Journal of Applied Computing Research, 3(2):103-117
July-December 2013
© 2013 by Unisinos - doi: 10.4013/jacr.2013.32.04

Conditional functional dependencies validation for XML data: an approach based on attribute grammar

Maria Adriana Vidigal de Lima, Aryadne Guardieiro Pereira Rezende, Caio Thomas Oliveira

Universidade Federal de Uberlândia. School of Computer Science. Av. João Naves de Ávila 2121, Bloco B, 38408-100, Uberlândia, MG, Brazil

madriana@facom.ufu.br, aryadne47@comp.ufu.br, caio@si.ufu.br

Abstract. The representation and the exchange of information originating from different data sources is an increasingly common need for companies and industries to integrate their operations and also to publish and trade information with government and other enterprises. For this purpose, there are many standards based on XML language that were created to allow effective data communication and exchange in a particular domain. In order to ensure data quality for XML data, this paper presents an approach based on conditional functional dependencies verification. Conditional dependencies are an extension of traditional database dependencies with the ability to enforce bindings of semantically related data values. The basis of our verification method is a generic *grammarware* for validating XML integrity constraints in one tree traversal. We use an attribute grammar to describe XML documents and constraints.

Keywords: conditional functional dependencies, data quality, XML integrity constraints.

Introduction

XML language has become the preferred format for data integration and information exchange between organizations, and has emerged as a well-accepted data model for heterogeneous data in many application domains. With the growing use of XML as a format for permanent storage of data, the topic of integrity constraints has received increased importance and has been identified as a challenging subject of XML research, as there is a relevant need for developing principles, algorithms and techniques for efficiently managing XML data, considering its semi-structured nature. Although XML provides many advantages based on the ability to contain both data and information about the data and to provide an extensible and adaptable data format, it is hard for XML to express semantic information. The definition and use of integrity constraints is one of the topics of XML semantics, which is fundamental to other XML research areas, such as normalization, query optimization and data quality.

Several types of integrity constraints have been studied in the context of XML language, as key constraints, functional dependencies, inclusion dependencies and path constraints (Buneman *et al.*, 2001; Deutsch and Tannen, 2005; Hartmann and Trinh, 2006; Karlinger *et al.*, 2009). Those constraints are a mechanism to express how elements contained in an XML document are associated to each other. Functional dependency is an important kind of integrity constraint and many definitions for XML functional dependencies with different notions were proposed in Arenas and Libkin (2004), Liu *et al.* (2003), and Wang and Topor (2005). As XML databases are designed without much consideration of integrity constraints, tools for dependencies discovery can be very useful to ensure data quality and knowledge exploration. For that reason, algorithms for dependencies discovery have been proposed in the XML field (Trinh, 2008).

A novel extension of traditional functional dependency referred to as conditional functional dependency was recently introduced. It was conceived to capture the notion of correct data in a specific situation. A conditional dependency represents a weaker form of dependency defined to act within a scope of data limited by conditions on several attributes. Only the tuples that satisfy these conditions should be evaluated and this conditional application allows contextualizing analysis on data, and in addition, to find and correct specific inconsistencies. In this way, conditional functional dependencies for XML (XCFD) extend functional dependencies with a conditional expression that allows the application of a functional dependency only over parts of an XML document, representing a specific subset of data. As an example, suppose a database of bank accounts in which we want to check (i) if the bank is B1 then a unique account number identifies each customer, (ii) if the credit card is PlatinumCard for bank B2, then the card type identifies the interest rate. Those conditions are useful to understand the characteristics of a given data subset, and also to assess quality.

In this work, an XML document is seen as a structure composed of an unranked node-labeled tree and some functions for handling this tree. A path expression defines a way of navigating XML trees and is fundamental for the specification of integrity constraints in an XML context. In this way, we want to express conditional functional dependencies (XCFD) based on path expressions using a homogeneous formalism for their verification, introduced in Bouchou *et al.* (2011), founded on attribute grammar and finite state automata. The validation of an XML document is done in one tree traversal, in the document reading order, going top-down until leaf nodes are reached and then, bottom-up, to finish each node visit, until the root node. In the top-down direction, the validation process uses attributes to specify the role of each node visited with respect to the defined constraints. In the bottom-up direction, the values concerned by the constraints are pulled up and treated via some other attributes. Our approach for XCFD validation does not depend on document schema and is established by a traversal function that receives an XML document and a set of XCFDs, and checks if each constraint is respected. The validation result is a Boolean value that is a conjunction of each XCFD Boolean result, as shown in Figure 1.

Paper organization: In Related Work we discuss related work on XML constraints and their verification. Next, we introduce our basic definitions. In Validation of conditional functional dependencies for XML we show our algorithms, based on attribute grammar to check if a set of XCFD is satisfied by an XML document and we discuss the verification process. In Framework for XML integrity constraints validation with conditions, we consider aspects of design patterns to implement the verification proposal taking into account homogeneous formalism. Finally, we present the conclusion of this paper and future work perspectives.

Related Work

In the relational context, many works are being developed to improve the quality of integrated data, with renewed interest in the application of classical dependencies (Fan, 2008; Liu *et al.*, 2011; Bakhtouchi *et al.*, 2011) and conditional dependencies (Bohannon *et*

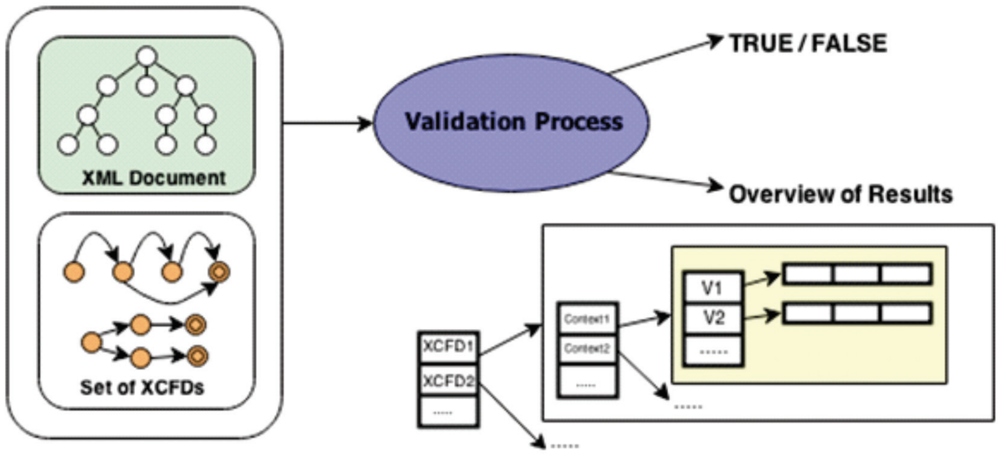

Figure 1. Validation structure. Overview of the verification process for a set of XCFDs.

al., 2007; Fan *et al.*, 2008; Ma *et al.*, 2014) for the preservation of semantics, detection and repairing of possible inconsistencies. In a similar way, classical and conditional dependencies on XML data have been proposed for data semantic verification. Several approaches for XML functional dependencies have been proposed (Buneman *et al.*, 2001; Liu *et al.*, 2003; Wang and Topor, 2005; Shahriar and Liu, 2008; Bouchou *et al.*, 2012; Tan and Zhang, 2011), and also for conditional functional dependencies (Vo *et al.*, 2011).

The implementation of constraint validators has received less attention. Our approach performance is comparable to implementations in Vincent and Liu (2005), Shahriar and Liu (2009), but contrary to them, it intends to be a generic model for XML constraints validation. The ideas guiding this work are the ones outlined in Bouchou *et al.* (2011). An incremental validation method for keys is defined in Bouchou *et al.* (2007), using the generic model considering multiple updates in an XML tree. In Gire and Idabal (2010), 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.

Basic definitions

Following the basic definitions used throughout this paper are shown.

Definition 1. XML Document: Let $\Sigma = \Sigma ele \cup \Sigma att \cup \Sigma data$ be an alphabet where Σele is the set of element names and Σatt is the set of attribute names. An XML document is represented by a tuple T = (*t*, *type*, *value*). The tree *t* is the function *t*: dom(*t*) $\rightarrow \Sigma$ where Σ is a set of tags and dom(*t*) is a set of positions. Given a tree position *p*, function *type* (*t*, *p*) returns a value in {*data*, *element*, *attribute*}. We also recall that, in an XML tree, attributes are unordered while elements are ordered.

A tree representing an XML document containing bank account information is illustrated in Figure 2. At each node, its position and its label (e.g., *t*(0) = *bank* and *t*(0.0.1) = *card*) are shown together and values (in italic) are associated to leaves. A path for an XML tree *t* is defined by a sequence of tags or labels. The path language *PL* is used to define integrity constraints over XML trees. In *PL*, [] 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* is a common fragment of regular expressions and XPath. A path *P* is valid if it conforms to the syntax of *PL* and for all tags *l* in *P*, if *l=data* or $l \in \Sigma att$ then *l i*s 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 2. 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. In a sequence of positions I= $v_1/ \ldots /v_n$, 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 $t(v_1)/\ldots /t(v_n) \in L(A_P)$.

A tuple is formed by the values or position values found at the end of a path instance for a path *P* over a tree *t*. The notion of tuple is important for integrity constraints for composing values that may be compared and verified. A functional dependency in XML (XFD) is denoted X → Y (where X and Y are sets of paths) and it imposes that for each pair of tuples t_1 and t_2, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. A XFD has a single path on the right-hand side and possibly more than one path on the left-hand side. This approach generalizes the proposals in Arenas *et al.* (2004), Liu *et al.* (2003), and Wang and Topor (2005).

The dependency can be imposed in a specific part of the document, and, for this reason, a context path can be specified. 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, which can be represented by V for value equality, or N for node equality.

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

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

where C is a path representing the 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 starting at the context node. The set $\{P_1, \ldots, P_k\}$ is the left-hand side or determinant of an XFD, and Q is the right-

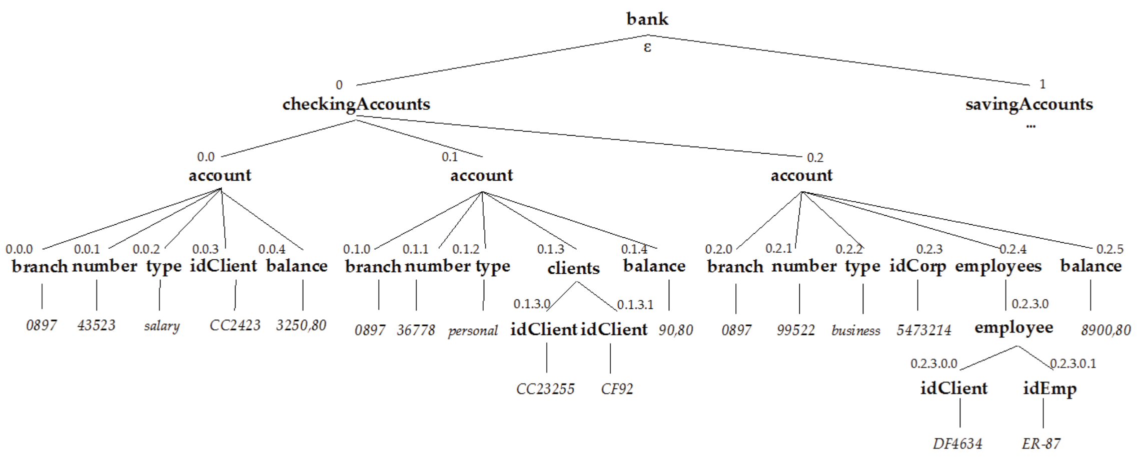

Figure 2. XML tree. Fragment of the XML document containing information about bank accounts.

hand side or the dependent path. The symbols $E_1,\ldots,E_k,E$ represent the equality type associated to each dependency path. When the symbols E or $E_1,\ldots,E_k$ are omitted, value equality is the default choice.

Definition 4. Satisfaction of functional dependency for XML: Let *t* be an XML tree, $\gamma = (C, (\{P_1\ [E_1], \ldots , P_k\ [E_k]\} \rightarrow Q\ [E]))$ an XFD. The set $S = \{C/P_1, \ldots, C/P_k, C/Q\}$ gathers all paths from the root for constraint γ. Each instance I of set S is defined by:

> $I_S = \{t_1, \ldots, t_k, t_q\}$ where t_i (i in [1..k]) is the tuple formed by the values (or position values, depending on E_i) found following $C/P_i\ [E_i]$ for instance I, and t_q is tuple obtained from C/Q [E].

The document represented by tree *t* satisfies the constraint γ if and only if for all two instances of S, namely I_S and I_R that coincide at least on their prefix C, we have:

> **if** $\{t_{S1}, \ldots, t_{Sk}\} = \{t_{R1}, \ldots, t_{Rk}\}$ **then** $t_{Sq} = t_{Rq}$

Conditional functional dependencies are similar to functional dependencies, but with a peculiarity of having a conditional expression that allows establishing a restriction associated to the values contained in a database.

Definition 5. Conditional functional dependency for XML: Given an XML tree *t*, a conditional functional dependency for XML (XCFD) is an expression of the form

> $(C, (Cond, \{P_1\ [E_1], \ldots , P_k\ [E_k]\} \rightarrow Q\ [E]))$

where C and $\{P_1\ [E_1], \ldots , P_k\ [E_k]\} \rightarrow Q\ [E]$ are defined similarly as in functional dependencies for XML, and Cond is a combination of expressions of the form: $expr_1\ \theta_1\ expr_2\ \theta_2 \ldots \theta_{n-1}\ expr_n$. Each expression $expr_i$ is a relational expression and represents a condition. The conditions are connected by logical operators (θ_i for operations $\wedge$, $\vee$), thus, several specific conditions can be imposed in the constraint definition. An expression $expr_i$ is defined as a sentence of the type $PC\ \varphi\ vc$, where PC is a path expression, φ is a relational operator ($=$; $\neq$; $<$; $>$; $\leq$; $\geq$) and vc is the expected value.

Definition 6. Satisfaction of conditional functional dependency for XML: Let *t* be an XML tree, $\gamma = (C, (Cond, \{P_1\ [E_1], \ldots , P_k\ [E_k]\} \rightarrow Q\ [E]))$ a XCFD. The set $S = \{C/PC_1, \ldots, C/PC_n, C/P_1, \ldots, C/P_k, C/Q\}$ gathers all paths from the root for constraint γ, being $C/PC_1, \ldots, C/PC_n$ the conditional paths. Each instance I of set S is defined by:

> $I = \{e_1,\ldots,e_n,t_1,\ldots,t_k, t_q\}$ where each e_i is a value obtained from C/PC_i and $t_1,\ldots,t_k, t_q$ are tuples obtained similarly to functional dependencies for an instance I.

The document represented by tree *t* satisfies the constraint γ if and only if for all two instances of S, namely I_S, I_R that coincide at least on their prefix C, we have:

> **if** $(((e_{S1}\ \varphi_1\ vc_1\ \theta_1 \ldots \theta_{n-1}\ e_{Sn}\ \varphi_n\ vc_n) = True)$ **and** $((e_{R1}\ \varphi_1\ vc_1\ \theta_1 \ldots \theta_{n-1}\ e_{Rn}\ \varphi_n\ vc_n) = True))$
>
> **then**
>
> **if** $\{t_{S1}, \ldots, t_{Sk}\} = \{t_{R1}, \ldots, t_{Rk}\}$ **then** $t_{Sq} = t_{Rq}$

Example 1: In a context of a banking environment we consider a situation where checking accounts are integrated, as illustrated in Figure 2. There are many kinds of checking accounts: personal, salary, university, business, etc., and for each one we can find differences in type of data stored, operations and also and restrictions that can be conditionally applied. For example:

1. If the checking account is of type salary, then it must be an individual personal account. In this situation, it is not possible to associate other clients to this account.
2. Many employees can be associated to a business account, but the same employee cannot be associated to more than one business account.
3. Pre-approved credit is offered to personal accounts depending on the average balance factor, if this factor is greater than 0.

Those rules can be translated in two conditional functional dependencies, respectively:

η_1 = *(/bank/checkingAccounts (/account/type = "salary", {account/number} $\rightarrow$ \account\idClient))*

η_2 = *(/bank/checkingAccounts (/account/type = "business", {account/employees/employee/idClient, account/employees/employee/idEmp} $\rightarrow$ \account\number))*

η_3 = *(/bank/checkingAccounts (/account/type = "personal" $\wedge$ /account/averageBalanceFactor > 0, {account/averageBalanceFactor} $\rightarrow$ \account\preapprovedCredit))*

Finite state automata for functional dependencies in XML

We use finite-state automata (FSA) or transducers (FST) to formalize paths in integrity constraints. The input alphabet for the finite-state automata is the set of XML tags. The output alphabet for transducers is composed by our equality symbols (for XFDs) and also by the expected values in conditions with the respective relational operator (for XCFDs). 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 $\Delta: \Theta \times V \rightarrow \Theta$ is the transition function. A FST is a 7-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

(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 and the dependent path, 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 by XML document tags.

- The determinant transducer $T' = (\Theta', \Sigma, \Gamma', \Delta', e', F', \lambda')$ expresses paths Pi ($i \in [1, k]$). The set of output symbols is $\Gamma' = \{V,N\} \times N^*$ 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 dependent side. 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 $q \in F''$.

For conditional functional dependencies, there is another set of paths, representing the conditions that must be verified. For this purpose, a new transducer is used to formalize the paths in the part Cond, specified in Definition 5:

- All paths contained in the Boolean expressions defined in Cond are expressed by the conditional transducer $T^C = (\Theta^c, \Sigma, \Gamma^c, \Delta^c, e^c, F^c, \lambda^c)$. The set of output symbols is $\Gamma^c = \Sigma \times \{= ; \neq; < ; > ; \leq ; \geq\} \times N^*$ such that the expected values defined in conditions and also the respective relational operation can be associated to each conditional path. Thus, the output function λ^c associates a triple (vc, φ, rank) to each final state $q \in F^c$.

A finite-state automaton is a machine that can be in one of a finite number of states, and in certain conditions, it can switch to another state by a transition. When the machine starts working it may begin from an initial state, in the case of XML data, the state representing the root node. Figure 3 illustrates FSAs and FSTs for XCFDs η_1 and η_3 defined in Example 1.

Attribute grammar

The general process for validating integrity constraints in XML documents can be performed with the use of an attribute grammar. Attribute grammars are extensions of context-free grammars that allow to specify

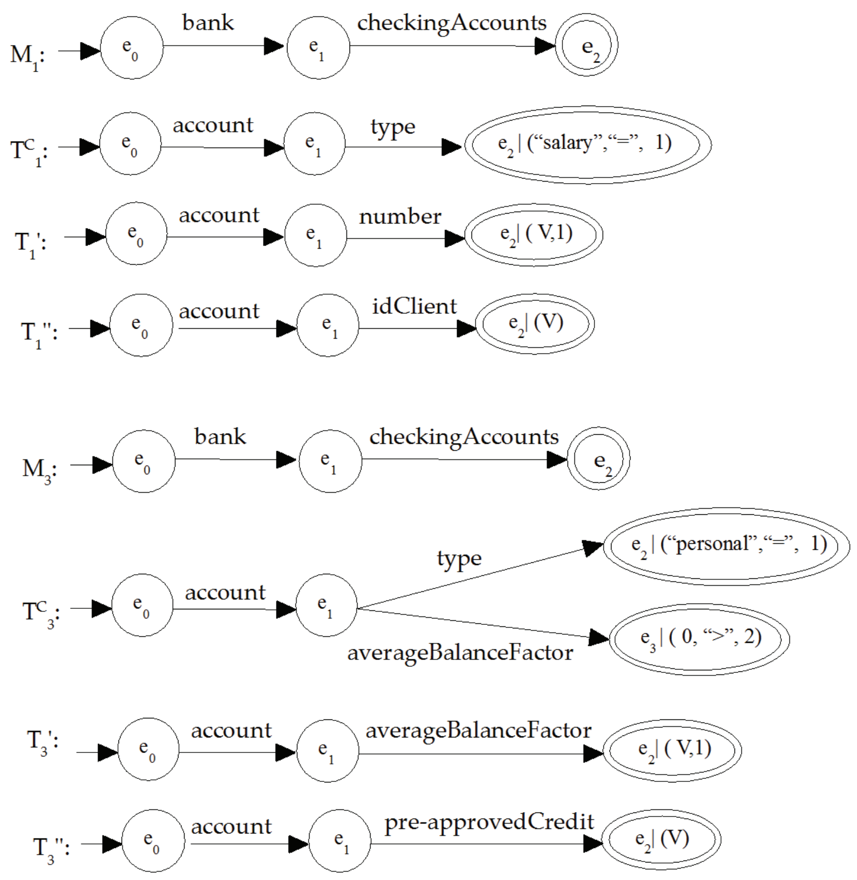
M1: e0 bank e1 checkingAccounts e2
TC1: e0 account e1 type e2 | ("salary","=", 1)
T1': e0 account e1 number e2 | (V,1)
T1'': e0 account e1 idClient e2 | (V)
M3: e0 bank e1 checkingAccounts e2
TC3: e0 account e1 type e2 | ("personal","=", 1) averageBalanceFactor e3 | (0, ">", 2)
T3': e0 account e1 averageBalanceFactor e2 | (V,1)
T3'': e0 account e1 pre-approvedCredit e2 | (V)

Figure 3. Examples of automata and transducers. Automata and transducers for XCFDs η_1 and η_3.

not only the syntax, but also the semantics of a language. We consider a context-free grammar G = (V_N, V_T, P, B) where V_N is the set of non-terminal symbols, V_T is a set of terminal symbols, P is the list of productions and B is the start symbol. In order to annotate extra information to a symbol, we attach semantic rules to its productions. In a semantic rule we can create attributes that may represent anything: a string, a number, a type, a memory location Aho *et al.* (1988). Those rules are declarative specifications describing how the attached attributes are computed. Two types of attributes can be found in a semantic rule: 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.

An attribute grammar is a triple GA = (G, A, F) where: G 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 ∊ $V_N \cup V_T$, we have A(X) = S(X)+I(X), i.e., A(X) is a set composed by the disjoint union of S(X), that is the set of synthesized attributes of X and I(X), the set of inherited attributes of X. For each production p: $X_0 \rightarrow X_1 \ldots X_n$, the set F_p contains the semantic rules that handle the set of attributes of p and describe its semantic features. In consequence, the semantic parsing of a sentence is executed using the set of actions associated to each production rule. In each action definition, the values of attribute occurrences are calculated in terms of other attribute values.

In this work we assume that G is a simple grammar describing any XML tree. To verify integrity constraints, one may augment G by semantic rules, using attributes that can constitute information to be used in the validation. Consider a context-free grammar G with the following three generic production rules:

(i) $Root \rightarrow \alpha_1 \ldots \alpha_m$, $m \in \mathbb{N}$.
(ii) $A \rightarrow \alpha_1 \ldots \alpha_m$, $m \in \mathbb{N}^*$
(iii) $A \rightarrow$ *data*

where (i) defines the production in which $\alpha_1 \ldots \alpha_m$ are direct descendent nodes from the root node, (ii) defines the production rule for an internal node that must have at least one direct descendent, and (ii) defines the production for a leaf node.

The grammar G can be augmented by semantic rules, containing grammar attributes, defining the exact actions that must be performed concerning integrity constraints validation. The parsing of an XML document is done by a top-down traversal in its tree using open-tag and close-tag events. During the descendent direction, the validation process defines the role of each node regarding the constraints being verified by using the finite state automata that formalize its paths. This information is stored in an inherited attribute, since it is calculated in the descendent direction. When leaves are reached then an upward trajectory begins to treat and store the encountered values, concerning the constraints, into synthesized attributes.

Validation of conditional functional dependencies for XML

The conditional functional dependencies validation process receives a set of XCFDs and a XML document. The validation of a XCFD is accomplished using an attribute grammar approach, wherein for each node in the XML tree, inherited and synthesized attributes are associated. Each association takes advantage of one of two parser events over the XML tree to be validated. The two events are: o*pen-tag* and *close*-tag. Considering ω a set of XCFDs to be verified, the traversal in the XML tree is performed according to Algorithm 1.

Algorithm 1 – Validation of conditional functional dependencies

Input:
(i) ω: a set of z XCFDs
(ii) Doc: an XML document
Output: the Boolean value *true* if the document satisfies the set of XCFDs, otherwise *false.*

Local Variables:
(i) tg : document tag referring to a node
(ii) InhStack: stack to store inherited attributes tuple
(iii) SyntStack: stack to store synthesized attributes tuple
(iv) InhAttList: k-tuple to organize inherited attributes for XCFDs
(v) SynAttList: k-tuple to organize synthesized attributes for XCFDs

```
(1)  for each XCFD ηi ∈ ω do
(2)     build Mηi, T′ηi, T″ηi, Tcondηi;
                              // FSA and FSTs for XCFDs
(3)  push (NULL,…, NULL) into SynStack;
(4)  push ({$M\eta_1.e_0$},..., {$M\eta_k.e_0$}) into InhStack;
                              // Initial states for FSAs
(5)  for each tag tg in XML document do
(6)     if tg is an opening tag then
(7)        $inhAttList_{parent}$ = top from InhStack;
(8)        for each XCFD $\eta_i \to \omega$ (i in [1..z]) do
(9)          $inhAtt\eta_i$ = calculateInhAttributes(tg,
             $inhAttList_{parent}$[i],
                  ($M\eta_i$, $T'\eta_i$, $T''\eta_i$, $T_{cond}\eta_i$));
(10)       inhAttList = ($inhAtt\eta_1$,...,$inhAtt\eta_z$);
(11)       push inhAttList into InhStack;
(12)       push (NULL,…, NULL) into SynStack;
(13)    else                          //closing tag
(14)       $inhAttList_{current}$ = pop from InhStack;
(15)       $synAttList_{current}$ = pop from SynStack;
(16)       $synAttList_{parent}$ = pop from SynStack;
(17)       if (leaf(tg)) then
(18)         for each XCFD $\eta_i \in \omega$ (i in [1..z]) do
(19)           $synAtt\eta_i$ = calculateSynAttributes-
               Leaf(tg, $inhAtt_{current}$[i], $synAtt_{current}$[i],
               $synAtt_{parent}$[i]);
(20)       else
(21)         for each XCFD $\eta_i \in \omega$ (i in [1..z]) do
(22)           $synAtt\eta_i$ = calculateSynAttribute-
               sInt(tg, $inhAtt_{current}$[i], $synAtt_{current}$[i],
               $synAtt_{parent}$[i]);
(23)       synAttList = ($synAtt\eta_1$,...,$synAtt\eta_z$);
(24)       push synAttList into SynStack;
(25) $syntAttList_{root}$ = pop from SynStack;
(26) result = calculateResult($syntAttList_{root}$);
(27) return result;
```

Algorithm 1 expresses a non-recursive function that uses stacks to direct the traversal of an unranked XML tree for the verification of *z* XCFDs. Two stacks are used to organize the association between grammar attributes and tree nodes. The first stack, *inhStack* is responsible for storing, for each node that is found (at open-tag event), a *z*-tuple containing the inherited attributes that were calculated for all XCFDs at that point. The second one is *synStack* and it is used for saving the *z*-tuple of the synthesized attributes computed, at close-tag event, for all constraints, during the tree visit. At the end of the tree traversal, the constraints verification is finally computed at the root node using its associated *z*-tuple at *synStack*. If, for all XCFDs, no violations were found then the function *calculateResult* returns true, otherwise, false.

Grammar attributes and their computation

In this section we detail inherited and synthesized attributes used in Algorithm 1 and their computation. The grammar attributes are responsible for storing values and partial results of treatments and comparisons between the encountered values that concern defined constraints. One inherited attribute is used for each constraint at each node to assign the rule of a node tag with respect to a given XCFD. On the other side, various synthesized attributes are needed for each constraint at each node, because they are used for creating the tuples for the encountered values (determinant and dependent side of the dependencies), to compute and store values referring to the conditions, and to store the result of manipulations and comparisons between dependencies values.

Inherited attribute

The inherited attribute used in the XCFD verification is named *conf*. As shown in Algorithm 1, line 7, it is calculated for each constraint when an opening tag is found. The computation of this attribute uses the automaton and transducers that were built using the path expressions given by an XCFD and also information from the *conf* attribute value associated to its parent node. The *conf* attribute calculation for each node (at open-tag event) for each XCFD is specified in Algorithm 2, considering the rules for XML grammar defined in section 3.2.

Algorithm 2 - Calculation of *conf* attribute

Function name: calculateInhAttributes
Input:
(i) A: tag opening (current node)
(ii) ParentConf: attribute *conf* (from parent node)
(iii) M, T′, T″, T_{cond} (FSA and FSTs for a XCFD η)
Output: conf attribute for node A

```
(1) if A is ROOT node then
(2)     conf := {M.$q_1$ | $\Delta(q_0, A) = q_1$};
(3) else
(4)     for each K.q ∈ ParentConf do
(5)         if $(K = M) \wedge (q \in F)$ then
(6)             conf := conf ∪ {T′.q1′ | $\Delta'(q0', A) = q1'$}∪
                    {T″.$q_1''$ | $\Delta''(q_0'', A) = q_1''$} ∪
                    {$T^C.q_1^c$ | $\Delta^c(q_0^c, A) = q_1^c$};
(7)         else
(8)             conf := {K.q′ | $\Delta_K(q, A) = q'$};
(9) return conf;
```

The *conf* attribute is calculated according to Algorithm 2 and it stores a set of configurations of type *M.e*, where *M* is a FSA or FST, and *e* is a state of *M*. In line (1) from Algorithm 2 we specify the case where the current node is the tree root. In this case the attribute *conf* is calculated by initializing the FSA M (for context path) and executing a transition using the root tag. If the current node is not the root (line (3)) then we must check if the final state from FSA M is reached. If it is the case, then transitions for FSTs T′, T″ and T^C from their initial states may be executed to check if this node is in their paths. Then the corresponding configurations are stored in *conf*. Figure 4 shows the computation of attribute *conf* for XCFD η_1 using the corresponding FSA and FSTs depicted in Figure 3.

Synthesized attributes for leaf nodes

When a leaf node is found it is necessary to verify if data contained in this node concerns any dependency being verified and, and if so, data is collected in synthesized attributes. To define the synthesized attributes, we recall the XCFD definition that is (C, (Cond, {P_1 [E_1], ... , P_k [E_k]} → Q [E])). Initially, we define the attributes ds_i, i in [1..k] to store values respectively to paths P_i , attribute *dc* to save values concerning path Q, and $dcond_j$, j in [1..n], to store the values obtained from conditional paths in Cond. Also, an attribute *inters* is defined to gather all values found for a constraint in tuples <*lcond*, *ldep*>, where *lcond* = <$dcond_1$, …, $dcond_n$>, and *ldep* is a tuple <*lds*, *dc*> to store the determinant part and the dependent part of the dependency (respectively l_1 and l_2). For this purpose, *lds* = <ds_1,…, ds_k>.

An extra attribute, called *c*, is defined to store a Boolean value representing the result of the validation for a context. At a leaf node, this result is not calculated yet. Synthesized attributes are grouped in a structure specified by ($dcond_j$, ds_i, *dc*, *{inters}*, *c*). Empty values are filled with the symbol ε as detailed in Algorithm 3.

Algorithm 3 - Calculation of synthesized attributes for leaf nodes

Function name: calculateSynAttributesLeaf
Input:
(i) A: tag closing (current node)

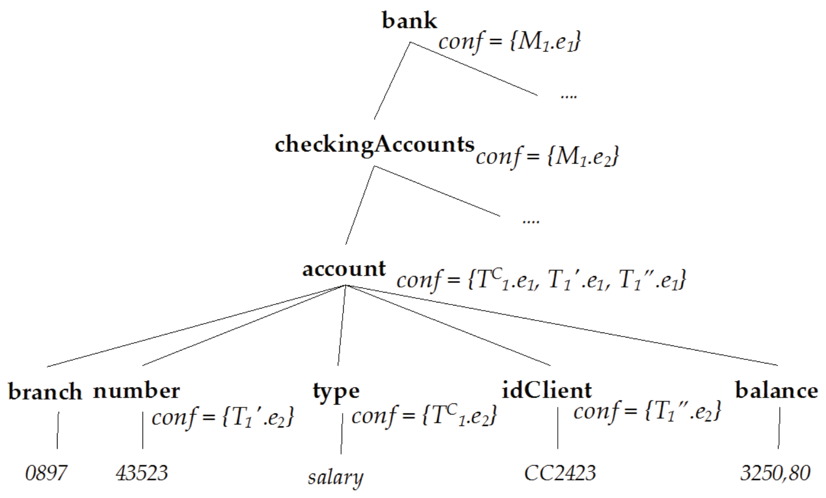

Figure 4. Fragment of the XML document. In this fragment the computation of the attribute *conf* for XCFD η_1 is shown.

(ii) CurrentConf: *conf* attribute (from current node)
(iii) CSA: synthesized attributes ($dcond_j$,ds_i,dc,{inters},c) of current node
(iv) PSA: synthesized attributes ($dcond_j$,ds_i,dc,{inters},c) of parent node

Output: PSA

```
(1)  for each configuration K.q ∈ CurrentConf do
(2)     if (K = T′) ∧ (q ∈ F′) then
(3)        y:= λ′(q);
(4)        i:= y.rank;
(5)        if y.equality = V then
(6)          CSA.ds_i:= <data_value(A)>;
(7)        else
(8)          CSA.ds_i:= <node_value(A)>;
(9)        CSA.inters := CSA.inters ∪ {<<ε,…,ε>,<<ε,…,dsi,…,ε>,ε>>};
(10)    if (K = T″) ∧ (q ∈ F″) then
(11)       if λ″(q) = V then
(12)         CSA.dc:=<data_value(A)>;
(13)       else
(14)         CSA.dc:=<node_value(A)>;
(15)       CSA.inters := CSA.inters ∪ {<<ε,…,ε>,<<ε,…,ε>,dc>>};
(16)    if (K = T^C) ∧ (q ∈ F^c) then
(17)       z := λ^c(q);
(18)       j := z.rank;
(19)       ev := z.expectedValue;
(20)       op := z.relationalOperator;
(21)       v :=<data_value(A)>;
(22)         CSA.dcond_j := eval(v,op,ev);
(23)       CSA.inters := CSA.inters ∪ {<<ε,…,dcondj,…,ε>,<<ε,…,ε>,dc>>};
(24)    PSA.inters:=PSA.inters∪mapping(CSA.inters);
(25) return PSA;
```

The process expressed in Algorithm 3 determines that all leaf values may be verified, and if they are values from the determinant part, we use the attributes ds_i to collect them first. The value concerning the dependent part of the dependency is stored in *dc*, and those from conditional paths are verified and the result is stored in $dcond_j$ for each conditional expression. The attribute *inters* is important because it provides a mechanism to build and group the complete tuples of the instances found for each dependency. The function mapping in line 24 is responsible for merging some tuples and eliminating empty values.

Synthesized attributes for internal nodes

Continuing the bottom-up visit in the XML tree, all synthesized attributes ($dcond_j$, ds_i, *dc*, *{inters}*, *c*) are calculated for internal nodes at close-tag events. This computation uses the synthesized attributes obtained until this point (from its child nodes). Those pieces of information must be treated and carried up to the parent node (which is not yet closed) to be regrouped with information from previous siblings. Algorithm 4 describes the process of calculating and associating synthesized attributes for internal nodes that carry the dependencies values and verifying their properties.

Algorithm 4 - Calculation of synthesized attributes for internal nodes

Function name: calculateSynAttributesInt
Input:
(i) A: tag closing (current node)

(ii) CurrentConf: *conf* attribute (from current node)
(iii) CSA: synthesized attributes ($dcond_j$,ds_i,dc,{inters},c) of current node
(iv) PSA: synthesized attributes (dcondj,dsi,dc,{inters},c) of parent node
Output: PSA

```
(1)  for each configuration $K.q \in$ CurrentConf do
(2)    if ($K = T'$) $\wedge$ ($q \in F'$) then
(3)      y:= $\lambda'(q)$;
(4)      i:= y.rank;
(5)      if y.equality = N then
(6)        CSA.$ds_i$:=<node_value(A)>;
(7)        CSA.inters := CSA.inters $\cup$ {<<$\varepsilon$,…,$\varepsilon$>,<<$\varepsilon$,…,dsi,…,$\varepsilon$>,$\varepsilon$>>};
(8)        PSA.inters := PSA.inters $\cup$ mapping (CSA.inters);
(9)    if($K = T''$) $\wedge$ ($q \in F''$) then
(10)     if $\lambda''(q)$ = N then
(11)       CSA.dc:=<node_value(A)>;
(12)       CSA.inters := CSA.inters $\cup$ {<<$\varepsilon$,…,$\varepsilon$>,<<$\varepsilon$,…,$\varepsilon$>,dc>>};
(13)       PSA.inters := PSA.inters $\cup$ mapping (CSA.inters);
(14)   if($K \neq M$) $\wedge$ ($q \notin F_K$) then
(15)     PSA.inters := PSA.inters $\cup$ mapping (CSA.inters);
(16)   if($K = M$) $\wedge$ ($q \in F_M$) then
(17)     CSA.c = true
(18)     CSA.c := <$\forall$ w,z $\in$ CSA.inters, w $\neq$ z:
                   validateCondition(w) $\wedge$
                   validateCondition(z) $\wedge$
                   w.ldep.lds = z.ldep.lds $\Rightarrow$
                   w.ldep.dc = z.ldep.dc >;
(19)     PSA.c := CSA.c $\wedge$ PSA.c;
(20)   if($K = M$) $\wedge$ ($q \notin F_M$) then
(21)     PSA.c := CSA.c $\wedge$ PSA.c;
(22)   return PSA;
```

As described in Algorithm 4, the values that are part of the conditional functional dependency are collected, treated and carried up to the context node. Attribute *inters* is responsible for gathering (bottom-up) the values that are in conditional, determinant and dependent path intersections. At the context nodes, these intersection values are compared in order to verify the XCFD satisfaction. Attribute *c* is used to carry the dependency validity (true or false) from the context level to the root. It can be observed that in line (17) of Algorithm 4 the *c* value remains neutral (true) if there are no instances of XCFD that respect the condition imposed by an XCFD.

In Figure 5, we show the computation of synthesized attributes for XCFD η_1 in a small portion of an XML document. Due to the determinant part of the XCFD, attribute ds_1 stores the values obtained from *number* (bank account number). As there are not any other paths in the determinant side, then *dc* stores the client identification code (*idClient*). For the conditional expression, we have only one conditional path, defining that the account type must be "*salary*", then the attribute $dcond_1$ stores the value *true*. The intersection sets are calculated for all leaf nodes and they are carried up to parent node, as the corresponding nodes are closed. At this point function mapping is responsible for combining those intersection tuples coming from the child node with the ones already at

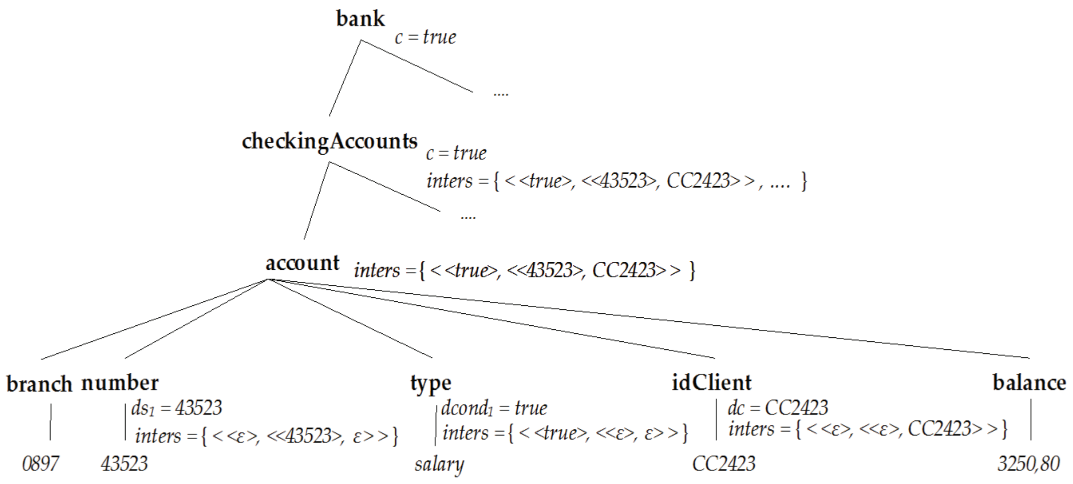
bank
c = true
....
checkingAccounts
c = true
inters = { < <true>, <<43523>, CC2423> > , ... }
....
account
inters = { < <true>, <<43523>, CC2423> > }
branch
number
type
idClient
balance
ds1 = 43523
inters = { < <ε>, <<43523>, ε> > }
dcond1 = true
inters = { < <true>, <<ε>, ε> > }
dc = CC2423
inters = { < <ε>, <<ε>, CC2423> > }
0897
43523
salary
CC2423
3250,80

Figure 5. Computation of attributes $dcond_1$, ds_1, *dc*, *inters* and *c* for XCFD η_1.

the parent node. After closing all child nodes, for node labeled *account*, we have *inters* = {<<*true*>,<<*43523*>,*CC2423*>>}. At the node labeled *checkingAccounts* all intersection sets are combined, and as it is a context node, all tuples contained in the intersection set are verified (according to Definition 6) and if no violations are identified, the result for the verification is *true*, and it is stored in attribute *c*.

Auxiliary algorithms

This section defines the auxiliary algorithms used in the previous main algorithms. In line (22) of Algorithm 3, a function *eval* is used to evaluate a relational expression. This function is detailed in Algorithm 5.

Algorithm 5 – Evaluation of Relational Expressions

Function Name: eval
Input:
(i) value : String
(ii) operator : String
(iii) expectedValue : String
Output: a Boolean value

```
(1)  if (op = "=") then
(2)     result := (value = expectedValue);
(3)  else if (op = "!=") then
(4)        result := (value != expectedValue);
(5)     else if (op = "<") then
(6)           result := (value < expectedValue);
(7)        else if (op = ">") then
(8)              result:= (value > expectedValue);
(9)           else if (op = "<=") then
(10)                result := (value <= expectedValue);
(11)             else if (op = ">=") then
(12)                   result := (value >= expectedValue);
(13)                else return false;
(14) return result;
```

It can be seen in lines (16-18) from Algorithm 4 that when a given node is reached in the XML document and at the same time a final state in the context path is also reached, a checking is executed to ensure that the XCFD instances respect the value constraints specified on the condition expression. The function *validateCondition* uses the intersection values (from attribute *inters* associated to current node) and verifies whether this tuple is complete and resolves the values contained in the conditional part of the tuple. Algorithm 5 specifies this process.

Algorithm 6 – Evaluation of conditional expression

Function Name: validateCondition
Input:
(i) i: inters attribute
Output: a Boolean value

```
(1) condValue = i.ldep.dcond$_1$;
(2) for each op$_i$ $(1 \le i \le n-1) \in$ Cond do
(3)    if (op$_i$ = '$\wedge$') then
(4)       condValue:= condValue $\wedge$ i.ldep.dcond$_{i+1}$;
(5)    if (op$_i$ = '$\vee$') then
(6)       condValue:= condValue $\vee$ i.ldep.dcond$_{i+1}$;
(7) return condValue;
```

The mapping function does all possible combinations with the values coming from child intersections tuples, replacing empty values. This process is important to complete instances of XCFDs with values that come up from different parts of the XML tree and is shown in Algorithm 7.

Algorithm 7 – Mapping for intersection values

Function Name: mapping
Input: curInters: set of inters attributes
Output: newInters: set of inters attributes

```
(1) newInters = {};
(2) for each tuple t$_i$ in curInters do
(3)    if ($\varepsilon \notin t_i$.ds $\wedge$ $\varepsilon \notin t_i$.dc $\wedge$ $\varepsilon \notin t_i$.dcond) then
(4)       newInters:= newInters $\cup$ t$_i$;
(5)    else
(6)       for each tuple t$_j$ in curInters (j$\neq$i) do
(7)          newInters:=newInters$\cup$combine(ti,tj);
(8) return newInters
```

When an empty value is found in an intersection tuple, we must try to replace it by looking up into all other intersection tuples that are in the same set for a node. This is shown in line (6) of Algorithm 7. For two intersection tuples, the combination between them is defined in Algorithm 8.

Algorithm 8 – Combination for two tuples with empty values

Function Name: combine
Input:
(i) t_1: inters tuple <lcond,ldep>

```
(ii) $t_2$: inters tuple <lcond,ldep>
Output: setInters: set of inters tuples

(1)  for each field f of an inters tuple do
(2)      if ($t_1$.f = ε) ∧ ($t_2$.f ≠ ε) then
(3)          $t_1$.f := $t_2$.f;
(4)      if ($t_1$.f ≠ ε) ∧ ($t_2$.f = ε) then
(5)          $t_2$.f := $t_1$.f;
(6)  if ($t_1$ = $t_2$)
(7)      setInters = {$t_1$}
(8)  else setInters = {$t_1$, $t_2$}
(9)  return setInters
```

Framework for XML integrity constraints validation with conditions

We propose a general framework to validate integrity constraints for XML data. In this section, the architectural design for this environment is treated. The framework is based on a homogeneous formalism used to express different integrity constraints and it is expected to validate not only traditional and conditional functional dependencies, but also traditional and conditional inclusion dependencies. The software is being developed in Java language and the choice of this technology is justified by its portability between different platforms. The component used to manipulate a set of XML documents is SAXParser and the server used is Tomcat, which allows integrating Java console and web applications. This research aims to develop a software in which non-proprietary APIs are used, that is, open-source components.

Considering that the same formalism is used to define and validate integrity constraints, based on path expressions and is evaluated using FSAs and FSTs, design patterns are very useful to define the software architecture with the purpose of facilitating code reuse and flexibility as discussed in Kuchana (2004) and Gama *et al.* (1994). We use UML diagrams to represent different views of our system model. Figure 6 partially illustrates the Class Diagram outlining the integrity contraints concerned. The dashed lines bypass dependences that are not yet implemented. The *AbstractConstraint* class defines the common characteristics of all constraints and is designed to be a base class for integrity constraints.

Figure 7 demonstrates the application packages organization for XML validation.

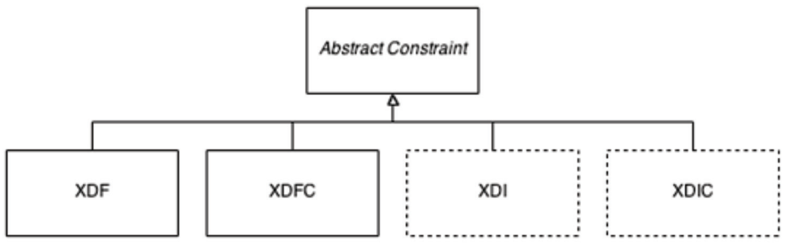

Figure 6. Framework classes. *AbstractConstraint* class its derived classes.

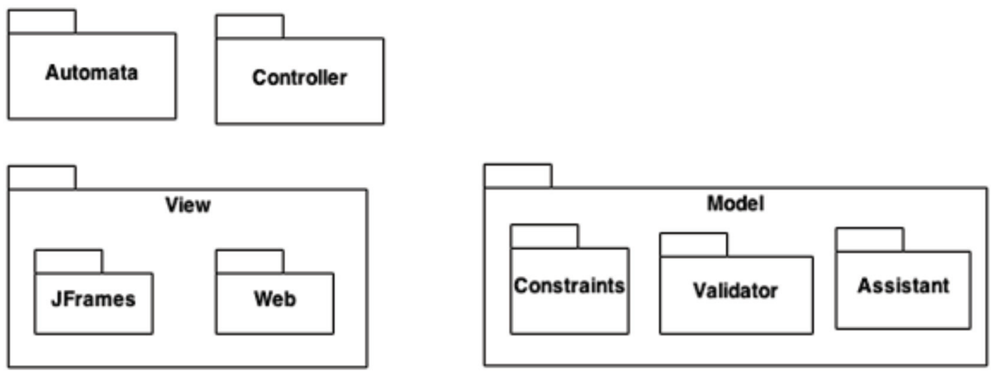

Figure 7. Packages. Overview of the packages organization.

The proposed environment applies the MVC (Model-View-Control) architectural design pattern, which is, in our case, useful to separate data model with validation rules from the user's interface. This approach helps to improve communication among developers, to increase the understanding of contents for each folder and allows better organization of the application.

The *Model* package contains the application object and is divided into three other sub-packages. The *Validator* package is responsible for organizing the constraints validation. *Constraints* package includes all basic features of restrictions. Synthesized and Inherited attributes are defined specifically for each constraint type. *Assistant* package contains helper classes used for building temporary structures, namely stacks, lists and hash tables. The *Automata* package includes all classes that involve the definition of finite automata and their corresponding operations. The *Controller* package contains classes for specifying actions used by Struts 2 to perform validation operations through web interface and to redirect actions. The *View* package implements the system interface and allows the user to interact through the graphical user interface (GUI) or browser user interface (BUI). For usability, the user can insert, edit or remove constraints for validation, and use the XML file sent to the last upload, avoiding the use of unnecessary bandwidth. Thus, it is also possible to view detailed results of the validation, thus creating a standard for the application.

In this project, the behavioral pattern Observer is implemented, where an object (called the subject) maintains a list of its dependents (called observers). These modifications enable the application to become extensible to accommodate new types of integrity constraints that are independent from each other. The Observer pattern is also a key part in Model View Controller (MVC) pattern.

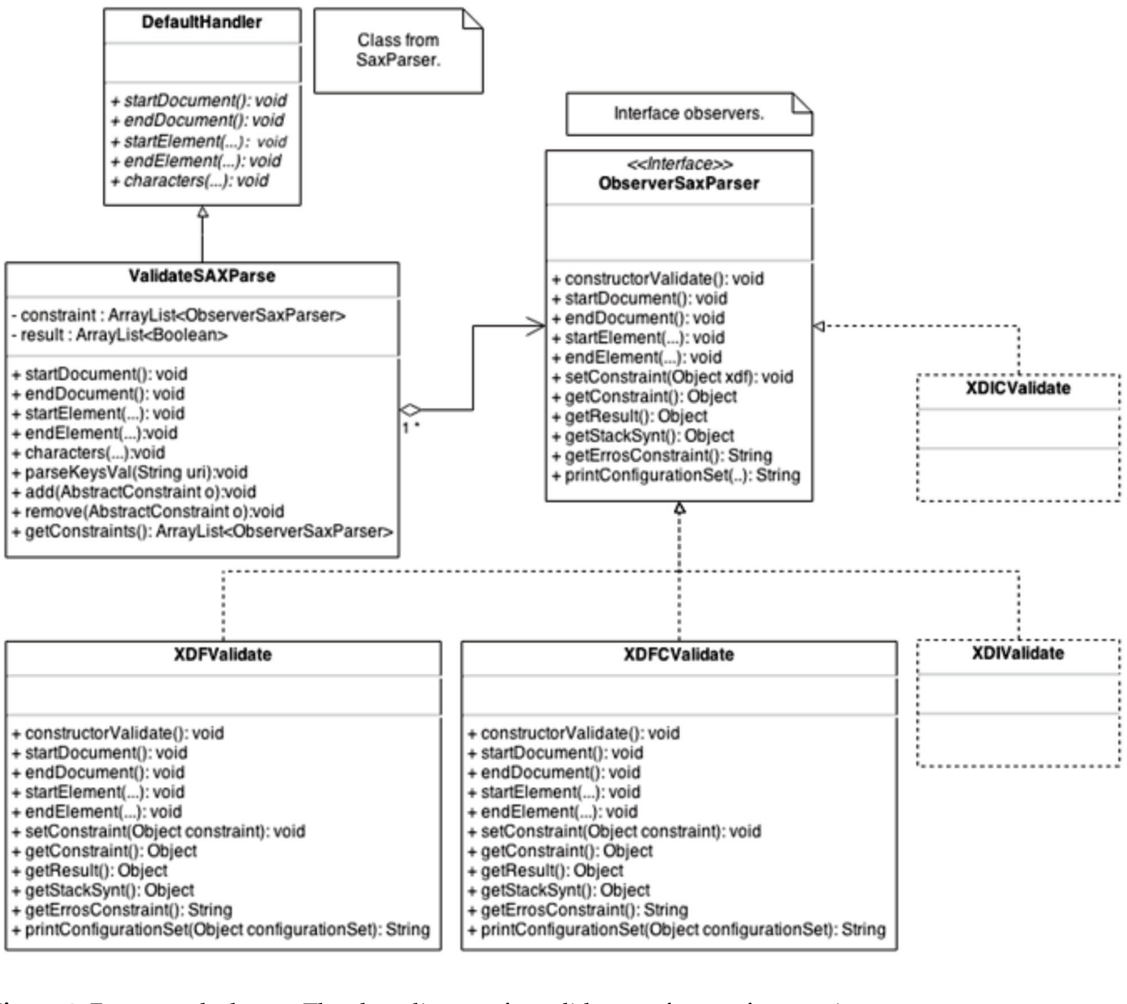
DefaultHandler
+ startDocument(): void
+ endDocument(): void
+ startElement(...): void
+ endElement(...): void
+ characters(...): void
Class from SaxParser.
Interface observers.
<<Interface>>
ObserverSaxParser
+ constructorValidate(): void
+ startDocument(): void
+ endDocument(): void
+ startElement(...): void
+ endElement(...): void
+ setConstraint(Object xdf): void
+ getConstraint(): Object
+ getResult(): Object
+ getStackSynt(): Object
+ getErrosConstraint(): String
+ printConfigurationSet(..): String
ValidateSAXParse
- constraint : ArrayList<ObserverSaxParser>
- result : ArrayList<Boolean>
+ startDocument(): void
+ endDocument(): void
+ startElement(...): void
+ endElement(...):void
+ characters(...):void
+ parseKeysVal(String uri):void
+ add(AbstractConstraint o):void
+ remove(AbstractConstraint o):void
+ getConstraints(): ArrayList<ObserverSaxParser>
1 *
XDICValidate
XDFValidate
+ constructorValidate(): void
+ startDocument(): void
+ endDocument(): void
+ startElement(...): void
+ endElement(...): void
+ setConstraint(Object constraint): void
+ getConstraint(): Object
+ getResult(): Object
+ getStackSynt(): Object
+ getErrosConstraint(): String
+ printConfigurationSet(Object configurationSet): String
XDFCValidate
+ constructorValidate(): void
+ startDocument(): void
+ endDocument(): void
+ startElement(...): void
+ endElement(...): void
+ setConstraint(Object constraint): void
+ getConstraint(): Object
+ getResult(): Object
+ getStackSynt(): Object
+ getErrosConstraint(): String
+ printConfigurationSet(Object configurationSet): String
XDIValidate

Figure 8. Framework classes. The class diagram for validators of a set of constraints.

Figure 8 illustrates a class diagram (from package *Validator*) that models the characteristics of the Observer pattern in this framework. The scenario contains the subject *ValidationSaxParser* and a set of observers. The subject maintains a dynamic list of observers, and they can change their state when receiving notification from the subject.

Conclusions

The research method employed in this work aims the application of basic concepts such as databases and compilers theory to create a set of algorithms to validate Conditional Functional Dependencies in XML documents. It also uses an analytical approach that intends to explain how XCFDs validation can be performed. The materials used here involved XML SAXParser, Tomcat server, and computer science theoretical concepts like functional dependencies, attribute grammar and design patterns such as Observer. With these tools we were able to define algorithms capable to validate any well-defined Conditional Functional Dependency in an XML document, as presented in this article.

An attribute grammar can be defined for the validation of integrity constraints over XML data by performing various annotations and calculations that are associated to the tree nodes during one tree traversal in an XML document. The main advantage of this proposal is to be based on a generic method founded on finite automata and grammar attributes, which can be adapted to the validation of other types of restriction. The validation of conditional functional dependencies allows the quality of the XML data to be analyzed from specific conditions imposed on the data (semantic imposition) which can be quite useful in data integration environments.

The use of the behavioral pattern Observer allows our framework to have a low coupling between the validation classes. Thus, each class is enabled to perform its validation process without interference from other types of constraints validation, once for every event occurring at tree traversal each class is individually notified and performs its own response action. Furthermore, this approach gives the opportunity to extend the system to other types of constraints validation and contributes to the identification and management of inconsistent information.

As a continuation of this work, a module that provides the correction of possible inconsistencies that are raised during the validation process is proposed. Those corrections might consider data semantics, error type, and can be formally defined with insertion, exclusion and substitution operations over branches and values. Another future improvement in this work is the processing of integrity constraints validation over XML collections stored in distributed storage, using a map-reduce framework.

References

AHO, A.V.; SETHI, R.; ULLMAN, J.D. 1988. *Compilers: principles, techniques, and tools.* Massachusetts, Addison-Wesley, 796 p.

ARENAS, M.; LIBKIN, L. 2004. A normal form for XML documents. *ACM Transactions on Database Systems*, **29**(1):195-232.
http://dx.doi.org/10.1145/974750.974757

BAKHTOUCHI, A.; BELLATRECHE, L.; AIT--AMEUR, Y. 2011. Ontologies and functional dependencies for data integration and reconciliation. *In:* International Conference on Advances in Conceptual Modeling: Recent Developments and New Directions, 30th, Brussels, 2011. *Proceedings…* Brussels, **6999**:98-107.
http://dx.doi.org/10.1007/978-3-642-24574-9_13

BOHANNON, P.; FAN, W.; GEERTS, F.; JIA, X.; KEMENTSIETSIDIS, A. 2007. Conditional functional dependencies for data cleaning. *In:* International Conference on Data Engineering, 23rd, Istanbul, 2007. *Proceedings…* IEEE, p. 756-755. http://doi.ieeecomputersociety.org/10.1109/ICDE.2007.367920

BOUCHOU, B.; CHERIAT, A.; HALFELD-FERRARI, M.; LAURENT, D.; LIMA, M.A.V.; MUSICANTE, M. 2007. Efficient constraint validation for updated XML databases. *Informatica,* **31**(3):285–310.

BOUCHOU, B.; HALFELD-FERRARI, M.; LIMA, M.A.V. 2011. Attribute Grammar for XML Integrity Constraint Validation. *In:* Database and Expert Systems Applications, 22nd, Toulouse, 2011. *Proceedings…* Toulouse, **6860**:94-109. http://dx.doi.org/10.1007/978-3-642-23088-2_7

BOUCHOU, B.; HALFELD-FERRARI, M.; LIMA, M.A.V. 2012. A Grammarware for the Incremental Validation of Integrity Constraints on XML Documents under Multiple Updates. *Transactions on Large-Scale Data and Knowledge-Centered Systems*, **6**:167-197.
http://dx.doi.org/10.1007/978-3-642-34179-3_6

BUNEMAN, P.; FAN, W.; SIMEON, J.; WEINSTEIN, S. 2001. Constraints for Semistructured Data and XML. *ACM Special Interest Group on Management of Data*, **30**(1):47-54.
http://doi.acm.org/10.1145/373626.373697

DEUTSCH, A.; TANNEN, V. 2005. XML Queries and Constraints, Containment and Reformulation. *Theoretical Computer Science*, **336**(1):57-87. http://dx.doi.org/10.1016/j.tcs.2004.10.032

FAN, W. 2008. Dependencies revisited for improving data quality. *In:* ACM SIGMOD-SIGACT--SIGART Symposium on Principles of database systems, 27th, Vancouver, 2008. *Proceedings…* ACM, p. 159-170.

http://doi.acm.org/10.1145/1376916.1376940

FAN, W.; GEERTS, F.; JIA, X.; KEMENTSIETSIDIS, A. 2008. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems*, **33**(2):1-48.

http://doi.acm.org/10.1145/1366102.1366103

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. 1994. *Elements of Reusable Object-Oriented Software*. Delhi, Pearson Education, 395 p.

GIRE. F.; IDABAL, H. 2010. Regular tree patterns: a uniform formalism for update queries and functional dependencies in XML. *In:* EDBT/ICDT Workshops, Lausanne, 2010. *Proceedings…* Lausanne, p. 18-1, 18-9.

http://doi.acm.org/10.1145/1754239.1754260

HARTMANN, S.; TRINH, T. 2006. Axiomatising functional dependencies for XML with frequencies. *In*: International Symposium FoIKS, 4th, Budapest, 2006. *Proceedings…* Budapest, **3861**:159-178.

KARLINGER, M.; VINCENT, M.; SCHREFL, M. 2009. Inclusion Dependencies in XML: Extending Relational Semantics. *In:* International Conference on Database and Expert Systems Applications, 20th, Linz, 2009. *Proceedings…* Linz, **5690**: 23-37. http://dx.doi.org/10.1007/978-3-642-03573-9_3

KUCHANA, P. 2004. *Software Architecture Design Patterns in Java*. Boston, Auerbach Publishers, 492 p. http://dx.doi.org/10.1201/9780203496213

LIU, J.; LI, J.; LIU, C.; CHEN, Y. 2011. Discover dependencies from data: a review. *IEEE Transactions on Knowledge and Data Engineering*, **24**(2):251-264.

http://dx.doi.org/10.1109/TKDE.2010.197

LIU, J.; VINCENT, M.; LIU, C. 2003. Functional Dependencies, From Relational to XML. *Lecture Notes in Computer Science*, **2890**:531-538.

http://dx.doi.org/10.1007/978-3-540-39866-0_51

MA, S.; FAN, W.; BRAVO, L. 2014, Extending inclusion dependencies with conditions. *Theoretical Computer Science*, **515**:64-95.

http://dx.doi.org/10.1016/j.tcs.2013.11.002

SHAHRIAR, M.S.; LIU, J. 2008. Preserving Functional Dependency in XML Data Transformation. Advances in Databases and Information Systems. *In:* East European Conference, ADBIS 2008, 12th, Pori, 2008. *Proceedings…* Lecture Notes in Computer Science, **5207:**262-278.

http://dx.doi.org/10.1007/978-3-540-85713-6_19

SHAHRIAR, M.S.; LIU, J. 2009. On the performances of checking XML key and functional dependency satisfactions. *In*: Confederated International Conferences, Vilamoura, 2009. *Proceedings...* On the Move to Meaningful Internet Systems: OTM, p. 1254-1271. Available at: http://link.springer.com/chapter/10.1007%2F978-3-642-05151-7_37. Accessed on: December 2nd, 2013.

TAN, Z.; ZHANG, L. 2011. Improving XML data quality with functional dependencies. *In:* International Conference on Database Systems for Advanced Applications, 16th, Hong Kong, 2011. *Proceedings…* Hong Kong, p. 450-465. Available at: http://dl.acm.org/citation.cfm?id=1997348. Accessed on: December 3th, 2013.

TRINH, T. 2008. Using Transversals for Discovering XML Functional Dependencies. *In:* International Symposium FoIKS, 5th, Pisa, 2008. *Proceedings…* Pisa, p. 199-218.

http://dx.doi.org/10.1007/978-3-540-77684-0_15

VINCENT, M.; LIU, J. 2005. Checking functional dependency satisfaction in XML. *In:* International XML Database Symposium (XSym05), Trondheim, 2005. *Proceedings…* Trondheim, p. 4–17. http://dx.doi.org/10.1007/11547273_2

VO, L.T.H.; CAO, J.; RAHAYU, W. 2011. Discovering Conditional Functional Dependencies in XML Data. *In:* Australasian Database Conference, 22nd, Perth, 2011. *Proceedings…* Perth, p. 143-152. Available at: http://crpit.com/confpapers/CRPITV115Vo.pdf. Accessed on: December 2nd, 2013.

WANG, J.; TOPOR, R. 2005. Removing XML data redundancies using functional and equality-generating dependencies. *In:* Australasian Database Conference, 16th, NewCastle, 2005. *Proceedings…* NewCastle, p. 65-74. Available at: http://crpit.com/confpapers/CRPITV39Wang.pdf. Accessed on: December 2nd, 2013.

Submitted on January 6, 2014
Accepted on April 29, 2014