Integrating Correction into Incremental Validation

Béatrice Bouchou Ahmed Cheriat Mírian Halfeld Ferrari Agata Savary

Université Francois-Rabelais de Tours, 3 place Jean Jaurès, 41000 Blois, France
{beatrice.bouchou, ahmed.cheriat, mirian.halfeld, agata.savary}@univ-tours.fr

Abstract Many data on the Web are XML documents. An XML document is an unranked labelled tree. A schema for XML documents (for instance a DTD) is the specification of their internal structure: a schema is a tree grammar, and validating a document w.r.t. a schema is done by a running of a tree automaton. Given a document, valid w.r.t. a DTD, and a sequence of updates (insertions, deletions and replacements of subtrees), we first recall how we incrementally check the validity of the resulting document. Indeed, updating a valid document requires re-checking the parts of the document concerned by the updates. Next, the core of the paper is a method to correct subtrees for which the re-validation fails: if the validator fails at node p, a correction routine is called in order to compute *corrections* of the subtree rooted at p, within a given *threshold*. Then re-validation continues. When the tree traversal is completed (i.e. all updates have been considered), the corrections generated by each call to the routine are merged, and different correction versions for the resulting document are proposed to the user. The correction routine uses *tree edit distance matrices*.

Keywords XML, DTD, updates, incremental validation, tree-to-tree edit distance, tree-to-language correction.

1 Introduction

The validation of an XML document *w.r.t* schema constraints consists in testing whether the document verifies a set of structural specifications. Supposing that updates are applied to the document, an incremental validator is the one that verifies whether the updated document complies with the schema, by validating only the parts of the document involved in the updates.

In this paper we associate the validation process with correction proposals. During the execution of our validation method, if a constraint violation is found, a correction routine is called in order to propose local solutions capable of allowing the validation process to continue. The correction is based on the notion of a tree-to-language edit distance. It relies on a threshold-controlled depth-first recursive exploration of finite state automata associated to regular expressions that determine the validity of tree nodes. Some optimizations come from the incremental aspect of the validation step. The calculation of the tree edit distance is accompanied by the generation of the corresponding update sequences.

1.1 Overview of our method

1.1.1 The incremental validation approach

The validator takes a tree automaton representing a DTD, an XML document and a sequence of updates to be performed on the document. The update sequence, resulting from a pre-processing over a set of update operations, is ordered by position, according to the document reading order. The sequence of updates is treated as one unique transaction. The incremental validation algorithm checks whether the updates should be accepted or not. It proceeds by treating the XML tree in the document order.

Example 1 Consider the XML tree of Fig. 1, valid w.r.t. some schema constraints, where update positions

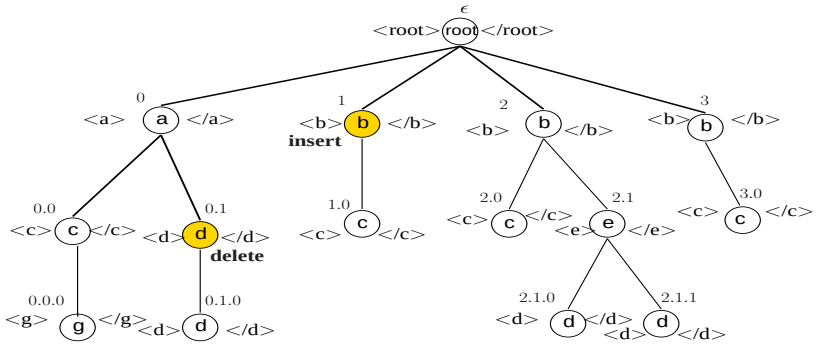

Figure 1: XML tree and update operations.

are marked. We suppose the insertion of a *locally valid* tree τ_1. When an open tag concerning an update position is reached, the incremental validation procedure takes the update operation into account (by deleting or inserting a subtree). Validation tests are performed only on parent nodes p of update positions $p.i$ (*i.e.* when the close tag of p is reached). On Fig. 1, a validation test is performed at position 0, due to the *delete* at 0.1, and at position ϵ due to the *insert* at 1. □

As detailed in section 3, a tree automaton having the same expression power as a DTD is used to express schema constraints. Thus, the incremental validation algorithm is a simplified version of the one proposed in [2, 5].

Each validation step corresponds to checking whether a word w is in a given language $L(E)$, where w results from the concatenation of the states associated to p's children and E is a regular expression defining the structure that p's children should respect. The complexity of the incremental validation algorithm for m updates is $O(m.n)$, where n is the maximum number of children of a node (fan out) in the XML tree.

1.1.2 The correction approach

If the validation test fails at node p, a correction routine is called. The goal of this routine is to correct the invalid subtree rooted at p. The correction routine assumes that p's label l is a correct one and considers corrections over its descendants. Indeed, to build a new and valid subtree with root l, the correction algorithm can propose changes on p's children. These modifications may assume changes on p's grandchildren and so forth, until reaching the leaves of the subtree. The following example illustrates our correction method.

Example 2 Consider Fig. 1, and assume that the validation test performed at node 0 fails. Let us suppose that the schema constraint associated to nodes labeled a is given by the regular expression $E_a = (cd)^* \mid m^*$, and the one associated to those labeled by d is given by $E_d = d^*$. Let $w = c$ be the word composed by the concatenation of the children of position 0, after the deletion. Clearly $w \notin L(E_a)$. The first possible correction is obtained by reinserting d at position 0.1 with cost 1. Assuming that m must respect the regular expression $E_m = g$, the second correction can be built by relabeling node at position 0.0 by m with cost 1. The third correction consists in deleting the subtree rooted at position 0.0. The total cost of this operation is 2 since it corresponds to the removal of two nodes (those at positions 0.0 and $0.0.0$). Within a threshold error $th = 2$, all three corrections can be proposed as possible solutions. □

To correct an invalid subtree at position p whose root label is assumed to be l, (*i.e.* to obtain a valid subtree whose root is labeled by l) our algorithm proposes changes that consist in: (i) changing labels of some p's descendants; (ii) deleting subtrees of the subtree rooted at p or (iii) inserting subtrees in the subtree rooted at p. In other words, given an invalid subtree T_p, we consider *update operations* capable of transforming T_p into a valid subtree T'_p. However, we want T'_p to be *close* to T_p. This means that a cost is assigned to each update operation and that the total cost of transforming T_p into T'_p is limited by a given threshold.

Our algorithm produces different local solutions for each invalid subtree. At the end of the revalidation process, global solutions are proposed to the user who can choose the best one for his update purposes.

Organization of the paper: In Section 2, we introduce some definitions necessary in the rest of the paper. In Section 3 we recall the method we use to incrementally validate a sequence of updates. In Section 4 we explain how to correct a given XML subtree (the one rooted at a node where the validation fails), and we present our correction routine. Section 5 overviews the integration of corrected trees as subtrees of the given XML tree. Section 6 concludes the paper.

2 Preliminaries

A tree t over an alphabet Σ is a function $t : dom(t) \rightarrow \Sigma \cup \{\lambda\}$. Each element from $dom(t)$ is a sequence of integers representing a node position of tree t and each node of position p is labeled with the symbol $t(p)$. The root is represented by the empty sequence ϵ and the empty tree is defined as $t = \{(\epsilon, \lambda)\}$. Let $\mathbb{N}^*$ be the set of finite words over the set of natural numbers $\mathbb{N}$ and λ be a special label. The domain $dom(t) = \{p \in \mathbb{N}^* \mid (p, l) \in t\}$ is closed under prefixes[1] and satisfies the following property: for $j \in \mathbb{N}$ and $u \in \mathbb{N}^*$; if $u.j \in dom(t)$, then for all $0 \leq i \leq j$, $u.i \in dom(t)$. The set of *leaves* of t is defined by $leaves(t) = \{u \in dom(t) \mid \neg\exists i \in \mathbb{N}$ such that $u.i \in dom(t)\}$.

Given a tree t we denote by t_p the subtree whose root is at position $p \in dom(t)$, *i.e.* $t_p = \{(r, t(r)) \mid r = p.u$ and $r \in dom(t), u \in \mathbb{N}^*\}$. Note that a subtree is not a tree (its root is not at ϵ), thus we define a tree t_p^{tree} resulting from subtree t_p as follows: $dom(t_p^{tree}) = \{s \mid p.s \in dom(t)$ and $s \in \mathbb{N}^*\}$ and for each $s \in dom(t_p^{tree})$ we have $t_p^{tree}(s) = t(p.s)$.

A tree may be changed through one or more update operations. In this paper, an update operation is seen as a high level operation defined over simple edit operations. Given a tree *t*, an *edit operation* may be applied to an *edit position* p provided that p respects some constraints depending on the type of the edit operation. In order to define edit and update operations, we use the following sets of positions:

- The *insert frontier* of t: $fr^{ins}(t) = \{u.i \notin dom(t) \mid u \in dom(t) \wedge i \in \mathbb{N} \wedge [(i = 0) \vee ((i \neq 0) \wedge u.(i-1) \in dom(t))]\}$. For an empty tree t, $fr^{ins}(t) = \{\epsilon\}$.
- The sets $DelPos_p$, $ShiftRightPos_p$, and $ShiftLeftPos_p$, defined according to a given position $p \neq \epsilon$. Let $p = u.i$, with $p \in dom(t)$, $i \in \mathbb{N}$ and $u \in \mathbb{N}^*$. We consider that $n + 1$ is the fan-out of p's father.
  - $DelPos_p = \bigcup_{k=i}^{n} \{w \mid w \in dom(t), w = u.k.u'$ and $u' \in \mathbb{N}^*\}$.
  - $ShiftRightPos_p = \bigcup_{k=i}^{n} \{w \mid w = u.(k+1).u', u.k.u' \in dom(t)$ and $u' \in \mathbb{N}^*\}$.
  - $ShiftLeftPos_p = \bigcup_{k=i+1}^{n} \{w \mid w = u.(k-1).u', u.k.u' \in dom(t)$ and $u' \in \mathbb{N}^*\}$.

$DelPos_p$ corresponds to all positions that have to be shifted left or right, in case of a node deletion or insertion at p, respectively. $ShiftRightPos_p$ is the set of all target positions resulting from shifting a part of a tree as a result of inserting a new node at p. $ShiftLeftPos_p$ is the set of all target positions resulting from shifting a part of a tree as a result of deleting a node at p.

Definition 1 - *Edit Operations:* Let ed be a tuple (op, p, l) where: $op \in \{add, remove, relabel\}$; $p = u.i$ ($i \in \mathbb{N}$ and $u \in \mathbb{N}^*$) is an edit position (defined according to the edit operation) and l is a label in Σ or null ($/$). Given a tree t, an edit operation is a function that transforms the tree t into a new tree t' ($t \xrightarrow{ed} t'$) iff the following conditions hold:

1. If $ed = (relabel, p, l)$ with $p \in (dom(t) \setminus \{\epsilon\})$ then $dom(t') = dom(t)$ and:
   $t'(p) = l$ and for all $u' \in (dom(t') \setminus \{p\})$ we have $t'(u') = t(u')$.
2. If $ed = (add, p, l)$ with $p \in ((dom(t) \cup fr^{ins}(t)) \setminus \{\epsilon\})$ then
   $dom(t') = [dom(t) \setminus DelPos_p] \cup ShiftRightPos_p \cup \{p\}$ and:

[1] The prefix relation in $\mathbb{N}^*$, denoted by $\preceq$ is defined by: $u \preceq v$ iff $u.w = v$ for some $w \in \mathbb{N}^*$. A set $dom(t) \subseteq \mathbb{N}^*$ is closed under prefix if $u \preceq v$, $v \in dom(t)$ implies $u \in dom(t)$.

• $t'(w) = t(w), \forall w \in (dom(t) \setminus DelPos_p)$.
• $t'(u.(k+1).u') = t(u.k.u')$ for each $u.(k+1).u' \in ShiftRightPos_p$ where $u' \in \mathbb{N}^*$ and $i \leq k \leq n$, with $n+1$ being the fan-out of p's father.
• $t'(p) = l$.

3. If $ed = (remove, p, /)$ with $p \in (leaves(t) \setminus \{\epsilon\})$ then $dom(t') = [dom(t) \setminus DelPos_p] \cup ShiftLeftPos_p$ and:

   • $t'(w) = t(w)$ for each $w \in (dom(t) \setminus DelPos_p)$
   • $t'(u.(k-1).u') = t(u.k.u')$ for each $u.(k-1).u' \in ShiftLeftPos_p$ where $u' \in \mathbb{N}^*$ and $(i+1) \leq k \leq n$, with $n+1$ being the fan-out of p's father.

For edit operations on position ϵ, for all $l_1 \in \Sigma \cup \{\lambda\}$, we define $\{(\epsilon, l_1)\} \overset{(relabel,\epsilon,l_2)}{\longrightarrow} \{(\epsilon, l_2)\}$ and $\{(\epsilon, l_1)\} \overset{(remove,\epsilon,/)}{\longrightarrow} \{(\epsilon, \lambda)\}$. Moreover, we define $\{(\epsilon, \lambda)\} \overset{(add,\epsilon,l)}{\longrightarrow} \{(\epsilon, l)\}$. All other edit operations are undefined. □

Intuitively, Definition 1 states that *relabel* changes the label associated with a given position p; *remove* allows the removal of a leaf and *add* allows the insertion of a single node at a position in $dom(t)$ or in fr^{ins}, except at the root of a non-empty tree.

Example 3 Consider the XML tree of Fig. 1 and the edit operation $ed = (add, 1, a)$. In this case the set of positions to be changed is $DelPos_1 = \{1, 1.0, 2, 2.0, 2.1, 2.1.0, 2.1.1, 3, 3.0\}$. The set of "new" positions is $ShiftRightPos_1 = \{2, 2.0, 3, 3.0, 3.1, 3.1.0, 3.1.1, 4, 4.0\}$. The domain $dom(T')$ of the resulting tree T' obtained after applying edit operation $ed = (add, 1, a)$ is $dom(T') = \{\epsilon, 0, 0.0, 0.1, 0.0.0, 0.1.0, 1, 2, 2.0, 3, 3.0, 3.1, 3.1.0, 3.1.1, 4, 4.0\}$. The new tree T' is shown on Fig. 2(a). □

Update operations extend edit operations by allowing the insertion, the deletion, and the replacement of subtrees. Indeed, we consider three update operations: $(insert, p, \tau)$, $(delete, p, \tau)$, and $(replace, p, \tau)$, where p is a position, and τ is a tree that must be empty for a $delete$.

Each update operation corresponds to a sequence (composition) of edit operations.

- The insertion of a tree τ at position p in a tree t is performed by adding each node of τ (one by one). To minimize the number of shifts, we add nodes of τ from its root to its leaves, and from left to right.
- The deletion of the subtree rooted at position p in a tree t is performed by removing all its nodes one by one, from leaves to root and from right to left.
- The replacement of the subtree at position p in a tree t by a tree τ can also be defined in terms of add, $remove$, and $relabel$, but it requires to introduce correction notions. Thus, it shall be done in section 4.

The definition below formalizes the link between update operations and edit operations, for insertions and deletions.

Definition 2 ***- Update Operations in terms of edit operations:*** Let upd be a tuple (op, p, τ) where: $op \in \{insert, delete\}$; p is an update position (defined according to op) and τ is a tree. Given a non empty tree t, an update operation is a partial function that transforms t into a new tree t' ($t \overset{upd}{\longrightarrow} t'$). Each update operation on t is defined as a sequence (composition) of edit operations, as follows:

1. ($t \overset{(insert,p,\tau)}{\longrightarrow} t'$), with $\tau \neq \{(\epsilon, \lambda)\}$ and p in $dom(t) \cup fr^{ins}(t)$, represents the sequence: $t = t_0 \overset{(add,p.v_1,\tau(v_1))}{\longrightarrow} t_1 \overset{(add,p.v_2,\tau(v_2))}{\longrightarrow} t_2 \cdots \overset{(add,p.v_n,\tau(v_n))}{\longrightarrow} t_n = t'$; where $v_1, \ldots, v_n$ are the positions of τ reached in its prefix order traversal and prefixed by p. If $\tau = \{(\epsilon, \lambda)\}$ then ($t \overset{(insert,p,\tau)}{\longrightarrow} t$).

2. $t \overset{(delete,p,\tau)}{\longrightarrow} t'$, and $\tau = (\epsilon, \lambda)$, with $p \in dom(t)$ represents the sequence: $t = t_0 \overset{(remove,p.v_1,/)}{\longrightarrow} t_1 \overset{(remove,p.v_2,/)}{\longrightarrow} t_2 \cdots \overset{(remove,p.v_n,/)}{\longrightarrow} t_n = t'$; where $p.v_1 \ldots p.v_n$ are the positions of t_p reached in its inverted postfix (right-to-left). □

As shown in Definition 2, each update operation can be translated into a sequence of simple edit operations. For any edit operation ed of Definition 1, we define $cost(ed) = 1$ to be the cost of performing ed. Given an update operation

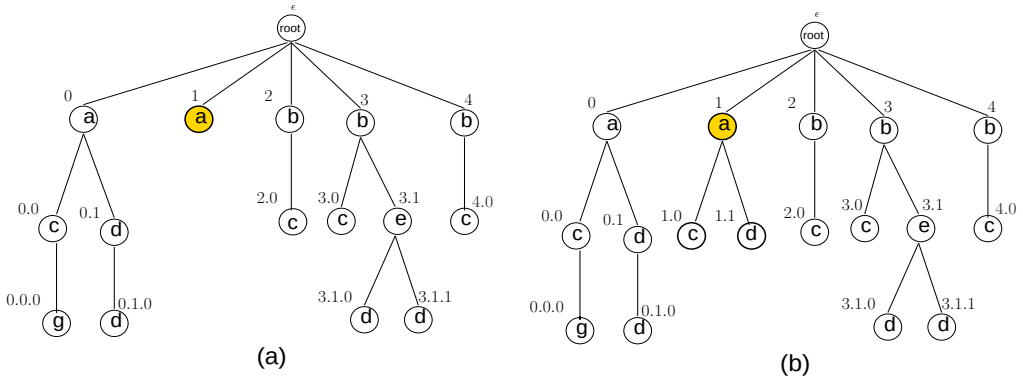

Figure 2: (a) Tree obtained by applying the edit operation $ed = (add, 1, a)$ over Fig. 1. (b) Tree obtained by applying the update operation $insert(1, \tau_1)$ over Fig. 1.

$t \xrightarrow{(upd)} t'$, equivalent to the sequence $t = t_0 \xrightarrow{ed_1} t_1 \xrightarrow{ed_2} t_2 \cdots \xrightarrow{ed_n} t_n = t'$ the cost of upd is $Cost(upd) = \Sigma_{i=1}^{n}(cost(ed_i))$. We will show in section 4 that the cost for a replacement of t_p by t'_p corresponds to the minimal distance between t_p and t'_p. We generalize the concept of update operation cost to introduce the cost of a sequence of update operations. Thus we note $t \xrightarrow{(updSeq)} t'$, to indicate $t = t_0 \xrightarrow{upd_1} t_1 \xrightarrow{upd_2} t_2 \cdots \xrightarrow{upd_n} t_n = t'$ and we define $Cost(updSeq) = \Sigma_{i=1}^{n}(cost(upd_i))$.

Example 4 Now, we consider the XML tree T of Fig. 1 and the update operation $(insert, 1, \tau_1)$ with τ_1=$\{(\epsilon, a), (0, c), (1, d)\}$. This update operation is translated into the sequence $(add, 1.\epsilon, a)$, $(add, 1.0, c)$, $(add, 1.1, d)$. The resulting tree is shown on Fig. 2(b). □

We can now define the notion of distances between two trees and between a tree and a tree language. These notions are important for our correction algorithm in order to find valid trees as close as possible to the invalid one.

Definition 3 - *Tree distances:* Let t and t' be trees. Let $\mathcal{S}$ be a set of update sequences each of which is capable of transforming t into t'. The distance between t and t' is defined by $dist(t, t') = \min_{S_i \in \mathcal{S}} \{Cost(S_i)\}$. The distance between a tree t and a tree language $\mathcal{L}$ is defined by: $DIST(t, \mathcal{L}) = \min_{t' \in \mathcal{L}} \{dist(t, t')\}$. □

3 Incremental validation

In [12] incremental validation of one edit operation is considered. We are interested in multiple updates, *i.e.*, we suppose an input file containing a sequence of update operations. Before actually replacing a valid XML tree with the one resulting from the given update sequence, updates must be validated, *i.e.*, the validity of the parts of the original document directly affected by the updates is checked [6, 3, 2, 5].

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. In other words, as a valid document is transformed by a sequence of operations, the document can be temporarily invalid, but in the end validity is restored.

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, the update operation op and the subtree τ to be *inserted*. In this paper we assume that $UpdateTable$ is the result of a preprocessing of a sequence of updates required by a user. In the resulting $UpdateTable$ the following properties hold:

P1 - An update on a position p excludes updates on descendants of p. In other words, there are, in $UpdateTable$ no two update positions p and p' such that $p \preceq p'$.

P2 - If a position p appears more than once in $UpdateTable$ then one of the operations involving p can be *replace* or *delete*, but all others are *insert*.

P3 - Updates in an $UpdateTable$ are ordered by position, according to the document order.
P4 - An update position in an $UpdateTable$ always refers to the original tree.

These properties impose some restrictions to our context: we assume that $UpdateTable$ is an update list resulting from a pre-processing over a set of updates. This pre-processing establishes some priorities (as in **P1**) and enhances the XML tree traversal in the document order.

It is a well known fact that XML schemas can be represented by tree automata [7, 10]. In this paper, we consider a bottom-up unranked tree automaton.

Definition 4 - Unranked tree automaton: An unranked tree automaton 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 $a, E \rightarrow q$ where *(i)* $a \in \Sigma$; *(ii)* E is a regular expression over Q and *(iii)* $q \in Q$. □

Example 5 Consider again Example 1: The tree automaton $\mathcal{A}$ representing the schema to be verified is the following: $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where,

- $\Sigma = Q = \{a, b, c, d, e, f, g, m, root\}$,
- $Q_f = \{root\}$ and
- $\Delta = \{$

$$root, (ab^*) \rightarrow root$$
$$a, ((cd)^*|m^*) \rightarrow a$$
$$b, (ce^*) \rightarrow b$$
$$c, (g^*f?) \rightarrow c$$
$$d, (d^*) \rightarrow d$$
$$e, (d^*) \rightarrow e$$
$$m, (g) \rightarrow m$$
$$g, () \rightarrow g$$
$$f, () \rightarrow f$$

$\}$ □

The execution of a tree automaton $\mathcal{A}$ over an XML tree corresponds to the validation of the XML document *w.r.t.* to the schema constraints represented by $\mathcal{A}$. 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 associated to a state q_p (only one state for a DTD). The assignment $r(p) = q_p$ is done by verifying whether the following constraints, imposed to p's children, are respected:

1. $T(p) = a$.
2. There exists a transition rule $a, E \rightarrow q$ in $\mathcal{A}$.
3. There is a word $w = q_1 \ldots q_n$ in $L(E)$ such that $q_1, \ldots, q_n$ are the states associated to children of p.

A run r is *successful* if $r(\epsilon)$ is associated to one final state. A tree T is *valid* if a successful run exists on it. A tree is *locally valid* if $r(\epsilon)$ is a state that belongs to $\mathcal{A}$ but that is not a final state. This notion is useful in an update context [6], since inserted trees must be locally valid.

Our incremental validation algorithm checks whether the updates should be accepted or not. It proceeds by treating the XML tree in document order. The following example illustrates our validation method.

Example 6 Consider again Example 1, with tree automaton $\mathcal{A}$ of Example 5. The original tree (Fig. 1.a) is valid w.r.t. $\mathcal{A}$, and subtree τ_1 being inserted is *locally valid w.r.t.* $\mathcal{A}$. The incremental validation is performed as follows:

- 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 rule in $\mathcal{A}$ associated to the parent of the update position. This test is performed when the close tag (position 0) is found. Notice that to perform this test we need to know the state assigned to position 0.0, but we do not need to go below this position (those nodes, when they exist, are skipped).

- When the open tag <b> (position 1) is reached, the insertion operation is taken into account and the new, locally valid subtree τ_1 is inserted. This implies that right-hand side siblings of position 1 must be shifted to the right.

  We continue by reading nodes at (original) positions 1, 2 and 3. Notice that we can skip all nodes below positions 2 and 3, since there is no update position below these points.

- The close tag </root> activates a validity test that takes into account the root's children. □

In [2] we present a complete version of our incremental validation algorithm. During its execution, the path from the root to the *current* position p defines a borderline between nodes already treated and those not already considered. For each current position p, the validation algorithm builds a list, called *SAC*(p) (State Attribution for the Children of p): when the open tag at position p is reached, *SAC*(p) is initialized; then the traversal continues (*i.e.* subtrees of p are visited while updates are considered) and each state assigned to a p child is appended to the list *SAC*(p). When the closing tag at position p is reached, *SAC*(p) contains the states associated to each one of its children.

Our incremental validation algorithm processes the XML document *à la* SAX [1]. 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.

- **replace**: The subtree rooted at p is changed to a new one. A state q_p indicates whether the locally valid subtree T_p is allowed at this position. The state q_p is appended to the list $SAC(father(p))$ to form the sequence that shall contain the states associated to each sibling of p. The (original) subtree rooted at p is skipped.

- **insert**: The validation process is similar to the previous case for each insertion at p, 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.

While performing validation tests, a new updated XML tree is 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 the current version. Otherwise, as detailed in next sections, each time a validation test fails at node p, a correction routine can be called in order to compute local corrections for subtree T_p.

4 Correcting an invalid tree

Let $\mathcal{L}$ be the tree language defined by the tree automaton $\mathcal{A}$. Let $\mathcal{L}_l \subseteq \mathcal{L}$ be the tree language defined by transition rules in $\mathcal{A}$ that contains all trees whose root is labeled with l.

Given an XML tree T, we assume that the validation fails at position p (*i.e.* $T_p^{tree} \notin \mathcal{L}_l$) and we propose a routine capable of correcting T_p. Our correction routine assumes that p has the correct label l and considers changes on p's descendants. It takes the language $\mathcal{L}_l$ and the tree $T_p^{tree} \notin \mathcal{L}_l$ as input. Then our algorithm looks for new trees $T_p^{tree'} \in \mathcal{L}_l$ such that $dist(T_p^{tree}, T_p^{tree'})$ is minimal. Each new tree $T_p^{tree'}$ can then be expressed as a new subtree T'_p of T.

Example 7 As seen in Example 2 our validation procedure fails at position 0 of the XML tree of Fig. 1 due to the deletion performed at position 0.1. At this point our correction routine can be activated. To perform corrections this routine considers the tree language $\mathcal{L}_a$ defined by the tree automaton whose transition rules are those shown in Example 2 but with $Q_f = \{a\}$. Clearly, as $T_0^{tree} = \{(\epsilon, a), (0, c), (0.0, g)\}$, we have $T_0^{tree} \notin \mathcal{L}_a$. □

REMARK: In the rest of this paper, in order to simplify notation, we use t to denote the tree T_p^{tree} and t' to denote the tree $T_p^{tree'}$.

4.1 Definition of correction

Our algorithm extends the ones proposed both in [13] and in [11]: it uses an edit distance matrix H_DIST. Each element $H\_DIST[i, j]$ contains the edit distance between *partial* trees $t\langle i\rangle$ and $t'\langle j\rangle$. A (partial) tree $t\langle i\rangle$ is composed by a root and subtrees $t_0, \ldots t_i$ (Fig. 3).

The matrix is calculated column per column. Each new element is deduced from its three top-left-hand neighbor elements which have been calculated previously ($H\_DIST[i-1, j]$, $H\_DIST[i, j-1]$ and $H\_DIST[i-1, j-1]$).

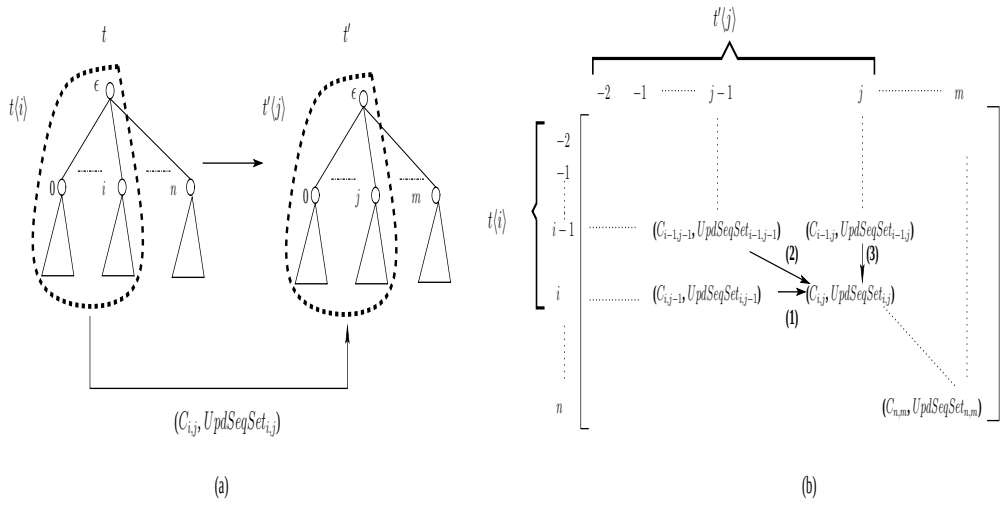

Figure 3: (a) Two (partial) trees $t<i>$ and $t'<j>$. (b) Matrix H_DIST for trees t and t', and computation of $H\_DIST[i,j]$.

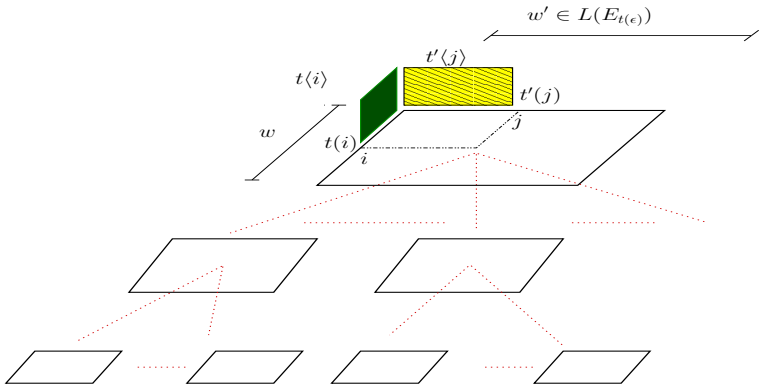

Figure 4: Illustration of the reasoning that guides the computation of edit distances between a tree $t \notin \mathcal{L}$ and a tree $t' \in \mathcal{L}$.

Each $H\_DIST[i,j]$ stores a cost and a set of update sequences obtained from its neighbors and from the updates that are necessary to: delete the subtree t_i (Fig. 3(b) edge (3) in the matrix), insert a new subtree t'_j (Fig. 3(b) edge (1)), or replace the subtree t_i by a new subtree t'_j (Fig. 3(b) edge (2)).

The computation of each $H\_DIST[i,j]$ follows a "horizontal-vertical" reasoning illustrated on Fig. 4. Recall that t is the tree to be corrected and t' is a correction candidate we have to construct so that its root's children respect the regular expression $E_{t(\epsilon)}$. We consider the word $w \notin L(E_{t(\epsilon)})$ composed by the concatenation of the states associated to the children of t's root. On the "horizontal" plan, our correction algorithm follows the one of [11], *i.e.* it tries to build new words $w' \in L(E_{t(\epsilon)})$, as close as possible to w via a threshold-controlled depth-first exploration of the Finite State Automaton (FSA) corresponding to $L(E_{t(\epsilon)})$. Each time a transition is followed a new column is added to the distance matrix. If a final state is reached and the current bottom-right hand element of the matrix does not exceed a given threshold then the current word w' is a valid correction candidate. Otherwise we may either continue to the next transition (if any element of the current column does not exceed the threshold) or backtrack, delete the current column, and explore a different path.

However, since in our approach each "character" in w is the root of a subtree, in order to obtain w', we also have to work in the "vertical" direction. Indeed, each $w'[j]$ has to yield a subtree $t'_j \in \mathcal{L}_{t'(j)}$, with root label $t'_j(\epsilon) = w'[j] = t'(j)$.

Therefore, while on the "horizontal" plan of Fig. 4 we deal with a matrix that associates a word w to a regular language, on the "vertical" plan, we deal with a tree t_i. In order to correct t_i, not only do we have to build a matrix for its root's sons, but possibly also for the grandchildren, and so on, as shown on Fig. 4. To conclude, as formalized in the following definition, matrix H_DIST stores the edit distance between the invalid tree t that we are trying to correct and its partial correction candidates, together with the set of update sequences necessary to transform t into each of the candidates.

Definition 5 - *Edit Distance Matrix:* Let t be a tree and let $\mathcal{L}$ ($\mathcal{L} = \mathcal{L}_{t(\epsilon)}$) be a tree language such that $t \notin \mathcal{L}$. An edit distance matrix, denoted $H\_DIST_{t,t'}$ (or just H_DIST when no confusion is possible), stores the edit distance between tree t and a partial tree $t' \in \mathcal{L}$.
The matrix H_DIST is a two dimensional matrix with indices starting from -2. Each element $H\_DIST[i,j] = (C, updSeqSet)$ where C is a cost and ***updSeqSet*** is a set of update sequences having cost C, as defined below.

1. Initializations

- For each row i: $H\_DIST[i,-2] = (\infty, \top)$ and for each column j: $H\_DIST[-2,j] = (\infty, \top)$, where $\top$ is a theoretical infinite update sequence of cost ∞.

- Element $H\_DIST[-1,-1]$ is calculated as follows :

$$\begin{array}{ll} (0, \{[]\}) & \textbf{if } t(\epsilon) = t'(\epsilon) \\ (1, \{[(relabel, \epsilon, t'(\epsilon))]\}) & \textbf{if } t(\epsilon) \neq \lambda,\ t'(\epsilon) \neq \lambda \\ & \text{and } t(\epsilon) \neq t'(\epsilon) \\ (1, \{[(add, \epsilon, t'(\epsilon))]\}) & \textbf{if } t(\epsilon) = \lambda \text{ and} \\ & t(\epsilon) \neq t'(\epsilon) \\ (1, \{[(remove, \epsilon, /)]\}) & \textbf{if } t'(\epsilon) = \lambda \text{ and} \\ & t(\epsilon) \neq t'(\epsilon) \end{array}$$

2. For all $j \geq 0$, and for all $i \geq 0$:

- $C = min(C_{replace}, C_{insert}, C_{delete})$ with

  $-\ C_{replace} := H\_DIST[i-1, j-1].C + dist(t_i, t'_j)$

  $-\ C_{insert} := H\_DIST[i, j-1].C + dist(\{(\epsilon, \lambda)\}, t'_j)$

  $-\ C_{delete} := H\_DIST[i-1, j].C + dist(t_i, \{(\epsilon, \lambda)\})$

- $updSeqSet = \{Seq \mid Seq \in \mathcal{S}_{replace} \cup \mathcal{S}_{insert} \cup \mathcal{S}_{delete}$ where *Cost*$(Seq) = C\}$ where:

  $-\ \mathcal{S}_{replace}$ is the Cartesian product of ***(i)*** the set of update sequences transforming $t\langle i-1\rangle$ into $t'\langle j-1\rangle$ and ***(ii)*** the set of update sequences transforming t_i into t'_j. Formally, $\mathcal{S}_{replace} = \mathcal{S}'_r \times \mathcal{S}''_r$ where:

  $\star\ \mathcal{S}'_r = H\_DIST[i-1, j-1].updSeqSet$

  $\star\ \mathcal{S}''_r = \{Seq'' \mid t_i \overset{(Seq'')}{\longrightarrow} t'_j$ and $Cost(Seq'') = dist(t_i, t'_j)\}$

  $-\ \mathcal{S}_{insert}$ is the Cartesian product of ***(i)*** the set of update sequences transforming $t\langle i\rangle$ into $t'\langle j-1\rangle$ and ***(ii)*** the set of update sequences inserting t'_j. Formally, $\mathcal{S}_{insert} = \mathcal{S}'_i \times \mathcal{S}''_i$ where:

  $\star\ \mathcal{S}'_i = H\_DIST[i, j-1].updSeqSet$

  $\star\ \mathcal{S}''_i = \{Seq'' \mid \{(\epsilon, \lambda)\} \overset{(Seq'')}{\longrightarrow} t'_j$ and $Cost(Seq'') = dist(\{(\epsilon, \lambda)\}, t'_j)\}$

  $-\ \mathcal{S}_{delete}$ is the Cartesian product of ***(i)*** the set of update sequences deleting t_i and ***(ii)*** the set of update sequences transforming $t\langle i-1\rangle$ into $t'\langle j\rangle$. Formally, $\mathcal{S}_{delete} = \mathcal{S}''_d \times \mathcal{S}'_d$ where:

  $\star\ \mathcal{S}'_d = H\_DIST[i-1, j].updSeqSet$

  $\star\ \mathcal{S}''_d = \{Seq'' \mid t_i \overset{(Seq'')}{\longrightarrow} \{(\epsilon, \lambda)\}$ and $Cost(Seq'') = dist(t_i, \{(\epsilon, \lambda)\})\}$ □

Notice that the first column of H_DIST ($H\_DIST[i,-2]$) stores very great costs (∞) and the second one ($H\_DIST[i,-1]$) stores the cost and update sequences needed to delete the subtrees $t_0, \ldots t_i$. In a similar way, the first row contains very great costs and the second one ($H\_DIST[-1,j]$) contains the cost and update sequences needed to insert the subtrees $t'_0, \ldots t'_j$. According to [13], the bottom right-hand position of a matrix H_DIST contains the edit distance between t and a tree $t' \in \mathcal{L}$. In our approach it is accompanied with all minimal cost update sequences transforming t into t'.

Example 8 As shown in Example 7 the validation procedure fails at position 0 and the correction routine is called for T_0^{tree} with $\mathcal{L}_a$ and $th = 2$. The finite-state automaton FSA_E corresponding to the regular expression of label a is shown on Fig. 5. The routine performs the following steps.

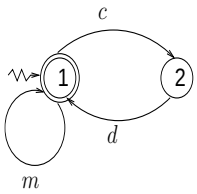

Figure 5: Finite state automaton FSA_E associated to regular expression $E_a = (cd)^*|m^*$.

Step 1: The tree t being considered for correction is $\{(\epsilon, a), (0, c), (0.0, g)\}$. The matrix H_DIST_1 is initialized as specified in Definition 5. The initial (and finite) state 1 in FSA_E becomes the current state. Column -2 imposes a matrix border, necessary in the computation. Notice that line -1 represents the root node a whose children are represented by lines $i \geq 0$ of H_DIST_1 .

Initial matrix H_DIST_1:

			a
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
a	-1	$(\infty, \top)$	$(0, \{[]\})$
c	0	$(\infty, \top)$	

As we assume that the root label of T_0^{tree} does not change (it stays as a), element $H\_DIST_1[-1, -1] = (0, \{[]\})$. To compute $H\_DIST_1[0, -1]$ we consider the three top-left-hand neighbors already calculated. In our case, neighbors $H\_DIST_1[0, -2]$ and $H\_DIST_1[-1, -2]$ are discarded since their cost exceeds the threshold. To compute $H\_DIST_1[0, -1]$ from $H\_DIST_1[-1, -1]$, we should assume the deletion of the subtree rooted at position 0 of T_0^{tree}. This deletion is treated by calling recursively the correction routine (*i.e.* by starting Step 2). Inputs are: the tree obtained from the subtree rooted at position 0 of T_0^{tree} and the tree language $\mathcal{L}_\lambda$ containing only the empty tree.

Step 2: We go down in the vertical direction (Fig. 4) and start the construction of a new matrix. The tree t being considered for correction is $\{(\epsilon, c), (0, g)\}$. The matrix H_DIST_2 is initialized as specified in Definition 5.

Initial matrix H_DIST_2:

			λ
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
c	-1	$(\infty, \top)$	$(1, \{(remove, \epsilon, /)\})$
g	0	$(\infty, \top)$	

Now $H\_DIST_2[-1, -1] = (1, \{(remove, \epsilon, /)\})$ because it is calculated by considering the removal of $t(\epsilon) = c$ (Definition 5). Similarly to Step 1, element $H\_DIST_2[0, -1]$ is computed from $H\_DIST_2[-1, -1]$ by assuming the deletion of the subtree $\{(0, g)\}$ of t. This deletion is treated by calling recursively the correction routine (*i.e.* by starting Step 3).

Step 3: We go down in the vertical direction (Fig. 4) and start the construction of a new matrix. The tree t being considered for correction is $\{(\epsilon, g)\}$. The matrix H_DIST_3 is initialized as specified in Definition 5 and as t has only one root node, we have

Matrix H_DIST_3:

			λ
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
g	-1	$(\infty, \top)$	$(1, \{(remove, \epsilon, /)\})$

Return to Step 2: Coming back (from the recursive call) to H_DIST_2 the result obtained in Step 3 corresponds to the deletion of the node at position 0 of the tree t considered in Step 2. The deletion of t's root is also considered. Thus, we have:

The bottom right-hand corner of H_DIST_2 contains the result to be sent back to Step 1.

Return to Step 1: Coming back (from the recursive call) to H_DIST_1, the result obtained in Step 2 corresponds to the deletion of the subtree rooted at position 0.0 of the tree t considered in Step 1. Thus, we have:

The current state in FSA_E is a final one and the bottom right-hand corner element of H_DIST_1 does not exceed the threshold 2. Thus, according to [11] and [13], this element contains a valid correction candidate. It is equivalent to the deletion of the subtree rooted at position 0.0 of Fig. 1. Notice that, in this case the word w' of Fig. 4 is the empty word which belongs to $L(E_a)$. To find more solutions within the threshold 2, we consider other words $w' \in L(E_a)$. We follow an outgoing transition in FSA_E, for instance the transition labeled with m, and we add a new column in H_DIST_1. The correction routine yields the following matrix H_DIST where the proposed solution consists in relabeling the node at position 0.0 of Fig. 1 (from c to m). As its cost (1) stays within the threshold and the current state (1) is final, we get another correction candidate.

Matrix H_DIST_2:

			λ
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
c	-1	$(\infty, \top)$	$(1, \{(remove, \epsilon, /)\})$
g	0	$(\infty, \top)$	$(2, \{(remove, 0, /)(remove, \epsilon, /)\})$

Matrix H_DIST_1:

			a
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
a	-1	$(\infty, \top)$	$(0, \{[]\})$
c	0	$(\infty, \top)$	$(2, \{(remove, 0.0.0, /)(remove, 0.0, /)\})$

			a	m
		-2	-1	0
	-2	$(\infty, \top)$	$(\infty, \top$	$(\infty, \top)$
a	-1	$(\infty, \top)$	$(0, \{[]\})$	$(2, \{(add, 0.0, m)(add, 0.0.0, g)\})$
c	0	$(\infty, \top)$	$(2, \{(remove, 0.0.0, /)(remove, 0.0, /)\})$	$(1, \{(relabel, 0.0, m)\})$

Notice that the element $H\_DIST[-1, 0]$ is obtained by considering the edge(1) of Fig. 3(b). Indeed it corresponds to the computation necessary to transform subtree $\{(0, a)\}$ into subtree $\{(0, a), (0.0, m), (0.0.0, g)\}$. Element $H\_DIST[0, 0]$ stores the minimum computation necessary to transform subtree $\{(0, a), (0.0, c), (0.0.0, g)\}$ into subtree $\{(0, a), (0.0, m), (0.0.0, g)\}$. The choice is done by considering the three cases explained in this section: Following edge(1) of Fig. 3(b), our algorithm inserts subtree $\{(0.0, m), (0.0.0, g)\}$ after deleting subtree $\{(0.0, c), (0.0.0, g)\}$ (as indicated in $H\_DIST[0, -1]$). In this case, the cost is 4. Following edge(2) of Fig. 3(b), our algorithm replaces subtree $\{(0.0, c), (0.0.0, g)\}$ by subtree $\{(0.0, m), (0.0.0, g)\}$. This is done by relabelling c by m with cost 1. Following edge(3) of Fig. 3(b), our algorithm deletes subtree $\{(0.0, c), (0.0.0, g)\}$ after inserting subtree $\{(0.0, m), (0.0.0, g)\}$ (as indicated in $H\_DIST[-1, 0]$). In this case, the cost is 4. □

4.2 Correction algorithms

The algorithm whose goal is to compute valid candidate trees within a given threshold is called *CorrectionRoutine* (see Algorithm 1). It receives as input a tree T_p^{tree} issued from an XML tree T whose validation failed at p. Before calling *CorrectSubtree*, *CorrectionRoutine* first uses procedure *intializeMatrix* (Algorithm 1, line 1), which initializes the first two columns of H_DIST according to Definition 5. It recursively calls *CorrectionRoutine* to compute the cost and the update sequence necessary to delete each subtree t_i. That is because deleting a subtree is equivalent to correcting it *w.r.t* the empty tree.

CorrectionRoutine contains a recursive procedure, called *CorrectSubtree* (Algorithm 1, line 5), which receives in its first call the initial state s_0 of FSA_E, an empty word w', the initialized matrix H_DIST, and an empty list $LCand$. It returns its solutions in $LCand$ (Algorithm 1, line 6).

In *CorrectSubtree* (see Algorithm 2), the finite state automaton FSA_E is explored in the depth-first order. Each time a transition is followed, on the "horizontal plan", the current word w' is extended (Algorithm 2, line 4) and a new column is added to H_DIST (Algorithm 2, line 5). This means that, on the "vertical plan", t' is also extended. That allows to check if t' may still lead to a candidate remaining within the threshold. If it does, the path is followed via a recursive call (Algorithm 2, line 7), otherwise the path gets cut off.

Algorithm 1 - *The main method to correct an invalid subtree*

Function CorrectionRoutine($t,\ p,\ a,\ th$)

Input:

(i) t: $t = T_p^{tree} \notin \mathcal{L}_{t(\epsilon)}$

(ii) p: position in $dom(T)$ such that $T_p^{tree} = t$

(iii) a: string (*root label of a new subtree which will replace the subtree* t)

(iv) th: integer corresponding to the error threshold

Output:

(i) $LCand$: set of tuples $(C, updSeqSet)$ with *updSeqSet* the set of update sequences having cost C

Local variables:

(i) H_DIST: edit distance matrix between two trees

(ii) FSA_E: deterministic FSA (FSA_E=$\langle Q, \Sigma, \delta, s_0, F \rangle$)

(iii) $w,\ w'$: words of states

1. $H\_DIST := InitializeMatrix$(t, p, a, th)
2. $w' := \epsilon$
3. $FSA_E := getFSA(a)$
4. $w := getSons(t, \epsilon)$
   // w is the concatenation of sons labels of the node at position ϵ in t
5. $LCand := CorrectSubtree$(t, w, w', th, H_DIST, FSA_E, s_0, $LCand$)
6. $return(LCand)$ □

In each case the transition is finally backed off (Algorithm 2, lines 8 and 9) and a new transition outgoing from the same state is tried out. If we arrive at a final state and the distance from t' to t does not exceed the threshold, then t' is a valid candidate that is inserted to the list $LCand$ of all candidates found so far (Algorithm 2, lines 1 and 2).

Function $Cuted$ allows to compute the cut-off edit distance [11] between t and t'. It checks if all elements of the current column (Algorithm 2, line 6) in the matrix exceed the threshold. If they do, there is no chance for t' to be a partial correction within the threshold.

Function *AddNewColumn* is used to compute all columns but the two first ones which are calculated by function *InitializeMatrix*.

In *AddNewColumn* (see Algorithm 3), each matrix element is deduced from its three upper left-hand neighbors as stated in Definition 5.

In case of insertion, the subtree t'_j is not known in advance although its root's label is (see input parameter a). Thus its insertion cost and update sequence is the one needed to create a minimal valid tree having a in its root. That is equivalent to correcting an empty tree *w.r.t* $\mathcal{L}_a$ by calling *CorrectionRoutine* (Algorithm 3, line 3). As only the minimal subtrees are considered we choose the first solution set (Algorithm 3, line 4). Each minimal solution must then be prefixed by the position where is should be inserted. Remember that in our model insertions take place left to the symbols at the positions concerned. Thus if we calculate row -1 and row i, the corresponding insertions appear with positions 0 and $i+1$, respectively (Algorithm 3, lines 5 and 9).

In case of deletion, the cost and update sequence are the ones needed to delete the subtree t_i. That is equivalent to converting this subtree into an autonomous tree and correcting it *w.r.t* the empty language $\mathcal{L}_\lambda$ (Algorithm 3, line 10). Only one minimal solution is possible (leaf by leaf deletion from right to left and bottom up, see Algorithm 3, line 11). This solution must be prefixed by the position i where the deletion is to take place (Algorithm 3, line 12).

In case of replacement, the cost and update sequences are the ones needed to correct t_i (converted into an autonomous tree) with respect to $\mathcal{L}_a$, also by calling *CorrectionRoutine* (Algorithm 3, line 13). Only the minimal solutions are considered thus we take the first solution set and prefix it by the position of the replacement i (Algorithm 3,

Algorithm 2 - Recursive Correction of a Subtree

Procedure CorrectSubtree(t, p, w, w', th, H_DIST, FSA_E, s, $LCand$*)*

Input

(i) t: $t = T_p^{tree} \notin \mathcal{L}_{t(\epsilon)}$

(ii) p: position in $dom(T)$ such that $T_p^{tree} = t$

(iii) w: invalid word (concatenation of states associated to t's root's children)

(iv) th: integer corresponding to the error threshold

(v) FSA_E: deterministic FSA for E appearing in $t(\epsilon), E \rightarrow q_{t(\epsilon)}$

(vi) s: current state in FSA_E

Input/Output

(i) w': partial valid word (concatenation of states associated to root's children in t')

(ii) H_DIST: edit distance matrix between t and t'

(iii) $LCand$: set of tuples $(C, updSeqSet)$ with *updSeqSet* the set of update sequences having cost C.

1. **if** s is a final state in FSA_E and $H\_DIST[|w|-1, |w'|-1].C \leq th$

   // New candidate found within the threshold

2. $LCand$:= $SortInsertion$(*LCand*, $H\_DIST[|w|-1, |w'|-1]$)
3. **for each** transition $\delta(s, a') = s'$ in FSA_E **do**
4. $w' = w'.a'$
5. $H\_DIST := AddNewColumn$(*t, p, w, w', th, H_DIST,* a')
6. **if** ($Cuted$(*w, w', th, H_DIST*) $\leq th$)
7. *CorrectSubtree*(*t, p, w, w', th, H_DIST,* FSA_E, s', $LCand$)
8. $H\_DIST := DelLastCol(H\_DIST)$
9. $w' := DelLastSymbol(w')$ □

lines 14 and 15).

Once all three hypotheses have been analyzed the full update sequences are constructed by combining (by a Cartesian product) sequences for smaller partial trees and those for the most recent subtrees (Algorithm 3, lines 16 through 18). Finally, we calculate the minimum possible cost (Algorithm 3, line 19), and we select all the candidates having this cost (Algorithm 3, line 20). These two data (minimum cost and set of minimal sequences) yield the current element $H[i, j]$.

Example 9 Consider the XML tree T of Fig.1 after the first update operation. During the correction of $t = T_0^{tree}$ *CorrectionRoutine* returns $LCand = \{(1, [(insert, 0.1, d)]), (1, [(relabel, 0.0, m)]), (2, [(remove, 0.0.0, /), (remove, 0.0, /)])\}$. The first element of $LCand$ (with cost 1) is found by computing matrix $H\_DIST_{\{(\epsilon,\lambda)\},\{(\epsilon,d)\}}$, the second one (with cost 1) by computing $H\_DIST_{T_{0.0}^{tree},\{(\epsilon,m),(0,g)\}}$, and the third one (with cost 2) is found by computing $H\_DIST_{T_{0.0}^{tree},\{(\epsilon,\lambda)\}}$. □

Theorem 1 *Let $\mathcal{L}$ be a local tree language and $t \notin \mathcal{L}$. The algorithm CorrectionRoutine is correct and complete, i.e. it computes all the candidates $t' \in \mathcal{L}$ such that $dist(t, t') = DIST(t, \mathcal{L})$ if $DIST(t, \mathcal{L}) \leq th$.* □

PROOF

In order to argue for the correctness we have to show that each solution t' proposed is valid and does not exceed the threshold th, *i.e.* $t' \in \mathcal{L}$ and $dist(t, t') \leq th$.

Let's suppose that t' is of depth 0 (it contains only one node). That means that the edit distance matrix between t and t' contains only two columns -2 and -1. Thus, candidate t' has been added to the list of solutions (in Algorithm 2 line 2) before having entered the loop in lines 3 through 9. This is possible only if, in the finite-state automaton corresponding to the root of t', the initial state is also a finite state. Thus, t' (with no children) is valid. Moreover, we have accepted t' only if the bottom right-hand element of the matrix does not

Algorithm 3 - ***Add a new column to an edit distance matrix***

Function AddNewColumn(t, p, w, w', th ,H_DIST, a)

Input:

(i) t: $t = T_p^{tree} \notin \mathcal{L}_{t(\epsilon)}$

(ii) p: position in $dom(T)$ such that $T_p^{tree} = t$

(iii) w: invalid word (concatenation of states associated to t's root's children)

(iv) w': partial valid word (concatenation of states associated to root's children in t')

(v) th: integer corresponding to the error threshold

(vi) H_DIST: edit distance matrix between t and t'

(vii) $a \in \Sigma$: label corresponding to the current column

Result: H_DIST

1. $j := width(H\_DIST)$-2
2. H_DIST[-2, j] := $(+\infty, \top)$
3. $LCand$:= **CorrectionRoutine**$(\{(\epsilon, \lambda)\}, p, a, th)$
4. $(CIns_{min}, SSIns_{min}) := LCand[0]$
   // the first element of list $[(cost_1, \{Seq_1^1, Seq_2^1, \ldots, Seq_k^1\}), \ldots, (cost_n, \{Seq_1^n, Seq_2^n, \ldots, Seq_k^n\})]$
5. $SS0 := SetAddPrefix(SSIns_{min}, 0)$ // we insert at first position '0'
6. H_DIST[-1, j] := (H_DIST[-1, j-1].$C + CIns_{min}$, $\{s' | s''$=$s \times s'$, $s \in H\_DIST$[-1, j-1].S, $s' \in SS0\}$)
7. **for** each (i=0 to $|w|$-1) do
8. $p' := concat(p, i)$

   // 1^{st} case: by insertion
9. $C_I := H\_DIST[i, j\text{-}1].C + CIns_{min}$; $SSins := SetAddPrefix(SSins_{min}, i+1)$

   // 2^{nd} case: by deletion
10. $LCand$:= **CorrectionRoutine**$(getSubTree(t, i), p', \lambda, th)$
11. $(Cdel, SSdel) := (LCand[0].C, LCand[0].S)$
12. $SSdel := SetAddPrefix(SSdel, i)$; $C_D := H\_DIST[i-1, j].C + Cdel$

    // 3^{rd} case: by replacement
13. $LCand$:= **CorrectionRoutine**$(getSubTree(t, i), p', a, th)$
14. $(Crep, SSrep) := (LCand[0].C, LCand[0].S)$; $C_R := H\_DIST[i\text{-}1, j\text{-}1].C + Crep$
15. $SSrep := SetAddPrefix(SSrep, i)$

    //comparison between the 3 cases
16. $s_I = \{s | s = s' \times s''$, $s' \in H\_DIST[i, j\text{-}1].S$, $s'' \in SSins\}$
17. $s_D = \{s | s = s'' \times s'$, $s'' \in SSdel$, $s' \in H\_DIST[i\text{-}1, j].S\}$
18. $s_R = \{s | s = s' \times s''$, $s' \in H\_DIST[i\text{-}1, j\text{-}1].S$, $s'' \in SSrep\}$
19. $H\_DIST[i, j].C$:= min$\{C_I, C_D, C_R\}$
20. $H\_DIST[i, j].S := \{s | s \in s_x, x \in \{I, D, R\}$ and $C_x = H\_DIST[i, j].C\}$
21. $return(H\_DIST)$ □

exceed th (line 1). According to [13], this element represents $dist(t, t')$.

Let's suppose that t' is of depth 1 (consists of a root and a set of leaves). This candidate (or more precisely, the set of updates needed to obtain it) is added to the list of solutions also in Algorithm 2 line 2. At this moment, we have reached a finite state of the local automaton, which means that the leaves of t' respect the regular grammar attributed to the root of t'. Thus t' is a valid tree. Moreover t'

remains within the threshold, similarly to the case above (depth 0).

Let's now suppose that any solution of depth e with $1 \leq e < d$ is correct. We will prove that each solution t' of depth d is correct. In order for t' to be valid, two conditions are sufficient : (i) each subtree of the root of t' is valid, (ii) the sons of the root of t' respect the regular expression associated to the root. The first condition is true as supposed above (they are of depths $e_i < d$). The second condition is true because we add t' to the list of solutions only if we have reached a final state of the automaton attributed to the root of t' (Algorithm 2 line 2). Moreover, the second condition of Algorithm 2 line 1 ensures that $dist(t, t') \leq th$. By induction, all solutions generated by our algorithm are correct.

We need now to prove that each candidate minimal within the threshold th is found by our algorithm, *i.e.* if $DIST(t, \mathcal{L}) \leq th$ and $dist(t, t') = DIST(t, \mathcal{L})$ then t' is found. Let's suppose that there exists a tree t' such that $dist(t, t') = DIST(t, \mathcal{L}) \leq th$, and t' is *not* found by our algorithm.

Let's suppose that t' is of depth 0 (contains only one node). If t' is valid than the label of its root is the finite state of the *tree automaton* associated to the language $\mathcal{L}_{t(\epsilon)}$. The initial call to $CorrectionRoutine$ necessarily takes label a equal to this finite state (because we correct t *w.r.t.* $\mathcal{L}_a$). Thus, $t'(\epsilon) = a$. Moreover, if t' is valid then the empty string is recognized by the regular expression of $t'(\epsilon)$. That means that the initial state of the corresponding finite-state automaton is also its finite state. Thus, while executing $CorrectSubtree$ must have necessarily generated t' in Algorithm 2 lines 1 and 2.

Let's now suppose that each correct solution of depth e with $1 \leq e < d$ is found. We will prove that each solution t' of depth d is found. If t' is valid then its root corresponds to the finite state of the *tree automaton* associated to the language $\mathcal{L}_a$. Similarly to the case above, a call to $CorrectionRoutine$ with $t'(\epsilon) = a$ must have been performed. While exploring the finite-state automaton associated to $t'(\epsilon)$, each subtree of t' has been found (according to the supposition above). Since the roots of these subtrees respect the regular expression of $t'(\epsilon)$, our algorithm necessarily finds the "word" consisting of these roots. Thus, t' itself must have been found.

□

It is worth noting that *CorrectionRoutine* calculates not only all the set of candidates $t' \in \mathcal{L}$ such that $dist(t, t')$ is minimal within a given threshold (*i.e.* $dist(t, t') = DIST(t, \mathcal{L}) \leq DIST(t, \mathcal{L}) \leq th$) but also some non-minimal candidates that respect the threshold. The procedure is however not complete *w.r.t* the set containing all the candidates respecting the threshold. This is due to two facts.

Firstly, while admitting an insertion we allow only minimal subtrees having the current symbol as a root (see Algorithm 3, line 4). In order to get *all*, possibly non minimal, solutions within the threshold we would have to take the whole list $LCand$ (Algorithm 3, line 3) into account, and not only its first element.

Secondly, while admitting a replacement we also retain only the minimal solution at the head of the list (Algorithm 3, line 14). For a complete set of candidates the whole list would have to be preserved instead.

5 Building corrected XML tree from local corrected subtrees

We have presented in previous section algorithms capable of computing a corrected tree from a given invalid tree. In order to integrate correction into the incremental validation procedure, given an invalid updated XML tree T, *CorrectionRoutine* is called for each position p where the incremental validation process fails. Then, each failure position p corresponds to a tuple $(p, LCand)$: these tuples are stored in a structure called $solStock$, managed by the validation process. Before describing how $solStock$ is managed and finally used to present corrections to the user, we notice that this structure also induces some optimizations: indeed, it avoids re-correcting the same subtrees in function *AddNewColumn*. To this end, we replace lines 13 to 15 of Algorithm 3 by the lines described in Fig. 6.

When correcting a subtree t, Algorithm $AddNewColumn$ (modified version) avoids correction over both the subtrees that had already

// 3^{rd} case: by replacement

1. **If** $(w[i] = a)$ //if we maintain the same label
2. **If** Exists(p', $solStock$) //the subtree has been corrected beforehand
3. $C_R := H[i-1, j-1].C + getCost(p', solStock)$; $SSrep := getSetSeq(p', solStock)$
4. $SSrep := SetAddPrefix(SSrep,\ i)$
5. **else** // the subtree is locally valid
6. $C_R := H[i-1, j-1].C$, $SSrep := \emptyset$
7. **else** // if we change the label
8. $LCand :=$ **CorrectionRoutine**$(getSubTree(t, i),\ p',\ a,\ th,\ solStock)$
9. $(Crep,\ SSrep) := (LCand[0].C,\ LCand[0].S)$; $C_R := H\_DIST[i\text{-}1, j\text{-}1].C + Crep$
10. $SSrep := SetAddPrefix(SSrep,\ i)$

Figure 6: The lines which modify lines 13 to 15 of Algorithm $AddNewColumn$

□

been corrected and those that are valid (lines 2-4 and 5-6 respectively).

We saw that each call to *CorrectionRoutine* at a failure position p returns its solutions in $LCand$. As said before, the incremental validation procedure (augmented with correction) uses relation $solStock$ to store solutions: during the validation process, each failure position p corresponds to a tuple $(p, LCand)$ in $solStock$. At the end of the validation-correction process, as some redundancies may exist in $solStock$, before proceeding to the integration of solutions, we eliminate from $solStock$ the set of tuples $\{(p, LCand) \in solStock \mid \exists p', p' \prec p \wedge (p', LCand') \in solStock\}$. This is because corrections at positions $p' \prec p$ take corrections at p into account. Indeed, during the correction at p', if a candidate tree for $T_{p'}$ relies on corrections on T_p (which belongs to $T_{p'}$) then the list $LCand$ computed for p is integrated into $LCand'$ (for p').

Once we have in $solStock$ all the "non redundant" local solutions remaining within the threshold, we combine them into global ones. If the sum of the costs of local solutions does not exceed the threshold, then the corresponding global solution may be presented to the user. For instance, one minimum cost solution (if any) can be computed by putting together for each tuple in $solStock$ the first candidate in $LCand$. Each global solution involves the correction of the invalid subtrees T_p (for each p appearing in the non redundant $solStock$) by the application of the corresponding update sequence to obtain candidate T'_p. It is interesting to remark that the user can choose to compute one unique solution or n ones, depending on whether he is interested in any correction with a minimal distance from T or in examining several possible corrections.

Example 10 Consider Example 9. Recall that error threshold $th = 2$. After the first correction at position 0 (subtree τ_0 in Fig. 7), $SolStock$ contains $S = (0, LCand_1)$, where $LCand_1$ represents the list of candidates found in Example 9. The second error is found when the validation test is performed at position ϵ. More precisely, the word $w = aabbb$ which is composed by the concatenation of states associated to children of ϵ, is not in $L(E_{root})$ where $E_{root} = ab^*$ (see Fig. 7). During the correction of $t = T_\epsilon^{Tree}$ (updated tree), *CorrectionRoutine* returns $null$, since all candidates have an edit distance that exceeds the threshold $th = 2$.
We consider now $th = 3$. As the validation routine fails at position 0 (Fig. 1), *CorrectionRoutine* of tree $t = T_0^{Tree}$ returns $LCand'_1$ (*i.e.* at position 0, $SolStock$ contains $S' = (0, LCand'_1)$). In this case, $LCand'_1$ stores the same solutions appearing in $LCand_1$ (obtained with $th = 2$) plus some extra solutions with cost 3.

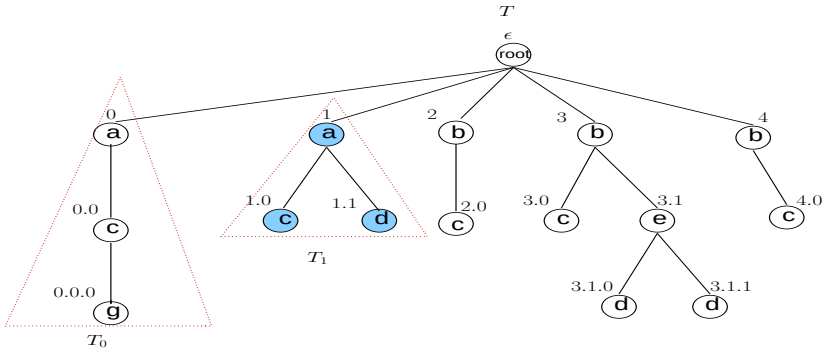

Figure 7: XML tree from Fig. 1 after updates.

The validation routine also fails at position ϵ of Fig.1. In order to propose corrected solutions, *Correction-Routine* considers the updated version of T, shown in Fig. 7. Since the correction of subtree T_1 in Fig. 7 needs at least two operations, only the cheapest corrections of T_0 can be kept (those with cost 1). More precisely, the *CorrectionRoutine* of tree $t = T_\epsilon^{Tree}$ returns the relation $LCand_2$ below (We recall that the positions indicated in this relation are those of the updated tree version illustrated in Fig. 7). Fig. 8 illustrates the corrected trees obtained by applying each one of these corrections.
$LCand_2 = \{$
S_1 = (3,[(*delete*,0.0.0,/),(*delete*,0.0,/),(*delete*,0,/)]),
S_2 = (3,[(*insert*,0.1,d),(*relabel*,1,b),(*relabel*,1.1,e)]),
S_3 = (3,[(*insert*,0.1,d),(*relabel*,1,b),(*delete*,1.1,/)]),
S_4 = (3,[(*relabel*,0.0,m),(*relabel*,1,b),(*relabel*,1.1,e)]),
S_5 = (3,[(*relabel*,0.0,m),(*relabel*,1,b),(*delete*,1.1,/)])$\}$

As $(\epsilon, LCand_2)$ is obtained at ϵ, we can eliminate $(0, LCand'_1)$, because ϵ is the prefix of 0. Then, in $SolStock$ we find only $(\epsilon, LCand_2)$. □

The global solutions can be presented to the user in different forms. If we consider that the correction method applies to relatively small XML documents visualized in an editor, we can display entire candidate trees. Consider Example 10, we can generate from $SolStock$ the tree candidates shown in Fig. 8. However, proposing a new corrected update sequence instead seems to be a better way of showing the user which operations he is allowed to do. Notice that the correct update sequence needed to transform the original valid XML tree $T0$ into the valid updated tree T' is simply computed by $H\_DIST_{T0,T'}$.

6 Conclusion

We have presented a correction routine associated to an incremental validation process. This routine introduces a tree-to-grammar correction approach: given a local tree language $\mathcal{L}$ and an invalid tree t, find valid trees t' whose distance from t is within a given threshold th. This is done by computing tree edit distance between t and $t' \in \mathcal{L}$. This algorithm extends our previous work in [8] which considers an incremental string-to-grammar correction method (based on [11, 16, 17]) . The tree-to-grammar correction is based on the tree-to-tree correction problems examined in [13, 14, 15] for two given trees t and t'. Their edit operations differ from ours. However, the main difference of our approach is that we generate several t' in a given tree language $\mathcal{L}$ close to t.

A correction algorithm for XML invalid trees *w.r.t* schema constraint is also presented in [4]. The correction is done from scratch and only one solution is presented (the authors argue that the distance between the original tree and the corrected one is minimal). Our approach differs from this one in the following aspects: (i) Our correction routine finds different solutions within a given threshold. In particular, all minimal solutions are computed. (ii) The user can choose the best solution for his purposes. (iii) Our correction routine is integrated in an incremental validation process and can use validation information to fulfill its task. (iv) We are capable of pointing at the sequences of edit operations needed to transform an invalid tree into a corrected one.

We have obtained good experimental results

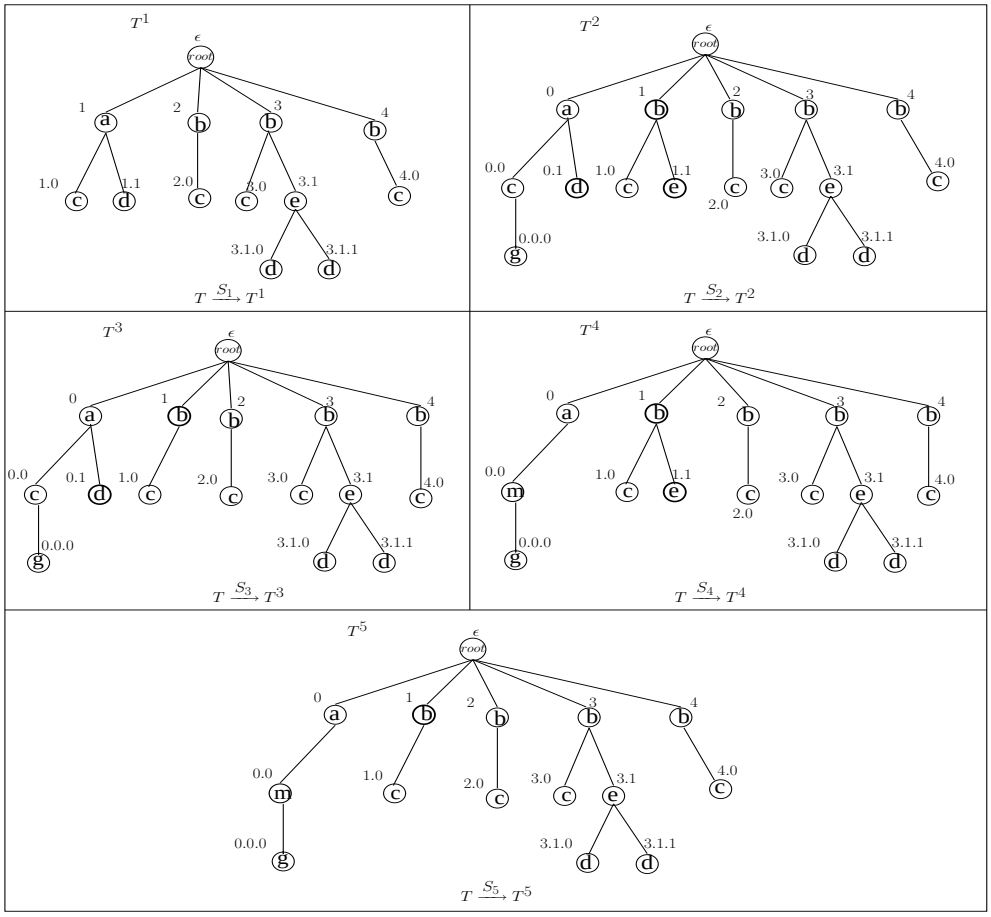

Figure 8: Correction candidates of XML tree of figure 7 with error threshold $th = 3$.

for the incremental validation routine [5]. Other experimental results show a good performance of our string-to-grammar correction algorithm, despite its high theoretical complexity [8]. The implementation of *CorrectionRoutine* (in progress) is based on [8, 13].

Our *CorrectionRoutine* is designed to fit the incremental validation process. We are currently studying a more general procedure capable of computing *all* solutions within a given threshold. Such kind of procedure might be applied in other application domains using trees. Complexity of our method should be evaluated *w.r.t* this general procedure, considering the advantages of avoiding correction of valid subtrees.

We continue studying the problem of how to help the user to choose the best correction: we plan to perform tests with intensive updates on huge documents in order to determine which procedure may be more suitable. Among other usability considerations, our proposition should be integrated in an XML update framework, using for instance UpdateX [9]: to this aim, more primitive operations should be considered. If we consider for example that inserting one node between a father and some of its children (*i.e.* inserting one new level) is an elementary operation, it means, at string level, to deal with the string edit operation that can replace a sequence of letters with only one letter (the inserted node, which becomes the father of the node sequence). This is a problem for which there is no efficient solution yet.

References

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

[2] M. A. Abrão, B. Bouchou, A. Cheriat, M. Halfeld-Ferrari, D. Laurent, and M. A. Musicante. Incremental Constraint Validation

of XML Documents Under Multiple Updates. In *Submitted to an international journal (under review)*, 2005.

[3] M. A. Abrão, B. Bouchou, M. Halfeld-Ferrari, D. Laurent, and M. A. Musicante. Incremental Constraint Checking for XML Documents. In *Database and XML Technologies, Second International XML Database Symposium*, volume 3186 of *LNCS*, pages 112–127. Springer-Verlag, 2004.

[4] U. Boobna and M. de Rougemont. Correctors for XML Data. In *Database and XML Technologies, Second International XML Database Symposium*, volume 3186 of *LNCS*, pages 97–111. Springer-Verlag, 2004.

[5] B. Bouchou, A. Cheriat, M. Halfeld Ferrari Alves, D. Laurent, and M. Musicante. Schema Constraint Validation in XML: an Incremental Approach under Multiple Updates. Technical Report 2101, LI, Université de Tours, 2006.

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

[7] A. Brüggeman-Klein and D. Wood. Regular tree languages over non-ranked alphabets. unpublished manuscript, 1998.

[8] A. Cheriat, A. Savary, B. Bouchou, and M. Halfeld Ferrari. Incremental String Correction: Towards Correction of XML Documents. In *Prague Stringology Conference 2005*, 2005.

[9] G. M. Gargi, J. Hammer, and J. Simeon. An XQuery-based language for processing updates in XML. In *Programming Language Technologies for XML (PLANX04)*, 2004.

[10] 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.

[11] K. Oflazer. Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics*, 22(1):73–89, 1996.

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

[13] S. M. Selkow. The Tree-to-Tree Editing Problem. *Information Processing Letters*, 6(6):184–186, 1977.

[14] D. Shasha and J. Tsong-Li Wang. New Techniques for Best-Match Retrieval. *Transactions on Information Systems*, 8(2):140–158, 1990.

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

[16] R. A. Wagner. Order-n Correction for Regular Languages. *Communications of the ACM*, 17(5):265–268, 1974.

[17] R. A. Wagner and M. J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.