EXTENDING TREE AUTOMATA TO MODEL XML VALIDATION UNDER ELEMENT AND ATTRIBUTE CONSTRAINTS

B. Bouchou D. Duarte M. Halfeld Ferrari Alves D. Laurent
Université de Tours - Laboratoire d'Informatique de Tours -Antenne Universitaire de Blois
3 Place Jean Jaurès, 41000, Blois, France
{*bouchou, mirian, laurent*}*@univ-tours.fr* *denio.duarte@etu.univ-tours.fr*

Key words: Regular tree automata, XML validation, XML views, DTD, attribute constraints, element constraints

Abstract: Algorithms for validation play a crucial role in the use of XML. Although much effort has been made for formalizing the treatment of elements, attributes have been neglected. This paper presents a validation model for XML documents that takes into account element *and* attribute constraints imposed by a given DTD. Our main contribution is the introduction of a new formalism to deal with both kinds of constraints. To this end we propose an extension of regular tree automata that allows the construction of a *deterministic* automaton having the *same* expression power as that of a DTD. Our formalism gives rise to an efficient validation method.

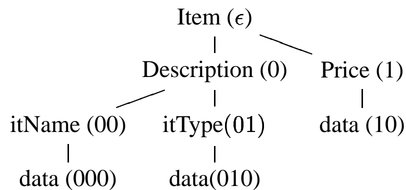
1 INTRODUCTION

In this paper we introduce an extension of regular tree automata in order to efficiently validate XML views (or documents), built from different data sources according to a particular schema. Efficiently validating XML views is particularly important when the schema and/or the view change.

An XML document is usually defined as a set of nested elements and represented by a *labeled tree*. However, it can contain other components, such as attributes, entities and hyper-textual references. Entities and references specify foreign information and once processed, the document content consists only of elements and attributes. By contrast to most of work concerning formal methods to treat XML documents (see (Neven, 2002b) for a survey), in this paper, we introduce a formalism which aims to treat not only *elements* but also *attributes*. To this end, we define a new tree automaton whose goal is to deal with finite trees having two different types of nodes: those that should be treated as members of a set (corresponding to attributes) and those that should be treated as part of a sequence (corresponding to elements).

Example 1.1 We consider part of an XML document (containing information about items in a shop) and its representation as a tree. We represent each node in the tree by showing its label ($Item$, for example) and its position (position ϵ for $Item$). We note that the element Description has two attributes and no sub-element.

```
<Item>
  <Description itName= "Les 400 Coups"
               itType= "DVD"/>
  <Price> 25.00 </Price>
</Item>
```



□

Our method for validating XML documents can be outlined as follows:

- XML documents are seen as unranked labeled trees (*i.e.*, trees whose nodes have a finite but arbitrary number of children) having two kinds of nodes.
- Some attribute and element constraints are imposed by a predefined schema. We use DTD (Document Type Definition) as a schema language, and we translate each DTD d into an *extended* tree automaton $\mathcal{A}$.
- A run of $\mathcal{A}$ on an XML tree corresponds to verifying whether the attribute and element constraints hold. The automaton $\mathcal{A}$ makes it possible to check validity by a single bottom-up pass on the document tree. If all the constraints are satisfied, the run is *successful*.

Element constraints are expressed by regular expressions E and, thus, they are verified by testing if the *sequences* of sub-elements correspond to E. At-

tribute constraints imply two kinds of tests, differing on whether the values associated to the attributes should be considered or not. The first kind of test does not take into account the values, only the existence of required and optional attributes is verified. The second kind of test involves attribute values. These tests verify the uniqueness of some identifier values (called ID values) in the whole document, and the existence of ID values corresponding to reference values (called IDREF or IDREFS values).

Here we propose an extended tree automaton capable of (i) verifying if an element constraint, specified by a regular expression E, is respected and (ii) performing the first kind of test concerning attribute constraints, *i.e.*, comparing the set of existing attributes to the specified one. Moreover, our validation method uses *external* procedures, while running the tree automata, to perform the tests on the attribute values.

Our tree automaton differs from regular tree automata (Brüggeman-Klein and Wood, 1998b) in the form of the transition rules. Ours are of the form $a, S, E \rightarrow q$ where a is a label, S is a set of state sets, E is a regular expression of states and q is a state. A *run* is defined as usual: given a labeled tree, the automaton starts its computation at the leaves and then simultaneously works up the paths of the tree. To *move* to a node p upward on a tree, *i.e.*, to *assume* a state q at p according to a given transition rule $a, S, E \rightarrow q$, our tree automaton should: (i) verify if the label associated to p is a, (ii) check if the states assumed at the children of p that are members of the set group (the attributes) correspond to those specified by S and (iii) verify if the concatenation of the states assumed at the children of p that are members of the sequence group (the sub-elements) satisfies the regular expression E. The tree automaton accepts a tree if there is a *successful* run, *i.e.*, if it is possible to perform move operations from the leaves to the root and to assume a final state at the root. The following example illustrates this computation.

Example 1.2 We suppose the tree t of Example 1.1 and a tree automaton $\mathcal{A}$ having the following transition rules:
(1)Item,$\{\emptyset, \emptyset\}$,$q_{Description}\ q_{Price} \rightarrow q_{Item}$
(2) Description,$\{\{q_{itName}, q_{itType}\}, \emptyset\}, \emptyset \rightarrow q_{Description}$
(3)itName,$\{\emptyset, \emptyset\}$,$q_{data} \rightarrow q_{itName}$
(4)itType,$\{\emptyset, \emptyset\}$,$q_{data} \rightarrow q_{itType}$
(5)Price,$\{\emptyset, \emptyset\}$,$q_{data} \rightarrow q_{Price}$
(6)data, $\{\emptyset, \emptyset\}$,$\emptyset \rightarrow q_{data}$
The run of $\mathcal{A}$ on t is obtained by applying, initially, rule 6 to all leaves of t to obtain the state q_{data} in positions 000, 010 and 10. The second step consists of applying rules 3, 4 and 5 to obtain the states q_{itName}, q_{itType} and q_{Price} in positions 00, 01 and 1 respectively. The automaton continues by verifying if rule 2 can be applied in order to associate the state $q_{Description}$ to the node at position 0. This is possible due to the following reasons: (i) the node is labeled $Description$, (ii) its two children are attributes corresponding to the specification of rule 2 (see Section 3 and 5 for details) and (iii) it has no sub-element, matching the empty regular expression of rule 2. Following a similar reasoning, the automaton associates q_{Item} to the position ϵ. □

The main contributions of our method are:

- To introduce an extended bottom-up finite tree automaton with the following features:
  - It is capable of treating *finite* trees with both element and attribute nodes.
  - It is *deterministic*, since it is constructed from a non ambiguous DTD.
  - It is *regular*, *i.e.*, it can be reduced to a regular tree automaton (Brüggeman-Klein and Wood, 1998b).
- To combine the running of the automaton on a tree with calls to an *external procedure* that stores ID/IDREF(S) values when their corresponding attribute nodes are found. To complete validation, after a successful run, the ID/IDREF(S) properties are verified over these stored values.
- To be a linear time validation method in the size of the XML document: each node is visited once.

The rest of this paper is organized as follows. In Section 2 we present a short introduction to XML and DTD. In Section 3 we define the labeled trees used to represent the XML documents and we introduce our notion of tree automaton. In Section 4 we show how to translate a DTD into a deterministic tree automaton. In Section 5 we present the algorithms for validating an XML document and we show that our method is correct and complete. Finally, in Section 6, we consider some related work, and we discuss our perspectives for further research. Proofs are omitted due to lack of space, they can be found in (Bouchou et al., 2003).

2 XML DOCUMENTS AND DTDs

In this section we use an example to recall the main features of XML documents and we define DTDs.

Example 2.1 We consider part of an XML document containing information about invoices in a shop. The basic component of XML is the *element*, delimited by matching tags such as <Invoice> and </Invoice>. An XML document is a set of elements with ordered sub-elements.

```
<?xml version='1.0'?>
  <Shop>
    <Customer idCust="C012" idInvoices="00123 00124">
      <Name> Manoel Dubois </Name>
      . . .
    </Customer>
    <Invoice invoiceNb="00123">
      <Date> 15/09/2002 </Date>
      <BillTo custNb="C012"/>
```

```
    <Item>
      <Description itName= "Les 400 Coups"
                   itType= "DVD"/>
      <Price> 25.00 </Price>
    </Item>
      . . .
  </Invoice>
</Shop>
```

In this document, the root element is Shop. Some elements, such as <Price>25.00</Price>, contain only text. Elements can have *attributes* defined as (*name, value*) pairs. Attributes cannot be nested and a given attribute may only occur once within an element. For instance, <BillTo custNb="C012"/> indicates that this element is empty and has attribute $custNb$ whose value is $C012$. □

In a data-exchange environment, we are mostly interested in documents that satisfy some specific constraints. A DTD is a schema for the data represented by XML documents. DTDs are classified as extended context-free grammars (Brüggeman-Klein and Wood, 1998b; Neven, 2002a) or as tree grammars (Murata et al., 2001). In this paper, a translation algorithm is used to build a tree automaton from a DTD organized as established by Definition 2.1.

Definition 2.1 - DTD: A DTD (file) is a text file having the form: <! DOCTYPE $firstEle$ [set of element and attribute declarations] > and following the specifications below:

- The element $firstEle$ is referred to as the outermost element. Its declaration should appear in the set of element declarations of the DTD file.
- *An element declaration* has the following form:
  <!ELEMENT $eleName$ $\underbrace{regExp|\#PCDATA|EMPTY}_{\text{content model}}$ >
  where $eleName$ is an element name and the content model describes the structure of the element in terms of sub-elements and data types. The content model is defined by $regExp$, a regular expression over element names, or by one of the reserved words #PCDATA or EMPTY[1]. #PCDATA indicates that the element $eleName$ has no sub-elements, only a text value is associated to it. EMPTY means that the element $eleName$ must have *no* sub-elements or text associated to it.
- *An attribute declaration* has the following form:
  <!ATTLIST $eleName$ $attSet$ >
  where $eleName$ is the name of the element whose attributes are those belonging to the set $attSet$. The set $attSet$ is a set of tuples containing, for each attribute, its name, its kind and its status, denoted by *att-name*, *att-kind* and by *att-status*, respectively. For each attribute, *att-name* is an attribute name, *att-kind* is a value from the set {CDATA, ID, IDREF, IDREFS} and *att-status* is a value from the set {#REQUIRED, #IMPLIED}. CDATA means that the attribute is an arbitrary text. ID means that the attribute uniquely identifies an element in the entire document. IDREF means that the attribute value is the identifier of another element and IDREFS denotes a list of IDREF. #REQUIRED and #IMPLIED are used to indicate whether an attribute is compulsory or optional, respectively. □

[1] We consider neither mixed nor ANY declaration in the content model.

Definition 2.2 - Alphabet of a DTD: Given a DTD d, let Σ_{ele} be the set of the element names appearing in d and Σ_{att} be the set of the attribute names in d. The alphabet of a DTD is the set $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$ where $data$ indicates the content model #PCDATA or attribute values in d. □

It is important to notice that, according to the requirements of W3C, in Definition 2.1 we consider only *unambiguous* DTDs (*i.e.*, DTDs where no lookahead is needed to decide whether a string matches a label (Brüggeman-Klein and Wood, 1998a)). The next example shows a DTD for our shop document.

Example 2.2 The DTD specifies that Shop is the outermost element and is composed by zero or several Customer-element and Invoice-element, in this order. We omit all declarations of the form <!ELEMENT eleName (#PCDATA)>.

```
<!DOCTYPE Shop[
  <!ELEMENT Shop (Customer*, Invoice*)>
  <!ELEMENT Customer (Name, Address)>
  <!ATTLIST Customer idCust ID #REQUIRED
                     idInvoices IDREFS #IMPLIED>
  <!ELEMENT Address (Street, (State|Province),
                     Country?)>
  <!ELEMENT Invoice (Date, BillTo, Item+)>
  <!ATTLIST Invoice invoiceNb ID #REQUIRED>
  <!ELEMENT BillTo EMPTY>
  <!ATTLIST BillTo custNb IDREF #REQUIRED>
  <!ELEMENT Item (Description, Price)>
  <!ELEMENT Description EMPTY>
  <!ATTLIST Description
       itName CDATA #REQUIRED
       itType CDATA #REQUIRED> ] >
```
□

3 TREES AND TREE AUTOMATA

To introduce the notion of tree, we consider a finite alphabet Σ and U denotes the set $\mathbb{N}^*$ of all finite strings of positive integers with the empty string ϵ.

Definition 3.1 - Prefix relation : The *prefix relation* in U, denoted by $\preceq$ is defined by: $u \preceq v$ iff $uw = v$ for some $w \in U$. Consider a finite subset $D \subseteq U$. We say that D is *closed under prefixes* if $u \preceq v$, $v \in D$ implies $u \in D$. □

Definition 3.2 - Σ-valued tree t: A Σ-valued tree t (or just a tree) is a mapping $t : dom(t) \rightarrow \Sigma \cup \{\lambda\}$ where $dom(t) \subseteq U$ is a nonempty set closed under

prefixes which satisfies: $j \geq 0, uj \in dom(t), 0 \leq i \leq j \Rightarrow ui \in dom(t)$ and where λ is a special symbol that indicates the empty tree. An *empty tree* t has $dom(t) = \{\epsilon\}$ and $t(\epsilon) = \lambda$. The set $dom(t)$ is also called the set of *positions* of t. □

We are interested in *unranked trees* (*i.e.*, there is no prior bound on the number of children of a node) and we use them to encode an XML document as follows: The outermost element is the *root* and every element has its sub-elements and attributes as children. Elements and attributes associated with an arbitrary text have a child labeled *data*, but its text value is not represented in the tree itself. A *leaf* node is an element or a node labeled *data*. Figure 1 shows the tree obtained from the XML document of Example 2.1. We notice that for each position $p \in dom(t)$ we write $t(p)$ to indicate the label associated with p.

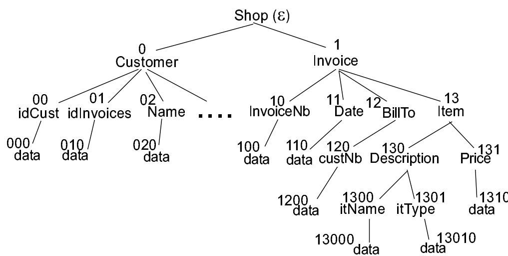


Figure 1: XML tree. Each node is represented by its position and its label.

Now, we propose to translate a DTD into a bottom-up tree automaton and to execute it over an XML tree in order to verify the validity of the corresponding XML document. This kind of automaton starts its computation at the leaves and then simultaneously works up the paths of the tree. To move upward, to a position p on a tree, an automaton has to check whether the children of p respect some attribute and element constraints. This move is represented by the assignment of a state q to position p.

In this paper we *extend* regular tree automata in order to allow them to deal with trees having two different types of nodes: those that should be treated as members of a set (the *set group*) and those that should be treated as part of a sequence (the *sequence group*).

Definition 3.3 - Extended non-deterministic bottom-up finite tree automaton (ENFTA): An ENFTA over Σ is a tuple $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where Q is a set of states, $Q_f \subseteq Q$ is a set of final states and Δ is a set of transition rules of the form $a, S, E \rightarrow q$ where *(i)* $a \in \Sigma$; *(ii)* S is a set of two disjoint sets of states, *i.e.*, $S = \{S_{comp}, S_{op}\}$ (with $S_{comp} \subseteq Q$ and $S_{op} \subseteq Q$); *(iii)* E is a regular expression over Q, and *(iv)* $q \in Q$. □

Now, in order to move upward, *i.e.*, to assume a state q at position p, our tree automaton performs the following tests:
1. If p has children in the set group then the states assumed for them should correspond to those specified by the sets in S, namely, S_{comp} and S_{op}, correspondingly, respectively, to p's children that *must* appear in the tree and those that *may* appear.
2. If p has children in the sequence group then a finite state automaton m_E, defined by E, should recognize the concatenation of p's children states.

The tree automaton accepts a tree if there is a *successful* run, *i.e.*, if it is possible to perform move operations from the leaves to the root and to assume a final state at the root. The following definition formalizes the concept of a run.

Definition 3.4 - Run of $\mathcal{A}$ on a finite tree t: Let t be a tree and $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ an ENFTA. A *run* of $\mathcal{A}$ on t is a tree r such that $dom(r) = dom(t)$ defined as follows: for each position p whose children are at positions[2] $p0, \ldots, p(n-1)$ (with $n \geq 0$), the state q is assumed at the position p, *i.e.*, $r(p) = q$ if all the following conditions hold:

1. $t(p) = a \in \Sigma$
2. There exists a transition $a, S, E \rightarrow q$ in Δ such that E is a regular expression equivalent to the finite state automaton m_E.
3. The children of p (the positions $p0, \ldots, p(n-1)$) can be classified according to the following rules:
   (a) an integer $0 \leq i \leq (n-1)$ is assumed, according to some criterion (such as the type of the children of p in t) and
   (b) the positions $p0, \ldots, p(i-1)$ are members of the set *setG* (possibly empty) and
   (c) the positions $pi, \ldots, p(n-1)$ are members of a set *seqG* (possibly empty) and
   (d) every child of p is a member either of *setG* or of *seqG*, but no position is in both sets[3].
4. The tree r is already defined for positions $p0, \ldots, p(n-1)$ that is, $r(p0) = q_0, \ldots, r(p(n-1)) = q_{n-1}$.
5. Either *seqG* is empty and so is E, or the concatenation $q_i \ldots q_{n-1}$ of the states associated with the positions in *seqG* is recognized by m_E.
6. The sets of S respect the following properties:
   (a) $S_{comp} \subseteq \{q_0, \ldots, q_{i-1}\}$ where $\{q_0, \ldots, q_{i-1}\}$ is the set of all states associated with the i positions in *setG* and

[2]Recall that 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.

[3]Clearly, when a position p has no children (*i.e.*, $n = 0$) then $setG = seqG = \emptyset$. If $n > 0$ and $i = 0$ then $setG = \emptyset$. Similarly, if $n > 0$ and $i = n$, then $seqG = \emptyset$.

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

A run r is *successful* if $r(\epsilon)$ is a final state and a tree t is accepted if a successful run exists on it. □

Definition 3.4 shows how a bottom-up automaton "works" on a tree, by assigning states to the children of a node p *before* associating a state to it. Each assignment $r(p) = q$ is also called a *move*.

Example 3.1 We consider the tree t of Figure 1 and we suppose a tree automaton $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, obtained from the DTD of Example 2.2 as will be detailed in Section 4. By this translation we know that Δ contains the following transition rules: $\delta_1 : data, \{\emptyset, \emptyset\}, \emptyset \rightarrow q_{data}$ and $\delta_2 : Invoice, \{\{q_{invoiceNb}\}, \emptyset\}, q_{Date}\ q_{BillTo}\ q_{Item}+ \rightarrow q_{Invoice}$. Due to δ_1, the state q_{data} can be assumed at all the leaves of r since all of them are labeled $data$.
Now, considering position 1, we suppose that its children, positions 10, 11, 12 and 13, are assigned to states $q_{invoiceNb}$, q_{Date}, q_{BillTo} and q_{Item}, respectively. As the element Invoice has one attribute and three sub-elements, we fix $i = 1$ in Definition 3.4 and thus $setG = \{10\}$ and $seqG = \{11, 12, 13\}$. We consider δ_2. We have $S_{comp} \subseteq \{q_{invoiceNb}\}$ and $(\{q_{invoiceNb}\} \setminus S_{comp}) \subseteq S_{op}$ and the word $q_{Date}q_{BillTo}q_{Item}$ is recognized by the finite state automaton associated with the rule. Then the state $q_{Invoice}$ can be assigned to position 1 in r. □

From the above example, we notice that the integer i is fixed according to the types of nodes in the tree t. In our approach, we use *external* functions to obtain extra information about nodes (such as the type). In this context, given a tree t and a position $p \in dom(t)$ we suppose the following external functions:

- $type(t, p)$ maps p is defined as follows:

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

- $value(t, p)$ returns a subset of $\mathbf{D}$ if $type(t, p) = data$, with $\mathbf{D}$ being an infinite (recursively enumerable) domain, and is $undefined$, otherwise.
- $children(t, p)$ maps p to the set containing all positions pi in $dom(t)$. If $children(t, p) = \emptyset$ we say that p is a *leaf* of t.
- $father(t, p)$ maps p to the position $u \in dom(t)$ such that there exists i in $\mathbb{N}$ for which $ui = p$. Define $father(t, \epsilon) = \epsilon$.

4 TRANSLATING DTDs INTO TREE AUTOMATA

In this section we consider the translation of a DTD into an tree automaton $\mathcal{A}$ and we show that the language generated by the DTD is equivalent to the language recognized by $\mathcal{A}$.

Definition 4.1 - Tree Automata from DTDs: Given a DTD d, define the extended tree automaton $\mathcal{A}_d = (Q, \Sigma, Q_f, \Delta)$ associated to d according to the following steps:

- The alphabet Σ is the alphabet of d.
- $Q = \{q_a \mid a \in \Sigma\}$.
- Q_f is a singleton set containing the state $q_{firstEle}$ that corresponds to the outermost element in d.
- The set of transition rules Δ is built as follows:

1. For each element declaration in d of the form <!ELEMENT *eleName content model*> build a new transition rule $a, S, E \rightarrow q_a$ where $S = \{\emptyset, \emptyset\}$ and where a, E and q_a are defined as follows:
   (a) The symbol a is the element name $eleName$ and thus q_a is the state corresponding to a.
   (b) If the content model is $regExp$ then E is the regular expression obtained by replacing all a in $regExp$ by q_a.
   (c) If the content model is #PCDATA then E is q_{data}
   (d) If the content model is EMPTY then E is $\emptyset$.
2. For each attribute declaration in d having the general form <!ATTLIST *eleName attSet*> do:
   - For each tuple[4] $\nu[$*att-name*, *att-kind*, *att-status*$]$ in $attSet$:
   (a) Build a new transition rule $att, \{\emptyset, \emptyset\}, q_{data} \rightarrow q_{att}$ such that $att = \nu($*att-name*$)$.
   (b) Update the set S in the transition rule $eleName, S, E \rightarrow q_{eleName}$ as follows:
   if $\nu($*att-status*$)$= #REQUIRED
   then $S_{comp} = S_{comp} \cup \{q_{\nu(att-name)}\}$
   else $S_{op} = S_{op} \cup \{q_{\nu(att-name)}\}$
3. Include the rule $data, \{\emptyset, \emptyset\}, \emptyset \rightarrow q_{data}$ in Δ. □

Now, we define a table to store the attribute properties found in a DTD d.

Definition 4.2 - Attribute table T: Given a DTD d, the *attribute table* T is built while constructing $\mathcal{A}_d$ in the following way: for each attribute declaration in d having the form <!ATTLIST *eleName attSet*> do: for each tuple $\nu[$*att-name*, *att-kind*, *att-status*$]$ in $attSet$ add the tuple $\nu_1[ele,$ *att-name*, *att-kind*$]$ in T such that $\nu_1(ele) = eleName$, $\nu_1($*att-name*$) = \nu($*att-name*$)$ and $\nu_1($*att-kind*$) = \nu($*att-kind*$)$ □

From Definition 4.1, we note that our translation method takes time $O(\mid \Sigma_{ele} \mid + \mid \Sigma_{att} \mid)$. Besides, the Glushkov method (Lee et al., 2000) is used to build, in time $O(m^2)$, a finite state automaton from a regular expression E having m occurrences of symbols.

In the following we state two interesting properties of our automaton $\mathcal{A}_d$.

[4] Here, we use the relational database notation: If ν is a tuple over V then $\nu(A)$ denotes the value of ν on attribute $A \in V$. Moreover, for $U \subseteq V$, $\nu[U]$ denotes the tuple u over U such that $u(A) = \nu(A)$ for each $A \in U$.

Proposition 4.1 *Given a DTD d, the tree automaton $\mathcal{A}_d$ is a bottom-up finite regular and deterministic tree automaton.* □

Theorem 4.1 *Given a DTD d and the ENFTA $\mathcal{A}_d$, the tree language generated by d is equivalent to the tree language recognized by $\mathcal{A}_d$, i.e., $L(d) = L(\mathcal{A}_d)$.* □

5 VALIDATING XML VIEWS

Given the XML tree t representing an XML view (or document), the tree automaton $\mathcal{A}_d$ and the auxiliary table T constructed from a given DTD d, the view validation process consists in the call **valid**$(\mathcal{A}_d, T, t)$ (which returns $true$ or $false$). Its implementation uses, basically, three algorithms: (i) **move** the state q to be assigned to given position p, (ii) **run** implements Definition 3.4 using **move**, and (iii) **valid** uses **run** to verify if an automaton $\mathcal{A}$ recognizes a tree t and tests if the ID/IDREFS attributes conform to DTD specifications. Below we only present the algorithms **move** and **run** and we refer to (Bouchou et al., 2003) for futher details on all these algorithms.

```
int function move (p, A, T, t) {
//* Computes the state q ∈ Q to be assigned to position p
//* or q_error if no assignment is possible according to A
Let children(t, p) = {p0, ..., p(n − 1)}, n ≥ 0
set_chSt = ∅;    seq_chSt = ε
//* Splits states of {p0, ..., p(n − 1)} into a set of attribute
//* states and a sequence of element (or data) states
for i = 0 to n − 1 do {
    if (type(t, pi) = attribute)   set_chSt = set_chSt ∪ {r(pi)}
    else seq_chSt = seq_chSt.r(pi)
}
if ( t(p) == a and the transition rule a, S, E → q ∈ Δ and
   S = {S_comp, S_op}  and the automaton m_E defined
   by E recognizes seq_chSt and (S_comp ⊆ set_chSt) and
   (set_chSt \ S_comp ⊆ S_op) )      return(q)
   else return(q_error)             }
```

```
procedure run (A, T, t, r) {
//* Builds a run r of A on t, i.e., a tree r such that
//* dom(r) = dom(t) and whose labels are elements of Q
C = {u ∈ dom(t) | ¬∃ i   such that ui ∈ dom(t)}
//* C is the current set of nodes to be treated
pos = noOneGreater(C)
while C ≠ ∅ do {
       if (type(t, p) == attribute)
         idIdref(T, t, p) //* stores ID / IDREF(S) values
       r(pos) =move(pos, A, T, t)
       C = C \ {pos}
       if pos ≠ ε {
         C = C ∪ {father(t, pos)}
         pos = noOneGreater(C)
} }    }
```

We note the use of two procedures called by **run**: (i) Procedure $idIdref$ is used to treat attributes. It feeds two tables of attribute values, namely V_{ID} and V_{IDREFS} (treated as global variables). If the node at position p is an attribute then it has just one child, at position $p0$. Procedure $idIdref$ verifies the kind of node p and adds the content of $value(t, p0)$ to V_{ID} (respectively to V_{IDREFS}) if it is of kind ID (respectively of kind IDREF or IDREFS). The function **valid** will check if V_{ID} has no duplicate value and if each value in V_{IDREFS} exists in V_{ID}. (ii) Function $noOneGreater$ returns a maximal position in C with respect to the prefix relation (Definition 3.1).

Theorem 5.1 below shows that only valid XML documents with respect to d have their trees t recognized by our validation method (correction) and, conversely, only trees recognized by our validation method correspond to valid XML documents with respect to d (completion).

Theorem 5.1 *Given a DTD d, the associated extended tree automaton $\mathcal{A}_d$ and the associated attribute table T, let X be an XML document and t its associated tree. The document X is valid w.r.t. d if and only if the call* **valid**$(\mathcal{A}_d, T, t)$ returns *true*. □

Considering the complexity of our validation method and supposing that c_i (resp. c_r) is the number of ID (resp. IDREF(S)) values in the XML document, inclusion tests that verify ID/IDREFS properties are in $O(c_i(c_i + c_r))$. Finally, for every XML document X, our validation method is in $O(n_e{+}n_a{+}c_i(c_i + c_r))$, where n_e is the number of elements and n_a the number of attributes in X.

6 RELATED AND FUTURE WORK

The advent of XML motivated some recent developments in the area of databases and automata (see, (Vianu, 2001) for a survey on database theory and XML). The interest in unranked tree automata is relatively new (Alon et al., 2001; Brüggeman-Klein and Wood, 1998b; Murata et al., 2001; Neven, 2002a; Papakonstantinou and Vianu, 2000) although their ranked counterparts have been thoroughly investigated (Comon et al., 1997; Thomas, 1997). In (Neven, 2002a) we find a survey dealing not only with unranked tree automata, but also with tree-walking automata and automata over infinite alphabets. A regular tree automaton is used to model XML data in (Chidlovskii, 2000). The author proposes an *XML query algebra* based on this model.

Although a lot of algorithms for *validating* XML documents with respect to some schema language exist, formal work on this area usually does not consider all the details of the problem. For instance, (Murata et al., 2001) defines four sub-classes of regular tree grammars and their corresponding schema

languages (DTDs, for instance, are classified as local tree grammars). The authors outline some algorithms for validation of documents against schemas, but instead of considering the details of how to deal with attributes and elements, they are interested in comparing schema languages for XML. In (Segoufin and Vianu, 2002) the authors propose a method for validating streaming XML documents using a finite state automaton. In (Milo et al., 2000), the authors consider the type checking problem expressed by k-pebble transducers, showing that the problem is decidable. In (Alon et al., 2001), they consider trees with labels from an infinite alphabet, showing that in this case type checking becomes undecidable.

Our work differs from those mentioned above mainly on the treatment of attributes, which has been usually neglected. Although the choice between elements and attributes to represent information is perennial and usually arbitrary, element constraints and attribute constraints are completely different. As already seen, contrary to elements, attributes are unordered, unique and atomic. Moreover, to completely validate an XML document with respect to a DTD, one should perform some tests on attribute values, since a DTD indicates the kind of attributes.

The main contribution of this paper is the introduction of a new formalism to deal with elements *and* attributes. Although attributes and elements have a similar representation in a document tree (both are nodes) they treated differently by our tree automaton. In this sense, our work is similar to (Hosoya and Murata, 2002). However, these authors do not deal with DTDs and present an approach where finite state automata change according to the attributes found in the document. In (Neven, 2002a) tree automata are extended to treat attributes and values, *i.e.*, using tree automata over infinite alphabets. In our work, we are interested in a validation method and, thus, we do not need to represent values as nodes in the tree. Moreover, our *external* functions are used to perform the ID/IDREF(S) tests without requiring trees on infinite alphabets.

Our validation method is based on the *tree model* (Murata et al., 2001) which considers an XML tree created in memory, accessed via some API such as DOM. The use of the *event model* does not imply changes in our formalism. We are currently implementing our validation method into these two different ways. The formalism introduced in this paper plays a crucial role in our present and future work, concerning updates on XML views. The use of an ENFTA to validate XML views allows the implementation of efficient update operations.

REFERENCES

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

Bouchou, B., Duarte, D., Halfeld Ferrari Alves, M., and Laurent, D. (2003). Tree automata for validating XML views under element and attribute constraints. In *Technical Report (To appear), LI, Université de Tours*.

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

Brüggeman-Klein, A. and Wood, D. (1998b). Regular tree languages over non-ranked alphabets. Technical report, unpublished manuscript.

Chidlovskii, B. (2000). Using regular tree automata as XML schemas. In *Proc. IEEE Advances in Digital Libraries Conference*.

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

Hosoya, H. and Murata, M. (2002). Validation and boolean operations of attribute-element constraints. In *Programming Languages Technologies for XML, PLAN-X*.

Lee, D., Mani, M., and Murata, M. (2000). Reasoning about XML schemas languages using formal language theory. In *Technical Report - IBM Almaden Research Center. Available in www.cobase.cs.ucla.edu/tech-docs/˜dongwon/ibm-tr-2000.ps*.

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

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

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

Neven, F. (2002b). Automata theory for XML researchers. *Sigmod Record*, 31(3).

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

Segoufin, L. and Vianu, V. (2002). Validating streaming XML documents. In *ACM Symposium on Principles of Database System*.

Thomas, W. (1997). Languages, automata and logic. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages*, volume 3. Springer Verlag.

Vianu, V. (2001). A web odyssey: from Codd to XML. In *ACM Symposium on Principles of Database System*.