Incremental Constraint Checking for XML Documents

Maria Adriana ABRÃO[1]*, Béatrice BOUCHOU[1], Mírian HALFELD FERRARI[1], Dominique LAURENT[2], and Martin A. MUSICANTE[3] **

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

Abstract. We introduce a method for building an XML constraint validator from a given set of schema, key and foreign key constraints. The XML constraint validator obtained by our method is a bottom-up tree transducer that is used not only for checking, in only one pass, the correctness of an XML document but also for incrementally validating updates over this document. In this way, both the verification from scratch and the update verification are based on regular (finite and tree) automata, making the whole process efficient.

1 Introduction

We address the problem of incremental validation of updates performed on an XML document that respects a set of schema and integrity constraints (*i.e.*, on a valid XML document). Given a set of schema and integrity constraints $\mathcal{D}$, we present a method that translates $\mathcal{D}$ into a bottom-up tree transducer $\mathcal{U}$ capable of verifying the validity of the document. We only address meaningful specifications [11], *i.e.*, ones in which integrity constraints are consistent with respect to the schema. The aim of this work is the construction of a transducer $\mathcal{U}$ that allows incremental validation of updates. In this paper, we deal mostly with the verification of key and foreign key constraints. The validation of updates taking into account schema constraints (DTD) is perform by $\mathcal{U}$ exactly as proposed in [5].
The main contributions of the paper are:

- A method for generating a validator from a given specification containing schema, key and foreign key constraints.
- An unranked bottom-up tree transducer, which represents the validator, where syntactic and semantic aspects are well separated.
- An incremental schema, key and foreign key validation method.
- An index tree that allows incremental updates on XML document. This key index can also be used for efficiently evaluating queries based on key values.

* Supported by CAPES (Brazil) BEX0706/02-7
** This work was done while the author was on leave at Université François Rabelais. Supported by CAPES (Brazil) BEX1851/02-0.

Section 2 gives an overview of the incremental constraint checking framework. Section 3 presents our method to build a tree transducer from a given specification containing a DTD and a set of keys and foreign keys. We also show how the transducer is used to efficiently verify all the imposed constraints. Section 4 shows how incremental validation is performed on updates. Section 5 concludes with our further research.

2 General Overview

An XML document is a structure $\mathcal{T}$ composed by an unranked labeled tree t 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). Fig. 1 shows part of the labeled tree representing the document used in our examples. It describes menus and combinations in some French restaurants. Differently from the *à la carte* style, a combination is a grouping of dishes and drinks, reducing both the choice and the price for clients. Each node in the tree has a position and a label. Elements and attributes associated with arbitrary text have a child labeled *data*. Attribute labels are depicted with a preceding @.

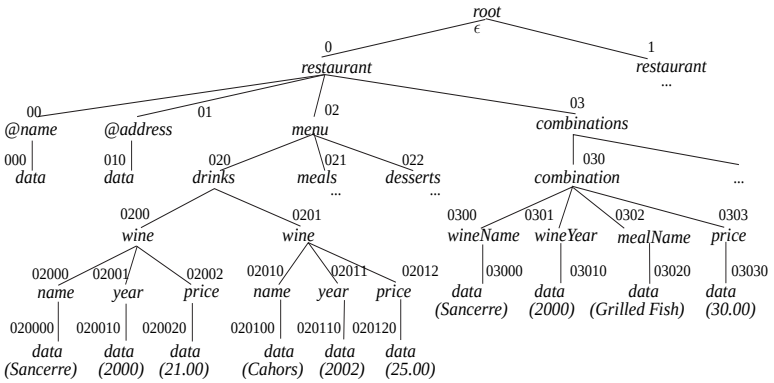

Fig. 1. Labeled tree t of an XML document.

Definition 1. Key and foreign key syntax [8]: A key is represented by $(P, (P', \{P^1, \ldots, P^m\}))$. A foreign key is represented by $(P_0, (P'_0, \{P^1_0, \ldots, P^m_0\})) \subseteq K$ where $K = (P, (P', \{P^1, \ldots, P^m\}))$ is a key such that $P = P_0$. In a key, path P is called the *context path*; P' the *target path* and $P^1, \ldots, P^m$ the *key paths*. The same applies for a foreign key ($P^1_0, \ldots, P^m_0$ are the *foreign key paths*). □

Context and target paths should reach element nodes. Key (or foreign key) paths are required to end at a node associated to a value, *i.e.*, attribute nodes or elements having just one child of type *data*. The next example gives the intuition of the semantics of key and foreign key constraints over the document of Fig. 1.

Example 1. Let K_1 = (/*restaurant*,(./*menu*/*drinks*/*wine*, {./*name*, ./*year*})) be a key constraint indicating that, in the context of a restaurant, a wine (the target node) can be uniquely identified

by its name and its year. Let FK_2 = (*/restaurant*, (*./combinations/combination*, {*./wineName*, *./wineYear*})) $\subseteq K_1$ be a foreign key constraint indicating that, for each restaurant, a combination is composed by a wine that should appear in the menu of the restaurant. □

Definition 2. Key and foreign key semantics: An XML tree $\mathcal{T}$ satisfies a key $(P, (P', \{P^1, \ldots, P^m\}))$ if for each context position p defined by P the following two conditions hold: (i) For each target position p' reachable from p via P' there exists a unique position p_h from p', for each $P^h (1 \leq h \leq m)$. (ii) For any target positions p' and p'', reachable from p via P', whenever the values reached from p' and p'' via $P^h (1 \leq h \leq m)$ are equal, then p' and p'' must be the same position. Similarly, an XML tree $\mathcal{T}$ satisfies a foreign key $(P_0, (P'_0, \{P_0^1, \ldots, P_0^m\})) \subseteq K$ if: (i) it satisfies its associated key K and (ii) each tuple τ of values, built following paths $P_0/P'_0/P_0^1, \ldots, P_0/P'_0/P_0^m$ (in this order), can also be obtained by following the paths $P/P'/P^1$, $\ldots$, $P/P'/P^m$ (in this order). □

In the following, we assume the existence of an XML tree $\mathcal{T}$ and a set of schema and integrity constraints $\mathcal{D}$ and we survey (i) the validation of $\mathcal{T}$ from scratch which is performed in only one pass on the XML tree and (ii) the incremental validation of updates over $\mathcal{T}$.

2.1 Validation from scratch

Our method consists in building a tree transducer capable of expressing all the constraints of a given specification $\mathcal{D}$. The tree transducer is composed by a bottom-up tree automata (to verify the syntactic restrictions) and a set of actions defined for each key and foreign key. These actions manipulate values and are used to verify the semantic aspects of constraints. The execution of the tree transducer consists in visiting the tree in a bottom-up manner[4], performing, at each level:

A) *The verification of schema constraints.* Schema constraints are satisfied if all positions of a tree t can be associated to a state and if the root is bound to a final state (defined by the specification). A state q is assigned to a position p if the children of p in t verify the element and attribute constraints established by the specification. Roughly, a schema constraint establishes, for a position labeled a, the type, the number and (for the sub-elements) the order of p's children. We assume that the XML document in Fig. 1 is valid wrt schema constraints (see [5] for details).

B) *The verification of key and foreign key constraints.* In order to validate key and foreign key constraints we need to manipulate data values. To this end, we define the values to be carried up from children to parents in an XML tree. The following example illustrates how the transducer treats values being carried up for each node. This treatment depends on the role of the node's label in the key or foreign key.

Example 2. We assume a tree transducer obtained from specification $\mathcal{D}$ (containing a given DTD together with K_1, FK_2 of Example 1) and we analyze its execution over $\mathcal{T}$ (Fig. 1):

1. The tree transducer computes the values associated to all nodes labeled *data*. We consider *value*(020000) = *value*(03000) = *Sancerre* and *value*(020010) = *value*(03001) = 2000 as some of the values computed in this step.

[4] Notice that it is very easy to perform a bottom-up visit even using SAX [13](with a stack).

2. The tree transducer analyzes the parents of the *data* nodes. If they are key or foreign key nodes, they receive the values computed in 1. Otherwise, no value is carried up. In our case, the value *Sancerre* is passed to key node 02000 and to foreign key node 0300. The value 2000 is passed to key node 02001 and to foreign key node 0301.
3. The tree transducer passes the values from children to parent until it finds a target node. At this level the values for each key or foreign key are grouped in a list. Node 0200 is target for K_1, and as the key is composed by two items, the list contains the tuple value $\langle$*Sancerre*, $2000\rangle$. Similarly, node 030 (target node for FK_2) is associated to $\langle$*Sancerre*, $2000\rangle$.
4. The transducer carries up the lists of values obtained in 3 until finding a context. At a context node of a key, the transducer tests if all the lists are distinct, returning a boolean value. Similarly, at a context of a foreign key, the transducer tests if all the tuples exist as values of the referenced key. In our case, *restaurant* is the context node for both K_1 and FK_2. As context node for K_1, it receives several lists, each containing a tuple with the wine name and year. The test verifies the uniqueness of those tuples. As context node for FK_2, it receives several lists, each containing a tuple with the name and year of a wine of a combination. The test verifies if each tuple is also a tuple for key K_1. For instance, $\langle$*Sancerre*$, 2000\rangle$ that represents a wine in a combination, appears as a wine in the menu of the restaurant.
5. The boolean values computed in 4 are carried up to the root. K_1 and FK_2 are satisfied if the conjunction of the boolean values results in *true*. □

2.2 Incremental validation of updates

Let us now consider updates over valid XML trees. To this end, we suppose that:

– Updates are seen as changes to be performed on the XML tree $\mathcal{T}$.
– Only updates that preserve the validity of the document (with respect to schema, key and foreign key constraints) 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.

We deal with two kinds of update operations. The insertion of a subtree $\mathcal{T}'$ at position p of $\mathcal{T}$ and the deletion of the subtree rooted at p in $\mathcal{T}$. To verify if an update should be accepted, we perform incremental tests, summarized as follows:

1. *Schema constraints:* We consider the run of the tree transducer on the subtree of $\mathcal{T}$ composed just by the updated position p, its siblings and their father. If the state assigned to p's father does not change due to the update, *i.e.*, the tree transducer maintains the state assignment to p's father as it was before the update, then schema constraints are not violated (see [5] for details).
2. *Key and foreign key constraints:* To facilitate the validation of keys and foreign keys for an update operation, we keep an index tree of those tuples in $\mathcal{T}$ defined by each key. For each tuple that is referenced by a foreign key, a reference counter is used in order to know how many times the tuple is used as a foreign key.
   The verification of key and foreign key constraints changes according to the update operation being performed. Firstly we have to find (for each key and foreign key) the corresponding context node p', concerned by the insertion or the deletion. Then, in order to insert a subtree $\mathcal{T}'$ at position p of $\mathcal{T}$ we should perform the following tests: (i) verify whether $\mathcal{T}'$ does not contain duplicate key values for context p', (ii) verify whether $\mathcal{T}'$ does not contain key values already appearing in $\mathcal{T}$ for context

p' and (iii) for each key tuple in context p' being referenced by a foreign key in $\mathcal{T}'$, increase its reference counter. Similarly, to delete a subtree $\mathcal{T}'$, rooted at position p, from an XML tree $\mathcal{T}$ we should perform the following tests, for each context p': (i) verify if $\mathcal{T}'$ contains only key values that are not referenced by foreign keys (not being deleted) and (ii) for each key tuple in context p' being referenced by a foreign key in $\mathcal{T}'$, decrease its reference counter.

The acceptance of an update over an XML tree $\mathcal{T}$ wrt keys and foreign keys requires information about key values in $\mathcal{T}$. Given an XML tree $\mathcal{T}$, the tree transducer is used once to verify its validity (from scratch). During this first execution of the tree transducer an index tree, called *keyTree*, is built for each key constraint K that should be respected by $\mathcal{T}$. Each *keyTree*$_K$ is a tree structure that stores the position of each context and target node together with the values associated to each key node in $\mathcal{T}$. Fig. 2 describes this index structure using the notation of DTDs and Fig. 3 shows a *keyTree* for key K_1 of Example 1. The next example illustrates the validation of updates.

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

Fig. 2. DTD specifying structure *keyTree*.

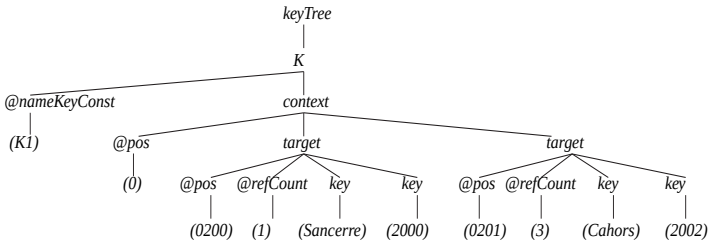

Fig. 3. *KeyTree*$_{K_1}$ built over the document of Fig. 1

Example 3. Given the XML tree of Fig. 1, we show its incremental verification due to the insertion of a new wine in the menu of a restaurant (*i.e.*, the insertion of a labeled tree t' at position $p = 0200$ of t). Moreover, we consider a specification stating that a position p labeled *drinks* should respect the following schema constraints: (i) for each child *pos* of p we have *type*$(t, pos) \neq$ *attribute* and (ii) the concatenation of the labels associated with p's children composes a word that corresponds to the regular expression ${q_{wine}}^*$.

The verification of the update with respect to schema constraints consists in: (i) considering that the update is performed (without performing it yet) and (ii) verifying if the state q_{drinks} can still be associated with position 020 (0200's father) by analyzing the schema constraint imposed over nodes labeled *drinks*. To this end, we build the sequence of states associated with 020's children. The insertion consists of shifting to the right the right siblings of p. Thus, we consider state q_{wine} associated to positions 0201 and 0202 and we only calculate the state associated with the update position 0200. As the root of t' (at position 0200) is associated to the state q_{wine}, we obtain the word $q_{wine}\ q_{wine}\ q_{wine}$. This word matches the regular expression ${q_{wine}}^*$. Thus, the update respects the schema constraints [5].

Now we verify whether K_1 and FK_2 (Example 1) are preserved by the insertion. As the inserted subtree contains only one key value, it contains no key violation by itself (no duplicate of key values). Then we assume that the update is performed (without performing it yet) and we verify whether the key value being inserted is not in contradiction with those already existing in the original document. In our case, we suppose that the wine being inserted is identified by the key tuple $\langle$*Bordeaux*, $1990\rangle$. Comparing this value to those stored in the *keyTree*$_{K_1}$ (Fig. 3), we notice that no violation exists. The inserted subtree does not contain foreign key values and, thus, we can conclude that the update is possible with respect to key and foreign key constraints.

As the above tests succeed, the insertion can be performed. The performance of an update implies changes not only on the XML tree but also on index trees *keyTree*s. □

3 Tree Transducers for XML

We first present the definition of our tree transducer. This transducer combines a tree automaton (expressing schema constraints) with a set of output functions (defining key and foreign key constraints).

Definition 3. Output function: Let **D** be an infinite (recursively enumerable) domain and let $\mathbf{D}^*$ denote the set of all lists of items in **D**. Let $\mathcal{T} = (t, \textit{type}, \textit{value})$ be an XML tree. An *output function* f takes as arguments: (i) a tree position $p \in \textit{dom}(t)$; (ii) a set s of pairs $(\textit{att}, l_v)$ where *att* is a tag associated to a list $l_v \in \mathbf{D}^*$ and (iii) a list l of items in **D**. The result of applying $f(p, s, l)$ is a list of items in **D**. In other words, $f : \textit{dom}(t) \times \mathcal{P}(\Sigma \times \mathbf{D}^*) \times \mathbf{D}^* \rightarrow \mathbf{D}^*$. □

We recall the process in Example 2: at each node, data values are collected from children nodes and can be used to perform tests. Output functions are defined to perform these actions: for the node at position p, each of them takes as parameters the set s containing data values coming from attribute children, and the list l of values coming from element children. One output function is defined for each key and foreign key.

Definition 4. Unranked bottom-up tree transducer (UTT): A UTT over Σ and **D** is a tuple $\mathcal{U} = (Q, \Sigma, \mathbf{D}, Q_f, \Delta, \Gamma)$ where Q is a set of states, $Q_f \subseteq Q$ is a set of final states, Δ is a set of transition rules and $\Gamma = \{f_1, ..., f_n\}$ is a set of output functions.

Each transition rule in Δ has the form $a, S, E \rightarrow q$ where *(i)* $a \in \Sigma$; *(ii)* S is a tuple 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)* $q \in Q$. Each output function in Γ has the form $f_j(p, s, l) = l'$ as in Definition 3. □

Key and foreign key constraints are expressed by the output functions in Γ. As the tree is to be processed bottom-up, the basic task of output functions is to define the values that have to be passed to the parent position, during the run.

3.1 Generating constraint validators

Given a specification $\mathcal{D} = (D, \mathbf{K})$ where D is a set of schema constraints and **K** is composed by the set of keys and foreign keys, we propose a method to translate $\mathcal{D}$ into a UTT. In this sense, we present an algorithm to generate a validator from a given specification. This validator is executed to check the constraints in $\mathcal{D}$ for any XML tree.

Let $\mathcal{U} = (Q, \Sigma, \mathbf{D}, Q_f, \Delta, \Gamma)$ be a UTT whose transition rules in Δ are obtained from the translation of a non-ambiguous DTD D (part of $\mathcal{D}$). The domain **D** is formed by pairs containing an (identified) finite state automaton state and a list of values [1].

To define the output functions we need to construct finite state automata for the paths appearing in the keys and foreign keys. Notice that context, target and key nodes in each key K_j or foreign key FK_j are defined in a top-down fashion. In order to identify these nodes using a bottom-up tree automaton, we must traverse the paths stated by each key K_j or foreign key FK_j in reverse.

Given a key constraint K_j ($1 \leq j \leq k$) or a foreign key constraint FK_j ($k+1 \leq j \leq n$) of the form $(P_j, (P'_j, \{P_j^1, ..., P_j^{m_j}\}))$, the following automata recognize the paths in reverse. For path P_j, we have $M_j = \langle \Theta_j, \Sigma, \delta_j, e_j, F_j \rangle$. For path P'_j, $M'_j = \langle \Theta'_j, \Sigma, \delta'_j, e'_j, F'_j \rangle$. For path $P_j^1 \mid \ldots \mid P_j^{m_j}$, $M''_j = \langle \Theta''_j, \Sigma, \delta''_j, e''_j, F''_j \rangle$. Additionally, we define $M_F = \langle \{e_0, e_f\}, \{root\}, \{\delta(e_0, root, e_f)\}, e_0, \{e_f\} \rangle$ as the finite state automaton recognizing the path formed just by the symbol *root*. Figure 4 illustrates the finite state automata for the paths in K_1 and FK_2 of Example 1 in reverse.

Remark: We denote by $M.e$ the current state e of the finite state automaton M, and we call it a *configuration*.

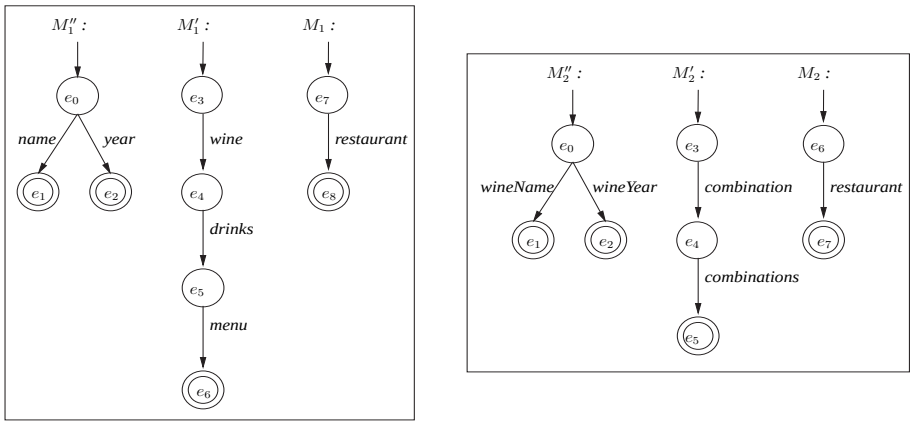

Fig. 4. Automata corresponding to the paths of K_1 and FK_2 in reverse.

Algorithm 1 - Key constraints as output functions:
Input: A set of k keys $K = \{K_j = (P_j, (P'_j, \{P_j^1, ..., P_j^{m_j}\})) \mid 1 \leq j \leq k\}$, a set of $(n-k)$ foreign keys $FK = \{FK_j = (P_j, (P'_j, \{P_j^1, ..., P_j^{m_j}\})) \subseteq K_i \mid (k+1) \leq j \leq n;\ K_i \in K\}$, and the finite state automata M_j, M'_j, M''_j, M_F, $(1 \leq j \leq n)$ that recognize key K_j and foreign key FK_j paths in reverse.
Output: A set of output functions $\Gamma = \{f_1, \ldots, f_n\}$.

Algorithm:
For each key K_j or foreign key FK_j $(1 \leq j \leq n)$ the output function f_j is defined as:
function f_j(p: position, s: set of pairs (att, l_v), l: list of items in **D**)
begin
  Let $t(p) = a$ $//a$ is the label of position p

(1) If $a =$ *data* then return $[(M''_j.e''_j, [value(p)])]$

(2) If a is a target label for K_j or FK_j
then return $[(M'_j.\delta'_j(e'_j, a), checkArity(concat(filter^j_{key}(c_0, \ldots, c_{z-1}))))]$
where z is the number of children of position p, $[c_0, \ldots, c_{i-1}] = orderByName(s)$ and $[c_i, \ldots, c_{z-1}] = l$. The function $orderByName$ sorts, in the lexicographic order, attribute tags of the pairs (att, l_v) coming from attribute children.
Function $filter^j_{key}$ leaves in the key lists only the values associated to key positions of K_j (or FK_j). It selects the lists of values whose configuration corresponds to a final state of M''_j. Function $concat$ returns the concatenation of all its argument lists into one list. If the length of the list does not correspond to the length m_j of K_j, then function $checkArity$ replaces it by an empty list. For foreign keys the length is not tested.

(3) If a is a context label for a key K_j
then return $[(M_j.\delta_j(e_j, a), checkKey(filter^j_{target}(l))]$
where $checkKey([v_1 \ldots v_m]) = \begin{cases} [\textit{true}] & \text{if } v_1 \ldots v_m \text{ are all nonempty distinct lists.} \\ [\textit{false}] & \text{otherwise.} \end{cases}$

(4) If a is a context label for a foreign key FK_j
then return $[(M.\delta_j(e_j, a), checkForeign(filter^j_{target}(l))]$
where $checkForeign([v_1 \ldots v_m]) = \begin{cases} [\textit{true}] & \text{if } v_1 \ldots v_m \text{ are lists whose values appear in the key } K_i \text{ (the one taking part in the definition of } FK_j). \\ [\textit{false}] & \text{otherwise.} \end{cases}$

Remark: In cases (3) and (4) above, function $filter^j_{target}$ rejects all the values not belonging to target lists of key K_j (or foreign key FK_j).

(5) If a is the root label then return $[(M_F.e_f, (filter^j_{context}(l)))]$
Function $filter^j_{context}$ rejects all the values not belonging to context lists.

(6) In all other cases
(*i.e.*, when $a \neq$ *data* and a is not a target label, nor a context label, nor the root)
return $carryUp^j([c_0, \ldots, c_{z-1}])$
where $[c_0, \ldots, c_{z-1}]$ is the list of pairs obtained from the children of p, such that $[c_0, \ldots, c_{i-1}] = orderByName(s)$ and $[c_i, \ldots, c_{z-1}] = l$.
Function $carryUp^j$ is defined as follows:

```
function carryUp^j (L : list of pairs)
var result : list of pairs
begin
result ← [ ]
foreach c = (M.e, v) in L //* M stands for M_j, M'_j or M''_j
   if δ(e, a) = e' is a transition in M then result ← concat(result, [(M.e', v)])
return result
end
```

end // end of function f_j □

Each output function returns a list composed by pairs. For instance, $f_1(020000, \emptyset, [\,]) = [(M''_1.e_0, [\textit{Sancerre}])]$. In cases (1) to (5) the list contains only one pair. In all cases, a pair is composed by:

(A) A configuration $\mathcal{M}.e$ where $\mathcal{M}$ is one of the finite automata representing paths in keys, and e is a state of $\mathcal{M}$. For example, in case (2), $\mathcal{M}$ is M'_j, the target automaton for K_j or FK_j. This configuration is obtained by performing the first transition at automaton M'_j, using the symbol a as input. Notice that $\delta'_j(e'_j, a)$ is a state of M'_j. Other cases are similar.

(B) A list of values. From data nodes to context nodes, these values represent those composing a key (or foreign key). From context nodes to the root they are boolean values indicating that within a given context, K_j or FK_j holds or not.

Notice that the result obtained at context level (case (3)) is a singleton list that contains a pair, formed by a configuration of M_j and a list containing a boolean value (the result of checking the validity of the key for each specific context). For foreign key context level (case (4)), FK_j and K_i have the same context and the tuples representing key K_i are computed before those that represent foreign key FK_j (since $i < j$). At root level (case (5)), we have the boolean values that were obtained for each subtree rooted at the context level. In case 6, values are carried up by function $carryUp^j$. This function selects pairs from children nodes belonging to key and foreign key paths, by checking configurations in these pairs. The resulting lists of these output functions can contain more than one pair. If nodes are not concerned by any key or foreign key, the function $carryUp^j$ does not transmit any value.

3.2 Validating XML documents

The verification of keys and foreign keys are performed simultaneously, in one pass, during the execution of the UTT over an XML tree. Example 4 illustrates such an execution. A tree index, necessary to perform incremental updates on XML documents, is dynamically built during this validation process. This index, called ***keyTree***, similar to the one proposed in [9], is a tree structure containing levels for the key name, context, target, key nodes and data (in this order) as defined in Fig. 2.

Example 4. We consider a specification $\mathcal{D}$ containing K_1 and FK_2 (Example 1). The finite state automata associated to K_1 and FK_2 are the ones given in Fig. 4. To verify if the XML tree $\mathcal{T}$ of Fig. 1 satisfies K_1 and FK_2 we run the transducer $\mathcal{U}$ (from $\mathcal{D}$) over $\mathcal{T}$ (recall that $\mathcal{U}$ contains two output functions f_1 and f_2 defined from K_1 and FK_2 (respectively), following Algorithm 1):

1. For the data nodes, each output function returns a singleton list that contains a pair: the initial configuration of the key (or foreign key) automaton M'', and the value of the node. Positions 020000 and 03000 are data nodes, then we have:
   $f_1(020000, \emptyset, [\,]) = [(M''_1.e_0, [\textit{Sancerre}])]$; $f_2(03000, \emptyset, [\,]) = [(M''_2.e_0, [\textit{Sancerre}])]$.
2. The fathers of data nodes which are key (or foreign key) nodes should carry up the values received from their children. Thus, each of them executes a first transition in M'' using each key (or foreign key) label as input. For each father of a data node which is not a key (or a foreign key) node, the output function returns an empty list.
   For instance, position 02000 is a key node for K_1 and position 0300 is a foreign key node for FK_2. Then, reading the label *name* from state e_0 of M''_1, we reach state e_1, and we carry up the value *Sancerre*. We obtain a similar result for FK_2 when reading label *wineName*:
   $f_1(02000, \emptyset, [(M''_1.e_0, [\textit{Sancerre}])]) = [(M''_1.e_1, [\textit{Sancerre}])]$;
   $f_2(0300, \emptyset, [(M''_2.e_0, [\textit{Sancerre}])]) = [(M''_2.e_1, [\textit{Sancerre}])]$.
   At this stage the construction of $\textit{keyTree}_{K_1}$ starts by taking into account the information associated to each key node (e.g., $\textit{keyTree}_{K_1}[t, 02000]$ is the subtree rooted at *key* and associated with the value *Sancerre* in Fig. 3).

3. For node 0200, *wine* is a target label of K_1 and for node 030, *combination* is a target label of FK_2. In order to transmit only key (or foreign key) values, the output function of a target label (i) selects those that are preceded by a final state of the key automaton M'', (ii) joins them in a new list, and (iii) executes the first transition of the target automaton M'. In this way, at a target position the tuple value of a key (or foreign key) is built:
   $f_1(0200, \emptyset, [(M_1''.e_1, [\textit{Sancerre}]), (M_1''.e_2, [2000])]) = [(M_1'.e_4, [\textit{Sancerre}, 2000])]$;
   $f_2(030, \emptyset, [(M_2''.e_1, [\textit{Sancerre}]), (M_2''.e_2, [2000])]) = [(M_2'.e_4, [\textit{Sancerre}, 2000])]$.
   The construction of $\textit{keyTree}_{K_1}$ continues and $\textit{keyTree}_{K_1}[t, 0200]$ is obtained taking into account the information available at position 0200. (See subtree rooted at *target* in Fig. 3).
4. The computation continues up to the context, verifying whether the labels visited are recognized by the target automaton or not and carrying up the key (or foreign key) values. For instance, we reach state e_5 in M_1' by reading the label "*drinks*" (Fig. 4):
   $f_1(020, \emptyset, [(M_1'.e_4, [\textit{Sancerre}, 2000])]) = [(M_1'.e_5, [\textit{Sancerre}, 2000])]$;
5. For the node 0, the label *restaurant* is a context label of both K_1 and FK_2. For K_1 (respectively FK_2) the output function selects the sublists associated to a final state of the target automaton M_1' (respectively M_2'). The output function of K_1 checks if all the selected sublists are distinct. The output function of FK_2 verifies if the selected sublists correspond to lists of values obtained for K_1. In both cases, the output functions return a boolean value that will be carried up to the root:
   $f_1(0, \emptyset, [(M_1'.e_6, [\textit{Sancerre}, 2000]), (M_1'.e_6, [\textit{Cahors}, 2002])]) = [(M_1.e_8, [\textit{true}])]$;
   $f_2(0, \emptyset, [(M_2'.e_5, [\textit{Sancerre}, 2000])]) = [(M_2.e_7, [\textit{true}])]$.
   At this point, we have $\textit{keyTree}_{K_1}[t, 0]$ represented by the subtree rooted at *context* in Fig. 3. Notice that the attribute `refCount` for tuple $\langle$*Sancerre*, 2000$\rangle$ has value 1 because at this context node, the tuple $\langle$*Sancerre*, 2000$\rangle$ exists for foreign key FK_2. Indeed, at the context level we increment the `refCount` of each key tuple that corresponds to a foreign key tuple obtained at this level. Supposing that the tuple $\langle$*Cahors*, 2002$\rangle$ appears in three different combinations (not presented in Figure 1), we would have `refCount`$= 3$ for it.
6. At the root position the last output function selects the sublists that are preceded by a final state of the context automaton M and returns all boolean values in these sublists. The construction of $\textit{keyTree}_{K_1}$ finishes by a label indicating the name of the key (Fig. 3). □

Definition 5. A run of $\mathcal{U}$ on a finite tree t: Let t be a Σ-valued tree and $\mathcal{U} = (Q, \Sigma, \mathbf{D}, Q_f, \Delta, \Gamma)$ be a UTT. Given the keys $K_1, \ldots, K_k$ and foreign keys FK_{k+1}, $\ldots, FK_n$ a **run** of $\mathcal{U}$ on t is: ***(i)*** a tree $r : dom(r) \rightarrow Q$ such that $dom(r) = dom(t)$; ***(ii)*** a function $\pounds : dom(r) \rightarrow (\mathbf{D}^*)^n$ and ***(iii)*** k index trees: in each $\textit{keyTree}_{K_j}$ the leaves contain the values that compose K_j.

For each position p whose children are those at positions[5] $p0, \ldots, p(z-1)$ (with $z \geq 0$), we have $r(p) = q$ and $\pounds(p) = l$ if and only 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 $0 \leq i \leq (z-1)$ such that the children of p (*i.e.*, the positions $p0, \ldots, p(z-1)$) can be classified according to the following rules:
   - the positions $p0, \ldots, p(i-1)$ are members of a set *posAtt* (possibly empty)
   - the positions $pi, \ldots, p(z-1)$ are members of a set *posEle* (possibly empty)
   - every child of p is a member of *posAtt* or of *posEle* but no position is in both sets.

[5] The notation $p(z-1)$ indicates the position resulting from the concatenation of the position p and the integer $z-1$. If $z = 0$ the position p has no children.

4. The tree r and the function £ are already defined for positions $p0, \ldots, p(z-1)$. We assume $r(p0) = q_0, \ldots, r(p(z-1)) = q_{z-1}$ and £$(p0) = l_0, \ldots,$ £$(p(z-1)) = l_{z-1}$ where each $l_i = \langle l_i^1, \ldots, l_i^n \rangle$ is a n-tuple.
5. The trees *keyTree*$_{K_j}[t, p0]$, ..., *keyTree*$_{K_j}[t, p(z-1)]$ are already computed, *i.e.*, the construction of *keyTree* for each key K_j has already taken into account the information associated to positions $p0 \ldots p(z-1)$.
6. The word $q_i \ldots q_{z-1}$, composed by the concatenation of the states associated to the positions in *posEle*, belongs to the language generated by E.
7. The sets of S ($S_{compulsory}$ and $S_{optional}$) respect the following properties: $S_{compulsory} \subseteq \{q_0, \ldots, q_{i-1}\}$ and $(\{q_0, \ldots, q_{i-1}\} \setminus S_{compulsory}) \subseteq S_{optional}$.
8. The output £(p) associated to position p is the n-tuple: $l =$ £$(p) = \langle f_1(p, s_1,$ *concat*$(l_i^1, \ldots, l_{z-1}^1)), \ldots, f_n(p, s_n,$ *concat*$(l_i^n, \ldots, l_{z-1}^n))\rangle$ where each $s_j = \{(t(ph), l_h^j) \mid 0 \leq h \leq (i-1)\}$ is the set of all pairs (*attName*, *listOfValues*) coming from the attribute children of p, for each K_j or FK_j.

Moreover, for each position p, the *keyTree*s are constructed as follows:

(a) If $t(p)$ is a key label of K_j, then *keyTree*$_{K_j}[t, p]$ is the tree: `<key>` $t(p) =$ *value*$(t, p0)$`</key>`
(b) If $t(p)$ is a target label of K_j, then *keyTree*$_{K_j}[t, p]$ is: `<target pos=p refCount=0>` *keyTree*$_{K_j}[t, p0] \ldots$ *keyTree*$_{K_j}[t, p(z-1)]$ `</target>`
(c) If $t(p)$ is a context label of K_j, then *keyTree*$_{K_j}[t, p]$ is: `<context pos=p>` *keyTree*$_{K_j}[t, p0] \ldots$ *keyTree*$_{K_j}[t, p(z-1)]$ `</context>`
(d) If $t(p)$ is a context label of FK_j, then increment the attribute `refCount` in the corresponding *keyTree*$_{K_i}$.
(e) If $t(p)$ is the root label then *keyTree*$_{K_j}[t, p]$ is the tree: `<root>` *keyTree*$_{K_j}[t, p0] \ldots$ *keyTree*$_{K_j}[t, p(z-1)]$ `</root>`
(f) In all other cases, for each key K_j, we define *keyTree*$_{K_j}[t, p]$ as the forest composed by all the trees *keyTree*$_{K_j}[t, p0] \ldots$ *keyTree*$_{K_j}[t, p(z-1)]$.

Notice that, although the *keyTree*s are defined in general as forests, for the special labels mentioned in cases (a) to (d) above, we build a single tree. □

Definition 6. Validity: An XML tree t is said to be valid with respect to schema constraints if there is a successful run r, *i.e.*, $r(\epsilon) \in Q_f$. An XML tree t is said to be valid with respect to key and foreign key constraints if the lists of £(ϵ) contain only the value $true$ for each key and foreign key. □

Notice that, in step 8 of Definition 5, the output for each position p in the XML tree is a tuple composed by one list for each key (or foreign key) being verified. Each list in the tuple is the result of applying the output function f_j, defined for the jth key or foreign key, over the following arguments:

- p: the position in dom(t).
- s_j: the set of pairs (*att*, l_h^j) where *att* is the attribute name of the hth child of p and l_h^j is a nonempty list containing the value associated to this attribute.
- *concat*$(l_i^j, \ldots, l_{z-1}^j)$: the list formed by the information carried up from the element children of p, concerning the jth key.

At the end of the run over an XML tree, each key K_j is associated to a *keyTree* K_j that respects the general schema given by Fig. 2. Attribute `pos` stores the target and context positions for a given key and attribute `refCount` indicates when a key K_j is referenced by a foreign key.

4 Incremental Validation of Updates

We consider two update operations, denoted by *insert*$(\mathcal{T}, p, \mathcal{T}')$ and *delete*$(p, \mathcal{T})$, where $\mathcal{T}$ and $\mathcal{T}'$ are XML trees and p is a position. Fig. 5 illustrates these operations on a Σ-valued tree. Only updates that preserve validity wrt the constraints are accepted.

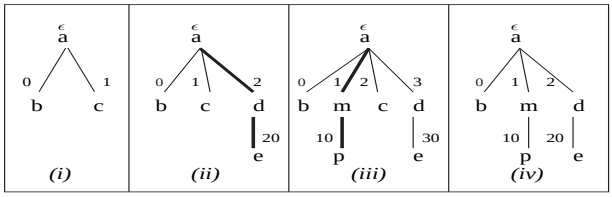

Fig. 5. (i) Initial Σ-valued tree t having labels a (position ϵ), b (position 0) and c (position 1). (ii) Insertion at $p = 2$. (iii) Insertion at $p = 1$. (iv) Deletion at $p = 2$.

4.1 Incremental key and foreign key validation

Let $\mathcal{T} = (t, type, value)$ be a valid XML tree, *i.e.*, one satisfying a collection of keys K_j $(1 \leq j \leq k)$ and foreign keys FK_j $((k+1) \leq j \leq n)$. Let $\mathcal{U} = (Q, \Sigma, \mathbf{D}, Q_f, \Delta, \Gamma)$ be a UTT specifying all the constraints that should be respected by $\mathcal{T}$. We should consider the execution of $\mathcal{U}$ over a subtree $\mathcal{T}'$ being inserted or deleted.

Given a subtree $\mathcal{T}' = (t', type, value)$, the execution of $\mathcal{U}$ over $\mathcal{T}'$ gives a tuple:

$$\langle q', \langle l_1, \ldots, l_n \rangle, \langle \textit{keyTree}_{K_1}[t', p], \ldots, \textit{keyTree}_{K_k}[t', p] \rangle \rangle \quad (1)$$

where q' is the state associated to the root of t', $\langle l_1, \ldots, l_n \rangle$ is a n-tuple of lists and $\langle \textit{keyTree}_{K_1}[t', p], \ldots, \textit{keyTree}_{K_k}[t', p] \rangle$ is a k-tuple containing the *keyTree* for each key. Notice that the n-tuple of lists has two distinct parts. Lists $l_1, \ldots, l_k$ represent keys and lists $l_{k+1}, \ldots, l_n$ represent foreign keys. Each l_j $(1 \leq j \leq n)$ is a list of pairs, *i.e.*, each l_j has the form $[c_1, \ldots, c_m]$ where each c_h is a pair containing an automaton configuration and a list of values.

When performing an insertion, we want to ensure that $\mathcal{T}'$ has no "internal" validity problems (as, for instance, duplicated values for K_j). Thus, we define $\mathcal{T}'$ as *locally valid* if the tuple (1) respects the following conditions: (A) q' is a state in Q and the ID attributes are unique in t' (see details in [5]); (B) for each list l_j $(1 \leq j \leq k)$ we have:

(i) if the root of t' is a target position for K_j then the list l_j has length m_j (*i.e.*, its length equals the number of elements composing a key tuple for K_j);
(ii) if the root of t' is a context position for K_j then the list l_j is $[(M_j.e, [true])]$;
(iii) if the root of t' is a position above the context positions for K_j then the list l_j is $[c_1, \ldots, c_m]$, where each pair c_h does not contain $[false]$ as its list of values.

Notice that no condition is imposed on foreign keys. A subtree $\mathcal{T}'$ can contain tuple values referring to a key value appearing in $\mathcal{T}$ (and not in $\mathcal{T}'$).

In the following, we assume that subtrees being inserted in a valid XML tree are locally valid and we address the problem of evaluating whether an update should be accepted with respect to key and foreign key constraints. Before accepting an update, we incrementally verify whether it does not cause any constraint violation. To perform these tests, we need the context node of a key or foreign key. To this end, we define procedure *findContext* that computes:

- The context position p' for a key K_j (or a foreign key FK_j) which is an ancestor of the update position p in the tree t.
- A list l' containing the key (or foreign key) values carried up from the subtree being inserted or deleted.[6]

The tests performed for insertion operation *insert*$(\mathcal{T}, p, \mathcal{T}')$ are presented next. Recall that $\mathcal{T}$ is valid and $\mathcal{T}'$ is locally valid.

Algorithm 2 - Incremental tests for update operation *insert*$(\mathcal{T}, p, \mathcal{T}')$

1. For each list $l_j \neq [\,]$ $(1 \leq j \leq k)$ obtained in the execution of $\mathcal{U}$ over $\mathcal{T}'$ for each key K_j do
   (a) If p is under a context node of K_j then
      i. Call *findContext*(p, l_j), that returns a context position p' and $l' = [v_1, ..., v_r]$.
      ii. For each list v in l' do
         If there exists a tuple *kval* in *keyTree*$_{K_j}[t, p']$ such that *kval* $= v$
         then the insertion violates K_j and must be rejected
         else the insertion respects K_j.
   (b) If p is the context position or it is between the root and a context node of K_j then the insertion respects K_j.
2. For each $l_j \neq [\,]$ $((k+1) \leq j \leq n)$ obtained in the execution of $\mathcal{U}$ over $\mathcal{T}'$ do
   (a) Call *findContext*(p, l_j), that returns a context position p' and $l' = [v_1, ..., v_r]$.
   (b) For each list v in l' do:
      If there exists a tuple *kval* in the *keyTree*$_{K_i}$ such that *kval* $= v$
      then the insertion respects the foreign key FK_j. The reference counter that corresponds to *kval* will be incremented at the end of the procedure, if the insertion is accepted.
      else the insertion does not respect the foreign key FK_j and must be rejected.
3. If all keys and foreign keys, together with schema constraints [5], are respected then accept the update and perform the modifications to $\mathcal{T}$ and all *keyTree*s.
   else reject the update. □

Before performing an insertion, Algorithm 2 tests if we are not adding key duplicates on $\mathcal{T}$ and if the new foreign key values correspond to key values. When we refer to a tuple in a *keyTree*, this tuple is obtained by concatenating the key values found inside target tags of this *keyTree*, taking into account a context position p' . The next example illustrates an insertion operation with respect to key and foreign key constraints.

[6] Let l_j be the list of pairs obtained for K_j or FK_j by the local validity check. Procedure *findContext* executes the automaton $\overline{M}$ (composition of M_j'' and M_j') starting from the configurations in l_j and using the labels associated to the ancestors of position p [1].

Example 5. We consider the update ***insert***$(\mathcal{T}, 0200, \mathcal{T}')$ presented in Example 3. The execution of $\mathcal{U}$ over $\mathcal{T}'$ gives the tuple: $\langle q_{wine}, \langle[\langle M_1'.e_4, [\textit{Bordeaux}, 1990]\rangle], [\,]\rangle, \langle \textit{keyTree}_{K_1}[t', \epsilon]\rangle\rangle$. We see that $\mathcal{T}'$ is locally valid and that the update affects only K_1. Procedure ***findContext*** returns the context position $p' = 0$ and the list $l' = [\langle \textit{Bordeaux}, 1990\rangle]$. We compare the tuples in l' with those in $\textit{keyTree}_{K_1}$ (Figure 3) for context $p' = 0$. All these tuples are distinct and thus the insertion is possible for K_1. As no other key is affected, the insertion is accepted. □

In a similar way, we define incremental tests for the operation ***delete***$(p, \mathcal{T})$. These tests check if the deletion of a subtree rooted at a position p does not violate constraints, before actually removing the subtree. The details are given in [1].

5 Conclusions

This paper extends and merges our previous proposals [5, 6]. In [5], we propose an incremental validation method, but only with respect to schema constraints. The validation of updates is also treated in [10, 16], and in [5] these approaches are compared to ours. In [6] we just consider the validation from scratch of an XML document associated to only one key constraint. In the current paper, we deal with incremental validation of updates taking into account schema constraints together with several key and foreign key constraints. Our verification algorithm uses only synthesized values (*i.e.*, values communicated from the children to the parents of a tree), making the algorithms suitable for implementation in any parser generator, or even using SAX [13] or DOM [18].

The algorithms presented here have been implemented using the ASF+SDF meta-environment [7]. The verification of keys and foreign keys uses *KeyTree*s, which can also be used for efficiently evaluating queries based on key values.

Validity verification methods for schema constraints have been addressed by [5, 10, 14–17]. Key constraints for XML have been recently considered in the literature (for instance, in [2, 4, 6, 8, 9]) and some of their aspects are adopted in XML Schema. In our paper, the definition of integrity constraints follows the key specification introduced in [8]. As shown in [11], it is easy to produce examples of integrity constraints that no XML document (valid wrt a schema) can verify. In our work, we assume key and foreign key constraints consistent with respect to a given DTD.

In [9] a key validator which works in asymptotic linear time in the size of the document is proposed. Our algorithm also has this property. In contrast to our work, in [3, 9] schema constraints are not considered and foreign keys are not treated in details. In [4] both schema and integrity constraints are considered in the process of generating XML documents from relational databases. Although some similar aspects with our approach can be observed, we place our work in a different context. In fact, we consider the evolution of XML data independently from any other database sources (in this context both validation and re-validation of XML documents can be required).

We are currently considering the following lines of research: (i) An extension of our method to deal with other schema specification, for instance XML-Schema and specialized DTDs. (ii) An implementation of an XML update language such as UpdateX [12] in which incremental constraint checking will be integrated. To this end, we shall consider a sequence of updates as one unique transaction and check validity of its result. We also shall take into account *KeyTree*s for efficiently locating update positions.

References

1. M. A. Abrao, B. Bouchou, M. Halfeld-Ferrari, D. Laurent, and M. A. Musicante. Update validation for XML in the presence of schema, key and foreign key constraints. Technical report, Université François Rabelais Blois-Tours-Chinon, 2004 (to appear).
2. M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *ACM Symposium on Principles of Database System*, 2002.
3. M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated update management for XML integrity constraints. In *Programming Language Technologies for XML (PLANX02)*, 2002.
4. M. Benedikt, C-Y Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constraints in data integration. In ACM Press, editor, *SIGMOD, San Diego, CA*, 2003.
5. 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.
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. 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.
8. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. C. Tan. Keys for XML. In *WWW10, May 2-5*, 2001.
9. Y. Chen, S. B. Davidson, and Y. Zheng. XKvalidator: a constraint validator for XML. In ACM Press, editor, *Proceedings of the 11th International Conference on Information and Knowledge Management*, pages 446–452, 2002.
10. B. Chidlovskii. Using regular tree automata as XML schemas. In *Proc. IEEE Advances in Digital Libraries Conference*, May 2000.
11. W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3):368–406, 2002.
12. 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.
13. W. S. Means and M. A. Bodie. *The Book of SAX: The Simple API for XML*. No Starch Press, 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. M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema language using formal language theory. In *Extreme Markup Language, Montreal, Canada*, 2001.
16. Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2003.
17. L. Segoufin and V. Vianu. Validating streaming XML documents. In *ACM Symposium on Principles of Database System*, 2002.
18. L. Wood, A. Le Hors, V. Apparao, S. Byrne, M. Champion, S. Issacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor, and C. Wilson. *Document Object Model (DOM) Level 1 Specification*. W3C Recommendation, http://www.w3.org/XML, 2000.