Updates and Incremental Validation of XML Documents

Béatrice BOUCHOU Mírian HALFELD FERRARI ALVES

Université de Tours - Laboratoire d'Informatique
Antenne Universitaire de Blois
3 place Jean Jaurès
41000, Blois, France
{bouchou, mirian}@univ-tours.fr

Abstract

We consider the incremental validation of updates on XML documents. When a *valid* XML document (*i.e.*, one satisfying some constraints) is updated, it has to be verified that the new document still conforms to the imposed constraints. Incremental validation of updates leads to significant savings on computing time when compared to brute-force validation of an updated document from scratch.

This paper introduces a correct and complete set of update operations that can be integrated in an XML manipulation language. Indeed, any document generated by a composition of our update operations is valid, and, every valid document can be generated by a composition of our update operations (from the empty document). To accept an update, the validity of the result is checked first (without any change on the original document). Validation tests are performed incrementally, *i.e.*, only the validity of the part of the document directly affected by the update is checked. Changes to the original document are effectively performed *only* when the update is accepted.

1 Introduction

We present a method for incrementally validating updates on an XML document. We assume a data-exchange environment where an XML document should respect schema constraints. When a valid XML document is updated, it has to be verified that the new document still conforms to the imposed schema. Validation from scratch requires reading the entire document after each update. An incremental method is undoubtedly very useful, in particular when we consider that the evolution of XML as an exchange format depends on its capability to support not only queries but also updates.

Proceedings of the 9th International Conference
on Data Base Programming Languages (DBPL),
Potsdam, Germany, September 6-8, 2003

We view an XML document as a structure $\mathcal{T}$ composed of an unranked labeled tree t (*i.e.*, a tree whose nodes have no fixed -or ranked- arity) and functions *type* and *value*. The function *type* indicates the type of a node (*element*, *attribute* or *data*). The function *value* gives the value associated with a leaf (a data node). Figure 1 shows part of the labeled tree representing the document used in our examples. Each node has a position and a label (for instance, position 0 is associated with label $Cust$). From this figure we see that an XML element has both its sub-elements and attributes as children in the tree. Elements and attributes associated with arbitrary text have a child labeled *data*. Attribute labels are depicted with a preceding @.

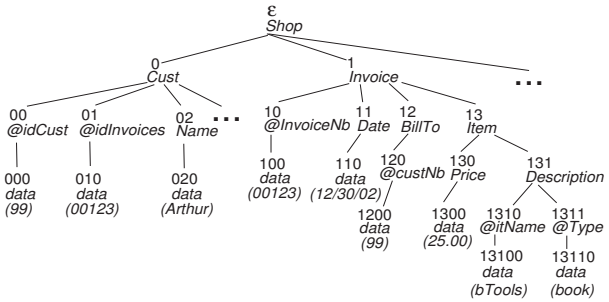

Figure 1: Labeled tree t representing an XML document.

XML documents should respect a schema $\mathcal{D}$ which corresponds to attribute and element restrictions. We see the schema $\mathcal{D}$ as a structure composed of a bottom-up tree automaton and some "extra" information about attribute values, both introduced in [5] where only validation from scratch is considered. Validating schema constraints means to execute the tree automaton over the labeled tree t. This computation results in another labeled tree r (called a running tree) with the same positions as t, but labeled with the *states* of the tree automaton, as illustrated in Example 1.1 below. Roughly, a state q is assigned to a position p in r if the children of p in t verify the element and attribute constraints established by the tree automaton.

Example 1.1 Figure 2 shows a running tree resulting from the execution of a given tree automaton over the tree t of

Figure 1. To illustrate the execution of this tree automaton, suppose that it has the transition rule

$$Invoice, \{\{q_{invoiceNb}\}, \emptyset\}, q_{Date}\ q_{BillTo}\ q^*_{Item} \rightarrow q_{Invoice}.$$

This rule states that a position p, labeled $Invoice$ in t, can be associated with the state $q_{Invoice}$ in r if the following attribute and element constraints are respected:
(i) position p has a required attribute child $invoiceNb$ (*i.e.*, the first child of p in r is associated with $q_{invoiceNb}$) and
(ii) children of p that are elements respect the regular expression $Date\ BillTo\ Item^*$ (*i.e.*, the second and the third children of p in r are associated with q_{Date} and q_{BillTo}, respectively; the other right children (if they exist) are associated with q_{Item}).
The above constraints are respected by, for instance, position 1 in r and thus 1 is associated with $q_{invoice}$ (Figure 2). The automaton executes bottom-up by considering each position and the transition rule that applies to it. A tree automaton accepts the document tree if and only if the root of the corresponding running tree is labeled with a *final* state. □

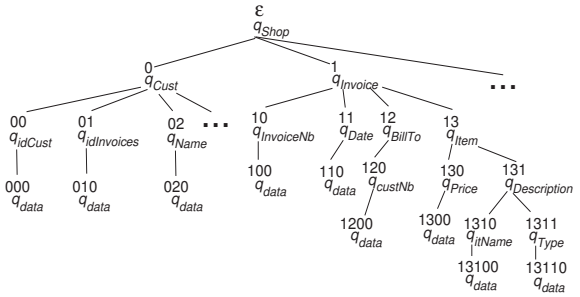

Figure 2: Running tree r resulting from the execution of a tree automata over t.

Given a valid XML document, an update is done taking into account the following features:

• Updates are seen as changes to be performed on the tree representation $\mathcal{T}$ of an XML document. We do not consider here the translation between an XML document and its tree representation, this is done by well known tools such as SAX and DOM.

• Only updates that preserve the validity of the document are accepted. If the update violates a constraint, then it is rejected and the XML document remains unchanged.

• The acceptance of an update relies on *incremental validation* tests, *i.e.*, only the validity of the part of the original document directly affected by the update is checked.

Based on the above points, we introduce a set of update operations capable of inserting, deleting or modifying parts of tree representing an XML document. Before accepting an update, we perform *incremental* validity tests which consist of verifying the validity of a small part of the document tree. If the desired update concerns position p, we just check if the subtree rooted at p's father continues to respect the validity conditions. Let $\mathcal{A}$ be the automaton in $\mathcal{D}$. Let δ be a transition rule of $\mathcal{A}$ that associates a position u, labeled a, with state q_a . An update at position p (a child of u) changes the sequence of states associated with u's children by the automaton. Considering the new children of u, we need to verify if they still respect the constraints established by δ. If these constraints are respected, δ can be applied and the label q_a is associated with u, otherwise the intended update is rejected since it violates constraints. To efficiently verify if δ applies to the new children of u, we only build a temporary *sequence of new states* of u's children. The following example illustrates this process.

Example 1.2 Let $\mathcal{D}$ be a schema containing an automaton $\mathcal{A}$ with the following transition rules (among others):

(1)Item, $\{\emptyset, \emptyset\}$, $q_{Price}\ q_{Description}$ $\rightarrow q_{Item}$
(2)Price, $\{\emptyset, \emptyset\}$, q_{data} $\rightarrow q_{Price}$
(3)data, $\{\emptyset, \emptyset\}$, $\emptyset$ $\rightarrow q_{data}$

Now, we assume the valid document of Figure 1 describing the customers and invoices of a shop. Each invoice contains the price and the description of the items bought by a customer. We consider the item depicted at position 13 and we assume the insertion of another price for this item. This operation corresponds to the insertion of a labeled tree t_1 (having positions ϵ and 0 associated with labels $Price$ and $data$, respectively) at position 131 of the tree t (Figure 1). The labeled tree t' in Figure 3 represents the requested change over t.
The verification of the update consists in: (i) considering that the update is performed (without performing it yet) and (ii) verifying if the state q_{Item} can still be associated with position 13 (131's father) by analyzing the unique transition rule whose head is q_{Item}.
To this end, we are going to build the sequence of states associated with 13's children. To better illustrate our example, we consider the subtree of t (Figure 1) whose root is at position 13 and its requested updated version (the corresponding subtree on Figure 3). We assume the bottom-up execution of $\mathcal{A}$ over t_1. We apply rule 3 over the leave of t_1 to obtain the state q_{data}. This leave shall be at position 1310 if the update is accepted (see requested updated tree partly depicted by Figure 3). Then, we apply rule 2 over the root of t_1 to obtain q_{Price}. The root of t_1 is at position 131 of the requested updated tree (Figure 3). Note that this update does not concern the subtrees on the left of position 131: nothing changes for the subtree rooted at position 130. Moreover, the subtrees on the right have just been shifted (see the subtree now rooted at position 132 of Figure 3). In other words, the update does not affect position 130 (associated with q_{Price}) and position 132 is the result of a shift (a new position, but associated with an "old" state, *i.e.*, one computed before the update). Thus, we only have to calculate the state associated with position 131 in order to obtain the complete new sequence of children states for position 13.
Now, we consider rule (1). It can be applied to position 13 if all the following conditions hold: (i) $t(13) = Item$, (ii) for all children pos of 13 we have $type(t, pos) \neq attribute$ and (iii) the concatenation of the labels associated with

13's children composes a word that corresponds to the regular expression $q_{Price}\ q_{Description}$. In our case this concatenation is $q_{Price}\ q_{Price}\ q_{Description}$ (positions 130, 131 and 132, respectively). This word does not match the regular expression $q_{Price}\ q_{Description}$, so condition (iii) does not hold. Rule 1 cannot be applied to position 13. The update is rejected, since it violates validity.
Notice that accepting or rejecting an update depends on the schema being considered. For instance, if we consider a schema $\mathcal{D}'$ similar to $\mathcal{D}$ except for transition rule (1) that is replaced by: Item, $\{\emptyset, \emptyset\}$, $q^*_{Price}\ q_{Description} \rightarrow q_{Item}$ then the insertion of t_1 at position 131 in t is accepted. Indeed, the concatenation $q_{Price}\ q_{Price}\ q_{Description}$ matches the regular expression $q^*_{Price}\ q_{Description}$. □

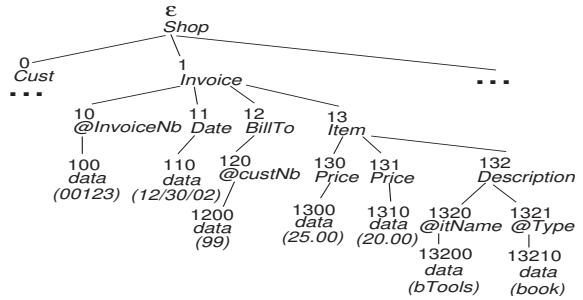

Figure 3: Labeled tree t' representing the requested changes on t: an insertion at position 131.

The main contributions of the paper are:

• The definition of a structure, called XML *dossier*, that formalizes and summarizes all the features necessary to the update validation.

• A correct and complete set of update operations. Indeed any XML dossier generated by a composition of our update operations is valid, and given a schema $\mathcal{D}$, every XML dossier valid with respect to $\mathcal{D}$ can be generated by a composition of these operations. The changes to an XML dossier due to an update are precisely defined. Four update operations are introduced, namely $insert$, $insertBefore$, $delete$ and $replace$.

• An incremental validation method that allows significant improvements over brute-force validation from scratch.

This paper is organized as follows: In Section 2, XML dossiers are defined and we discuss each component of this structure. In Section 3 we define the set of update operations and in Section 4 we show how incremental validation is performed. Finally, Section 5 concludes with related work and our perspectives for further research.

2 XML Dossiers

An XML dossier $\mathcal{X}$ contains all the components necessary to the validation of updates. It is a tuple $(\mathcal{D}, \mathcal{T}, \mathcal{R})$ where: $\mathcal{D}$ is a schema that defines attribute and element constraints, $\mathcal{T}$ is the tree representation of an XML document and $\mathcal{R}$ is a structure built from $\mathcal{D}$ and $\mathcal{T}$, over which (i) the validity of $\mathcal{T}$ with respect to $\mathcal{D}$ can be easily determined (only with few tests on values contained in $\mathcal{R}$), and (ii) the incremental validity test performed while updating is also easily applied.

2.1 XML document representation

There are different ways to view an XML document as a tree. Before introducing our choice of representation, we recall the notion of unranked Σ-valued trees [21]. Let $\mathbb{N}^*$ be the set of all finite strings of positive integers with the empty string ϵ as the identity. The following definition assumes that $dom(t) \subseteq \mathbb{N}^*$ is a nonempty set closed under prefixes[1], *i.e.*, if $u \preceq v$, $v \in dom(t)$ implies $u \in dom(t)$. Clearly, this set $dom(t)$ represents the set of nodes of t, uniquely identified with a Dewey like prefix schema.

Definition 2.1 - Σ-valued tree t [21]: Given an alphabet Σ, a nonempty Σ-valued tree t is a mapping $t : dom(t) \rightarrow \Sigma$ where $dom(t)$ satisfies: $j \geq 0, uj \in dom(t), 0 \leq i \leq j \Rightarrow ui \in dom(t)$. The set $dom(t)$ is also called the set of *positions* of t. We write $t(p) = a$, for $p \in dom(t)$, to indicate that the Σ-symbol associated with p is a. For each position p in $dom(t)$, $children(t, p)$ denotes the positions pi in $dom(t)$, and $father(t, p)$ denotes the *father* of p. Define an *empty tree* t as the one having $dom(t) = \emptyset$. □

Definition 2.2 - XML tree $\mathcal{T}$: Let $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$ be an alphabet where Σ_{ele} is the set of element names and Σ_{att} is the set of attribute names. An XML tree is a tuple $\mathcal{T} = (t, type, value)$ where:

• t is a Σ-valued tree (*i.e.*, $t : dom(t) \rightarrow \Sigma$).

• $type$ and $value$ are functions defined as follows for a position $p \in dom(t)$:

$$type(t,p) = \begin{cases} data & \text{if } t(p) = data \\ element & \text{if } t(p) \in \Sigma_{ele} \\ attribute & \text{if } t(p) \in \Sigma_{att} \end{cases}$$

$$value(t,p) = \begin{cases} setval \subset \mathbf{D} & \text{if } type(t,p) = data \\ undefined & \text{otherwise} \end{cases}$$

where **D** is an infinite (recursively enumerable) domain. □

In Figure 1 we have, for instance, $type(t, 13) = element$ and $value(t, 1300) = \{25.00\}$.

To define update operations we need the notions of *frontier* and *insert frontier*. The frontier corresponds to the set of leaves while the insert frontier is the set of positions (not in $dom(t)$) where the simple insertion of new subtrees is possible.

Definition 2.3 - Frontier and insert frontier of a finite tree t: Given a tree t and considering $i \in \mathbb{N}$, the *frontier* of t, denoted by $fr(t)$ is defined by $fr(t) = \{u \in dom(t) \mid \neg\exists\ i \text{ such that } ui \in dom(t)\}$ while the *insert frontier* of t, denoted by $fr^{ins}(t)$ is defined by $fr^{ins}(t) = \{ui \notin dom(t) \mid u \in dom(t) \wedge [(i = 0) \vee ((i \neq 0) \wedge u(i-1) \in dom(t)]\}$. For an empty tree t, define $fr^{ins}(t) = \{\epsilon\}$. □

[1]The *prefix relation* in $\mathbb{N}^*$, denoted by $\preceq$ is defined by: $u \preceq v$ iff $uw = v$ for some $w \in \mathbb{N}^*$.

2.2 Schema representation

We assume that XML views are built from different data sources according to a particular schema. In our approach, a schema $\mathcal{D}$ is specified by an extended non-deterministic bottom-up finite tree automaton (ENFTA) enhanced with an attribute table.

Definition 2.4 - Extended non-deterministic bottom-up finite tree automaton (ENFTA) [5]: An ENFTA over an alphabet 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 set of two 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)* $q \in Q$. □

Definition 2.5 - Schema $\mathcal{D}$ for XML documents: Let $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$ be a schema alphabet. A schema $\mathcal{D}$ for XML documents is a tuple $\mathcal{D} = (\mathcal{A}, Æ)$ where $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ is an ENFTA over Σ and $Æ[att\text{-}name, att\text{-}kind, ele]$ is an attribute table having one tuple for each pair $(att\text{-}name, ele)$ that associates an attribute $att\text{-}name \in \Sigma_{att}$ with an element $ele \in \Sigma_{ele}$. We assume that attribute kinds $att\text{-}kind$ in Æ are those possible in a DTD (*i.e.*, CDATA, ID, IDREF and IDREFS). □

Definition 2.4 extends classical tree automata in order to deal with trees with different kinds of nodes. In $\mathcal{T}$, the children of any position $p \in dom(t)$ can be classified into two groups: those that are unordered, corresponding to the attributes of the node, and those that are ordered, corresponding to the sub-elements.

In a schema $\mathcal{D}$, element constraints are expressed by regular expressions (part (iii) of $\mathcal{A}$'s transition rules). Attribute constraints imply two levels of specification. In the first level, for each element, the specification (part (ii) of $\mathcal{A}$'s transition rules) indicates the attributes that are obligatory ($S_{compulsory}$) and optional ($S_{optional}$). In the second level, for each attribute, the specification indicates its kind (attribute table Æ). Thus, as discussed in [5], the validity of attribute requires some tests on attribute values. These tests verify the uniqueness of identifier values (called ID values) in the whole document, and the existence of ID values corresponding to reference values (called IDREF or IDREFS values).

Example 2.1 We consider the schema $\mathcal{D} = (\mathcal{A}, Æ)$ (concerning customers and invoices in a shop) which has been used to build the running tree of Figure 2 from the tree of Figure 1. The schema alphabet Σ contains all the labels appearing in Figure 1.
The ENFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ has $Q = \{q_a \mid a \in \Sigma\}$, $Q_f = \{q_{Shop}\}$ and twenty-one rules in Δ: four of them are presented in Examples 1.1 and 1.2.
The table Æ has the following tuples: { ⟨idCust, ID, Cust⟩, ⟨idInvoices, IDREFS, Cust⟩, ⟨invoiceNb, ID, Invoice⟩, ⟨custNb, IDREF, BillTo⟩, ⟨itName, CDATA, Description⟩, ⟨Type, CDATA, Description⟩ }. □

2.3 XML documents respecting a schema

Given a schema $\mathcal{D} = (\mathcal{A}, Æ)$ and an XML tree $\mathcal{T} = (t, type, value)$, we want to verify if $\mathcal{T}$ respects the validity constraints imposed by $\mathcal{D}$. Consider first the execution of $\mathcal{A}$ over t. To assume a state q at position p, the automaton $\mathcal{A}$ performs the following tests:

1. If p has *attribute* children then their states should match those specified by the sets in S, namely $S_{compulsory}$ and $S_{optional}$, corresponding, respectively, to attributes that *must* appear in the tree and to those that *may* appear.

2. If p has *element* children then the concatenation of their states must belong to the language generated by the regular expression E.

We call *running tree* the Q-valued tree resulting from the execution of a tree automaton $\mathcal{A}$ over t.

Definition 2.6 - Running tree r [5]: Let t be a Σ-valued tree and $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ an ENFTA. A *running tree* r, corresponding to an execution of $\mathcal{A}$ over t, is a tree such that $dom(r) = dom(t)$ defined as follows: for each position p whose children are at positions $p0, \ldots, p(n-1)$ (with $n \geq 0$), $r(p) = q$ if all the following conditions hold:

1. $t(p) = a \in \Sigma$

2. There exists a transition $a, S, E \rightarrow q$ in Δ

3. There exists an integer $0 \leq i \leq (n-1)$ such that the children of p can be classified as follows:

(a) the positions $p0, \ldots, p(i-1)$ are members of a set *posAtt* (possibly empty) and

(b) the positions $pi, \ldots, p(n-1)$ are members of a set *posEle* (possibly empty) and

(c) every children of p is a member of *posAtt* or of *posEle* but no position is in both sets.

4. The tree r is already defined for p's children *i.e.*, $r(p0) = q_0, \ldots, r(p(n-1)) = q_{n-1}$.

5. The word $q_i \ldots q_{n-1}$, composed by the concatenation of the states associated with the positions in *posEle*, belongs to the language generated by E.

6. The sets of S respect the following properties:

(a) $S_{compulsory} \subseteq \{q_0, \ldots, q_{i-1}\}$ and

(b) $(\{q_0, \ldots, q_{i-1}\} \setminus S_{compulsory}) \subseteq S_{optional}$.

A running tree r is *successful* if $r(\epsilon)$ is a final state. □

From Definition 2.6, one can see that t is accepted by $\mathcal{A}$ if and only if r is *successful*. This is one of the conditions an XML tree $\mathcal{T}$ must respect to be valid relative to a schema $\mathcal{D}$.

Consider now the ID/IDREF constraints. During the run of $\mathcal{A}$ over t, bags of ID and IDREF(S) values are filled, according to the table Æ in $\mathcal{D}$. It is then straightforward to verify the ID/IDREF constraints. Precisely, we say that $\mathcal{T}$ respects $\mathcal{D}$ if all the following conditions hold:

C1- The running tree r constructed according to Definition 2.6 from $\mathcal{D}$ and $\mathcal{T}$ is successful.

C2- The ID attributes in t are unique.

C3- The IDREF/IDREFS attributes refer to existing ID attributes.

To facilitate the verification of conditions C1-C3, we define the structure $\mathcal{R}$ containing the running tree r, together with ID and IDREF values.

Definition 2.7 - Run $\mathcal{R}$: Given a schema $\mathcal{D} = (\mathcal{A}, Æ)$ and an XML tree $\mathcal{T} = (t, type, value)$, a run $\mathcal{R}$ is a tuple $\mathcal{R} = (r, V_{ID}, V_{IDREF})$ where r is the running tree (Definition 2.6) and V_{ID} and V_{IDREF}, called id-storage, are *bags* filled using $Æ$ during the run of $\mathcal{A}$ on t, according to the steps below:

• If there exists a tuple in $Æ$ that indicates that an attribute at position p has kind ID, then insert $value(t, p0)$ in V_{ID}.

• If there exists a tuple in $Æ$ that indicates that an attribute at position p has kind IDREF or IDREFS, then insert $value(t, p0)$ in V_{IDREF}. □

Condition C2 holds when V_{ID} has no duplicate and condition C3 is verified when V_{IDREF} only contains values that appear in V_{ID}.

Example 2.2 We consider $\mathcal{X}$, composed by the schema $\mathcal{D}$ of Example 2.1, the tree $\mathcal{T}$ depicted in Figure 1 and the run $R = (r, V_{ID}, V_{IDREF})$. The structure $\mathcal{R}$ contains the running tree r (Figure 2) and[2] $V_{ID} = V_{IDREF} = \{99, 00123\}$. As $r(\epsilon) = q_{Shop}$ and $Q_f = \{q_{Shop}\}$, the running tree r is successful. Notice that conditions C1-C3 are respected by $\mathcal{X}$. □

2.4 Validity of XML dossiers

Now XML dossiers and their validity are defined according to the preceding sections.

Definition 2.8 - XML dossier and validity: An XML dossier is a triple $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ where $\mathcal{D}$ is a schema as specified in Definition 2.5, $\mathcal{T}$ is an XML tree as introduced in Definition 2.2 and $\mathcal{R}$ is a run obtained according to Definition 2.7. Two XML dossiers are equal if their components are equal. An empty dossier has $dom(t)$, $dom(r)$, V_{ID} and V_{IDREF} empty.
An XML dossier $\mathcal{X}$ is *valid* if it is empty or if its run $\mathcal{R} = (r, V_{ID}, V_{IDREF})$ respects the following conditions: (i) r is *successful* (Definition 2.6), (ii) V_{ID} is a *set* and (iii) values in V_{IDREF} exist in V_{ID}. □

We distinguish between two types of validity: the *global* validity of Definition 2.8 and the *local* one, introduced below. The id-storage V_{IDREF} of a *locally* valid dossier can refer to ID attributes not present in this dossier. This notion is very useful in an update context. For instance, when inserting a locally valid dossier $\mathcal{X}_1$ into a valid dossier $\mathcal{X}$, it is reasonable to suppose that the id-storage V_{IDREF1} contains references to attributes in V_{ID} that are not in V_{ID1}.

Definition 2.9 - Local validity: Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be an XML dossier where $\mathcal{T} = (t, type, value)$ and $\mathcal{R} = (r, V_{ID}, V_{IDREF})$. Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be the ENFTA in $\mathcal{D}$. An XML dossier $\mathcal{X}$ is locally valid if its run $\mathcal{R}$ respects the following conditions: (i) $r(\epsilon) = q$ and $q \in Q$ and (ii) V_{ID} is a *set*. □

Example 2.3 The dossier of Example 2.2 is valid.
Now, we consider the dossier $\mathcal{X}_1 = (\mathcal{D}, \mathcal{T}_1, \mathcal{R}_1)$ with the same schema $\mathcal{D}$ as Example 2.1. Let the Σ-valued tree t_1 in $\mathcal{T}_1$ be a subtree similar to the one rooted at position 1 of Figure 1 (*i.e.*, having the same labels). The running tree r_1 in R_1 corresponds to the subtree rooted at position 1 in Figure 2. It has $r_1(\epsilon) = q_{Invoice}$, (with $q_{Invoice} \in Q$ and $q_{Invoice} \notin Q_f$), $V_{ID1} = \{00123\}$ and $V_{IDREF1} = \{99\}$. Clearly, $\mathcal{X}_1$ is not valid. However, it is locally valid. □

In the following, we introduce the notion of *sub-dossier* and one important property concerning them.

Definition 2.10 - Sub-dossier: Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be an XML dossier where $\mathcal{T} = (t, type, value)$ and $\mathcal{R} = (r, V_{ID}, V_{IDREF})$. Let p be a position in $dom(t)$. The XML dossier $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$, where $\mathcal{T}_p = (t_p, type, value)$ and $\mathcal{R}_p = (r_p, V_{IDp}, V_{IDREFp})$, is the sub-dossier of $\mathcal{X}$ at position p if the following conditions hold:

1. $dom(t_p) = \{u \mid pu \in dom(t)\}$
2. $t_p(u) = t(pu)$ for each $pu \in dom(t)$
3. Similarly to t_p, the new functions $type$ and $value$ (associated with $\mathcal{T}_p$) are mappings over $dom(t_p)$ and, therefore, are defined following the same principle as the definition of t_p.
4. The run $\mathcal{R}_p$ is obtained according to Definition 2.7, from $\mathcal{D}$ and $\mathcal{T}_p$. □

Proposition 2.1 *If $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ is a valid XML dossier then for every position $p \in dom(t)$, its associated sub-dossier $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$ is locally valid.* □

In Example 2.3, the dossier $\mathcal{X}_1$ is the sub-dossier of $\mathcal{X}$ at position 1.

3 Updating Valid XML Documents

We define four update operations over XML dossiers showing all changes (on structure and values) that should be performed on their components. Our update processing transforms a valid XML dossier into a (sometimes new) valid XML dossier. Updates that do not preserve validity are rejected. Given a valid dossier $\mathcal{X}$ and a position p, it is possible to update it by performing one of the following operations: $insert(\mathcal{X}_p, p, \mathcal{X})$ (inserts a dossier

[2] In table $Æ$ of Example 2.1, the only ID attributes considered are at positions 00 and 10. From Figure 1, $value(t, 000) = \{99\}$ and $value(t, 100) = \{00123\}$

$\mathcal{X}_p$ in $\mathcal{X}$ at $p \in fr^{ins}(t)$), $insertBefore(\mathcal{X}_p, p, \mathcal{X})$ (inserts $\mathcal{X}_p$ in $\mathcal{X}$ at $p \in dom(t)$), $delete(p, \mathcal{X})$ (deletes from $\mathcal{X}$ the sub-dossier associated to $p \in dom(t)$) and $replace(\mathcal{X}_p, p, \mathcal{X})$ (replaces in $\mathcal{X}$ the sub-dossier associated with p by $\mathcal{X}_p$). Figure 4 illustrates these operations by showing the changes occurring in a Σ-valued tree t.

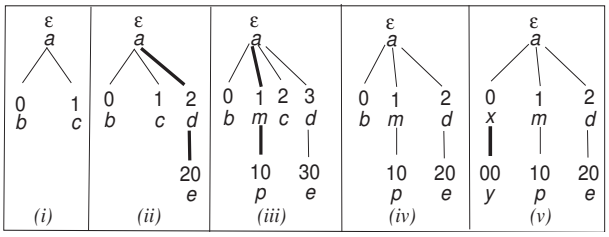

Figure 4: (i) Initial Σ-valued tree t having labels a (position ϵ), b (position 0) and c (position 1). (ii) Insertion at $p = 2$. (iii) Insertion before $p = 1$. (iv) Deletion at $p = 2$. (v) Replace at $p = 0$.

Next we formally define our set of update operations. Notice that we only consider insertion and deletion of non empty locally valid dossiers.

Definition 3.1 - Update: Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be a valid dossier. The result of applying an update operation on $\mathcal{X}$ at position p is a valid dossier $\mathcal{X}'$ defined by:

$$\mathcal{X}' = \begin{cases} (\mathcal{D}, \mathcal{T}', \mathcal{R}') & \text{if } (\mathcal{D}, \mathcal{T}', \mathcal{R}') \text{ is a valid} \\ & \text{dossier different from } \mathcal{X}. \\ \mathcal{X} & \text{otherwise.} \end{cases}$$

where $\mathcal{T}' = (t', type, value)$ and $\mathcal{R}' = (r', V'_{ID}, V'_{IDREF})$ respect the three properties stated below, which are based on the following assumptions:

• The update position is $p = ui$, with $i \in \mathbb{N}$ and $u \in \mathbb{N}^*$. It is defined according to the update operation: (i) $p \in fr^{ins}(t)$, for *Insertion*,
(ii) $p \in dom(t)$ and $p \neq \epsilon$, for *Insertion before* p and
(iii) $p \in dom(t)$, for *Deletion* and *Replace*.

• $n = |children(father(t, p))| - 1$ for $p \neq \epsilon$.

• $DelPos = \bigcup_{k=i}^{k=n} \{w \mid w \in dom(t) \text{ and } w = uku'\}$ if $p \neq \epsilon$, otherwise $DelPos = dom(t)$.

• $ShiftRightPos = \bigcup_{k=i}^{k=n} \{w \mid w = u(k+1)u' \text{ and } uku' \in dom(t)\}$.

• $ShiftLeftPos = \bigcup_{k=i+1}^{k=n} \{w \mid w = u(k-1)u' \text{ and } uku' \in dom(t)\}$ if $p \neq \epsilon$, otherwise $ShiftLeftPos = \emptyset$.

• $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$ is a non empty locally valid XML dossier with $\mathcal{T}_p = (t_p, type, value)$ and $\mathcal{R}_p = (r_p, V_{IDp}, V_{IDREFp})$.

Properties

1. t' is a Σ-valued tree over $dom(t')$ whose definition depends on the type of update :

(a) *Insertion*: $dom(t') = dom(t) \cup \{pv \mid v \in dom(t_p)\}$ and

$$\begin{cases} t'(w) = t(w) & \forall w \in dom(t) \\ t'(pv) = t_p(v) & \forall v \in dom(t_p) \end{cases}$$

(b) *Insertion before* p: $dom(t') = [dom(t) \setminus DelPos] \cup ShiftRightPos \cup \{pv \mid v \in dom(t_p)\}$ and
• $t'(w) = t(w), \forall w \in dom(t)$ and $w \notin DelPos$
• $t'(u(k+1)u') = t(uku')$ for each $u(k+1)u' \in ShiftRightPos$ where $k \in [i..n]$
• $t'(pv) = t_p(v)$ for each $v \in dom(t_p)$

(c) *Deletion*: $dom(t') = [dom(t) \setminus DelPos] \cup ShiftLeftPos$ and
• $t'(w) = t(w)$ for each $w \in dom(t)$ and $w \notin DelPos$
• $t'(u(k-1)u') = t(uku')$ for each $u(k-1)u' \in ShiftLeftPos$ where $k \in [(i+1)..n]$

(d) *Replace*: $dom(t') = [dom(t) \setminus \{v \mid v \in dom(t) \wedge v = pu'\}] \cup \{pv \mid v \in dom(t_p)\}$ and

$$\begin{cases} t'(w) = t(w) & \forall w \in dom(t) \text{ and } w \neq pu' \\ t'(pv) = t_p(v) & \forall v \in dom(t_p) \end{cases}$$

2. The Σ-valued tree r' and the new functions $type$ and $value$ are defined following the same principle as t' (property 1 above).

3. The id-storage V'_{ID} and V'_{IDREF} are defined according to the type of update:

(a) *Insertion* and *Insertion before* p:
• V'_{ID} is the set $V_{ID} \cup V_{IDp}$.
• V'_{IDREF} is the bag $V_{IDREF} \cup V_{IDREFp}$ such that every value in V'_{IDREF} exists in V'_{ID}.

(b) *Deletion*: Let $\mathcal{X}_p$ be the sub-dossier of $\mathcal{X}$ at position p.
• V'_{ID} is the set $V_{ID} \setminus V_{IDp}$.
• V'_{IDREF} is the bag $V_{IDREF} \setminus V_{IDREFp}$ such that every value in V'_{IDREF} exists in V'_{ID}.

(c) *Replace at* p: Let $oldp = p$ and let $\mathcal{X}_{oldp} = (\mathcal{D}, \mathcal{T}_{oldp}, \mathcal{R}_{oldp})$, with $\mathcal{R}_{oldp} = (r_{oldp}, V_{IDoldp}, V_{IDREFoldp})$, be the sub-dossier of $\mathcal{X}$, at position $oldp$, to be replaced by the new dossier $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$.
• V'_{ID} is the set $(V_{ID} \setminus V_{IDoldp}) \cup V_{IDp}$.
• V'_{IDREF} is the bag $(V_{IDREF} \setminus V_{IDREFoldp}) \cup V_{IDREFp}$ such that every value in V'_{IDREF} exists in V'_{ID}. □

The following theorem states that when the resulting dossier is different from the original one, the update operation has *effectively* been performed.

Theorem 3.1 *Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be a valid dossier and let $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$ be a locally valid dossier. Let $\mathcal{X}' = (\mathcal{D}, \mathcal{T}', \mathcal{R}')$ be a dossier* different *from $\mathcal{X}$ and let p be a position.*

• *If $\mathcal{X}' = insert(\mathcal{X}_p, p, \mathcal{X})$ and $p \in fr^{ins}$ then $\mathcal{X}'$ is valid and $\mathcal{X}_p$ is the sub-dossier of $\mathcal{X}'$ at p.*

• *If $\mathcal{X}' = insertBefore(\mathcal{X}_p, p, \mathcal{X})$ then $\mathcal{X}'$ is valid and $\mathcal{X}_p$ is the sub-dossier of $\mathcal{X}'$ at p. Each sub-dossier of $\mathcal{X}$ at p and its right siblings is a sub-dossier of $\mathcal{X}'$, shifted one position to the right.*

• *If $\mathcal{X}' = delete(p, \mathcal{X})$ then $\mathcal{X}'$ is valid and $\mathcal{X}_p$ is the sub-dossier of $\mathcal{X}$ at p, but it is not the sub-dossier of $\mathcal{X}'$ at p. Each sub-dossier of $\mathcal{X}$ at a position that is a right sibling of p is a sub-dossier of $\mathcal{X}'$, shifted one position to the left.*

• *If $\mathcal{X}' = replace(\mathcal{X}_p, p, \mathcal{X})$ then $\mathcal{X}'$ is valid and $\mathcal{X}_p$ is the sub-dossier of $\mathcal{X}'$ at p and there exists $\mathcal{X}_{oldp}$ which is the sub-dossier of $\mathcal{X}$ at p.* □

Proof (Sketch) : From Definition 3.1, it can be verified for each update operation that the validity conditions (Definition 2.8) hold.

We finish this section by stating the correction and the completeness of our update operators. In other words, we show that after an update, a valid XML dossier remains valid and that any valid dossier can be obtained from a sequence of our update operations.

Lemma 3.1 *Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be a valid dossier. The XML dossier $\mathcal{X}'$ resulting from the update of $\mathcal{X}$ according to Definition 3.1 is valid.* □

Theorem 3.2 *Let $\mathcal{X}$ and $\mathcal{X}'$ be valid dossiers with respect to a schema $\mathcal{D}$. There exists a sequence* u *of update operations (of Definition 3.1) such that $\mathcal{X}'$ is the result of applying* u *over $\mathcal{X}$.* □

Proof: The proof is straightforward since $\mathcal{X}' = replace(\mathcal{X}', \epsilon, \mathcal{X})$.

4 Incremental validation

In this section we explain how to perform *incremental* validity tests before accepting an update. From Definition 3.1, we notice that an update over $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ at position p only results in changes to p's right siblings (including itself). Thus, only the subtree rooted at $father(p)$ has to be checked to assure the validity of the updated document.

All update procedures have the dossier $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ and the update position p as input. Procedures $insert$, $insertBefore$ and $replace$ receive a dossier $\mathcal{X}_p$=$(\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$, to be added to $\mathcal{X}$. We consider that $\mathcal{X}$ is valid, p is a correct update position and $\mathcal{X}_p$ is locally valid (or valid, according to the type of update). These assumptions can be easily verified at the beginning of update procedures.

To implement incremental validity tests efficiently, we *simulate* the update using a small auxiliary run $\mathcal{R}_{aux} = (r_{aux}, ID_{aux}, IDREF_{aux})$. The id-storage ID_{aux} and $IDREF_{aux}$ are *bags*. They are computed according to the type of update, implementing the operations used in the Property 3 of Definition 3.1. The Σ-valued tree r_{aux} is always an 1-depth tree. As shown below, and illustrated by Figure 5, r_{aux} is built with the siblings of p in r, and by re-computing the state of p's father.

Construction of r_{aux}:
Leaves: Let $p = ui$ be the update position, with $u \in \mathbb{N}^*$ and $i \in \mathbb{N}$:

1. Copy the left siblings of position p:
   For $j \in [0..i-1]$ do $r_{aux}(j) = r(uj)$
2. Let[3] $n = |children(r, father(r,p))| - 1$ and compute the other leaves according to the update operation:

   For $insert$: $r_{aux}(i)$=$r_p(\epsilon)$. In this case, i is the rightest child.

   For $insertBefore$:
   $r_{aux}(i)$=$r_p(\epsilon)$
   for $k \in [i..n]$ do $r_{aux}(k+1) = r(uk)$

   For $delete$:
   for $k \in [i..(n-1)]$ do $r_{aux}(k) = r(u(k+1))$

   For $replace$:
   $r_{aux}(i)$=$r_p(\epsilon)$
   for $k \in [(i+1)..n]$ do $r_{aux}(k) = r(uk)$

Root: Compute the root $r_{aux}(\epsilon)$ by applying the transition rule associated with the label $t(father(t,p))$.

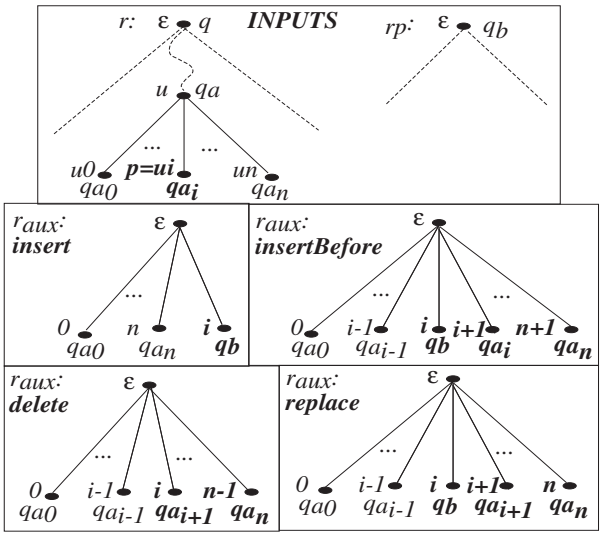

Figure 5: Auxiliary trees r_{aux} for each type of update.

Once $\mathcal{R}_{aux}$ is built, the validity tests are performed by checking (i) if $r(father(t,p))$ equals $r_{aux}(\epsilon)$ and (ii) if ID_{aux} and $IDREF_{aux}$ respect the validity conditions stated in Definition 2.8. Notice that $r_{aux}(\epsilon)$ represents the state that should be associated with position $father(t,p)$ if the update was accepted. In fact, when we test if $r(father(t,p))$ equals $r_{aux}(\epsilon)$, we are taking into account the good properties of our tree automaton $\mathcal{A}$. Since $\mathcal{A}$ is a translation of an unambiguous DTD [9], each label $a \in \Sigma$ is associated with a *unique* state q_a. Thus, both running trees r (before an update) and r' (after an accepted update) should associate the same state with position $father(t,p)$. To verify if an update respects this property, we perform the test on r_{aux} (instead of building the whole r').

[3] We recall that $dom(t) = dom(r)$ and that functions $children$ and $father$ return positions in $dom(r)$ (or $dom(t)$). Thus, $|children(r, father(r,p))|$ gives the number of children of p's father.

Now we make some remarks concerning complexity. Given a dossier $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$, the construction of $\mathcal{R}$ from $\mathcal{D}$ and $\mathcal{T}$ is linear in the number of nodes appearing in the Σ-valued tree t (in $\mathcal{T}$). To visit a subtree of t in order to fill bags containing ID values is linear in the number of nodes of the subtree. The construction of ID_{aux} and $IDREF_{aux}$, from the original bags V_{ID}, V_{IDREF}, etc., is linear in the number of ID/IDREF(S) values present in these original bags. Considering m as the maximum number of values in ID_{aux} and $IDREF_{aux}$ checking the conditions (ii) and (iii) of Definition 2.8 takes, in the worst case, time $O(m^2)$. The construction of $\mathcal{R}_{aux}$ takes linear time in the number of p's siblings. Applying the transition rule over $\mathcal{R}_{aux}$ also takes linear time in the number of p's siblings (*i.e.*, positions in $dom(r_{aux})$).

Therefore, in comparison with a naive method that first applies the update and then checks validity from scratch, it is easy to see that our approach is far better. Supposing that the input of the update operations respect the assumptions of Definition 3.1, we just need to construct some small auxiliary structures and to apply a transition rule *once*. In general the gain is very important: imagine for example that we delete, insert or replace a "phone number" of an element "person" in a large document. The incremental validity test will check only the element "person" concerned by the update, and not the whole resulting document[4]! Not counting that, in the naive method, if the resulting document is not valid then all update process must be rolled back.

5 Conclusions

Tree automata are used in XML research in different ways (see [15, 18] as surveys). For instance, several static typing techniques for XML transformers (including static type checking [4, 14] and type inference [11, 16]) have been modeled with tree automata or tree transducers. In [14], the authors consider the type checking problem expressed by k-pebble transducers, showing that it is decidable. In [4], they consider trees with labels from an infinite alphabet in order to represent both elements and their values, showing that in this case type checking becomes undecidable. The problem of type check transformations of XML trees (given a tree, its schema and a transformation, check whether the resulting tree conforms to a specified schema) is complementary to the one we address in this paper, as it focus on the task of extracting (sometimes with restructuring) information from a given document.

In this paper we have applied unranked tree automata to the *incremental validation of updates*. Our validity test is static, as we perform it before applying the update. For this purpose, we use the extended tree automaton introduced in [5] whose aim is to deal with both element and attribute constraints. This automaton has the same expression power as a DTD and gives rise to an efficient validation method. It has already been implemented as a validator from scratch using both SAX and DOM. The key proposition to perform incremental validation efficiently is to build a temporary sequence of states which represents the requested change (together with bags of ID/IDREF(S) values) and to perform tests upon this small auxiliary structure.

Updates and incremental validation are very useful for many application such as XML databases and XML editors. To our knowledge no information is provided on the incremental validation of updates for the available products on XML [17, 1].

A set of primitive XML update operations (different from ours) is proposed in [20]. Contrary to us, the authors are not interested in the problem of validation. Their goal is to define an XML update language and to translate update operations into updates on the associated relational database. The same problem is addressed in [7] where, in order to specify when XML views are updatable, the authors use the nested relational algebra as the formalism for defining them. As XML views must respect the schema induced by the view specification, in [7] only updates that do not violate it are considered. However, contrary to us, the goal of the authors is not to build an update environment that assures the validity of XML views. In this sense, our work is complementary to theirs.

Our choice of update operations is based on existing propositions for extending XQuery with updates. For instance,[13] enumerates studies of XML update operations and conduct experimental study to compare their incremental checking method against re-validating the whole document after an update. Note that they use a quite different approach to pre-validate updates, based on constraint check queries, which seems to add unnecessary computation overhead. As in [13, 17], we also have implemented a *rename* operation, which gives a new name to a node (element or attribute), not presented here because it is just a simplified version of a *replace*.

Updates on trees appear in papers, such as [19], dealing with the notion of distance between two trees. Their update operations are general while ours reflect desired changes on XML documents. The notion of distance between two trees is also explored in [12], where the goal is to detect changes and not to propose updates that allow such changes.

In [17] the authors propose an incremental validation of XML documents. They first describe a way to incrementally check the sequence composed by sub-element states. They maintain a kind of B-tree as auxiliary information structure. This idea is extended to incremental DTD validation in the case of *one element renaming*. Finally, for specialized DTDs they propose to use as auxiliary structure a binary tree encoding of the document. This structure is of size $O(n)$, where n is the size of the document, and their incremental validation is in time $O(log^2 n)$.

Although the aim of the work presented in [17] is similar to ours, the two proposals differ in many aspects. Con-

[4] As another example, consider that an element "person" is added in a long list of "person" elements (originally valid): the incremental validity test will check only the local validity of this new element and will accept the update if this element is locally valid. This is possible because adding an element in a list is obviously correct when this list of sub-elements is specified in the schema (and it is, since the original list is valid).

trary to [17], we deal not only with element constraints but also with attribute constraints. In [17], only elementary updates affecting *one* node at a time are considered. Our update method allows sophisticated update operations (dealing with trees) without loosing the capacity of effectively performing elementary ones.

In terms of complexity, in our approach, considering only the insertion or deletion of a leaf (at position p) in the XML tree (t being the Σ-valued tree), we improve the complexity bounds. Our auxiliary structure has size $|dom(r_{aux})|$ and our validation time is linear in the number of elements in $dom(r_{aux})$. In [17] the same operation takes time $O(log^2\ n)$ where n is the size of the entire document, *i.e.*, n equals $|dom(t)|$. However, contrary to them, we do not consider specialized DTDs.

We are currently considering the following lines of research:

(i) The construction of a framework for manipulating XML documents. This framework is intended to be a formal laboratory to test query and update languages for XML [2, 3]. We are currently implementing our update operations using the ASF+SDF [8] meta-environment. Next, we shall consider the development of an XML update language as an extension of some existing query language such as XQuery. To this end, we shall define a method to determine the update position p by the evaluation of some predicates (*e.g.*, XPath). In doing this, we intend also to investigate how the intermediate validation necessary to determine p can optimize the complete validation for the data modification operation.

(ii) The generalization of our method to treat other kinds of updates such as those that change the schema since they can help a lot the administration of a data exchange environment.

(iii) The generalization of the update process to consider "global" updates, *i.e.*, a sequence (or a set) of updates treated as one unique transaction, instead of just a single primitive update operation. In this case, we are interested in assuring 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 using a sequence of primitive operations, the document can be temporarily invalid but in the end validity is restored.

(iv) The extension of our method to deal with specialized DTDs, as well as to treat integrity constraints. Our goal is to incrementally validate updates over XML documents, taking into account both schema and key constraints (as defined in [10]). To this end, we aim at merging the method presented here with the proposal in [6]. This first extension is our way to start the investigation of how our validation process can work when XML Schema (instead of DTD) is considered.

References

[1] XML editor products. Available at http://www.perferctxml.com/soft.asp?cat=6.

[2] XML query working group. Available at http://www.w3.org/XML/Query.

[3] XUpdate - XML:DB Working draft. Available at http://www.xmldb.org/xupdate/xupdate-wd.html.

[4] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *ACM Symposium on Principles of Database System*, 2001.

[5] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, and D. Laurent. Extending tree automata to model XML validation under element and attribute constraints. In *ICEIS*, 2003.

[6] 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.

[7] V. P. Braganholo, S. B. 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.

[8] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling rewrite systems: The ASF+SDF compiler. *ACM, Transactions on Programming Languages and Systems*, 24, 2002.

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

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

[11] B. Chidlovskii. Using regular tree automata as XML schemas. In *Proc. IEEE Advances in Digital Libraries Conference*, May 2000.

[12] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Data Engineering*, 2002.

[13] B. Kane, H. Su, and E. A. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In *Proceedings of WIDM 02*, 2002.

[14] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *ACM Symposium on Principles of Database System*, pages 11–22, 2000.

[15] F. Neven. Automata, logic and XML. In *CSL'02 - Annual Conference of the European Association for Computer Science Logic (invited talk)*, 2002.

[16] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *ACM Symposium on Principles of Database System*, pages 35–46, 2000.

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

[18] D. Suciu. On database theory and XML. *SIGMOD Record*, 30(3), 2001.

[19] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26(3), 1979.

[20] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *ACM SIGMOD*. ACM, 2001.

[21] W. Thomas. Automata of infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.