Attribute Grammar for XML Integrity Constraint Validation *

Béatrice Bouchou[1], Mirian Halfeld Ferrari[2], and Maria Adriana Vidigal Lima[3]

[1] Université François Rabelais Tours, Laboratoire d'Informatique, France
[2] LIFO - Université d'Orléans, Orléans, France
[3] Faculdade de Computação, Universidade Federal de Uberlândia, MG, Brazil

Abstract. The main contribution of this paper is a generic *grammarware* for validating XML integrity constraints. Indeed, we use an attribute *grammar* to describe XML documents and constraints. We thus explain the main parts of this novel algorithm and we report on experiments showing that our method allows for an effective and efficient validation of XML functional dependencies (XFD).

1 Introduction

This paper deals with integrity constraint validation on XML documents. Our validation method can be seen as a *grammarware*, since it is based on a grammar describing an XML document to which we associate attributes and semantic rules. Our grammar is augmented by semantic rules that define, for each integrity constraint, the verification process. In this way we show that XML integrity constraints can be compiled to an attribute grammar [1, 15]. To instantiate an integrity constraint we introduce a set of finite state automata (FSA). Indeed, XML integrity constraints are defined by using path expressions which can be seen as simplified regular expressions over XML labels. These finite state automata help us to determine the role of each node in a constraint satisfaction.

To explain the main parts of our method, we focus on the validation of functional dependencies (XFD). The approach presented here implements the general proposal introduced in [7], where we present a homogeneous formalism to express different kinds of integrity constraints, including XFDs, and introduce the basis of our general validation method.

Our method validates an XML tree in one tree traversal. In the document reading order, we first go top-down until reaching some leaves and then, bottom-up, as closing tags are reached. During the top-down visit, the validation process uses attributes to specify the role of each node with respect to a given integrity constraint. In the bottom-up visit the values concerned by the constraints are pulled up via some other grammar attributes. Its running time is linear in the size of the XML document, in the number of paths composing the constraint and in the number of obtained tuples containing the constraint values.

* Partially supported by: Codex ANR-08-DEFIS-04

Related work: Although several approaches for XML functional dependencies have been proposed in the literature (we refer to [3, 11, 18, 19, 21] as examples of ongoing works on the subject over the decade), the implementation of constraint validators has received less attention. Proposals in [4, 5, 8] just address specific constraints such as primary or foreign keys. Our approach performance is comparable to implementations in [17, 16], but contrary to them, it intends to be a *generic model for XML constraints validation*, provided constraints are defined by paths. The ideas guiding this work are the ones outlined in [7, 10]. We describe an incremental validation method for keys in [6]. In [9] the notion of incremental validation is considered via the static verification of functional dependencies with respect to updates. However, in that work, XFDs are defined as tree queries which augments considerably the complexity of an implementation.

Indeed, one important difference among all the XFD proposals concerns the expressive power of the language used to specify the components of the dependency. Even if path languages in [2, 18–20] are slightly different, they all express unary queries. In [2, 18, 20] only simple paths are allowed while in [19] simple, composed, ascendant or descendent paths are permitted. Differently, the approach in [11] uses an n-ary path language. Similarly to us, in all these approaches, once a path is defined, one needs to determine instantiations of this path. The way it is done depends on the path language used and on the assumption of an underlying schema (that allows to define tree tuples [2] and generalized tree tuples [20]). Notice also that paths defining constraints induce a notion of tree pattern that the document must conform to (called *Paths(D)* in [2], *Schema Graph* in [12] or *Legal paths* in [18]), which can be seen as a *schema* (though less restrictive than a DTD). Thus, following [15] and [13], an extended attribute grammar can always be used to represent the selecting part of any of these different notions of XML functional dependencies. Then, functions for checking features of the selected components must be defined. In this way, attribute grammars are generic enough to accommodate many kind of integrity constraints. For instance we plan to consider the possibility of using it to compute XFD defined by tree queries such as in [9].

Paper organisation: Section 2 introduces our XFD definitions. In Section 3 we explain how finite state automata can help us to develop our validation algorithm. Section 4 presents the XFD verification process based on attribute grammar. In Section 5 we consider complexity and experimental results.

2 Functional Dependencies in XML

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 document is represented by a tuple $\mathcal{T} = (t, type, value)$. The tree t is the function $t : dom(t) \to \Sigma$ where Σ is a set of tags and where $dom(t)$ is the set of positions $u.j$, such that $(\forall j \geq 0)$ $(u.j \in dom(t)) \Rightarrow (\forall i \;\; 0 \leq i < j)$ $(u.i \in dom(t))$; where i and $j \in \mathbb{N}$ and $u \in U$ (U is a set of sequences of symbols in $\mathbb{N}$, and the symbol ε which is the empty sequence). Given a tree position p, function $type(t, p)$ returns a value in $\{data$, $element$, $attribute\}$. Similarly, $value(t, p) = \begin{cases} p & \text{if } type(t, p) = element \\ val \in \mathbf{V} & \text{otherwise} \end{cases}$ where $\mathbf{V}$ is an infinite recursively enumerable domain. We also recall that, in an XML tree, attributes are unordered while elements are ordered. As many other authors, we distinguish two kinds of equality in an XML tree, namely, *value and node equality.* Two nodes are *value equal* when they are roots of isomorphic subtrees. Two nodes are *node equal* when they are the same position. To combine both equality notions we use the symbol E, that can be represented by V for value equality, or N for node equality.

Figure 1 illustrates an XML document that models the projects of a company. Notice that each node has a position and a label. For instance, $t(\epsilon) = bd$ and $t(1.0) = pname$. Nodes in positions 1.2.1.1 and 0.1.2.1 are value equal, but nodes 0.1.2 and 1.2.1 are not value equal (element quantity in their subtrees is associated to different data values).

A path for an XML tree t is defined by a sequence of tags or labels. The path languages PL_s (defined by $\rho ::= \ l \mid \rho/\rho \mid \_$) and PL (defined by $\upsilon ::= [\ ] \mid \rho \mid \upsilon//\rho$) are used to define integrity constraints over XML trees.

In PL and PL_s, $[\ ]$ represents the empty path, l is a tag in Σ, the symbol "/" is the concatenation operation, "//" represents a finite sequence (possibly empty) of tags, and "_" is any tag in Σ. The language PL_s describes a path in t, while PL is a generalization of PL_s including "//". Then, one path in PL describes a set of PL_s paths. In this work we adopt the language PL that is a common fragment of regular expressions and XPath. A path P is **valid** if it conforms to the syntax of PL_s or PL and for all tag $l \in P$, if $l = data$ or $l \in \Sigma_{att}$, then l is the last symbol in P. We consider that a path P defines a finite-state automaton A_P having XML labels as its input alphabet.

Definition 1. Instance of a path P over t : Let P be a path in PL, A_P the finite-state automaton defined according to P, and $L(A_P)$ the language accepted by A_P. Let $I = v_1/\ldots/v_n$ be a sequence of positions such that each v_i is a direct descendant of v_{i-1} in t. Then I is an instance of P over t if and only if the sequence $t(v_1)/\ldots/t(v_n) \in L(A_P)$. □

As an example, consider the path $bd/project/supplier$. From Figure 1, we can see that $\epsilon/0/0.1$ or $\epsilon/1/1.1$ are instances of this path. Integrity constraints in XML are expressed by sets of paths. A set of paths can form a *pattern* M if all paths have a common prefix and for all path $P \in M$, if P_1 is a subpath of P, then $P_1 \in M$. Thus, a pattern is a tree pattern.

Definition 2. Pattern and Pattern Instance : A *pattern* is a finite set of *prefix-closed* paths in a tree t. Let $Long_M$ be the set of paths in M that are not prefix of other paths in M. Let $Instances(P, t)$ be the set of all instances of a path P in t. Let $PInstanceSet^i$ be the set of path instances that verifies:

1. For all paths $P \in Long_M$ there is one and only one instance $inst \in Instances(P, t)$ in the set $PInstanceSet^i$.
2. For all $inst \in PInstanceSet^i$ there is a path $P \in Long_M$.

3. For all instances $inst$ et $inst'$ in $PInstanceSet^i$, if $inst \in Instances(P, t)$ and $inst' \in Instances(P', t)$, then the longest common prefix of $inst$ and $inst'$ is an instance of path Q in t, where Q is the path with the longest common prefix for P and P'.

An instance of a pattern M is a tuple $I = (t^i, type^i, value^i)$ where $type^i(t^i, p) = type(t, p)$, $value^i(t^i, p) = value(t, p)$ and t^i is a function $\Delta \to \Sigma$ in which:

- $\Delta = \bigcup_{inst\ \in\ PInstanceSet^i} \{p \mid p \text{ is a position in } inst\}$
- $t^i(p) = t(p)$, $\forall p \in \Delta$ □

A functional dependency in XML (XFD) is denoted $X \to Y$ (where X and Y are sets of paths) and it imposes that for each pair of tuples[4] t_1 and t_2 if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. We assume that an XFD has a single path on the right-hand side and possibly more than one path on the left-hand side - generalizing the proposals in [3, 18, 14, 19]. The dependency can be imposed in a specific part of the document, and, for this reason, we specify a *context path.*

Definition 3. XML Functional Dependency : Given an XML tree t, an XML functional dependency (XFD) is an expression of the form

$$\gamma = (C, (\{P_1\ [E_1], \ldots, P_k\ [E_k]\}\ \to Q\ [E]))$$

where C is a path that starts from the root of t (*context path*) ending at the *context node*; $\{P_1, \ldots, P_k\}$ is a non-empty set of paths in t and Q is a single path in t, both P_i and Q start at the context node. The set $\{P_1, \ldots, P_k\}$ is the left-hand side (LHS) or determinant of an XFD, and Q is the right-hand side (RHS) or the dependent path. The symbols $E_1, \ldots, E_k, E$ represent the equality type associated to each dependency path. When symbols E or $E_1, \ldots, E_k$ are omitted, value equality is the default choice. □

Definition 4. XFD Satisfaction : Let $\mathcal{T}$ be an XML document, $\gamma = (C, (\{P_1\ [E_1], \ldots, P_k\ [E_k]\}\ \to Q\ [E]))$ an XFD and let M be the pattern $\{C/P_1, \ldots, C/P_k, C/Q\}$. We say that $\mathcal{T}$ satisfies γ (noted by $\mathcal{T} \models \gamma$) if and only if for all I_M^1, I_M^2 that are instances of M in $\mathcal{T}$ and coincide at least on their prefix C, we have: $\tau^1[C/P_1, \ldots, C/P_k] =_{E_i, i\in[1\ldots k]} \tau^2[C/P_1, \ldots, C/P_k] \Rightarrow \tau^1[C/Q] =_E \tau^2[C/Q]$ where τ^1 (resp. τ^2) is the tuple obtained from I_M^1 (resp. I_M^2), cf. preceding footnote. □

Notice that our XFD definition allows the combination of two kinds of equality (as in [19]). We consider some XFDs, verified by the document in Figure 1:

XFD_1: $(db, (\ \{/project/pname\} \to /project\ [N]\ ))$
Project names are unique and identify a project. The context is *db*, so in this case the dependency must be verified in the whole document.

XFD_2: $(db, (\ \{/project/pname\} \to /project))$
Subtrees of projects which have the same name are identical.

XFD_3 : $(db/project, (\{/supplier/@sname, /supplier/component/@cname\}$
$\to /supplier/component/quantity\ ))$

[4] Tuples formed by the values or nodes found at the end of the path instances of X and Y in a document T.

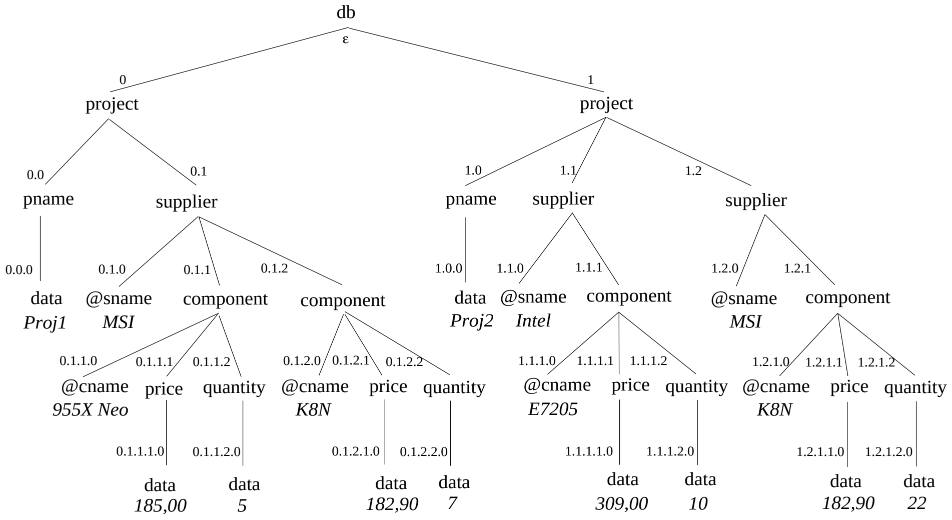

Fig. 1. Tree representing an XML document containing projects information.

3 Finite State Automata for XFD

To model the paths of an XFD, we use finite-state automata (FSA) or transducers (FST). The use of finite state machines allows (i) to clearly distinguish each path (e.g. the context path) and so to define the computation of needed attributes, and (ii) to easily deal with the symbol // and thus to deal with different instantiations for a unique path (*e.g.*, instances $a.b$ and $a.x.b$ for path $a//b$).

The input alphabet of our finite machines is the set of XML tags. The output alphabet of our transducers is composed by our equality symbols. As usual, we denote a FSA by 5-tuple $A = (\Theta,\ V,\ \Delta,\ e,\ F)$ where Θ is a finite set of states; V is the alphabet; $e \in \Theta$ is the initial state; $F \subseteq \Theta$ is the set of final states; and Δ: $\Theta \times V \to \Theta$ is the transition function. A FST is a 6-tuple $A = (\Theta,\ V,\ \Gamma,\ \Delta,\ e,\ F,\ \lambda)$ such that: (i) $(\Theta,\ V,\ \Delta,\ e,\ F)$ is a FSA; (ii) Γ is an output alphabet and (iii) λ is a function from F to Γ indicating the output associated to each final state.

From Definition 3 we know that in an XFD, path expressions C, P_i and Q $(i \in [1, k])$ specify the constraint context, the determinant paths (LHS) and the dependent path (RHS), respectively. These paths define path instances on an XML tree t. To verify whether a path instance corresponds to one of these paths we use the following automata and transducers:

- The *context automaton* $M = (\Theta, \Sigma, \Delta, e, F)$ expresses path C. The alphabet Σ is composed of the XML document tags.
- The *determinant transducer* $T' = (\Theta', \Sigma, \Gamma', \Delta', e', F', \lambda')$ expresses paths P_i $(i \in [1, k])$. The set of output symbols is $\Gamma' = \{V, N\} \times \mathbb{N}^*$ such that V (value equality) and N (node equality) are the equality types to be associated to

each path. Each path is numbered because there may be more than one path in the LHS. Thus, the output function λ' associates a pair $(equality, rank)$ to each final state $q \in F'$;

- Path Q is expressed by the *dependent transducer* $T'' = (\Theta'', \Sigma, \Gamma'', \Delta'', e'', F'', \lambda'')$. The set of output symbols is $\Gamma'' = \{V, N\}$ and the output function λ'' associates a symbol V or N to each final state $q \in F''$.

Figure 2 illustrates FSA and FST for constraints XFD_1 and XFD_2. We remark that the context automaton and the determinant transducers are equal for XFD_1 and XFD_2. For XFD_1, the dependent transducer expresses the application of node equality, while, for XFD_2, the application of value equality (for all values obtained from nodes rooted *project*). The automaton and the corresponding transducers for XFD_3 are illustrated in Figure 3. The determinant transducer (T_3') gathers the attribute values of @*sname* and @*cname* that, together, determine the quantity of a component. This dependency employs value equality for @*sname*, @*cname* and *quantity* with respect to context *project*, defined by M_3.

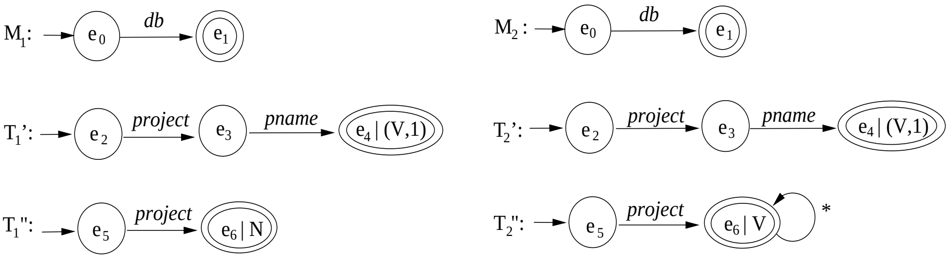

Fig. 2. Automata and transducers corresponding to XFD_1 and XFD_2.

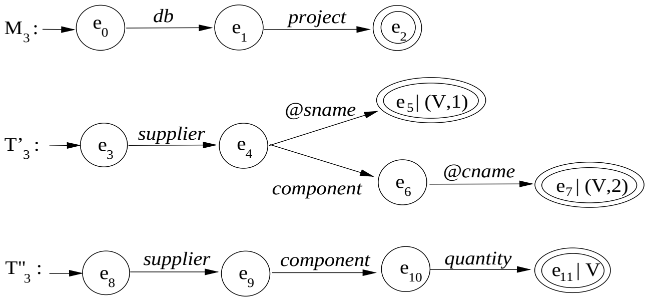

Fig. 3. Automaton and transducers for XFD_3.

4 XFD Validation: Attribute Grammar Approach

The integrity constraint validation process for an XML document can be accomplished with the use of an attribute grammar. Attribute grammars are extensions of context-free grammars that allow to specify not only the syntax, but also the semantics of a language. This is done by attaching a set of semantic rules to each production of a context-free grammar. In a semantic rule, two types of attributes can be found: synthesized and inherited. Synthesized attributes carry information from the leaves of a tree to its root, while inherited ones transport information inversely, from root to leaves.

Definition 5. Attribute Grammar [1]: An attribute grammar is a triple $GA = (G, A, F)$ where: $G = (V_N, V_T, P, B)$ is a context-free grammar; A is the set of attributes and F is a set of semantic rules attached to the productions. For $X \in V_N \cup V_T$, we have $A(X) = S(X) + I(X)$, *i.e.*, $A(X)$ is the disjoint union of $S(X)$, the set of synthesized attributes of X and $I(X)$, the set of inherited attributes of X. If a is an attribute of $A(X)$, we denote it $X.a$. For a production $p : X_0 \rightarrow X_1 \dots X_n$, the set of attributes of p is denoted by $W(p) = \{X_i.a \mid a \in A(X_i), i \in [0 \dots n]\}$. For each production $p : X_0 \rightarrow X_1 \dots X_n$, the set F_p contains the semantic rules that handle the set of attributes of p. □

According to Definition 5, a set $A(X)$ of attributes is associated to each grammar symbol X to describe its semantic features. This gives rise to the following definition for the semantic rules:

Definition 6. Semantic rules attached to production rules: In an attribute grammar, each production $p : X_0 \rightarrow X_1 \dots X_n$ where $X_0 \in V_N$ and $X_i \in (V_N \cup V_T)^*$, $i \in [1 \dots n]$ is associated to a set of semantic rules of the form $b := f(c_1, c_2, ..., c_k)$, where f is a function and: (i) b is a synthesized attribute of X_0 and $c_1, c_2, ..., c_k$ are attributes of non-terminal symbols X_i, or (ii) b is an inherited attribute of a symbol X_i and $c_1, c_2, ..., c_k$ are attributes of X_0 and/or non-terminal symbols X_j, $j \in [1, \dots, i]$. □

Definition 6 establishes that the semantic analysis of a sentence using an attribute grammar is accomplished by a set of actions that is associated to each production rule. In each action definition, the values of attribute occurrences are defined in terms of other attribute values.

In the context of XFD validation, it would be possible to consider the XML type (or schema) as the grammar to be enriched with semantic rules. However, because in our approach integrity constraints are treated independently from schemas, we use a general grammar capable of describing any XML tree. Thus, we consider a context-free grammar G with the following three generic production rules where $\alpha_1 \dots \alpha_m$ denote children nodes (being either XML elements or attributes) of an element A, or the $ROOT$ element:

- Rule for the root element: $ROOT \rightarrow \alpha_1 \dots \alpha_m$, $m \in \mathbb{N}$.
- Rule for an internal element node: $A \rightarrow \alpha_1 \dots \alpha_m$, $m \in \mathbb{N}^*$.

– Rule for an element containing data and for an attribute: $A \rightarrow \mathit{data}$.

Grammar G is extended with semantic rules composed by attributes and actions concerning integrity constraints. Reading an XML document means visiting the XML tree top-down, opening tags, and then bottom-up, closing them. During a top-down visit (to reach the leaves), the validation process specifies (with the aid of FSAs) the role of each node with respect to a given XFD. This role is stored in an inherited attribute. Once the leaves are reached, we start a bottom-up visit in order to pull up the values concerned by the integrity constraints. These values are stored into different synthesized attributes. In the rest of this section, Tables 1-4 introduce our attributes and semantic rules.

Inherited attribute. Firstly, we consider the inherited attribute *conf* which represents for each node in t its role concerning the given XFD. Its value is a set of FSA configurations. All nodes in t, except nodes of type *data* are bound to a *conf* attribute, but for some nodes the value of *conf* is the empty set, which means that this node is not on any path of the XFD.

Tables 1 and 2 show the semantic rules that specify the operations to be executed on *conf*, considering root or internal nodes (except leaves). The first instruction associated to production $ROOT \rightarrow \alpha_1 \ldots \alpha_m$ in Table 1 sets the attribute *conf* of the root node to be $M.q_1$, provided that the root node label is the first transition label in M. Then the value of *ROOT.conf* is transmitted, using the descending direction, to $\alpha_i.conf$. If *ROOT.conf* contains a configuration with a final state for M, then it is necessary to start considering the transducers for calculating configurations for $\alpha_i.conf$.

Table 1. Semantic Rule for Root Production: Attribute *conf*.

Production	**Attributes**
$ROOT \rightarrow \alpha_1 \ldots\ \alpha_m$	$ROOT.conf := \{\ M.q_1\ \mid \delta_M\ (q_0, ROOT) = q_1\}$ for each α_i $(1 \leq i \leq m)$ do $\alpha_i.conf := \{\ M.q'\ \mid \delta_M\ (q_1, \alpha_i) = q'\ \}$ if $(q_1 \in F_M)$ then $\alpha_i.conf := \alpha_i.conf \cup \{\ T'.q'_1\ \mid \delta_{T'}(q'_0, \alpha_i) = q'_1\ \}$ $\cup \{\ T''.q''_1\ \mid \delta_{T''}(q''_0, \alpha_i) = q''_1\ \}$

Similarly, Table 2 specifies how values are assigned to attributes *conf* of internal node children (using rule $A \rightarrow \alpha_1 \ldots\ \alpha_m$). For each α_i, we consider each configuration $\overline{M}.q$ in the parent's *conf* ($\overline{M}$ standing for either M, T' or T”): the transition $\delta_{\overline{M}}\ (q, \alpha_i)$ gives us a new configuration that is stored in α_i's *conf*. Furthermore, we verify if we must change from context automaton to determinant and dependent transducers, as illustrated in Example 1.

Example 1. - We consider XFD_3 (Section 2) for a company and its projects: this dependency is depicted by automaton M_3 and transducers T'_3 et T''_3 of Figure 3. The inherited attribute *conf* is calculated from the root to the leaves as shown in Figure 4.

Table 2. Semantic Rule for Internal Node Production: Attribute *conf*.

Production	**Attributes**

$A \rightarrow \alpha_1 \ldots\ \alpha_m$	for each α_i $(1 \leq i \leq m)$ do
	    for each $\overline{M}.q \in A.conf$ do
	        $\alpha_i.conf := \{\ \overline{M}.q' \mid \delta_{\overline{M}}\ (q, \alpha_i) = q'\ \}$
	        if $(\overline{M} = M) \wedge (q \in F_M)$
	            then $\alpha_i.conf := \alpha_i.conf \cup \{\ T'.q'_1 \mid \delta_{T'}(q'_0, \alpha_i) = q'_1\ \}$
	                $\cup\ \{\ T''.q'_1 \mid \delta_{T''}(q'_0, \alpha_i) = q'_1\ \}$

In the root node, attribute *conf* has configuration[5] $\{M_3.e_1\}$ obtained for node labelled *db*. This configuration comes from the first transition in M_3. For the node in position 0.1 (with label *supplier*), attribute *conf* contains $T'_3.e_4$ and $T"_3.e_9$, as its parent contains a configuration with M in a final state, which denotes a context node. □

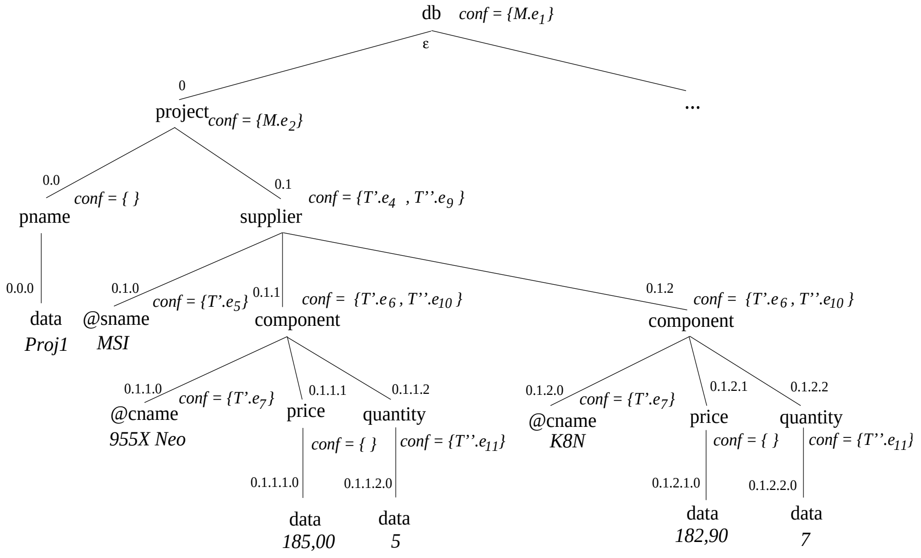

Fig. 4. Inherited attributes *conf* for XFD_3.

Synthesized attributes. We use the ascending direction to compute synthesized attributes: the values that are part of the dependency are collected, treated and carried up to the context node. At the context nodes, these values are compared in order to verify XFD satisfaction.

For each functional dependency, with possibly many paths, there are $k + 3$ synthesized attributes, where k is the number of paths in the determinant part of

[5] To simplify notations M_3,T'_3 and T''_3 do not contain indexes in Figure 4.

the dependency (Definition 3). They are denoted by c, $inters$, dc and ds_j ($1 \leq j \leq k$). Attribute c is used to carry the dependency validity ($true$ or $false$) from the context level to the root. Attribute $inters$ gathers (bottom-up) the values from the nodes that are in determinant and dependent path intersections. Finally, ds_j and dc are attributes for storing the values needed to verify the dependency. These values can be of type $data$ (leaves of t) or node positions, according to the XFD definition of E and E_j.

Attribute $inters$ builds the tuple $<l_1,\ l_2>$ where l_1 is a tuple containing the values of the determinant part and l_2 contains the value of the dependent part. Consider, for instance, the validation of XFD_3 in Figure 5. The context node at position 0 has just one supplier (position 0.1) that provides two distinct components. In this context, there is a path instance for *supplier/@sname* and two instances for the pattern formed by paths *supplier/component/@cname* and *supplier/component/quantity.* Thus, tuple $< l_1, l_2 >$ assigned to $inters$ in position 0.1 is $\{<< MSI, 955XNeo >, 5 >, << MSI, K8N >, 7 >>\}$. To compute the value of this tuple we combine the values carried from the determinant part and assemble this combination with the value of the dependent part. Then, attribute $inters$ carries the XFD values up to the context level.

Table 3. Semantic Rule for Leaf Production: Attributes ds_j and dc.

Production	**Attributes**

$A \rightarrow data$	for each configuration $\overline{M}.q$ in $A.conf$ do
	    if $(\overline{M} = T') \wedge (q \in F_{T'})$
	        $y := \lambda'_{T'}(q)$
	        $j := y.rank$
	        if $(y.equality = V)$
	            then $A.ds_j := < value(t, data) >$
	            else $A.ds_j := < value(t, A) >$
	    if $(\overline{M} = T'') \wedge (q \in F_{T''})$
	        if $(\lambda''_{T''}(q) = V)$
	            then $A.dc := < value(t, data) >$
	            else $A.dc := < pos(t, A) >$

The values of attributes c, $inters$, ds_j and dc are defined according to the role of the parent node *w.r.t.* the XFD. We recall that this role is given by the value of attribute *conf.* Table 3 shows how to calculate ds_j and dc for parents of nodes $data$ (the grammar rule for leaves). As seen in Section 3, transducer T' has an output function λ' and associates the couple $(equality, rank)$ to each final state $q \in F'$ of T', where $equality$ stores the equality type (V ou N) and $rank$ is the rank j of P_j. Transducer T'' follows the same idea, but in this case, as the dependent part of an XFD has just one path, the output function λ'' associates to each final state $q \in F''$ only one symbol representing the equality type. In Table 3, the function $value(t, A)$ returns the parent *position.* In Figure 5 we depict values of attributes ds_1, ds_2 and dc for XFD_3.

Table 4. Semantic Rule for Internal Node Production: Attributes ds_j, dc, $inters$, c.

Production	**Attributes**

$A \to \alpha_1 \ldots\ \alpha_m$	$intersFlag := true$
	for each configuration $\overline{M}.q$ in $A.conf$ do
	(1) if $(\overline{M} = T'') \wedge (q \in F_{T''})$ then
	    if $(\lambda''_{T''}(q) = N)$ then $A.dc := < value(t, A) >$
	            else $A.dc := < \alpha_1.dc, \ldots, \alpha_m.dc >$
	(2) if $(\overline{M} = T') \wedge (q \in F_{T'})$ then
	    $y = \lambda'_{T'}(q)$ $j = y.rank$
	    if $(y.equality = N)$ then $A.ds_j := < value(t, A) >$
	            else $A.ds_j :=< \alpha_1.ds_j, \ldots, \alpha_m.ds_j >$
	(3) if $(\overline{M} = T'') \wedge (q \notin F_{T''})$ then
	    for each α_i $(1 \leq i \leq m)$ do
	      if $(\alpha_i.inters =<>)$ then $A.dc := \alpha_i.dc$
	   if $(\overline{M} = T') \wedge (q \notin F_{T'})$ then
	    for each α_i $(1 \leq i \leq m)$ do
	      if $(\alpha_i.inters =<>)$ then
	        for each j $(1 \leq j \leq k)$ do $A.ds_j \mathrel{+}= \alpha_i.ds_j$
	(4) if $(\overline{M} = M) \wedge (q \in F_M)$
	    then $A.inters := A.inters + \alpha_w.inters$
	      $A.c := < \forall\ w, z$ in $A.inters, w \neq z$: $w.l_1 = z.l_1$
	          $\Rightarrow w.l_2 = z.l_2 >$
	      $intersFlag := false$
	(5) if $(\overline{M} = M) \wedge (q \notin F_M)$
	    then $A.c := < (\forall\ x_w : \alpha_w.c =< x_w >\Rightarrow \bigwedge_{w=1}^{m} x_w) >$
	      $intersFlag := false$
	end for
	(6) if $(intersFlag = true)$ then
	   for each j $(1 \leq j \leq k)$
	    if $(A.ds_j =<>)$ then $A.ds_j := \varepsilon$ // ε is the empty string
	   if $(A.dc =<>)$ then $A.dc := \varepsilon$
	   $temp := < A.ds_1 \times \cdots \times A.ds_k >$
	   $A.inters := < temp \times A.dc >$
	   if $(\ \forall\ \overline{M}.q \in A.conf\ : q \notin F_{T'} \wedge q \notin F_{T''})$ then
	    for each α_i $(1 \leq i \leq m)$ do
	      $A.inters \mathrel{+}= \alpha_i.inters$
	   $A.inters := mapping(A.inters)$

Table 4 defines the synthesized attribute computation for internal nodes. In the following, we explain their computation, respecting the numbering in Table 4. We denote p the parent's position whose synthesized attributes are computed.

1. When p is a node in the XFD dependent path (transducer T'') and is also the last node, we have: if $E = V$, values for attribute dc are obtained from the values of p descendants whose type is *data*; if $E = N$ then dc stores p.
2. When p is the last node for an XFD determinant path P_j we have: if $E = V$ the value of an attribute ds_j is obtained from p descendants whose type is

data; if $E = N$ then ds_j stores p. In this case, *rank* j and the equality type of P_j come from λ' ($y = \lambda'_{T'}(q)$).

3. When p is a node corresponding to the intersection of paths in XFD, then p is seen as a point where obtained values should be combined in order to build a tuple containing the values of the determinant and the dependent parts. We denote this tuple by $< l_1, l_2 >$, where l_1 is a tuple of determinant values, and l_2 is the dependent value. As a result, $< l_1, l_2 >$ contains the combined XFDs values to be carried up.
4. When p is an XFD context node then attribute $inters$ should keep all n-tuples (the final dependency values) from attributes $inters$ of descendant nodes. At this point the XFD is validated for a specific context, and if it is satisfied, attribute c is assigned *true*, otherwise *false*.
5. When p is a node in the context path, then c is assigned the conjunction of values obtained from attribute c of its descendants (one attribute c for each context node). If c is *true* then the XFD is respected up to node p. If p is the root node then the XFD is respected in the whole document.
6. To combine values from the determinant and dependent parts we use a boolean variable ($intersFlag$), a variable (*temp*) and a *mapping* function. Variable $intersFlag$ indicates when there are no more intersections to calculate (which means that the node is in the context path). When there are intersections to calculate, we proceed by steps. Firstly we build a n-tuple for ds_j and dc of a node A. This is done by assigning to *temp* the Cartesian product of each ds_j. Afterwards, a second Cartesian product is done to combine attribute values from *temp* and *dc*. The resulting tuple $< l_1, l_2 >$ is stored in *A.inters*. At this point, function *mapping* verifies, for each tuple l_1, if there are empty values that may be replaced by non-empty values obtained from another tuple l_1. It returns a completed tuple.

Example 2. In Figure 5, we show the computation of attributes c, $inters$, ds_j and dc for XFD_3 of Example 1. Due to the determinant part of XFD_3, attributes ds_j store the values obtained from $@fname$ (supplier name) and @*cname* (component name). On the other hand, dc stores a value for *quantity*. The attributes ds_j and dc carry the dependency values up to the first intersection node (*component*). Notice that for position 0.1.1 we have $ds_1 = \{\epsilon\}$ $ds_2 = \{955XNeo\}$ and $dc = \{5\}$. To compute *temp* we perform $ds_1 \times ds_2$ which gives $temp = \{\{\epsilon, 955XNeo\}\}$. The attribute $inters$ is computed using Cartesian products between ds_j and dc. Thus $inters = temp \times dc$, which gives $inters = \{\{\{\epsilon, 955XNeo\}, 5\}\}$. The function *mapping* is not executed for nodes *component* (positions 0.1.1 and 0.1.2) because there is only one tuple l_1 in each attribute $inters$. The next intersection, for node *supplier* in position 0.1, creates the following tuple, obtained from ds_1 of @*sname*: $< <MSI,\varepsilon>, \varepsilon>$. The attribute $inters$ of node *supplier* stores the new tuple and also puts together the tuples of attributes $inters$ from subnodes: $< < <MSI,\varepsilon>, \varepsilon>,< <\varepsilon, 955XNeo>,5>,< <\varepsilon, K8N>,7> >$. Next, function *mapping* verifies for each two binary tuples in $inters$ if their values can be joined. The result is: $< < <MSI, 955XNeo>,5>, < <MSI, K8N>,7> >$. In the context node, labelled *project*, the dependency is verified (according to Definition 4) and the value *true* is assigned to the attribute c. This last attribute is carried up to the tree root as well as attributes c from other context nodes. □

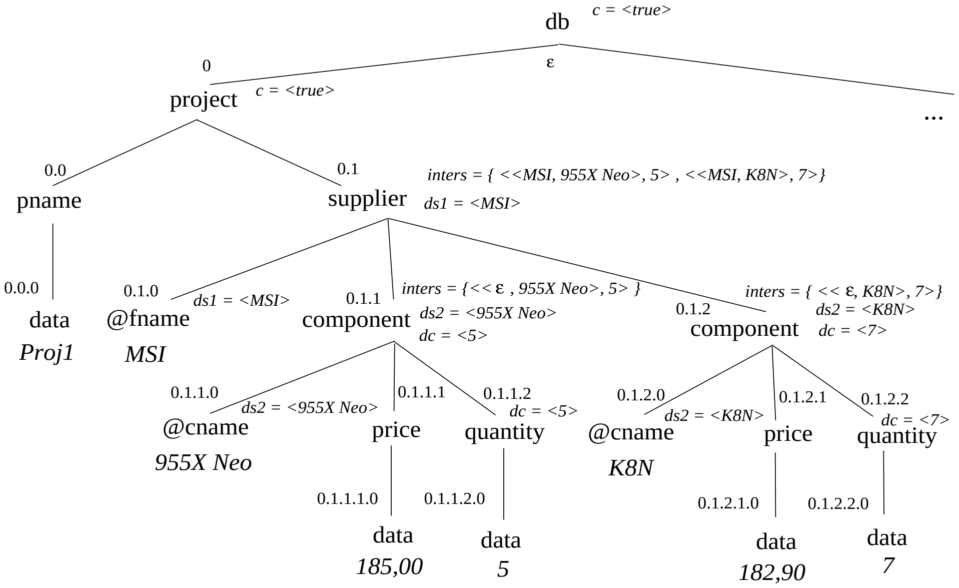

Fig. 5. Synthesized Attributes c, $inters$, ds_j and dc for XFD_3.

To finish this section, we notice that the grammar presented here generates any well-formed XML document (having elements or attributes in Σ_{elem} or Σ_{att}). It can be seen that the documents which respect a given set of XFD are exactly the ones having a value **true** as the attribute c for their context nodes.

5 Algorithm Analysis and Experimental Results

As we have discussed in [7, 10], our *grammarware* can be regarded as a generic way of implementing constraint verification from scratch that requires only one pass on a XML document. Indeed, our way of using attribute grammar for verifying integrity constraints consists in the following stages:
(1) define a generic grammar capable of generating any labelled tree;
(2) define inherited attributes to distinguish nodes which are involved in the integrity constraints, specified by using FSA;
(3) define synthesized attributes whose values are computed by functions that check the properties stated by a given constraint.
Thus, our *generic aspect* refers to the fact that, by adapting some parameters, the same reasoning is used to validate different constraints: in particular, by determining which nodes are important in a constraint definition and, as a consequence, by establishing which FSA and attributes are needed.

The following table illustrates the parameter adaptation for XFD, keys (XKeys) and foreign keys (XFK). The case XFD was discussed in this paper. We refer to [6] for details concerning key and foreign key validation. Notice that besides context and leaf nodes, key specification also needs an extra special node denoted by *target*. Consequently, three finite state automata are used, one associated to each special node in the key (or foreign key) specification.

Constr.	Path expression	FSA	Attributes
XDF	$(C, (\{P_1\ [E_1], \ldots, P_k\ [E_k]\} \to Q\ [E]))$	M, T and T'	Inherit.: : *conf* Synth.: $c, inters, ds_j, dc$
XKeys	$(C, (Tg, \{P_1, \ldots, P_k\}))$	A_C, A_{Tg} et A_P	Inherit.: $conf$ Synth.: c, tg et f
XFK	$(C, (Tg^R, \{P_1^R, \ldots, P_k^R\}) \subseteq (Tg, \{P_1, \ldots, P_k\}))$	A_C, A^R_{Tg}, A^R_P	Inherit: $conf$ Synth.: c, tg et f

As shown in Section 4, XFD validation can be divided in two parts: (i) generation of tuples and (ii) checking, at a context level, the distinctness or (value) equivalence of the obtained tuples. Tables 3-4 have offered the details of these operations: the rules describe how to compose tuples, how to verify when we reach a context node and, in this case, how to perform appropriate checking. The validation of other integrity constraints is done in a similar way, changing the tests performed and the actions in concerned nodes.

The generation of tuples and their verification for a given XFD is done while parsing the XML document $\mathcal{T}$ and its time complexity is $\mathcal{O}(n.n_p.n_t)$ where n is the number of nodes in $\mathcal{T}$, n_p is the number of paths for the given XFD and n_t is the number of obtained tuples (instances of the XFD to be compared). This complexity is not affected by the shape of the XML document, but it can be affected by the number of XFD instances existing in the document. When there is a large number of XFD instances, the comparisons performed are time consuming at context level.

Each XFD is checked by running the finite state automaton that corresponds to its path and we use two stack structures to store the inherited and synthesized attributes. The synthesized attributes are collected to compose the XFD tuples until a context level. At this point, we use a hash table to store the formed tuples. The index/value pair for the hash table is defined by the tuple determinant and dependent parts, respectively. Thus, a tuple insertion in the hash table is valid if its determinant part is not already an index. Otherwise, the dependant value of the tuple that exists in the hash table is compared with the dependant value of the one to be inserted: if they are distinct then the XFD is not respected under the particular context and this part of the validation returns *false*.

The implementation of our validation method was done in Java. XML documents have been created specifically for our tests using the template-based XML generator ToXGene[6]. By using the Xerces SAX Parser documents are read and the necessary information stored into our data structures. The experiments were performed using a PC with Intel Pentium Dual CPU TE2180 at 2.00GHz, 2GB

[6] http://www.cs.toronto.edu/tox/toxgene/

RAM under Microsoft Windows XP. For the tests illustrated in Figure 6, we used 4 XML documents containing projects information (as shown in Figure 1) with varying sizes (8MB, 41MB, 83MB and 125MB) where we considered strings of size 10 and integers of size 3 for random data generation in ToXGene templates. In Part (a) we show time validation when we have a fixed number of paths (2 determinant paths as XDF_3 shown in Section 2) but a varying number of concerned tuples. The validation time increases linearly *w.r.t.* the number of tuples. In Part (b) we do the inverse: we have a varying number of determinant paths for a fixed number of tuples (1000K).

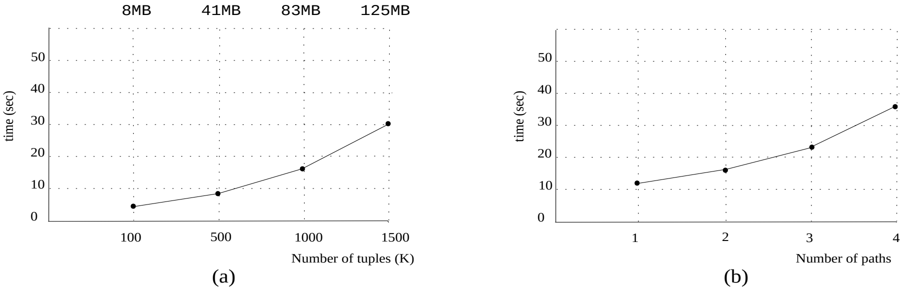

Fig. 6. (a) Validation time with number of LHS paths fixed to 2. (b) Validation time with number of tuples fixed to 1000K.

6 Conclusions

An attribute grammar can be used as an integrity constraint validator. This paper shows its application as an XFD validator while in [6] the same reasoning has been used for keys. As in [17, 16], the validation is performed in linear time *w.r.t.* the document size and the number of XFD paths and instances. The added value of our proposal lies in *its generic nature*, since our generic attribute grammar can stand for any XML constraint validator (provided that the constraint is expressed by paths), by adjusting attributes, tests and the needed FSA. An incremental version of XFD validation is obtained by extending the proposals introduced in [6]. We consider the possibility of adapting our approach to implement more powerful languages such as the tree patterns proposed in [9].

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1988.
2. M. Arenas and L. Libkin. A normal form for XML documents. In *ACM Symposium on Principles of Database System*, 2002.

3. M. Arenas and L. Libkin. A normal form for XML documents. *ACM Transactions on Database Systems (TODS)*, 29 No.1, 2004.
4. 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.
5. N. Bidoit and D. Colazzo. Testing XML constraint satisfiability. *Electr. Notes Theor. Comput. Sci.*, 174(6):45–61, 2007.
6. B. Bouchou, A. Cheriat, M. Halfeld Ferrari, D. Laurent, M. Lima, and M. Musicante. Efficient constraint validation for updated XML databases. *Informatica*, 31(3):285–310, 2007.
7. B. Bouchou, M. Halfeld Ferrari, and M. Lima. Contraintes d'intégrité pour XML. visite guidée par une syntaxe homogène. *Technique et Science Informatiques*, 28(3):331–364, 2009.
8. Y. Chen, S. Davidson, and Y. Zheng. XKvalidator: A constraint validator for XML. In *Proceedings of ACM Conf. on Information and Knowledge Management*, 2002.
9. F. Gire and H. Idabal. Regular tree patterns: a uniform formalism for update queries and functional dependencies in XML. In *EDBT/ICDT Workshops*, 2010.
10. M. Halfeld Ferrari. *Les aspects dynamiques de XML spécification des interfaces de services web avec PEWS*. Habilitation à diriger des recherches, Université François Rabelais de Tours, 2007.
11. S. Hartmann, S. Link, and T. Trinh. Solving the implication problem for XML functional dependencies with properties. In *Logic, Language, Information and Computation*, volume 6188 of *LNCS*. Springer Berlin-Heidelberg, 2010.
12. S. Hartmann and T. Trinh. Axiomatising functional dependencies for XML with frequencies. In *Foundations of Information and Knowledge Systems (FoIKS), 4th Int. Symposium*, pages 159–178, 2006.
13. Christoph Koch and Stefanie Scherzinger. Attribute grammars for scalable query processing on XML streams. *The VLDB Journal*, 16:317–342, July 2007.
14. J. Liu, M. W. Vincent, and C. Liu. Functional dependencies, from relational to XML. In *Ershov Memorial Conference*, pages 531–538, 2003.
15. F. Neven. Extensions of attribute grammars for structured document queries. In *Proceedings of International Workshop on Database Programming Languages*, 1999.
16. Md. S. Shahriar and J. Liu. On the performances of checking XML key and functional dependency satisfactions. In *OTM Conferences (2)*, pages 1254–1271, 2009.
17. M. W. Vincent and J. Liu. Checking functional dependency satisfaction in XML. In *XSym*, pages 4–17, 2005.
18. M. W. Vincent, J. Liu, and C. Liu. Strong functional dependencies and their application to normal forms in XML. *ACM Transactions on Database Systems*, 29(3), 2004.
19. J. Wang and R. Topor. Removing XML data redundancies using functional and equality-generating dependencies. In *Proceedings of the 16th Australasian Database Conference*, 2005.
20. Cong Yu and H. Jagadish. XML schema refinement through redundancy detection and normalization. *The VLDB Journal*, 17:203–223, 2008.
21. X. Zhao, J. Xin, and E. Zhang. XML functional dependency and schema normalization. In *HIS '09: Proceedings of the 9th International Conference on Hybrid Intelligent Systems*, pages 307–312, 2009.