. If p is a context node for a foreign key and the tuple values carried up by p's children are also key values, then the attribute c (wrt the foreign key FK) is assigned with a tuple containing the value *true*. Otherwise the tuple contains *false*.

6. If p is in a context path, then the attribute c is assigned with a tuple containing the conjunction value of c's values obtained from p's children.

Finally, Table 2 shows how to compute synthesized attributes for the root, distinguishing whether it is a context (for a key or a foreign key) or not.

From the above explanation, we see that the values of attributes c are computed from those of attributes t which are, in turn, built from the values of the attributes k.

Example 8 - As in Example 7 we show in Fig. 10 how the synthesized attributes are computed for K_2 and FK_4 for nodes in the key, target and context paths:

Key path: The data values for K_2, obtained from nodes $name$ and $author$, are collected in each k_2. For FK_4, *recipe_name* and *author_name* are the foreign key nodes and their data values are collected in attributes k_4.

Target path: For K_2, the key values are also concatenated into a tuple and kept in a singleton set when reaching target node *recipe*. At target node *top_recipe*, the attribute t_4 receives a singleton set containing the tuple obtained by the concatenation of the key values for FK_4. Node *top_recipes* is in the target path and its attribute t_4 groups all the target tuples coming from target nodes.

Context path: At the context node *collection*, as all the tuples collected in the various t_2 are all distinct, then the attribute c^{K_2} is set to $true$. Still at node *collection*, the set of tuples coming from t_4 is compared to those coming from the various t_2. As all the tuples in t_4 are contained in the set formed by the various t_2, then the attribute c^{FK_4} is set to $true$. It means that for the concerned *collection* node, the foreign key FK_4 is valid.

At the root node, the attributes c_2 and c_4 are set to $true$, indicating that the document respects K_2 and FK_4. In order to show neatly the attribute values that are synthesized, those attributes (c, t or k) for K_2 or FK_4 that are not concerned in the position are hidden. □

At the same time that we compute the synthesized attributes for a key, we build its corresponding *keyTree*. The

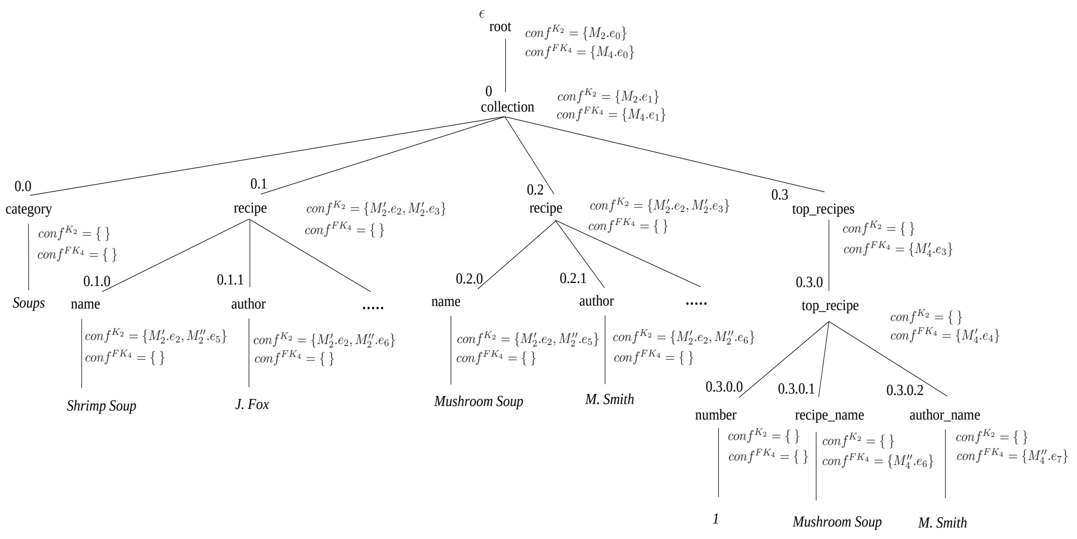

Figure 9: Inherited Attribute *conf* for K_2 and FK_4.

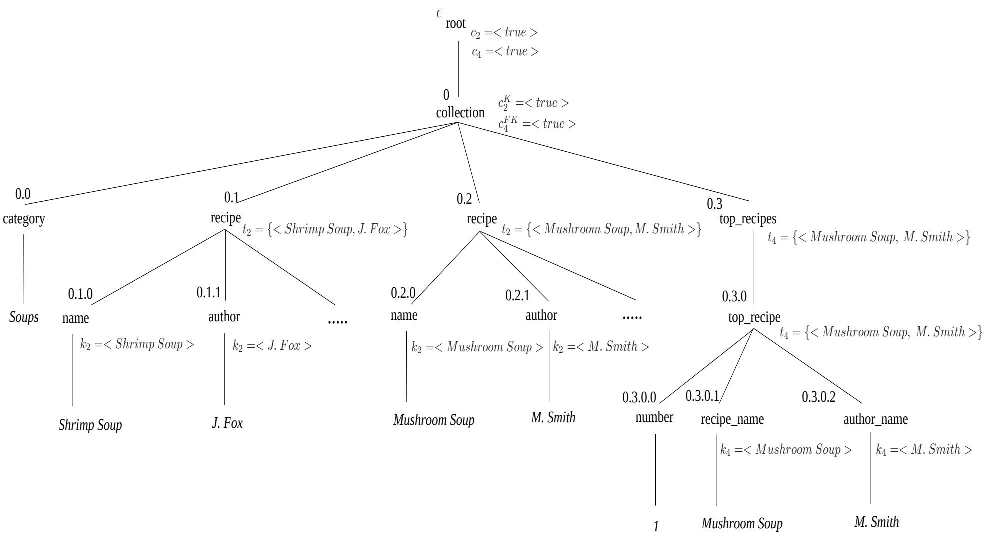

Figure 10: Synthesized Attributes for K_2 and FK_4.

*keyTree*s are structures storing the position of each context and target nodes together with the key values associated to each key node. Fig. 11 describes this structure index using the notation of DTDs. For each key constraint K that should be respected by the XML document, we keep its *keyTree*$_K$. Also, for each key, a reference counter *refCount* is used to store how many times the key is referenced by a foreign key. The *keyTree*s are kept to facilitate validation of keys and foreign keys in update operations, as the acceptance of an update operation wrt integrity constraints relies on information about key values. Fig. 12 shows *keyTree*$_{K_2}$ that stores the key values and information for K_2.

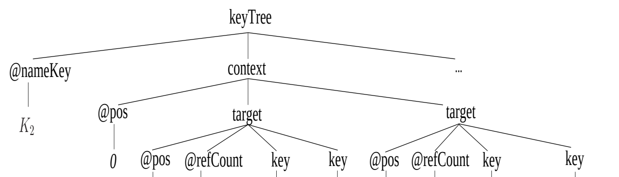

Figure 12: *KeyTree*$_{K_2}$ built over the XML document of Fig. 1.

4.2 Incremental integrity constraint verification

We consider a collection of keys K_j $(1 \leq j \leq m)$ and foreign keys FK_j $((m+1) \leq j \leq n)$ constraints that should be respected by a subtree T' being inserted, replaced or deleted. The execution of our key and foreign key constraints validator over a tree T' gives a tuple $\langle\langle l_1,\ldots,l_n\rangle$, $\langle \mathit{keyTree}_{K_1}[T',\epsilon],\ldots,\mathit{keyTree}_{K_m}[T',\epsilon]\rangle\rangle$ where:

- $\langle l_1,\ldots,l_n\rangle$ is a n-tuple of tuples $\langle c,t,k\rangle$. Each tuple $\langle c,t,k\rangle$ represents the synthesized attributes computed for one key (or foreign key) at the root position ϵ of T'.
- $\langle \mathit{keyTree}_{K_1}[T',\epsilon],\ldots,\ \mathit{keyTree}_{K_m}[T',\epsilon]\rangle$ is a m-tuple containing one *keyTree* for each key.

Notice that the n-tuple that represents the synthesized attributes has two distinct parts: tuples $l_1,\ldots,l_m$ represent keys and tuples $l_{m+1},\ldots,l_n$ represent foreign keys.

We introduce now the notion of local validity for keys and foreign keys. When performing an insertion we want to ensure that the new subtree T' has no internal validity problems (as, for instance, duplicated values for a key K).

Definition 6 - Local Validity: Let T' be an XML tree. Let K_j $(1 \leq j \leq m)$ be a collection of keys and FK_j $((m+1) \leq j \leq n)$ be a collection of foreign keys, both defined according to language CTK. The tree T' is locally valid if the result of the validation gives a tuple $\langle\langle l_1,\ldots,l_n\rangle, \langle \mathit{keyTree}_{K_1}[T',\epsilon],\ldots,\mathit{keyTree}_{K_m}[T',\epsilon]\rangle\rangle$ respecting the conditions below.

For each tuple l_j $(1 \leq j \leq n)$ we have:

(i) If the root of T' is a target position for K_j (or FK_j) or a position in the target path then all tuples in the set specified by the attribute t_j, which is in tuple l_j, has length m_j (*i.e.*, its length equals the number of elements composing a key (or foreign key) tuple).

(ii) If the root of T' is a context position for K_j (or FK_j), or a position in the context path, then the attribute c_j in l_j is a tuple containing the value *true*. □

Given a tree T and a sequence of updates over T, the incremental validation problem wrt integrity constraints consists in checking whether the updated tree does not violate any constraints, by visiting only the part of T involved by the updates. We propose an algorithm, similar to Algorithm 1, to perform the incremental validation of an XML tree T.

Example 9 Consider the XML tree of Fig. 3 where update positions are marked. As in Section 3, updates are treated in the document order. Now, we remark that, in the incremental validation wrt integrity constraints the following steps are performed:

1. When the open tag <d> (position 0.1) is reached, the deletion operation is taken into account and a search of key and foreign key values is triggered in order to verify whether there are key or foreign key values in the subtree rooted at the deletion position. If such kind of values are found, we use the corresponding *keyTree* to test whether the deletion does not imply any violation of integrity constraints. If a violation is detected we mark position 0.1 (in *keyTree*) and the document is considered to be temporarily invalid. Only after analyzing late updates we can decide whether this deletion operation can be performed. A late update can indeed restore document validity (*e.g.*, by removing foreign key references from a key involved in our delete operation). In this case, our deletion can be performed.

2. When the open tag <a> at position 1 is reached, we use the corresponding *keyTree* to test whether the insertion does not imply any violation of integrity constraints. We consider the insertion as performed (similarly to Section 3). If the insertion represents a violation of an integrity constraint, then there is a subtree where key values are duplicated wrt those being inserted. We mark this subtree in *keyTree* and proceed as in the above item. Notice that, similarly to Section 3, we can skip nodes below position 2.

3. The replace operation at position 3 combines the effects of a deletion and an insertion.

4. At the end of the tree traversal a final validity test is executed. It consists on verifying whether the resulting *keyTree* does not contain any violation label (meaning that a postponed violation correction was not performed).

The implementation of this approach is done by the following algorithm. Notice that it uses the *UpdateTable* to obtain the update positions. When an update position is reached, different tests are performed, according to the update operation.

Algorithm 2 - Incremental Validation of Keys in Multiple Updates

Input:

(i) *doc*: An XML document.

(ii) $UpdateTable$: A relation that contains updates to be performed on *doc*. Each tuple has the general form $\langle pos, op, T_{pos}, \Phi\rangle$, similar to Algorithm 1.

(iii) $KeysFSA$: Set of finite state automata describing the paths that appear in the keys and foreign keys.

(iv) $KeyTree$: Structure that contains the key values resulting from the last validation performed on *doc*.

Output:

If *doc* remains valid after all operations in $UpdateTable$ the algorithm returns the Boolean value $true$, otherwise $false$.

Local Variables:

(i) $CONF$: structure storing the inherited attributes.

(ii) $SYNT$: structure storing the synthesized attributes.

(iii) $keyTreeTmp$: copy of the initial $keyTree$.

```
(1)  $keyTreeTmp := KeyTree$
(2)  $CONF_\epsilon := InitializeInhAttributes(KeysFSA)$
(3)  foreach event v in doc do
(4)     switch (v) do
(5)     case start of element a at position p
(6)       Compute $CONF_p$ using $KeysFSA$
(7)       foreach $u = (p, \mathsf{insert}, T_p, \Phi) \in UpdateTable$ do
(8)         if $(\neg \mathsf{insert}(doc, p, T_p, keyTreeTmp))$
(9)         then report "invalid" and halt
(10)        if $\nexists u' = (p', op', T', \Phi') \in UpdateTable$
              such that $p \prec p'$
(11)        then $skipSubTree(doc, a, p)$;
(12)    case end of element a at position p
(13)      Compute $SYNT_p$ using $CONF_p$
(14)      if $\exists\, u = (p, \mathsf{delete}, \Phi) \in UpdateTable$
(15)      then if $(\neg \mathsf{delete}(doc, p, SYNT_p, keyTreeTmp))$
(16)           then report "invalid" and halt
(17)      if $\exists\, u = (p, \mathsf{replace}, T_p, \Phi) \in UpdateTable$
(18)      then if $\neg \mathsf{replace}(doc, p, SYNT_p, T_p, keyTreeTmp)$
(19)           then report "invalid" and halt
(20)    case value
(21)      $str := value(p)$
(22)      Compute $SYNT_p$ using $CONF_p$ and $str$
(23)      if $(\neg valid(keyTreeTmp))$
             then report "invalid" and halt
```

```
(24) return $true$ □
```

Algorithm 2 describes the incremental validation of key and foreign key constraints while reading the XML tree in the document order. Firstly, we make a work version of the initial valid *keyTree*. This work version will be changed as we process a sequence of update operations. At the end of this sequence we check whether the obtained *keyTreeTmp* is valid and, only in this case, it will replace the original *keyTree*.

The algorithm uses two structures to store the attribute values, namely *CONF* and *SYNT*. At each position p, the structure $CONF_p$ keeps the roles of p wrt the keys and foreign keys being verified. Indeed, $CONF_p$ contains one inherited attribute *conf* (as defined in Tables 2 and 3) for each key and foreign key at position p. The structure $SYNT_p$ contains a tuple formed by the synthesized attributes k, t, and c (see Tables 2, 3 and 4) for each key and foreign key at position p.

When reaching an open tag at position p, the inherited values to be stored in $CONF_p$ are computed and all requested insertions at this position are performed according to our insertion method (Algorithm 3). On the other hand, when reaching a close tag in position p, we compute the synthesized values to be kept in $SYNT_p$ and we perform the requested deletions and replacements. Recall that the computation of synthesized values depends not only on $CONF_p$ but also on the result of each $SYNT_{p'}$ (where p' is a child of p). Notice that $SYNT_p$ is necessary in delete and replace operations, since for each key and foreign key, we need to remove key and foreign key values from the corresponding *keyTreeTmp* (see Algorithm 4).

Algorithm 3 defines the operation $\mathsf{insert}(T, p, T', keyTreeTmp)$ to include a new subtree T' in an XML tree T at position p. It first computes the tuple τ which is the result of the local validation for T'. Recall that τ is composed by tuples containing the synthesized attributes and by the (partial) *keyTree*s associated to T'.

An insert operation is accepted or rejected. If an insertion is accepted then the key or foreign key values found in T' are inserted in the corresponding *keyTreeTmp*. Notice that an insertion is temporarily accepted in two cases, namely:

• If the key tuple value corresponding to a foreign key instance being inserted does not exist. In this case, we add the new key tuple value to *keyTreeTmp* and we mark it with a null value at attribute pos.

• If the insertion of a key instance generates duplicated tuple values for a key. In this case, we mark the original tuple in *keyTreeTmp* as a duplicate value.

In both cases, we postpone the final decision of rejecting or accepting the insertion. We just mark the involved tuples in *keyTreeTmp*, keeping the document temporarily invalid. We should notice the difference between duplicating key values and key (or foreign key) components. Indeed, if the subtree to be inserted requests a duplication of key (or foreign key) component nodes, then it is rejected since we assume that nodes composing a key (or a foreign key) must be unique. For instance, if we consider the key K_1 of Example 6, then the insertion at position 0.0 of a new category under a collection (position 0) is rejected. As *category* is the key node for K_1, the subtree rooted at *collection* cannot have more than one *category* child.

Algorithm 3 - The insert operation: $\mathsf{insert}(T, p, T', keyTreeTmp)$

```
(1)  if (($\tau$ := LocalValidation($T, p, T'$)) = 'invalid')
       then return false
(2)  for each tuple $l_i = \langle k_i, t_i, c_i\rangle$ $(1 \le i \le m)$ in $\tau$,
       corresponding to $K_i$ do
(3)    if ($k_i \neq <>$) then return false
(4)    if ($t_i \neq \emptyset$) then
         Find the context position $p'$ (above $p$) for $K_i$
(5)    for each tuple $v \in t_i$ do
(6)      if $\exists u = v$ in $keyTreeTmp_{K_i}[T, p']$ then
(7)        if $u$ is associated to an attribute $pos \neq null$
(8)          then Mark $u$ in $keyTreeTmp_{K_i}[T, p']$
               with dup = "yes"
(9)      Add $keyTree_{K_i}[T', p]$ to $keyTreeTmp_{K_i}[T, p']$
(10)   if ($c_i = \langle true\rangle$) then
(11)     Add $keyTree_{K_i}[T', p]$ to $keyTreeTmp_{K_i}[T, \epsilon]$
(12) for each tuple $l_j = \langle k_j, t_j, c_j\rangle$ $(m + 1 \le j \le n)$,
       corresponding to $FK_j$ do
(13)   if ($k_j \neq <>$) then return false
(14)   if ($t_j \neq \emptyset$) then
(15)     Find the context position $p'$ (above $p$) for $FK_j$
(16)     for each tuple $v \in t_j$ do
(17)       if $\exists u = v$ in $keyTreeTmp_{K_i}[T, p']$ then
(18)         increment refCount of tuple $u$ by 1
(19)       else build a subtree $\beta$ which corresponds to
             tuple $u$ as follows:
(20)           ⋆ Attribute $pos$ is null
(21)           ⋆ Attribute $refCount$ is 1
(22)           ⋆ Key values compose the tuple $u$
(23)         add the subtree $\beta$ in $keyTreeTmp_{K_i}[T, p']$
(24)         as rightmost child of context node $p'$.  □
```

The insertion of a new subtree rooted at position p is possible if the following tests succeed:

1. The subtree being inserted is locally valid.
2. For each key K_i $(1 \le i \le m)$:

(a) If p is a position in the key path (*i.e.*, a position below the target node) then the update operation implies the duplication of key nodes. In this case, the insertion is rejected.

(b) If p is a position in the target path, then we insert the new key values in $keyTreeTmp_{K_i}$ under the corresponding context. The key value may already exist in $keyTreeTmp_{K_i}$. It is necessary to test if it exists as a duplicate or as an "incomplete" insertion triggered by the previous insertion of a foreign key. The test consists in verifying whether attribute pos is null.

If there are duplicated key values wrt the one being inserted, then the duplication is annotated by adding *dup* = *"yes"* in $keyTreeTmp_{K_i}$. If the insertion corresponds to an incomplete previous insertion, it is completed by changing the value of attribute pos.

(c) If p is a position in the context path, then the insertion is accepted since tree T' is locally valid wrt key and foreign keys (item (1)) .

3. For each foreign key FK_j $(m+1 \leq j \leq n)$:

(a) If p is a position in the foreign key path, then we proceed as in item 2(a).

(b) If p is a position in the target path, then we are inserting new foreign keys wrt a key K_i. We increment the *refCount* (in *keyTreeTmp*$_{K_i}$) of each K_i tuple corresponding to a foreign key tuple being inserted. If there is not a corresponding tuple in *keyTreeTmp*$_{K_i}$ then a subtree containing the key values which correspond to the inserted foreign key is added to *keyTreeTmp*$_{K_i}$ as the rightmost child of the concerned context. For this subtree, attributes $pos = null$ and $refCount = 1$.

Algorithm 4 defines the update operation delete$(T, p, \sigma, keyTreeTmp)$ where p is the position to be removed from the XML tree T, and σ is the tuple resulting from the local validation of the subtree T' originally rooted at p.

If a deletion is accepted, then the key and foreign key values are removed from the corresponding *keyTreeTmp*. Notice that a deletion is temporarily accepted if it concerns a key tuple whose *refCount* is not 0. Indeed, in this case, the deletion is finally accepted only if all the foreign keys referencing this tuple are also removed from T. We postpone the final decision of accepting or rejecting the deletion to the end of the update sequence. We consider that key (or foreign key) nodes cannot be deleted.

Algorithm 4 - The delete operation: delete$(T, p, \sigma,$ ***keyTreeTmp***$)$

```
(1)  for each tuple $l_i = \langle k_i, t_i, c_i \rangle$ $(1 \leq i \leq m) \in \sigma$,
       corresponding to $K_i$ at position $p$ do
(2)    if ($k_i \neq <>$) then return false
(3)    if ($t_i \neq \emptyset$) then
(4)      Find the context position $p'$ (above $p$) for $K_i$.
(5)      for each tuple $v \in t_i$ do
(6)        Find the tuple $u = v$ in $keyTreeTmp_{K_i}[T, p']$
(7)        if refCount associated to tuple $u$ is 0
(8)        then remove $u$ from $keyTreeTmp_{K_i}[T, p']$
(9)        else mark tuple $u$ in $keyTreeTmp_{K_i}[T, p']$
             with del = "yes"
(10)   if ($c_i = \langle true \rangle$) then
(11)     for each context pos $p'$ under or equal $p$ do
(12)     remove $keyTreeTmp_{K_i}[T,p']$ from $keyTreeTmp_{K_i}[T,p]$
(13)     for each tuple $l_j = \langle k_j, t_j, c_j \rangle$ $(m+1 \leq j \leq n)$
           in $\sigma$, corresponding to $FK_j$ do
(14)       if ($k_j \neq <>$) then return false
(15)       if ($t_j \neq \emptyset$) then
(16)         Find the context position $p'$ (above $p$) for $FK_j$
(17)         for each tuple $v \in t_j$ do
(18)           Obtain the tuple $u$ that is referenced by $v$ in
                 $keyTreeTmp_{K_i}[T, p']$;
(19)           Decrement refCount associated to $u$ by 1;
(20)           if (refCount = 0) and (del = yes) in $u$
(21)           then remove $u$ from $keyTreeTmp_{K_i}[T, p']$    □
```

To remove a subtree rooted at position p we execute the following tests for keys and foreign keys:

1. For each key K_i $(1 \leq i \leq m)$:

(a) If p is a position in the key path, then the deletion is rejected.

(b) If p is a position in the target path and if each target to be deleted is not referenced by any foreign key, then the deletion is accepted. Otherwise, the corresponding target in *keyTreeTmp*$_{K_i}$ is labeled to be deleted subsequently.

(c) If p is a position in the context path, then the deletion is possible for all contexts, since we are removing key values and also the foreign key values that reference them.

2. For each foreign key FK_j $(1 \leq j \leq n)$:

(a) If p is a position in the foreign key path, then the deletion is rejected.

(b) If p is a position in a target path, then the deletion is accepted and we decrease the corresponding *refCount*s in *keyTreeTmp*$_{K_i}$. If *refCount* turns to 0 and the target tuple in *keyTreeTmp*$_{K_i}$ was earlier marked to be deleted, then this tuple is removed.

The replace operation combines the deletion and the insertion but it is not equivalent to the update sequence (insert$(T, p, T', \tau,$ *keyTreeTmp*); delete$(T,$ $p,$ $\sigma,$ *keyTreeTmp*)) since it allows the replacement of key (or foreign key) nodes. For instance, the replace operation allows the substitution of a recipe author, even if *author* is part of key K_2 of Example 6 (recall that the delete operation does not allow this removal). If there is a replace operation to be performed at position 0.2.1 (to change the value *M. Smith* to *L. Greene*), it is accepted if key K_2 is still respected after the update.

Example 10 We suppose the XML tree of Fig. 1 and an update sequence composed by: (1) An insertion of a new *recipe* at position 0.1; (2) A deletion at position 0.2; (3) A deletion at position 0.3.0.

The initial XML tree is valid, and we keep a work version of its *keyTree* in the new structure *keyTreeTmp*, used here to verify the validity of the updates wrt key and foreign keys. In this example we consider keys K_2 and FK_4 defined in Example 6. The update sequence is examined while reading the XML tree, as shown in Algorithm 2. In this way, the tests are performed in the following order:

1. When we reach the open tag of element *recipe* at position 0.1, we find that there is an insertion to be performed. The new subtree to be inserted is a new recipe that contains instructions and ingredients for preparing a broccoli soup. To check if this insertion is valid, the insert operation in Algorithm 3 is performed. The new key values are inserted in *keyTreeTmp*$_{K_2}$ under the collection of soups (context at position 0, prefix of position 0.1), as shown in Fig. 13. Notice that attributes *pos* are not updated yet.
2. When the close tag of element *recipe* at position 0.2 is reached, there is a deletion to be performed (the removal of the mushroom soup recipe). To verify if this deletion is accepted, the delete operation in Algorithm 4 is performed. It also concerns key K_2, and the key values under position 0.2 must be removed from *keyTreeTmp*$_{K_2}$ under the collection of soups (context at position 0). Fig. 14 shows that this deletion is postponed, since the *refCount* associated to mushroom soup in *keyTreeTmp*$_{K_2}$ is 1.

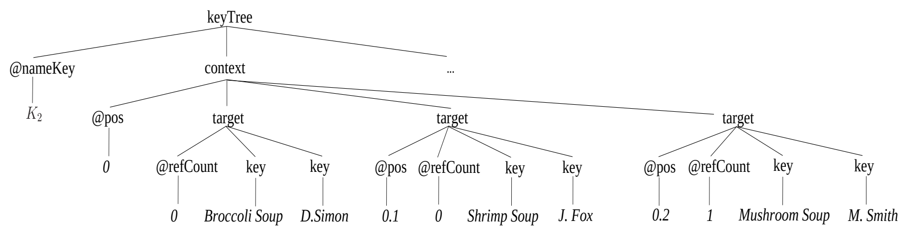

Figure 13: $keyTreeTmp_{K_2}$ after insertion at position 0.1.

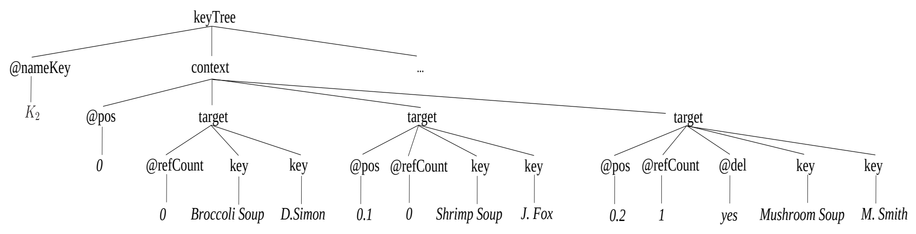

Figure 14: $keyTreeTmp_{K_2}$ after deletion at position 0.2.

3. When the close tag of element *top_recipe* at position 0.3.0 is reached, the deletion of the top recipe with *number = 1* (mushroom soup) should be done. Since this deletion concerns FK_4 (that references K_2), the foreign key values under position 0.3.0 are removed from *keyTreeTmp*$_{K_2}$ in the collection of soups (context at position 0). This is done by decreasing of 1 the corresponding *refCount* for mushroom soup. At this point, as *refCount* becomes 0 and the mushroom soup tuple was already marked to be deleted, then the deletion (earlier postponed) is accepted, as illustrated in Fig. 15. At the end of the update sequence, we traverse *keyTreeTmp*$_{K_2}$ in order to: (a) update attributes *pos* and (b) verify the existence of violation marks.

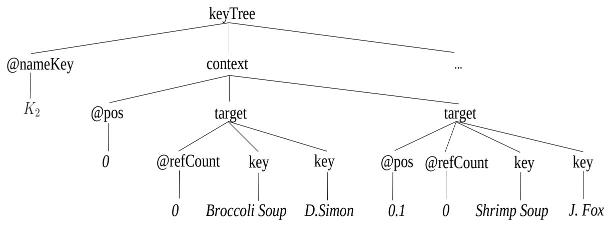

Figure 15: $keyTreeTmp_{K_2}$ after deletion at position $0.3.0$.

Notice that we consider that the user provides our algorithm with a *set* of updates. The processing of these updates is performed in the document order, *i.e.*, in the order in which the nodes of the XML tree are visited[6]. This ordering is natural to our algorithm and is the most efficient one, since we will perform only one pass over the tree.

The semantics of our method is independent of the order in which the update operations are processed. Our method has the following properties:

[6]This is why we present *UpdateTable* as a sequence built in a preprocessing phase from a given set of updates and respecting properties stated in Section 2.

• The operations are performed only whenever possible. In the presence of errors in the verification of constraints, our algorithm will not perform the updates. (this condition is similar to that for transactions, in the context of relational databases).

• The updates are performed if and only if there exists an ordering for them, whose final result preserves the validity of the document wrt the verified constraints.

Notice that the second condition above is verified since our method is as general as possible. It has the same effect as the ordering which enables us to accept most update sequences, that is: (i) remove foreign keys; (ii) remove primary keys; (iii) insert primary keys; (iv) insert foreign keys.

This condition means that our method is as general as possible, accepting the largest class of possible sets of updates.

4.3 Complexity and experimental results

The validation method proposed in Section 4.1 requires only one pass on the XML document. As in [24], its running time is linear in the size of the XML document. This complexity is not affected by the shape of the XML document, but it can be affected by the size of the *keyTree*. Indeed the time and space complexities of our method are based on the size of the *keyTree* which indicates the number of key instances existing in the document (*i.e.*, the number of target nodes). When the *keyTree* is big, the following steps are time consuming: the set inclusion test done (once) at the root level (Table 2) and list comparisons at the context level, which are performed once for each key. The space complexity for our method corresponds to the size of *keyTree*. Each *keyTree* is a subset of the document, containing only the necessary information to verify the validity wrt a given integrity constraint.

In Algorithm 2, each update position p activates a *validation step* (*i.e.*, Algorithm 3 or Algorithm 4) which depends on the kind of the required update. To find if key and foreign key values are involved in the update, algorithms consult the list τ containing the result of the local validation of the subtree T_p (being inserted of deleted). The list τ is built in time $O(|T_p|)$. We recall that insertions and deletions trigger a subtree traversal to find the key and foreign key values.

Algorithm 3 and 4 go through the resulting list τ in order to detect non empty lists (*i.e.*, those that correspond to the key (or foreign) values being inserted or deleted). Then the algorithms compare these values to those in the *keyTree*. To this end, Algorithm 3 and 4 visit key nodes in the *keyTree* corresponding to the context under which the insertion or deletion is required. Let n be the number of key and foreign key constraints and let n_{target} be the maximum number of target nodes (*i.e.*, key tuples). This operation runs in time $O(n.n_{target}.c_2)$ where c_2 is the maximum number of components of a composed key (*i.e.*, the number of key nodes under a target).

In the worst case, each insertion (validation of the subtree being inserted and Algorithm 3) and each deletion (Algorithm 4) runs in time $O(|T_p| + n.n_{target}.c_2)$. Let m be the number of updates. Since validation steps are needed only on update positions, if we disregard the value of c_2 then the cost of incremental validation under m updates is $O(m.(|T_p| + n.n_{target}))$. The last step of our incremental validation corresponds to visiting and updating *keyTree*s. To this end, we need to visit all the attribute nodes of each *keyTree* concerned by the updates. This routine runs in time $O(n.n_{target})$.

The implementation of our validation method was done in Java, and the Xerces SAX Parser was used for reading the XML documents and loading them into our data structures. Each key or foreign key is checked by running the finite state automata that corresponds to its path. We use two stack structures to store the inherited and synthesized attributes.

We now present the results of a preliminary experimental analysis of our validation method. All tests were executed on the same 1.7GHz Pentium 4 machine with 256MB memory, running Windows 2000. For these tests, we used 4 XML documents, varying in the number of nodes (XML elements and attributes) from $250,000$ to $1,000,000$.

To verify the validity of a set of keys and foreign keys over an XML document, we fixed the number of keys to 2 and foreign keys to 1, and we varied the size of the XML document to verify the method's performance. The CPU time required to check the validation from scratch of the XML document wrt the given keys and foreign key is shown in Fig. 16 (a).

To verify the checking time wrt the number of integrity constraints, we fixed the size of the XML document to $500,000$ and we varied the number of keys and foreign keys from 1 to 5. The results (for validation from scratch) are shown in Fig. 16 (b).

The implementation of our incremental validation method was also done in Java. The sequence of updates is treated as a unique transaction and the updates are tested in one traversal of the XML tree. In fact, the sequence of updates is treated before being applied, so that the update positions are sorted in the sequential order of the XML tree lecture.

Our method incorporates the operations *insert* and *delete* and tests for each operation, by using the $KeyTree$, if the XML tree still respects the predefined set of integrity constraints. The $KeyTree$ is stored in a hash table structure that associates each constraint (primary key) with values (the various contexts obtained for the constraint and their corresponding targets, attributes, key data values and foreign key references). The hash table structure organizes the constraints information reducing look up time and it also improves the verification performance (Fig. 17).

The local validation of each subtree to be inserted or deleted is triggered when the update position is reached. The result of this local validation is loaded into new data structures and a new $KeyTree$ is built for the subtree. We can note that if the update does not concern the set of keys and foreign keys, then the local validation is an empty structure, meaning that there are no verifications to be executed. On the other hand, updates concerning constraints that have composed key paths are the ones which demand more comparisons.

Fig. 18 summarizes the behavior of our algorithm for the incremental validation of keys and foreign keys. We ran two experiments, similarly to the ones for the validation from scratch, using a sequence of 50 updates. In the first one we varied the size of the XML document to test the updates wrt five fixed keys and foreign keys. The second experiment was done by fixing the XML document size $(500,000$ nodes) and varying the number of constraints.

Our last experiments aim at assessing the systems capabilities with respect to keyTree construction. As an example consider an absolute key, composed of three parts (an order is identified by its number, the product number and the supplier number). The validation of this key generates a keyTree containing all values found for this key if they are all different. Clearly, the size of the keyTree may vary drastically with respect to the number of key instances found in the document, and it is of particular interest for incremental validation. Fig. 19 shows the time response considering XML documents that contain from 10^2 to 10^5 key instances for validation from scratch. A sequence of 50 updates was considered for incremental validation.

4.4 Related work

In this paper, we present the validation of XML documents wrt schema and its verification wrt integrity constraints in an independent way. However, if we assume that the given integrity constraints are consistent with the given schema, then the integration of both validation routines is a straightforward generalization of our method.

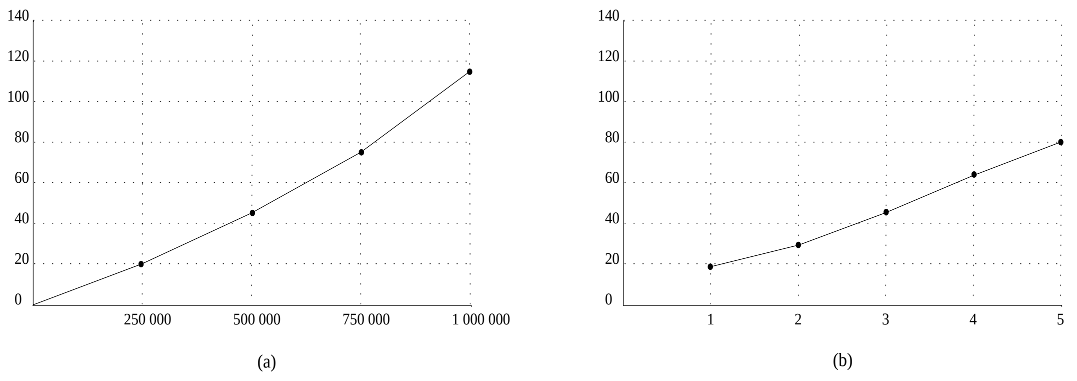

Figure 16: Validation from scratch: (a) Number of nodes × checking time (in seconds). (b) Number of integrity constraints × checking time (in seconds).

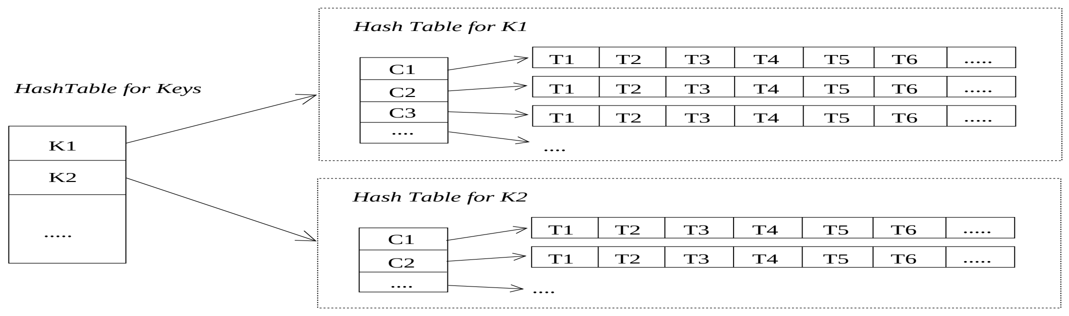

Figure 17: Two-level hash table structure to represent $KeyTrees$.

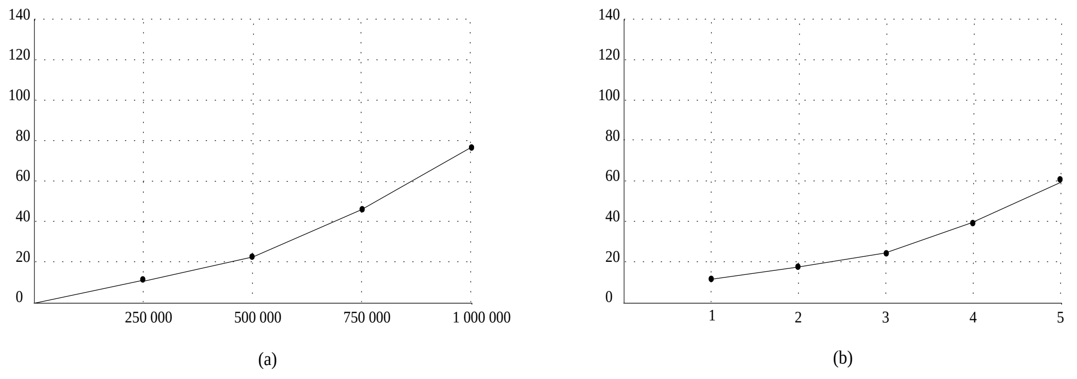

Figure 18: Incremental validation: (a) Number of nodes × checking time (in seconds). (b) Number of integrity constraints × checking time (in seconds).

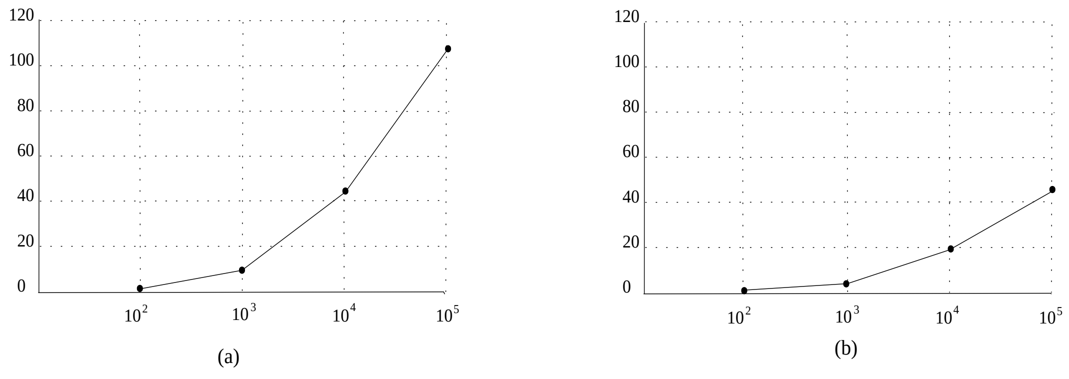

Figure 19: Number of key instances × checking time (in seconds): (a) Validation from scratch for an absolute key. (b) Incremental validation.

In [21], the independence of key syntax from schema constraints is presented as an advantage. One can wonder if this advantage is so clear since structural and integrity constraints are both parts of database design, implementation and optimization. However, as shown in [10], verifying consistency of structural and integrity constraints for XML documents is a difficult problem. In several cases, in order to reason about satisfiability and implication, it is better to consider only integrity constraints.

Proposals that deal with both structural and integrity constraints usually impose restrictions on schema definitions. In [30], for instance, the authors propose an Unified Constraint Model (UCM) which is tightly coupled with the schema model. The schema is defined using *the type system* of XQuery algebra [32] which captures the structural aspects of XSD. Validating a document wrt schema constraints corresponds to finding a *unique* type assignment to each position on an XML tree. In other words, schemas must only define LTL (Section 3). In UCM it is possible to define the "key of a type" by specifying the type name together with the key components (defined by path expressions). A drawback of [30] is the lack of relative and foreign key constraints.

In [31] key paths are composed of a (possibly empty) ranked sequence of up symbols (*i.e.*, Kleene closure cannot be used) followed by a nonempty simple (downward) path. For instance, consider a document with information about conference proceedings. Proceedings are identified by an absolute key $key_{Proc} = (/bib/proc/, \{confName, confYear\})$ containing the name of the conference and its year. As articles are identified by numbers inside a proceedings, we need a relative key to express this constraint. Thus, we write $key_{Article} = (/bib/proc/article/, \{numArticle, [up]/confName, [up]/confYear\})$ where up is a wildcard denoting a move up on the XML tree. Such a syntactic possibility can be translated into CTK. Let $K_1 = (/, (bib/proc/, \{confName,\ confYear\}))$ and $K_2 = (/bib/\ proc/, (article/, \{numArticle\}))$ be two CTK keys. Notice that $key_{Article}$ holds if K_1 and K_2 do. More generally, let k be the length of the longest upward wildcard sequence in a constraint C. It can be shown that there is a set $\{C_1, ..., C_k\}$ of CTK constraints such that if $\{C_1, ..., C_k\}$ is verified then C is verified too.

The goal of [12], where XPath and XSD are used to express constraints, is to perform incremental constraint checking. Their approach differs from ours since in [12] constraints are translated into logic formula and updates on documents are related to computation of incremental versions of the formula. They use schema information to optimize their computation.

In [24], a validator for XML constraints is presented for incrementally checking updates. Its performance is linear in the size of the part being updated, for each key being checked. Although this work is similar to ours, it is worth noting that in [24] foreign keys are not treated and the incremental validation is considered only for a single update. In our approach, as the validation of keys and foreign keys is done for multiple updates, the maintenance of the *keyTree* is performed progressively (while considering the update sequence). The *keyTree* can be temporarily invalid. In this way, our complexity is bound to the number and type of update operations, the number of constraints that are being verified and the number of tuple values corresponding to a constraint.

The current work extends and merges our previous proposals which have considered XML document validation only under single updates. In [17], we propose an incremental schema validation method, but only wrt DTD. In [7, 18] we use a tree transducer to address the problem of the key validation. In the current paper integrity constraints are specified by an attribute grammar which makes our validation strategy simpler. At this point, some similar aspects with [13] can be observed. In [13] XML documents are mapped to relational databases. Indeed, relational DBMS are tuned to efficiently validate integrity constraints. Nevertheless, when relying on a relational DBMS, the system must factorize XML documents before it can store them in its data structures. This entails format mappings and interchanges between XML's hierarchical structure and the DBMS structures, which raises non trivial, theoretical questions about the relational database design on the one hand, and XML update (and XML query) translations on the other hand (as shown in, *e.g.*, [19, 48, 38]).

Proposals for mapping XML in RDBMS by considering XML integrity constraints exist [25, 28, 27, 39]. However, in [28, 27], the authors state that it is impossible to effectively propagate all forms of XML constraints supported by XML Schema, including keys and foreign keys, even when the transformations (from XML to relational) are trivial. In contrast [39] does not deal with such theoretical consideration, but the authors present an ad hoc translation of XML constraints in a relational framework. Experimental results reported in [39] compare this proposal to the one in [9], and conclude that the translation of keys and foreign keys leads to improve query execution time in the context of relational databases.

Authors in [25] notice that, in current propositions, the design of the relational database aimed to store XML data is tuned either for updates (enforcing the constraints efficiently), or for queries workload (to achieve better performance), however, to find a good compromise for both seems to be still an open problem. Indeed, as noticed in [38], one big, open challenge is to efficiently process queries to hierarchical XML data, in a database whose fundamental storage is table-based and whose fundamental query engine is tuple-oriented.

Update processing is even a bigger challenge in this context: for instance one can notice that since a single XML update may affect several tuples in the relational store, transactions must be carefully used to prevent anomalies. Moreover, the XML "view" of the relational database must be *updatable*, *i.e.* there must be a unique, *side effect free* translation from any update on this view to the underlying relations. In [19] the authors show that this is still an open problem for XML views that are not defined by general nested relational algebra or that can not be rewritten into a *nest-last* relational form.

To conclude, the use of relational databases for allowing data to be imported, accessed and exported in the XML format is still an important challenge, addressed by some of the so-called "native XML data stores". As argued in [48], a natural implementation of such systems keeps an XML logical format (even if the underlying storage is relational or object-oriented) in order to achieve scalability, data-access speed and reliability. We can place our work in this context.

5 Conclusions and perspectives

In this paper we present a method to incrementally verify multiple updates in the presence of schema and integrity constraints in XML documents. Update operations are the insertion, deletion and replacement of any subtree of the XML tree. The validity of the resulting XML document is determined only after having analyzed all update operations in a given set of updates. If the resulting document does not violate the imposed constraints, then updates are committed and the document is permanently changed.

Though there exist previous works on schema validation on the hand, and key verification on the other hand, our approach is new in the sense that it considers them both simultaneously, in one single pass over the XML document being processed. This implies to deal with interesting research topics, including incremental validation, unranked tree processing (not translated into ranked trees) and attribute grammars, as well as with non-elementary and non-single update operations on XML documents.

The incremental verification of schema constraints is performed by using a bottom-up tree automaton to re-validate just the parts of the XML document affected by the updates. Our algorithm is not restricted to schemas specified by DTDs or XML Schema, but it also works on schemas obtained from any regular tree grammar, even those which are not local or single typed tree grammars. Attribute grammars are used to formalize the process of integrity constraints verification.

The algorithms presented here have been implemented in Java, and experimental results show that the incremental schema verification has advantages over the verification from scratch, for multiple updates and for large XML documents. In large scale tests, our incremental schema validation algorithm (although implemented without optimization) uses almost one third of the time needed for the validation from scratch performed by a highly optimized commercial product.

The experimental results obtained with our key and foreign key validation routines are also encouraging. Despite their theoretical complexity, their behavior led to a graphic which grows almost linearly with the size of the processed document (Fig. 16). Results for the incremental key verification programs are similar, but more efficient, to those of the verification from scratch. Indeed, the incremental verification of keys is, in the limit, better than the verification from scratch (compare the results in Fig. 16 and 18).

Our asymptotic time complexity and experimental results show the efficiency of our incremental validation of updates: it is at least as efficient as those proposed by [23, 45] (even if it is hard to compare precisely). Moreover, it has some advantages over them, such as space requirements, multiple updates on any tree node and flexibility of integrating schema and key constraint validation.

Several directions have to be investigated for future work:

Integration of our approach into an existing update framework: We have designed algorithms to run in the "back office" of a framework enabling the performance of updates over XML documents. As discussed in Introduction, it can be either a text editor and/or an update language. Next step will be to integrate our routines in such a framework.

New update operations: Our set of update operations conforms to recent recommendations for updates in

XQuery ([22]), but it does not include insertion or deletion of a node in a path, say under position p, *i.e.*, an update operation allowing changes on the hierarchy by the addition or the removal of one level. Although these operations are not explicitly recommended by [22], they may be useful for some applications. Nevertheless, before implementation, their semantics must be carefully specified: this new kind of insertion consists in adding to the child axis a new node that becomes the father of one child of p. Moreover, it can also be the nesting of several children of p under one new level: in that case, the update must contain the specification of which children become children of the newly inserted node. The new kind of deletion might be the inverse operation.

New classes of constraints: Our integrity constraint validator can be adapted to verify different constraints such as XML functional (XFD) and inclusion dependencies. Such constraints can be expressed in a language very similar to CTK. The approach to implement them can be exactly the same as the one presented here for keys and foreign keys (using the attribute grammar): we just need to adapt the positions where tests must be performed and the kind of tests to be performed.

Use of a more expressive constraint language: One of the first possibilities to consider is to extend our key constraint language to include other XPath expressions, such as predicates or the use of other axes. XPath has been widely used in XML query languages and in XML specifications. In practice, many applications do not need the excessive power of the full XPath (which makes it rather expensive to process); they use only a fragment of XPath. Considering the sub-languages of XPath studied in [15], we notice that path expressions in CTK belongs to $\mathcal{X}_r$ the XPath sublanguage which allows navigation along the ancestor and descendant axes (while others permit only parent and child axes); but does not allow upward navigation or the use of qualifiers (predicates). Indeed, since $\mathcal{X}_r$ is the sublanguage used by XML Schema to specify integrity constraints ([15]) we have (until now) considered that our CTK was well adapted to most of the practical cases. Nevertheless, it would be interesting to extend our proposition to a quite general constraint language such as the one presented in [29].

References

[1] Altova XMLSpy. At http://www.altova.com.

[2] Apache XML editor. Available at http://www.apache.org/xerces-j/.

[3] Official website for SAX. Available at http://www.saxproject.org/.

[4] XML schema. Available at http://www.w3.org/XML/Schema.

[5] XMLmind. Available at http://www.xmlmind.com/xmleditor/.

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[7] M. A. Abrão, B. Bouchou, M. Halfeld Ferrari, D. Laurent, and M. A. Musicante. Incremental constraint checking for XML documents. In *XSym*, number 3186 in LNCS, 2004.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1988.

[9] Sihem Amer-Yahia, Fang Du, and Juliana Freire. A comprehensive solution to the xml-to-relational mapping problem. In Alberto H. F. Laender, Dongwon Lee, and Marc Ronthaler, editors, *WIDM*, pages 31–38. ACM, 2004.

[10] M. Arenas and L. Libkin. A normal form for XML documents. In *ACM Symposium on Principles of Database System*, 2002.

[11] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4), 2004.

[12] M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated update management for XML integrity constraints. In *Program Language Technologies for XML (PLANX02)*, 2002.

[13] M. Benedikt, C-Y Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constraints in data integration. In ACM Press, editor, *SIGMOD, San Diego, CA*, 2003.

[14] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In *XIME-P*, 2005.

[15] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In *ICDT*, 2003.

[16] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Simï£¡on. XQuery 1.0, w3c candidate recommendation 8 june 2006. Available at http://www.w3.org/TR/xquery.

[17] B. Bouchou and M. Halfeld Ferrari Alves. Updates and incremental validation of XML documents. In Springer, editor, *The 9th International Workshop on Data Base Programming Languages (DBPL)*, number 2921 in LNCS, 2003.

[18] 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.

[19] V. Braganholo, S. Davidson, and C. A. Heuser. On the updatability of xml views over relational databases. In *Web and Databases (WebDB)*, San Diego, CA, USA, June 2003.

[20] A. Brüggeman-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over nonï£¡ranked alphabets. Technical Report HKUST TCSC 2001 05, Hong Kong Univ. of Science and Technology Computer Science Center (available at http://www.cs.ust.hk/tcsc/RR/2001ï£¡05.ps.gz), 2001.

[21] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. C. Tan. Keys for XML. In *WWW10, May 2-5*, 2001.

[22] D. Chamberlin, D. Florescu, and J. Robie. XQuery update facility, W3C Working Draft 11 July 2006. Available at http://www.w3.org/TR/2006/WD-xqupdate-20060711/.

[23] 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.

[24] Y. Chen, S. B. Davidson, and Y. Zheng. XKvalidator: a constraint validator for XML. In ACM Press, editor, *Proceedings of the 11th International Conference on Information and Knowledge Management*, pages 446–452, 2002.

[25] Yi Chen, Susan B. Davidson, and Yifeng Zheng. Constraints preserving schema mapping from xml to relations. In *WebDB*, pages 7–12, 2002.

[26] J. Clark. XSL transformations (XSLT 1.0) -w3c recommendation 16 november 1999. Available at http://www.w3.org/TR/xslt.

[27] Susan B. Davidson, Wenfei Fan, and Carmem S. Hara. Propagating xml constraints to relations. *J. Comput. Syst. Sci.*, 73(3):316–361, 2007.

[28] Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Jing Qin. Propagating xml constraints to relations. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 543–. IEEE Computer Society, 2003.

[29] A. Deutsch and V. Tannen. XML Queries and Constraints, Containement and Reformulation. *Theoretical Computer Science*, 336(1), 2005.

[30] W. Fan, G. M. Kuper, and J. Siméon. A unified constraint model for XML. In *WWW10, May 2-5*, 2001.

[31] W. Fan, P. Schwenzer, and K. Wu. Keys with upward wildcards for xml. In *DEXA '01: Proceedings of the 12th International Conference on Database and Expert Systems Applications*, pages 657–667. Springer-Verlag, 2001.

[32] Mary F. Fernández, Jérôme Siméon, and Philip Wadler. An algebra for xml query. In *FSTTCS*, 2000.

[33] G. Ghelli, C. Rï£¡, and J. Simï£¡on. XQuery! an XML query language with side-effects. In *DATA-X colocated with EDBT 2006*, 2006.

[34] Mírian Halfeld Ferrari Alves. *Aspects dynamiques de XML et spécification des interfaces des services web avec PEWS*. Habilitation à diriger des recherches, Université François Rabelais de Tours, 2007. In preparation.

[35] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley Publishing Company, second edition, 2001.

[36] George Karakostas, Richard J. Lipton, and Anastasios Viglas. On the complexity of intersecting finite state automata. In *IEEE Conference on Computational Complexity*, 2000.

[37] C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. In *Workshop on Data Base Programming Languages (DBPL)*, pages 233–256, 2003.

[38] Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikutty, James W. Warner, Vikas Arora, and Susan Kotsovolos. Query rewrite for xml in oracle xml db. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 1122–1133. Morgan Kaufmann, 2004.

[39] Qiuju Lee, Stéphane Bressan, and J. Wenny Rahayu. Xshrex: Maintaining integrity constraints in the mapping of xml schema to relational. In *DEXA Workshops*, pages 492–496. IEEE Computer Society, 2006.

[40] M. Mernik and D. Parigot (Eds). Special issue on attribute grammars and their applications. In *Informatica, Vol 24 No 3, September*, 2000.

[41] M. Murata. Relax (REgular LAnguage description for XML). Available at http://www.xml.gr.jp/relax/, 2000.

[42] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema language using formal language theory. In *Extreme Markup Language, Montreal, Canada*, 2001.

[43] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML Schema Language using Formal Language Theory. In *ACM, Transactions on Internet Technology (TOIT)*, 2004.

[44] Frank Neven. Attribute grammars for unranked trees as a query language for structured documents. *J. Comput. Syst. Sci.*, 70(2):221–257, 2005.

[45] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2003.

[46] Stylus Studio. XMLStylus. Available at http://www.stylusstudio.com.

[47] G. M. Sur, J. Hammer, and J. Simï£¡on. An XQuery-based language for processing updates in XML. In *PLAN-X - Programming Language Technologies for XML A workshop colocated with POPL 2004*, 2004.

[48] Athena Vakali, Barbara Catania, and Anna Maddalena. Xml data stores: Emerging practices. *IEEE Internet Computing*, 9(2):62–69, 2005.