Tree Automata to Verify XML Key Constraints

Béatrice Bouchou
Université de Tours
LI/Antenne Univ. de Blois
3 place Jean Jaurès
41000 Blois, France
bouchou@univ-tours.fr

Mírian Halfeld Ferrari
Université de Tours
LI/Antenne Univ. de Blois
3 place Jean Jaurès
41000 Blois, France
mirian@univ-tours.fr

Martin A. Musicante*
Univ. Federal do Paraná
Dep. de Informática
C.P. 19081
81531-970 - Curitiba - Brazil
mam@inf.ufpr.br

ABSTRACT

We address the problem of checking key constraints in XML. Key constraints have been recently considered in the literature and some of their aspects are adopted in XMLSchema. However, only few works have appeared concerning the verification of such constraints.

Unranked deterministic bottom-up tree automata can be used to validate XML documents against a schema. These automata work over (unranked) trees used to represent XML documents.

In this paper we show how key constraints can be integrated in such automaton by extending the automaton to carry up values from the leaves to the root, during its run. In fact the tree automaton becomes a tree transducer. Under these conditions, the key verification is done in asymptotic linear time on the size of the document.

Keywords: XML key constraints, tree automata, XML

1. INTRODUCTION

We consider a data-exchange environment where an XML document should respect two kinds of constraints: schema constraints and key constraints. Schema constraints correspond to attribute and element restrictions. Key constraints give the possibility of identifying data without ambiguity and, therefore, introduce a value-based method of locating items in a document. Keys for XML have received more attention recently. They exist in XMLSchema [2] and some formal definitions have been introduced in [9, 10].

We address the problem of validating both schema and key constraints. To this end we use bottom-up tree transducers as an extension of the tree automata introduced in [7]. Tree automata are a natural way of describing structural constraints. However, in order to validate key constraints we need to manipulate data values. To this end, we introduce output functions whose basic task is to define the values to be carried up from children to parents in an XML tree.

In this work we concentrate our attention to the validation of key constraints as defined in [9]. We present an efficient key validator whose work consists in executing a tree transducer over an XML document.

We see an XML document as a structure composed by an unranked labeled tree 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 to a node. Figure 1 shows part of the labeled tree representing the document used in our examples. Each node is represented by a label and a position (for instance, position 0 is associated to the label *politicPos*). Moreover, in this figure, an XML element has both its sub-elements and attributes as children. Elements and attributes associated with an arbitrary text have a child labeled *data*. Attribute labels are depicted with a preceding @. The following example illustrates how the tree transducer performs the validation of a key constraint.

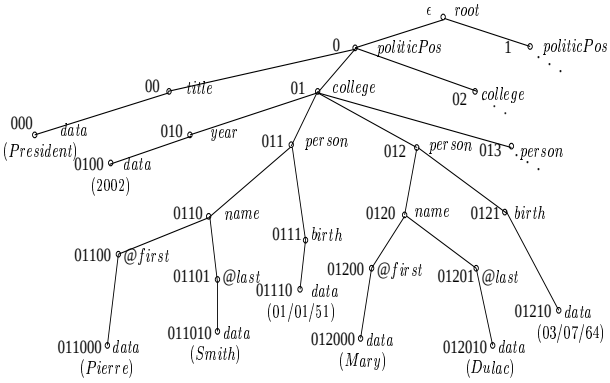

Figure 1: Tree representation of an XML document.

Example 1.1 We suppose the document of Figure 1, describing the organization of indirect elections, *i.e.*, the electoral colleges that vote for different political positions. Now consider the constraint expressed by $(/politicPos, (./college, \{./year\}))$ to indicate that for a political position, an electoral college can be uniquely identified by the year of the election. In other words, *year* is a key for an electoral college voting for a given political position. We say that *politicPos* defines context nodes, *college* defines target nodes and *year* defines key nodes.
In order to verify the above constraint, we execute our tree transducer over the tree of Figure 1. This execution consists of visiting the tree in a bottom-up manner, according to the following steps:

1. The tree transducer computes the values associated to all nodes labeled *data*.

*On leave at Université de Tours. Partly supported by CAPES (Brazil) BEX1851/02-0.

International Workshop on the Web and Databases (WebDB).
June 12–13, 2003, San Diego, California.

We consider $value(0100) = 2002$, $value(0200) = 1998$ as some of the values computed in this step[1].

2. The tree transducer analyzes the parents of the *data* nodes. If they are key nodes, they receive the values computed in step 1. Otherwise, no value is carried up.

   In our case, the values 2002 and 1998 are passed to the key nodes 010 and 020, respectively.

3. The tree transducer continues its execution just passing the values from children to parent until it finds a target node. At this level the values for each key are grouped in a list.

   In our case, the node 01 (labeled *college*) is a target node. As the key is composed by just one item, the list contains only the value 2002.

4. This step consists of carrying up the lists of values obtained in the previous step until finding a context node. At this level, the transducer tests if all the lists are distinct, returning a boolean value.

   In our case, *politicPos* is a context node. It receives several lists, each one containing the year of a college. The test verifies the uniqueness of those years.

5. The boolean values computed in step 4 are carried up to the root. At the root, the conjunction of these boolean values is obtained.

The key constraint $(/politicPos, (./college, \{./year\}))$ is satisfied if the conjunction computed at step 5 results in $true$. □

Given the tree representation $\mathcal{T}$ of an XML document, we can test schema and key constraints by a single bottom-up visit of the labeled tree. Example 1.1 illustrates how the key constraint verification is performed. In [7] schema constraint verification is treated in details (see Section 3 for a short explanation on this aspect).

The main contributions of our work are

- An efficient validator for both schema and key constraints. The validation is performed by the bottom-up visit of the XML tree, in only one pass.
- A method for allowing the use of DTDs with key constraints. DTDs are easily translated into a tree automata [7]. In this work we give an algorithm to add key constraint verification to the tree automata. In this way, our verification takes into account both *unique* and *not null* properties of the key.
- An unranked bottom-up tree transducer (a generalization of the ranked one [11]) where syntactic and semantic aspects are well separated. Schema validation deals with syntactic (structural) features of an XML tree (a plain tree automata can be used to this end). Key validation is about semantics. It requires an extension of tree automata allowing the manipulation of data values.

The rest of this paper is organized as follows. In Section 2 we recall the definitions of unranked labeled trees and key constraints. In Section 3, unranked tree transducers are introduced as an extension of tree automata. We present a method to express key constraints as output functions and we show that, in this way, an efficient verification of key constraints is possible. Finally, in Section 4, we consider some related work, and we discuss our perspectives for further research. Proofs are omitted due to lack of space.

[1]Node 0200 does not appear in Figure 1, but from the definition of positions in Section 2, it is easy to see that it is a grand-child of node 02.

2. XML TREES AND KEY CONSTRAINTS

In this section we recall the notions of unranked Σ-valued trees and key constraints.

Firstly, let U be the set of all finite strings of positive integers with the empty string ϵ as the identity. In the following definition we assume that $dom(t) \subseteq U$ is a nonempty set closed under prefixes[2], *i.e.*, if $u \preceq v$, $v \in dom(t)$ implies $u \in dom(t)$.

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

Unranked trees can be used to represent an XML document. In fact, there are different ways to encode an XML document as a tree. The following definition introduces our choice of representation.

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

- t is a Σ-valued tree (*i.e.*, $t : dom(t) \to \Sigma$).
- $type$ and $value$ are functions defined as follows for a position $p \in dom(t)$:

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

$$value(p) = \begin{cases} val \in \mathbf{V} & \text{if } type(p) = data \\ undefined & \text{otherwise} \end{cases}$$

where $\mathbf{V}$ is an infinite (recursively enumerable) domain. □

An XML key constraint over $\mathcal{T}$ is defined in three steps. In the first step we identify a set of positions from the root as the context in which the key must hold. In the second step, we obtain a set of target positions on which the key is being defined. Finally, we specify the set of values that distinguish each target position. We use a subset of XPath expressions, as in [10], to specify context and target positions and to obtain the values that compose keys.

Using the syntax of [9] a key can be written as

$$(P, (P', \{P_1, ..., P_k\}))$$

where P, P' and $P_1, ..., P_k$ are path expressions. P is called the *context path*, P' the *target path* and $P_1, ..., P_k$ the *key paths*. Figure 2 shows a Σ-valued tree with the positions we can reach by following each path. Level 3 corresponds to the root of the tree (reached by the path "/"). The context path begins at the root and specifies a set of *context positions*, shown in level 2. We say that these positions are associated to context labels. From each context position p, we define a set of *target positions* (associated to target labels) corresponding to the nodes reachable from p by following the path P' (level 1). The key constraint specified by $P_1, ..., P_k$ must hold for every target position. Level 0 corresponds to the positions composing a key (each of them associated to a key label).

Now, in [9], we find different types of keys, *i.e.*, different definitions of the semantics of a key. In our work, we adopt the definition called *strong keys*. Moreover, we consider that key paths

[2]The *prefix relation* in U, denoted by $\preceq$ is defined by: $u \preceq v$ iff $uw = v$ for some $w \in U$.

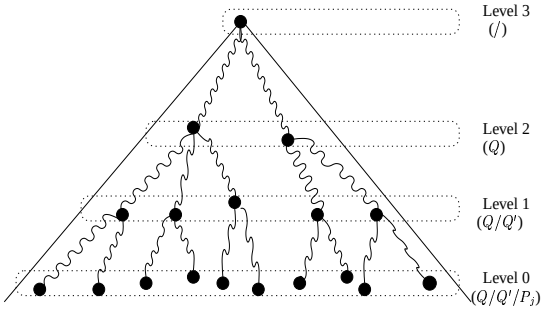

Figure 2: Context, target and key positions in an XML tree.

must define attributes or elements which occur *exactly once* and are associated to a *data* node. This restriction is also present in XLMSchema [10]. In the following we formalize the notion of key satisfaction and we present some examples of XML keys.

Definition 2.3 - Key satisfaction [9]: An XML tree $\mathcal{T}$ is said to satisfy a key $(P, (P', \{P_1, ..., P_k\}))$ iff for each context position p defined by P the following conditions hold:
(i) For each target position p' reachable from p via P' there exist a unique position p_j from p', for each $P_j (1 \leq j \leq k)$.
(ii) For any target positions p', p'', reachable from p via P', whenever the values reached from p' and p'' via $P_j (1 \leq j \leq k)$ are equal, then p' and p'' must be the same position. □

Example 2.1 Considering the document represented by Figure 1, we write the following keys:
• $K_1 = (/(./politicPos, \{./title\}))$
Within the context of the whole document ("/" denotes the empty path from the root), a political position is identified by its title.
• $K_2 = (/politicPos, (./college, \{./year\}))$
For a political position, an electoral college can be uniquely identified by the year of the election.
• $K_3 = (\ /politicPos/college, (./person,$
$\{./name/@first, ./name/@last, ./birth\}))$
Within a electoral college, a person can be uniquely identified by the composition of his/her first, last name and birthday.
Its worth to remark that in K_3 we cannot replace "$/name/@first$, $./name/@last$" by "$./name$" since in this case the key values are XML trees rather than a *data* node (*i.e.*, a text). □

3. TREE TRANSDUCERS FOR XML

We consider an XML tree $\mathcal{T}$ that should respect a given schema and some key constraints. We introduce an unranked tree transducer capable of verifying both the schema and the key validity. This transducer extends the tree automaton of [7] by associating an output function to each transition rule. In this way, we allow values to be carried up from leaves to the root. In the following we formalize the concepts of output function and unranked tree transducer.

Definition 3.1 - Output function: Let **D** be an infinite (recursively enumerable) domain and $\mathcal{T} = (t, type, value)$ be an XML tree. An *output function* f takes as arguments: (i) a position $p \in dom(t)$, (ii) a set s of pairs (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, $f : dom(t) \times \mathcal{P}(\Sigma \times \mathbf{D}^*) \times \mathbf{D}^* \rightarrow \mathbf{D}^*$. □

The notation $\mathbf{D}^*$ denotes all the lists of items in **D**.

Definition 3.2 - Unranked bottom-up tree transducer (UTT): An 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 Γ is a set of output functions. *For each transition rule in Δ, there is an output function f in Γ.*
Each transition rule in Δ has the form $a, S, E \rightarrow q$ where *(i)* $a \in \Sigma$; *(ii)* S is a set of two disjoint sets of states, *i.e.*, $S = \{S_{compulsory}, S_{optional}\}$ (with $S_{compulsory} \subseteq Q$ and $S_{optional} \subseteq Q$); *(iii)* E is a regular expression over Q and *(iv)* $q \in Q$. Each output function in Γ has the form $f(p, s, l) = l_1$ as in Definition 3.1. □

Now, we consider the execution of an UTT on a Σ-valued tree t (in $\mathcal{T}$). As t represents an XML document, the children of any position $p \in dom(t)$ can be classified into two groups: those that are unordered, corresponding to the attributes of the node, and those that are ordered, corresponding to the sub-elements.

The transducer states two types of constraints: schema constraints and key constraints. Schema constraints are stated by the transition rules in Δ. In order to verify these constraints, *i.e.*, to assume a state q at position p, the transducer $\mathcal{U}$ performs the following tests:

1. If p has attribute children then the states assumed for them should correspond to those specified by the sets in S, namely, $S_{compulsory}$ and $S_{optional}$, corresponding, respectively, to p's children that *must* appear in the tree and those that *may* appear[3].
2. If p has element children then the concatenation of the states assumed at them must belong to the language generated by the regular expression E.

Key constraints are expressed by the output functions in Γ (see Algorithm 3.1). As the tree is to be processed bottom-up, the basic task of the output functions is to define the values that will be passed to the parent position, during the run.

Definition 3.3 - 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 an UTT. A **run** of $\mathcal{U}$ on t is: *(i)* a tree $r : dom(r) \rightarrow Q$ such that $dom(r) = dom(t)$ and *(ii)* a function $\pounds : dom(r) \rightarrow \mathbf{D}^*$, defined as follows:
For each position p whose children are those at positions[4] $p0$, ..., $p(n-1)$ (with $n \geq 0$), 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 with an associated output function f in Γ.
3. There exists an integer $0 \leq i \leq (n-1)$ such that the children of p (*i.e.*, the positions $p0, \ldots, p(n-1)$) can be classified[5] according to the following rules:
   (a) the positions $p0, \ldots, p(i-1)$ are members of a set *posAtt* (possibly empty) and
   (b) the positions $pi, \ldots, p(n-1)$ are members of a set *posEle* (possibly empty) and
   (c) every children of p is a member of *posAtt* or of *posEle* but no position is in both sets.

[3]These sets correspond to the *required* and *implied* attributes of a DTD.
[4]The notation $p(n-1)$ indicates the position resulting from the concatenation of the position p and the integer $n-1$. If $n = 0$ the position p has no children.
[5]In an XML tree, the children at positions $p0, \ldots, p(i-1)$ correspond to attributes and the positions $pi, \ldots, p(n-1)$ correspond to elements.

4. The tree r and the function $\pounds$ are already defined for positions $p0, \ldots, p(n-1)$. We suppose $r(p0) = q_0, \ldots, r(p(n-1)) = q_{n-1}$ and $\pounds(p0) = l_0, \ldots, \pounds(p(n-1)) = l_{n-1}$.
5. The word $q_i \ldots q_{n-1}$, composed by the concatenation of the states associated to the positions in *posEle*, belongs to the language generated by E.
6. The sets of S ($S_{compulsory}$ and $S_{optional}$) respect the following properties:
   (a) $S_{compulsory} \subseteq \{q_0, \ldots, q_{i-1}\}$ and
   (b) $(\{q_0, \ldots, q_{i-1}\} \setminus S_{compulsory}) \subseteq S_{optional}$.
7. Given $s = \{(t(pj), l_j) \mid 0 \leq j \leq (i-1), l_j \neq [\,]\}$, the output associated to position p is
$$l = \pounds(p) = f(p, s, concat(l_i, \ldots, l_{n-1})).$$

We say that a run r is *successful* if $r(\epsilon)$ is a final state of the automaton. The *output of a run* is given by $\pounds(\epsilon)$. □

For a given XML tree, the existence of a successful run of an UTT implies that the document conforms to the DTD [7].

Notice that, in step (7) of Definition 3.3, the output function for each node is defined in terms of the position p (the first argument of the function f) and the result of the output functions of its children (the second and third arguments). The argument s corresponds to the set of pairs (*att*, *l*). This set is formed by pairs containing the name and the outputs of the attributes of p. The third argument is obtained by concatenating the outputs coming from the sub-elements of p.

In what follows, we aim to construct the output function of an UTT, in such a way that the value associated to the root after a successful run ($\pounds(\epsilon)$) is a list containing a truth value, which will be true if and only if the key constraint is verified.

In order to verify a key $K = (P, (P', \{P_1, ..., P_k\}))$ we should:

1. Collect the values associated to the positions defined by the paths $P_1, ..., P_k$. Carry up these values taking into account if they correspond to attributes or elements.
2. At the target positions defined by P', group the values for each key, in lists (of k elements). Carry up these lists to the context level.
3. At context positions defined by P, verify the uniqueness condition. Carry up a boolean value denoting the result of this test.
4. At the root, calculate the conjunction of these boolean values.

The above operations should be performed for all paths leading from the root to a key node. Values not belonging to these paths should be discarded.

Context, target and key nodes in K are defined in a top-down fashion. In order to identify these nodes with a bottom-up automaton, we must traverse the paths stated by K in reverse. We keep a representation of the reversed paths in the form of finite automata.

Before presenting the algorithm that translates keys into output functions, we give an example of such translation. We use the notation $M.e$ to represent a configuration of the automaton M, *i.e.*, the current state e of the automaton M.

Example 3.1 We consider the verification of K_3 of Example 2.1 over the tree of Figure 1. We suppose an UTT whose transition rules represent some schema constraints and we want to add to it the output functions implementing the verification of K_3. The finite state automata (FSA) associated to K_3 are the ones given in Figure 3. Notice that M and M' represent respectively the paths $/politicPos/college$ and $./person$ of K_3 in reverse. The automaton M'' represents the disjunction of the paths $./name/@first$, $./name/@last$ and $./birth$ in reverse.

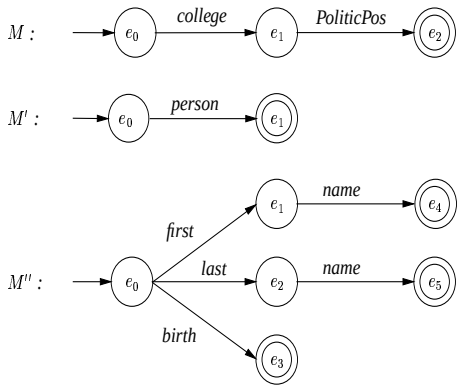

Figure 3: Automata corresponding to the paths of K_3 in reverse.

Following Figure 1, we consider firstly how the values concerning the element "$person$" at position 011 are carried up.

1) For the data nodes, the output functions have to get the data values, as well as to initiate the execution of the automaton to recognize the key paths. The output function, in this case, will return a singleton list, containing a pair (the initial configuration of the key automaton M'' and the value of the node):

$$f(011000, \emptyset, [\,]) = [(M''.e_0, [Pierre])]\,;$$
$$f(011010, \emptyset, [\,]) = [(M''.e_0, [Smith])]\,;$$
$$f(01110, \emptyset, [\,]) = [(M''.e_0, [01/01/51])].$$

2) The fathers of the data nodes mentioned in the previous step have key labels. For each of them the output function must transmit this information together with the value received from its child. Doing that, it executes a first transition of the key FSA M'', using each key label as input. For instance, reading the label $first$ from state e_0 we reach state e_1. Thus we should define:

$$f(01100, \emptyset, [(M''.e_0, [Pierre])]) = [(M''.e_1, [Pierre])];$$
$$f(01101, \emptyset, [(M''.e_0, [Smith])]) = [(M''.e_2, [Smith])];$$
$$f(0111, \emptyset, [(M''.e_0, [01/01/51])]) = [(M''.e_3, [01/01/51])]$$

where e_1, e_2, e_3 are the states reached by the key FSA M'' when reading the key labels (Figure 3).

Notice that data nodes which are not part of a key should not pass values to their fathers. Consider, for instance, the node at position 00. We have $f(00, \emptyset, [(M''.e_0, [President])]) = [\,]$.

3) Next we consider position 0110. This position has two attribute children. In this case, the output function must promote the attribute values only if the current label belongs to the inversed key paths (represented by M''). In our case we have:

$$f(0110, \{(@first, [(M''.e_1, [Pierre])]),$$
$$(@last, [(M''.e_2, [Smith])])\}, [\,]) =$$
$$[(M''.e_4, [Pierre]), (M''.e_5, [Smith])].$$

where e_4 and e_5 are the states reached by M'' when reading the label "$name$" from e_1 and e_2, respectively (Figure 3).

4) For the node 011, the label "$person$" is the target label; it receives from its children two lists of values (one from attributes and one from elements). In order to transmit only key values, the output function of a target label should (i) select those that are preceded by a final state of the key automaton M'', (ii) join them in a new list, and (iii) execute the first transition of the target FSA M'. In our case we have, for node 011:

$$f(011, \emptyset, [(M''.e_4, [Pierre]), (M''.e_5, [Smith]), (M''.e_3, [01/01/51])]) = [(M'.e_1, [Pierre, Smith, 01/01/51])].$$

The other target nodes will get their output values in a similar way. For instance, at position 012 we obtain the list $[(M'.e_1, [Mary, Dulac, 03/07/64])]$.

5) For the node 01, the label "$college$" is the context label, the output function should work in a similar way as in stage 4: it should (i) select the sublists that are preceded by a final state of the target automaton M'; (ii) check if all these sublists are distinct and (iii) execute the first transition of the context FSA M. So, in our case, for node 01, we have:

$$f(01, \emptyset, [(M'.e_1, [Pierre, Smith, 01/01/51]), (M'.e_1, [Mary, Dulac, 03/07/64)], \dots]) = [(M.e_1, [B])].$$

where B is true iff all the sublists selected in the third argument of f are distinct.

6) The computation should continue up to the root, verifying whether the labels visited are recognized by the context FSA or not: in our example,

$$f(0, \emptyset, [(M.e_1, [b])]) = [(M.e_2, [b])]$$

where e_2 is the state reached by M when reading the label "*Politic-Pos*" (Figure 3).

7) At the root position the last output function should select the sublists that are preceded by a final state of the context FSA M and should return the conjunction of all boolean values in these sublists. In our example:

$$f(\epsilon, \emptyset, [(M.e_2, [b_1]), \dots, (M.e_2, [b_m])]) = [(M_F.e_f, [B])]$$

where B is true iff all b_i are true. The (final) configuration $M_F.e_f$ is introduced to keep the homogeneity of the lists returned by the output function. It corresponds to the final configuration of an automaton M_F accepting only the root label. □

The following algorithm shows how output functions can be defined in order to represent a given key. Note that, since our UTT is an extension of a deterministic tree automaton having the same expression power of a non ambiguous DTD [7], a label $a \in \Sigma$ corresponds to a unique transition function and thus to a unique output function.

Algorithm 3.1 - Key constraints as output functions: Let $K = (P, (P', \{P_1, \dots, P_k\}))$ be a key. Let $M = \langle \Theta, \Sigma, \delta, e_0, F \rangle$ (respectively, $M' = \langle \Theta', \Sigma, \delta', e_0, F' \rangle$ and $M'' = \langle \Theta'', \Sigma, \delta'', e_0, F'' \rangle$) be the finite state automaton that recognizes the path P in reverse (respectively, the paths P' and $P_1 \mid \dots \mid P_k$ in reverse). Let $M_F = \langle \Theta_F, \{root\}, \delta_F, e_0, \{e_f\} \rangle$ be the automaton recognizing the path root (in reverse).

Let $\mathcal{U} = (Q, \Sigma, \mathbf{D}, Q_f, \Delta, \Gamma)$ be an UTT whose transition rules represent some schema constraints. The domain $\mathbf{D}$ is defined as $(\Theta \uplus \Theta' \uplus \Theta'' \uplus \Theta_F) \times \mathbf{V}^*$. We assume that the transition rules of $\mathcal{U}$ have the general form $a, S, E \rightarrow q_a$. In order to express the key K, each transition rule of $\mathcal{U}$ is associated to an output function defined according to its label a:

1. If $a = data$ (the rule has the form $data, \{\emptyset, \emptyset\}, \emptyset \rightarrow q_{data}$) then the output function is $f(p, s, l) = [(M''.e_0, [value(p)])]$.

2. If a is a target label then the output function is defined as: $f(p, s, l) = [(M'.\delta'(e_0, a), concat(\mathit{filter}_{key}(c_1, \dots, c_m)))]$ where:
$[c_1, \dots, c_{j-1}] = \Pi_{pair}(\mathit{orderByName}(s))$ and
$[c_j, \dots, c_m] = l$.
For the target nodes, the output list is composed by a pair containing (i) the configuration of the target automaton reached from its initial state e_0 by reading a and (ii) a list of all the values composing a key.
We impose an order to the pairs coming from attribute children (*i.e.*, those contained in s); this is done by sorting them in the lexicographic order of their tags and then eliminating these tags.
The function "filter_{key}" filters the key lists, leaving only the values associated to key positions. Notice that each c_k is a pair of an automaton configuration and a list of values. The filter selects the lists of values in the pairs whose configuration corresponds to a final state of M''. The list operation "*concat*" returns the concatenation of all its argument lists into one list.
Notice that at this step, the output function will return a singleton list, whose only element is a pair formed by a configuration of M' and a list of all the values that belong to the key.

3. If a is a context label then the output function is $f(p, s, l) = [(M.\delta(e_0, a), g(\mathit{filter}_{target}(l)))]$ where

$$g([v_1 \dots v_m]) = \begin{cases} [true] \text{ if } v_1 \dots v_m \\ \quad \text{are all distinct lists} \\ [false] \text{ otherwise} \end{cases}$$

The function "filter_{target}" simply filters the target lists in a similar way as key lists were filtered in the previous case. For the context nodes, we must check that the lists formed at the target level are all different. The result of this level is a singleton list that contains a pair. This pair is formed by a configuration of M and a list containing a boolean value (the result of checking the validity of the key for each specific context).

4. If a is the root label then the output function is $f(p, s, l) = AND(\mathit{filter}_{context}(l))$ where
$AND([b_1], \dots, [b_m]) = [(M_F.e_f, [\bigwedge_{j=1}^{j=m} b_j])]$.

At the root level, we calculate the conjunction of the truth values that were obtained for each subtree rooted at the context level. The function "$\mathit{filter}_{context}$" simply filters the context lists, as in the previous cases.

5. In all other cases (*i.e.*, when $a \neq data$ and a is not a target label, nor a context label, nor the root) the output function is defined as:

$$f(p, s, l) = h([c_1, \dots, c_m])$$

where , $[c_1, \dots, c_m]$ is the list of pairs obtained from the children of p, such that $[c_1, \dots, c_{j-1}] = \Pi_{pair}(\mathit{orderByName}(s))$ and $[c_j, \dots, c_m] = l$. In other words, key pairs coming from the attribute children of the node are sorted and then projected, as in step 2.

The function h is defined by the following procedure:

procedure h(L : **list of pairs**);
var result : **list of pairs**;
begin
result $\leftarrow$ [ ];
foreach $c = (\mathcal{M}.e, v)$ **in** L //* $\mathcal{M}$ stands for M, M' or M''
  if $\delta(e, a) = e'$ is a transition in $\mathcal{M}$ **then**
    result $\leftarrow$ concat($result$, $[(\mathcal{M}.e', v)]$);
return result;
end;

This definition concerns the positions that are at the key levels or that does not belong to levels 1, 2 or 3 in Figure 2. The resulting list is formed by pairs, each of which contains an automaton configuration and a list of values. □

The key verification is done in asymptotic linear time on the size of the document. Given the key constraint $(P, (P', \{P_1, ..., P_k\}))$ the validation process consists in a bottom-up visit of each XML tree we want to validate. The copy, concat and filtering operations performed during this validation have constant time complexity $O(k)$. Thus, verifying if the lists are distinct, at the context level, takes time $O(c^2)$ where c is the number of target nodes, *i.e.*, those reachable by the path $/P/P'$. Note that c usually represents a very small number when compared to the size of the tree.

The following theorem states that tree transducer generated by Algorithm 3.1 validates the key constraints.

Theorem 3.1 Let $K = (P, (P', \{P_1, ..., P_k\}))$ be an XML key over an XML tree $\mathcal{T}$. Let $\mathcal{U}$ be a tree transducer that expresses K according to Algorithm 3.1. Then,

$$\mathcal{T} \text{ satisfies } K \text{ iff } £(\epsilon) = [(M_F.e_f, [true])].$$ □

4. CONCLUSIONS

In this paper we show how both schema and key constraints can be represented and validated by bottom up tree transducers. Both constraints are verified in a single bottom-up visit of an XML tree. This approach can be very useful in the context of the incremental validation of updates to XML documents.

In [5, 9] we find different proposals for expressing keys. Some of them had been incorporated to schema languages such as XML-Schema [2] and Schematron [1]. In our work we use the notion of strong keys of [9]. Moreover, we consider that there is no inconsistency between key and schema. We refer to [4] as a survey on the problem of statically verifying the consistency of schema and key constraints.

Key validation is the subject of recent research [5, 10, 6]. In [5] a constraint language, designed to support incremental validation, is proposed. The incremental validation of constraints is done by translating constraints into logic formulas and then generating incremental constraint checking code from them. In [10] a key validator is proposed which works in asymptotic linear time in the size of the document. Our algorithm also has this property. In [10] (incremental) validation relies on the use of index. In contrast to our approach, schema constraints are not considered in [5, 10]. In [6] both schema and integrity constraints are considered in the process of generating XML documents from relational databases. They propose a formalism inspired by attribute grammars [12] with both synthesized (bottom-up evaluation) and inherited (top-down evaluation) data. 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 pursuing the following lines of research:

(i) The generalization of our method to treat more than one key constraint, as well as to treat more general schema definitions. We are in the process of verifying some properties respected by UTTs, such as their closure under intersection.

(ii) The introduction of a notion similar to attribute inheritance to our output functions (they already implement synthesized attributes). This notion should be similar to those of L-attributed grammars, as used in compiler construction [3] and should be implemented in a single-pass bottom-up tree transducer. This feature will allow the validation of other kinds of constraints.

(iii) The use of tree transducers for the incremental validation of key constraints. We aim at the extension of the incremental validation method for XML documents under schema constraints proposed in [8]. This method relies on the execution of a tree automaton only on the part of the XML tree affected by the update.

5. REFERENCES

[1] The Schematron: An XML structure validation language using patterns in trees. Available at http://www.ascc.net/xml/resource/schematron.

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

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

[4] M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *ACM Symposium on Principles of Database System*, 2002.

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

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

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

[8] B. Bouchou and M. Halfeld Ferrari Alves. Updates and incremental validation of XML documents. Submitted paper, 2003.

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

[10] Y. Chen, S. Davidson, and Y. Zheng. Validating constraints in XML. Technical Report MS-CIS-02-03, Department of Computer and Information Science, University of Pennsylvania, 2002.

[11] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Available on: *http://www.grappa.univ-lille3.fr/tata*, 1997 (new version 2002).

[12] P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars: Definitions, systems and bibliography. Number 323 in LNCS - Lecture Notes in Computer Science, 1988.