Computing and Informatics, Vol. 31, 2012, 573–595

SEARCH-BASED EVOLUTION OF XML SCHEMAS

Julio Cesar Teodoro Silva, Aurora Trinidad Ramirez Pozo
Silvia Regina Vergilio

Department of Informatics
Federal University of Paraná
CP: 19081, CEP: 81531-970 Curitiba – PR, Brazil
e-mail: julioteodoro@gmail.com, {aurora, silvia}@inf.ufpr.br

Martin A. Musicante

Department of Informatics and Applied Mathematics
Federal University of Rio Grande do Norte
CP: 1647, CEP: 59000-000 Natal – RN, Brazil
e-mail: mam@dimap.ufrn.br

Communicated by Jacek Kitowski

Abstract. The use of schemas makes an XML-based application more reliable, since they contribute to avoid failures by defining the specific format for the data that the application manipulates. In practice, when an application evolves, new requirements for the data may be established, raising the need of schema evolution. In some cases the generation of a schema is necessary, if such schema does not exist. To reduce maintenance and reengineering costs, automatic evolution of schemas is very desirable. However, there are no algorithms to satisfactorily solve the problem. To help in this task, this paper introduces a search-based approach that explores the correspondence between schemas and context-free grammars. The approach is supported by a tool, named EXS. Our tool implements algorithms of grammatical inference based on LL(1) Parsing. If a grammar (that corresponds to a schema) is given and a new word (XML document) is provided, the EXS system infers the new grammar that: i) continues to generate the same words as before and ii) generates the new word, by modifying the original grammar. If no initial grammar is available, EXS is also capable of generating a grammar from scratch from a set of samples.

Keywords: XML-based applications; DTD; LL parsing

Mathematics Subject Classification 2010: 68N30

1 INTRODUCTION

XML (eXtensible Markup Language) has become one of the most popular and widely used technologies for storage, manipulation, and transference of data. This is due to its very simple and strict formation rules, which allow XML data to be easily defined and verified with respect to a given schema [36]. Schemas for XML are languages for the definition of specific formats of XML documents. They define a set of (typing) rules according to which it is legal to create valid XML documents to be manipulated by a XML-based application. Examples of schema languages are DTD (Document Type Definition) [13] and XML Schema [35].

With the advent of Internet and the increasing demand for web applications, schema languages and XML-based applications have gained importance. These types of applications present some special requirements, among which faster maintenance is considered crucial [33]. The technologies related to XML-based applications evolve more rapidly than traditional application technologies. Maintenance of these applications is very frequent and evolution needs to be done more efficiently.

When the applications evolve, the schemas may need to be modified in order to match the new requirements. Modifications are usually application-domain dependent and they are frequently promoted by data administrators whose domain of expertise is not computer science. Moreover, schema extensions must usually be conservative: the new version of the schema must be able to continue validating previous versions of the XML data. Because of this, it is important to accomplish this maintenance task automatically. The automatic evolution of schemas contributes to reduce time and failures in the development and maintenance of XML applications, decreasing costs and working effort.

Another point considered by some authors [18, 32] is that many XML applications and databases are schema-less. However, an implicit format for the XML documents usually exists and many times it is desired to construct the schema to validate the data and, consequently, make the application more reliable. Support for the generation of such schemas is also fundamental to maintenance and reengineering tasks.

Several techniques have been used to generate and/or evolve a schema for XML data. Bouchou et al. [8] propose an algorithm, GREC, to dynamically evolve XML types by means of inducing the regular expressions contained in DTD documents. In that work, only one modification can be made to the schema per run of the algorithm. This is a very restrictive limitation: it is interesting to devise methods to allow several modifications to the schema per run. Some of these disadvantages were addressed in a new version, called dGREC [12]; however, that work concentrates

its efforts on the generation of regular expressions appearing on the schema. Bex et al. [7] used another approach to build regular expressions inside the schema rules. They propose a probabilistic algorithm that generates a regular expression from a set of positive examples. None of the works above consider the evolution of the whole set of rules for the schema.

To overcome these limitations, in the present work, we explore the use of a search-based approach in the context of Grammar Inference (GI) to generate schemas from scratch.

In the literature, there are some related works for context-free grammar inference. Examples of these works are: a strategy proposed in [26] to induce grammars by using Genetic Programming (GP) [22] and, the algorithm ICYK proposed in [27] and [28], an inductive algorithm based in CYK [21].

However, neither the GP approach nor the ICYK algorithm, nor other works on GI, were designed to work with schemas for XML. This means that they are not concerned with keeping the structure of the induced grammars to be as similar as possible to the original schema. This is a fundamental issue for us, since the new version of the schema after evolution should be similar in structure to the original one, in order to preserve readability and familiarity with the data administrator. For example, the GP algorithm generates and changes the grammars randomly. To keep the proximity between grammars (schemas), to manipulate grammars randomly is not interesting. The ICYK approach is also not suitable for XML, since it uses a very restrictive normal form to represent grammars. This normal form is very far from the usual representation of schemas.

In spite of this, to explore the main characteristics of both proposals in the context of XML is very promising, and this is the objective of our work.

This work introduces a search-based tool, named EXS, for XML Schema evolution. This tool was implemented considering the advantages of the GP and ICYK approaches and implements two algorithms (previously described in [31]): ILLA (Inductive LL Algorithm) and SILLA (Synthesis with ILLA). The implemented algorithms are based on the LL(1) Parsing algorithm [1]. LL is an alternative to CYK and presents some desired characteristics in the context of schemas for XML. Our algorithms use some structures (a table, and a stack) to control the parsing and it works with production rules very similar to those of schemas for XML, with different sizes, and of any length. Furthermore, the complexity of LL is $O(n)$, while CYK is $O(n^3)$, so the use of LL instead of CYK is preferable.

EXS facilitates the evolution of schemas for XML by evolving context-free grammars. If a grammar (schema) is given and a new desired word (XML document) is provided, the EXS system infers the new grammars; that keeps recognizing the same words (documents) as before, as well as the new word. If no initial grammar is available, EXS is also capable to generate a grammar from scratch, for a set of samples.

The paper is organized as follows. Section 2 describes related works. Section 3 introduces the architecture of EXS and describes its main modules and algorithms. Section 4 contains some evaluation results obtained using EXS. Section 5 discusses

the applicability and scalability of our algorithms. Section 6 presents our conclusions and future works.

2 RELATED WORK

XML emerged as a standard for data representation and exchange on the Web, therefore devising schema evolution algorithms is of capital importance to the life-cycle of applications that use schemas. Research in this field is beginning. XML Schema evolution is not an easy task.

The problem of extracting a schema from semi-structured data has been addressed, for instance in [7, 10, 19, 29]. However, the schema extracted by these early works attempt to find a typing for semi-structured data assuming a graph-based model. Since the schemas in semi-structured databases are expressed using plain sequences or sets of edges, they cannot be used to infer, for example, DTDs containing arbitrary regular expressions.

On the other side, more related to this work is the inference of formal languages from examples with the extensive study of the inference of Deterministic Finite Automata (DFA) [4, 16, 17]. The above line of work is purely theoretical and it focuses on investigating the computational complexity of the language inference problem. Following this line of research, the works described in [2, 3] propose an approach for automatically generating context-free grammars from structured text documents. Their method essentially produces a union finite-state automaton for all example documents and then simplifies/generalizes that automaton and the corresponding regular expression by merging states. A potential problem with this approach is that the resulting regular expressions may need to be manipulated further in order to produce meaningful structural descriptions.

A system, named XTRACT, was proposed in [15] to infer a DTD schema for a database of XML documents by learning regular expressions. XTRACT inference algorithms employ a sequence of steps that involve:

1. finding patterns in the input sequences and replacing them with regular expressions to generate general candidate DTDs,
2. factoring candidate DTDs using adaptations of algorithms from the logic optimization literature, and
3. applying the Minimum Description Length (MDL) principle to find the best DTD among the candidates.

XTRACT manipulates regular expressions and not automata graphs. These groups of works focus to infer a DTD for a collection of XML documents and once the DTD is learned no evolution can be made.

Other group of works focus on schema evolution. Bouchou et al. [8] proposes an algorithm, named GREC. Regular expressions of a DTD document are transformed in automata and then, the automata are transformed until the new document is recognized. At the end of the process, the automata are converted in the new

regular expression. The GREC algorithm is an extension of the reduction process proposed in [9] to transform a Glushkov automaton into a regular expression. Some of the disadvantages of GREC were treated in a new version, called dGREC. dGREC manipulates regular expressions and not automata graphs [12]. This group of works focuses on the evolution of an already known DTD; however, an XML document may not always have an accompanying DTD. In fact, several papers (e.g., [18, 32]) claim that it is frequently possible that only specific portions of XML databases will have associated DTDs, while the overall database is still schema-less.

A novel approach is introduced here, where both important tasks would be included, namely:

1. learning from scratch when no DTD is provided and
2. the incremental learning of a DTD when a new document that violates the restrictions comes.

To do this, in this paper we propose the study of the new field of grammar inference. It can be made by exploring the correspondence between schemas and grammars.

Grammatical Inference (GI) (or induction) is a process of learning a grammar from a set of training data. The most traditional field of GI is pattern recognition and other areas such as gene analysis, sequence prediction, cryptography and information retrieval [6, 14, 30].

Two works on GI mentioned before are the basis of our study. A first similar work introduced an inductive algorithm based on CYK parsing algorithm [21], described in [27] and [28]. This algorithm, named Inductive CYK (ICYK), is a component of an inductive grammar inference system called Synapse. ICYK adds new production rules to the grammar when it does not derive a positive sample.

At first, the system has no production rules. For a given positive sample string, it generates minimum production rules to derive this string. Then it checks that the rules do not derive any given negative samples. This process continues until the system finds a rule set which derives all the positive samples and none of the negative samples. For generating production rules, the system uses Inductive CYK algorithm, which generates sets of rules required for parsing positive samples. The inductive inference is based on incremental search, or iterative deepening, in the sense that the rule sets are searched within the limits of the minimum numbers of non-terminal symbols and rules. When the search fails, the system iterates the search with larger limits. This system is able to learn from scratch and to incrementally learn a new grammar.

Other works use Evolutionary Computation (EC) techniques [5] to induce context-free grammars. EC techniques have been successfully applied in many areas related to software development, for example test data generation [24, 25], software evolution [20], scheduling [34], etc.

In the GI area, most works are based on Genetic Programming [22]. In the work reported in [26], the grammars are represented as programs constructed by sets of terminal and non-terminal symbols. The positive and negative samples are

used to calculate the fitness function of the algorithm. Beside the basic operators of GP, heuristic operators are proposed in [26]. Also a better initial population is created instead of random generation. As noticed in [26], this algorithm infers only grammars that have a small number of productions rules.

In [23], another Genetic Programming-based method is presented. It induces a classification mechanism for positive and negatives samples of a language. The individuals are represented by finite-state automata, and the fitness function is defined according to the classification of a training set. GP approaches are not incremental.

Works on GI do not address evolution of schemas. Neither the GP approaches nor the ICYK algorithm were designed to work with schemas for XML. They do not worry about keeping the structure of the induced grammars similar to a schema. The advantage of ICYK is to implement a control and structures to manipulate the grammars during the search. Its advantage is to work only with grammars in a specific normal form that is not similar to schemas. Beside this, the asymptotic time complexity of the parsing algorithm CYK is $O(n^3)$. The advantage of the GP algorithm is to allow induction from positive and negative samples.

The tool described in this paper combines the main ideas and advantages of the above-mentioned works and present characteristics that make them suitable to the context of XML schema evolution. The algorithms implemented by EXS use LL to control the parsing and it works with production rules very similar to schemas, with different sizes, and of any length. Further, the complexity of LL is $O(n)$. EXS allows the evolution of a schema given a new XML document, as well as learning from scratch when no schema is provided.

3 THE EXS TOOL

In this section we describe the EXS tool. Its architecture is given in Figure 1. The tool uses a *Configuration file* in a pre-defined format. This file can be created either by the EXS *Interface* or directly by the user. In this initial version only DTD schemas are evolved by EXS. The tool contains two main modules – *ILLA* and *SILLA* that are controlled by the *Controller* module. *Controller* interprets the *Configuration File* and according to the information given by the user, one of the two modules is called providing the necessary data.

The *ILLA* and *SILLA* modules implement the algorithms ILLA (Inductive LL Algorithm) and SILLA (Synthesis with ILLA) that are based on LL parsing algorithm. *ILLA* is responsible for evolution of a schema and *SILLA* calls *ILLA* for generation of schemas from scratch.

The algorithms implemented by EXS, as well as the configuration file, are detailed next by using a simple XML example shown in Figure 2. The DTD is related to an application that manipulates contact information of clients, such as phone numbers.

Let us now consider now the XML documents of Figures 3 and 4. Notice that the schema of Figure 2 only validates documents of Figure 3. However, suppose that

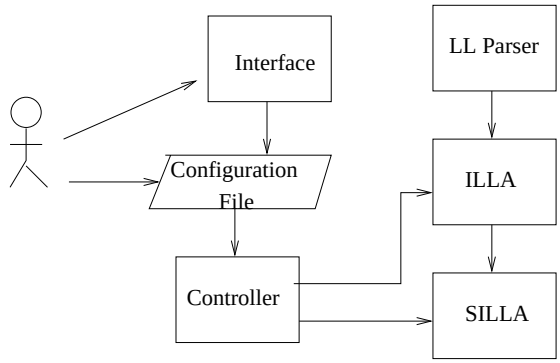

Fig. 1. EXS architecture

```
<?xml version="1.0"?>
<!DOCTYPE contact [
  <!ELEMENT contact (name,email*,telephone)>
  <!ELEMENT name    (#PCDATA)>
  <!ELEMENT email    (#PCDATA)>
  <!ELEMENT telephone  (homephone,celphone)>
  <!ELEMENT homephone  (#PCDATA)>
  <!ELEMENT cellphone  (#PCDATA)>
]>
```

Fig. 2. A schema for contact information

a new kind of phone is now necessary for the application, for instance, *officephone*. So, it is necessary to evolve the given DTD to allow the validation of the document presented in Figure 4.

```
<contact>
  <name>John Winston</name>
  <email>johnwinston@wch.com</email>
  <telephone>
    <homephone>555-5555</homephone>
    <celphone>999-9999</celphone>
  </telephone>
</contact>
```

Fig. 3. A valid document

```
<contact>
  <name>John Winston</name>
  <email>johnwinston@wch.com</email>
  <telephone>
    <homephone>555-5555</homephone>
    <celphone>999-9999</celphone>
    <officephone>123-4567<officephone>
  </telephone>
</contact>
```

Fig. 4. A positive example

3.1 Configuration File

The Configuration File contains information necessary to call the EXS algorithms. This information includes:

- the name of a file that contains the initial grammar to be evolved. If such name is not provided, SILLA is called.
- name of a file that will contain the result. At the end the first solution found by ILLA is written in this file, or according to an option given by the user, the best or all solutions found by SILLA.
- set of positive and negative examples. Negative examples are not necessary for ILLA (they are optional).
- other information to control the evolution process, such as the maximum number of recursive calls (or maximum depth tree), etc.

Suppose an use of EXS to evolve the schema of our example. In this case, ILLA is called. The tester needs to provide: 1) a file containing the LL(1) grammar of Figure 5 obtained from the DTD of Figure 2; and 2) as positive example w obtained from the new XML document to be validated (Figure 4). In the grammar a number was associated to each rule to ease referencing to next sections.

3.2 LL

LL is a predictive, non recursive parsing algorithm described in [1]. It uses a parsing table and a stack to control the grammatical derivation as can be seen in Figure 6.

The parsing table is a bi-dimensional table, of non-terminal by terminal symbols. Each cell [`X`, `a`] keeps which production rule must be consumed when `X` is on the top of the control stack and `a` is the next input symbol.

The parsing table is constructed with the aid of two functions: First and Follow. Given a context-free grammar, the function First receives a sequence of grammar symbols as its argument. It returns the set of terminal symbols that can start strings derived from that sequence. The function Follow receives a non-terminal symbol and

```
1) contact   ::= name.email.telephone
2) name      ::= #PCDATA
3) email     ::= #PCDATA.email
4) email     ::= e
5) telephone ::= homephone.celphone
6) homephone ::= #PCDATA
7) cellphone ::= #PCDATA
```

Fig. 5. Grammar translated from Figure 2

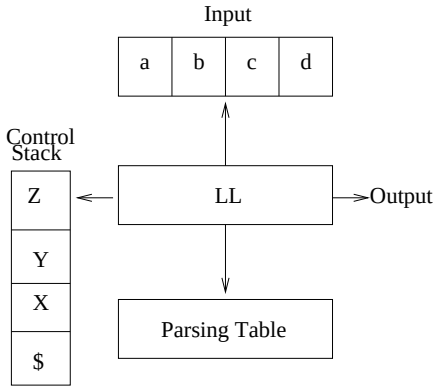

Fig. 6. LL Parser

returns the set of terminal symbols that can appear immediately to the right of that non-terminal symbol in any derivation of the grammar. The LL(1) parsing table for the grammar of our example is in Table 1, where the number of each rule is used.

	#PCDATA	#PCDATA	#PCDATA	#PCDATA	$
contact	1				
name	2				
Email		3	4		
telephone				5	
homephone				6	
celphone					7

Table 1. LL(1) parsing table

3.3 ILLA

ILLA is an iterative-deepening search algorithm. The procedure starts when ILLA receives a grammar and a word (DTD and document). ILLA calls the LL Parser to validate the word, if the LL Parser detects that the example being parsed is not

derived by the grammar, ILLA creates new production rules based on the parsing table used in the derivation. The system controls the search by iterative deepening on the number of rules to be generated. First, the number of the rules in the initial set of rules is assigned to the limit K of the number of initial rules. When the system fails to generate enough rules to parse the samples within this limit, it increases the limit by one and iterates the search. By this control, it is assured that the procedure finds a grammar with the minimum number of rules at the expense that the system repeats the same search each time the limit is increased. In general, iterative deepening is the preferred search method when there is a large search space and the depth of the solution is not known.

Algorithm 1 shows the main function of ILLA. The appendix presents the graph of functions called by ILLA (Figure 9), as well as all the corresponding pseudocodes.

ILLA receives a LL(1) grammar G, a string w and an integer n. It verifies whether G derives w. If not, it calls the function Test_TS, which adds to G production rules of a set of options TS, created by the function Create_TS_Set, and verifies whether the resulting grammar derives w. The parameter n is the maximum depth allowed to the search tree.

The function Test_TS adds an element of TS to G, and recursively calls the main function ILLA, with the adjusted parameters. The function Create_TS_Set estimates, according to the error occurred in the parsing, how the production rules must be created, in the sense of filling correctly the parsing table of LL. Given a combination of parameters, the function Create_TS_Set can call the functions Create_Optional_Terminal and Create_Options_From.

The function Create_Optional_Terminal is called when the symbol on the top of P is a terminal symbol. This function modifies all the production rules of the grammar where the terminal symbol `a` appears, by rendering `a` optional.

Algorithm 1 ILLA

1: Input: grammar G, string w, integer n;
2: Try to derive w using LL Parser;
3: If it is well-succeeded then $NG := G$;
4: Else
5: If $n <= nMax$, then
6: { TS := Create_TS_Set(G, P, a),
7: where P is the stack used in LL, and
8: a is the terminal symbol of w
9: that was being read when the parsing failed;
10: NG := Test_TS(G, w, TS, n, $nMax$);}
11: Output: returns a grammar NG;

The new test set (TS) is formed by substituting the production rule '$X ::= aa\beta$' by a pair of rules, to make the symbol 'a' optional. This rule replacement is conservative, in the sense that all strings that can be generated using the old rule will be

generated using the new ones. The function Create_Options_From creates new production rules for the grammar when the error on the parsing process occurs because the cell $[X, a]$ in parsing table is empty. The options are generated according to the properties of context-free grammars, aiming to fill correctly the parsing table. ILLA only generates new non-terminal symbols when necessary, to avoid left recursion or ambiguity.

To illustrate the use of ILLA, consider our XML example again. When the LL(1) algorithm tries to read the cell [officephone,#PCDATA] of Table 1, LL fails because this cell does not exist. That means grammar G does not derive w. Then ILLA has the task of adding production rules to G until w is derived. When the parsing fails, ILLA calls the function Create_TS_Set, which creates the set TS below.

```
TS[1] = [cellphone ::= #PCDATA]
TS[2] = [telephone ::= homephone.celphone.officephone,
                officephone ::= #PCDATA, officephone ::= e]
```

ILLA randomly chooses an element of TS and adds its production rules to G. For demonstration, it is chosen the element $TS[2]$. The resulting grammar G' is shown in Figure 7.

```
1) contact   ::= name.email.telephone
2) name      ::= #PCDATA
3) email     ::= #PCDATA.email
4) email     ::= e
5) telephone ::= homephone.celphone.officephone
6) homephone ::= #PCDATA
7) cellphone ::= #PCDATA
8) officephone ::= #PCDATA
9) officephone ::= e
```

Fig. 7. Obtained G' grammar

ILLA is called recursively with G', w, and the increased iterator as input parameters. The parsing table of the second calling to ILLA is shown in Table 2.

When ILLA tries to parse w in grammar G', the parsing is successful. Then, G' is returned as the inferred grammar.

A DTD document that can be generated from the inferred grammar is shown in Figure 8.

3.4 SILLA

For the purpose of inferring a grammar only from sets of samples, we propose an extension to ILLA, an algorithm called SILLA. It generates an initial population of grammars based on the structure of the positive samples. Then, it uses ILLA to infer grammars for each positive sample.

	#PCDATA	#PCDATA	#PCDATA	#PCDATA	#PCDATA	$
contact	1					
name	2					
Email		3	4			
telephone				5		
homephone				6		
celphone					7	
officephone					8	9

Table 2. Parsing table to the second calling of ILLA

```
<?xml version="1.0"?>
<!DOCTYPE contact [
  <!ELEMENT contact (name,email*,telephone)>
  <!ELEMENT name        (#PCDATA)>
  <!ELEMENT email       (#PCDATA)>
  <!ELEMENT telephone (homephone,celphone,officephone?)>
  <!ELEMENT homephone  (#PCDATA)>
  <!ELEMENT cellphone  (#PCDATA)>
  <!ELEMENT officephone  (#PCDATA)>
]>
```

Fig. 8. DTD correspondent to G'

SILLA uses concepts of the Evolutionary Computation (EC) [5], like population, fitness, and selection of individuals. However, it does not use gene combination and mutation, because this would result in losing the LL properties of the synthesized grammars. SILLA gets as input parameters sets of positive and negative samples, and returns the grammar of the final generation with best fitness. The used fitness function is as follows:

$$F(x) = \frac{\frac{correct\_positive\_samples}{all\_positive\_samples} + \frac{correct\_negative\_samples}{all\_negative\_samples}}{2}. \quad (1)$$

SILLA creates the initial population, evaluates the fitness of every individual, and verifies if there is a complete solution (a grammar which classifies correctly all the samples). If not, it starts the evolutionary process:

- For each positive sample, and for each grammar of the population, SILLA calls ILLA, and keeps the resulted grammars in a list of candidates to the next generation.
- The candidates in the list are evaluated and, if there is no complete solution, the selection of individuals is performed, resulting in the new generation. So, the process restarts.
- SILLA ends when a complete solution is found, or when the maximum number of generations is reached.

The main function of SILLA is presented in Algorithm 2. It controls the evolution of the grammars, by means of calling the functions Create_Initial_Population, Evaluate_Fitness, Select_Next_Population, and ILLA. Figure 10 shows the sequence of calls for SILLA. The called functions are described in the appendix.

The function Create_Initial_Population creates grammars based on positive samples in two ways: with large number of production rules, in Chomsky Normal Form and grammars that, seen as trees, have the leaf nodes in the deepest level.

Algorithm 2 SILLA.

```
1:  SILLA(set of positive samples P, set of negative samples N, integer nMax): returns
    a grammar;
2:    Let Population and Candidates be sets of grammars;
3:    Population := Create_Initial_Population(P);
4:    Let i := 1;
5:    Loop
6:      If Population includes a complete solution, then
7:        Return the complete solution;
8:      If i > nMax, then
9:        Return the grammar in Population with best fitness;
10:     Else,
11:       Population := Evaluate_Fitness(Population, P, N);
12:       For each sample w ∈ P, do
13:         For each grammar G ∈ Population, do
14:           Candidates := Candidates ∪ ILLA(G, w, n, nMax);
15:           Evaluate_Fitness(Candidates, P, N);
16:           Population:= Select_Next_Population (Population, Candidates);
17:     i++;
18:   End;
```

Notice that when SILLA is used in the context of schema for XML evolution, the initial population is given only by the original schema. The function Create_Initial_Population is not called.

The function Evaluate_Fitness updates the set of grammars received, calculating the parameters fitness, correct_positive, and correct_negative of every grammar.

The function Select_Next_Population fills the new population according to the parameters fitness (*correct_positive*, *correct_negative*) and their respective weights (previously configured) F_Weight, P_Weight, and N_Weight.

4 EMPIRICAL RESULTS

This section presents some results obtained by using EXS. We describe an experiment for context-free grammar inference and discuss some characteristics of the inferred grammars, as well as EXS limitations.

An experiment was made to evaluate the use of EXS for GI. To do this, we repeated the experiment described in [23]. In that work, the authors used a benchmark

named "Tomita Language Set" (TLS). Table 3 shows seven languages extracted from TLS and used in the experiment.

	Positive Samples	Negative Samples
2	ϵ, 10, 1010, 101010, 10101010, 10101010101010	1, 0, 11, 00, 01, 101, 100, 1001010, 10110, 110101010
3	ϵ, 1, 0, 01, 00, 11, 00, 100, 110, 111, 000, 100100, 110000011100001, 111101100010011100	10, 101, 010, 1010, 1110, 10001, 111010, 1001000, 11111000, 0111001101, 1011, 11011100110
4	ϵ, 1, 0, 10, 01, 00, 100100, 001111110100, 0100100100, 11100, 010	000, 11000, 0001, 000000000, 00000, 11111000011,10111101111, 110101000001011, 11001
5	ϵ, 11, 00, 1001, 0101, 1010, 1000111101, 111111, 0000, 1001100001111010	1, 0, 111, 010, 00000000, 1000, 01, 10, 011, 1110010100, 010111111110, 0001
6	ϵ, 10, 01, 1100, 101010, 111, 000000, 10111, 0111101111, 100100100	1, 0, 11, 00, 101, 011, 00000000, 010111, 10111101111, 11001, 1001001001, 1111
1	ϵ, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111	0, 10, 01, 00, 011, 110, 11111110, 10111111
2	ϵ, 10, 1010, 101010, 10101010, 10101010101010	1, 0, 11, 00, 01, 101, 100, 1001010,10110, 110101010
3	ϵ, 1, 0, 01, 00, 11, 00, 100, 110, 111, 000, 100100, 110000011100001, 111101100010011100	10, 101, 010, 1010, 1110, 10001, 111010, 1001000, 11111000, 0111001101, 1011, 11011100110
4	ϵ, 1, 0, 10, 01, 00, 100100, 001111110100, 0100100100, 11100, 010	000, 11000, 0001, 000000000, 00000, 11111000011,10111101111, 110101000001011, 11001
5	ϵ, 11, 00, 1001, 0101, 1010, 1000111101, 111111, 0000, 1001100001111010	1, 0, 111, 010, 00000000, 1000, 01, 10, 011, 1110010100, 010111111110, 0001
6	ϵ, 10, 01, 1100, 101010, 111, 000000, 10111, 0111101111, 100100100	1, 0, 11, 00, 101, 011, 00000000, 010111, 10111101111, 11001, 1001001001, 1111
7	ϵ, 1, 0, 10, 01, 11111, 111, 00110011, 0101, 0000100001111, 00, 00100, 011111011111	1010, 00110011000, 0101010101, 1011010, 10101, 010100, 101001, 100100110101

Table 3. Tomita Language Set

Table 4 summarizes the results described in [23] and obtained with EXS. Column *TL* indicates which set of TLS was used. *Avg. Evals* is the average number of evaluations per run before a solution to the training set was found. *Gen. Accuracy* is the percentage of strings correctly classified, calculated as shown below:

$$\frac{correct\_positiveexamples + correct\_negative\_examples}{all\_positive\_examples_all\_negative\_examples}. \quad (2)$$

Similarly to [23], EXS was executed 50 times for each set of strings from TLS. Each run has a population of 50 individuals. The deepness limit is 50 generations.

The general average of accuracy in the experiment using SILLA (89.6 %) was better than in the experiment using GP (76.65 %). The worst average in the experiment with GP is 65.25 % and with SILLA it is 77 %. While the experiment with GP has no 100 % of accuracy for any set, 2 sets of TLS were 100 % correctly classified using SILLA. For more complex samples, that is, grammars with a large number or rules, the metrics using SILLA were worse.

Hence, it can be said that GP explores a large search space, where the GP algorithm not always converges to complete solutions, even in simple cases, but produces good solutions in the end. On the other hand, for these simple cases,

	GP				EXS			
TL	Avg. Evals	Avg (%)	Var	Best (%)	Avg. Evals	Avg (%)	Var.	Best (%)
1	30	88.39	0.0391	100	53.91	100	0	100
2	10̇10	84.00	0.0232	100	109	100	0	100
3	12̇450	66.28	0.0174	100	1 425.99	86.10	1 252	92
4	7̇870	65.25	0.0324	100	2 072.53	89.67	1 701	91.33
5	13̇670	68.65	0.0147	82.94	2 539.20	77	2 081	84.08
6	2̇580	95.94	0.0269	100	2 517.34	83.11	1 390	87
7	11̇320	67.69	0.0221	100	3 011.33	91.43	1 563	93.94

Table 4. Results of the experiment

SILLA always converged. The space searched by SILLA is not as large as the one searched by GP. We observed in the experiment that in more complex cases, its search was limited in local maximums and SILLA did not find a good solution to the inference. We are now working on this problem.

The language recognized by the grammar inferred by our tool contains the language generated by the old grammar. However, adding new production rules can give to the grammar a power of derivation much bigger than necessary. For this reason, ILLA accepts as parameter a set of negative samples, and during the inference process, the algorithm discards any grammar that derives at least one of these negative samples.

5 DISCUSSION

Let us now discuss how our algorithms can be applied to the evolution of more realistic DTDs. Despite of its simplicity, the example used in Section 3 illustrates how the algorithms can deal with real-life schema evolution. This is because the example contains the most common structures found in *DTDs for data* [11]. This kind of DTDs are the main target of our algorithms and the most largely used in database applications.

We can characterize *data DTDs* as follows [11]:

Deterministic: Following the W3C recommendations, schemas for XML should be deterministic. It is a well-known fact that ambiguous grammars derive into non-deterministic schemas. Our algorithms generate LL grammars, which cannot be ambiguous [1].

Non-recursive: It is unusual to have recursively defined elements in *data DTDs*. Our tool does deal both with recursive and non-recursive grammars. The recursive part of our grammars can be used to generate the content models of the DTD.

Use of `PCDATA`: Most of the elements of this kind of DTDs (around 60 %) are defined as textual content. Our algorithm are well-adapted to this feature. The example given in Section 3 is typical in this sense.

Absence of empty rules: Most of the production rules for this kind of schema are non-empty (only around 2 % of the rules are empty). The number of production rules generated by our algorithms is lower when empty rules are not part of the original grammar.

The items above clearly show the applicability of our algorithms in more complex cases. The scalability of our algorithms is evidenced by the fact that real-life *data DTDs* present the same structural characteristics as our example.

6 CONCLUDING REMARKS

This paper introduces an approach and a supporting tool to allow the evolution of XML schemas. EXS explores the correspondence between grammars and schemas. It implements two algorithms to infer a grammar either from a given grammar and a positive sample (ILLA), or from sets of samples (SILLA).

In this initial version only DTD schemas are evolved by EXS. In this way, the main contributions of this tool are:

a) Incremental evolution of a schema given a new XML document, allowing more than one modification per run. EXS does not impose changes on documents, but rather computes a new schema that preserves the consistency of the documents. Beside of this, the schema presents a small number of changes in relation to the original one;

b) learning from scratch, when no schema is provided.

The algorithms implemented by EXS are based on the LL parsing technique. ILLA generates new production rules based on the existent rules in the grammar. So, grammars with large numbers of rules result in a large number of new production rules, having a (natural) negative impact in the algorithm performance. The search implemented by ILLA, in an average case, has the exponential complexity $O(a^n)$, where a is the average number of production rules generated per iteration, and n is the depth of the search tree. If either a or n is very large, the search will be intractable. However, we have observed that the number of generated production rules is reduced if the grammar has no empty production rules, which is usual for grammars representing *data DTDs*, as mentioned before.

EXS can also be used in the context of GI. Some results comparing EXS with the GP approach in this context were presented in Section 4. It can be observed that the search space examined by SILLA is not as large as in GP. However, SILLA was more efficient to infer grammars from simpler samples.

Some future improvements of EXS are related to:

a) an evaluation function to measure the probability of the utility of the production rules. This metric can be used during the learning process to reduce the search space without losing the convergence to solution;

b) a heuristic function to evaluate different grammars with respect to the original one. This function makes possible the application of different meta-heuristic methods and supports hybrid strategies such as the combination of machine learning and evolutionary computation. Still in this context, SILLA can be extended to work with all the concepts of evolutionary computation, mainly gene combination;

c) generation of the initial individuals.

A APPENDIX

Figures 9 and 10 present the call graph for ILLA and SILLA, respectively. The algorithms of each function are described next.

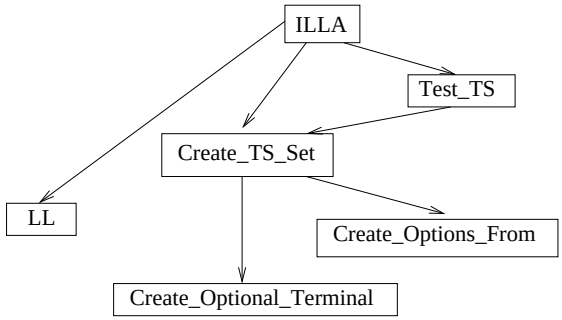

Fig. 9. Call graph for ILLA

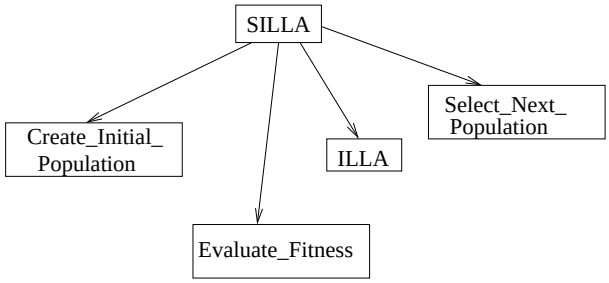

Fig. 10. Call Graph for SILLA

ILLA Called Functions

1: **Test_TS**(grammar G, string w, set TS, integer n, integer nMax): returns grammar;
2: repeat
3: Choose an element TS[i];
4: NG := G added with the production rules in TS[i];

```
   Remove element i from TS;
   Return ILLA(NG, w, n + 1, nMax);
until NG derives w, or TS is empty;
```

```
Create_TS_Set(grammar G, stack P, terminal symbol a): returns a list of sets of production rules;
If a = $ (that is, if w was derived until the last symbol, but there was still non-terminal symbols on P), then
   If the symbol on top of P is a terminal symbol, then
      Returns Create_Optional_Terminal(P[top], G);
   Else
      Returns the set {Y ::= e || Y is a non-terminal in P};
Else
   Let TS be a set of sets of production rules;
   For each X in P, do
      If X is a terminal symbol, then
         Add Create_Optional_Terminal(X, G) to TS;
      Else,
         Add Create_Options_From(X, a, G) to TS;
         From the second element X on, if the production rule 'X ::= e' doesn't belong to G, it is added to all sets generated from X;
   Return TS;
```

```
Create_Optional_Terminal(terminal symbol a, grammar G): returns a set of production rule sets;
Let TS =  // TS is a set of sets of production rules
For each 'X ::= αaβ' in G, do
   TS = TS - { X ::= αaβ } / ∪ { X ::= aAβ, A ::= a || ϵ }
Return TS;
```

```
Create_Options_From(non-terminal X, terminal a, grammar G):
      returns a set of sets of production rules;
   Let TS :=  ; // TS is a set of sets of production rules
   //a) Add a simple production rule TS:
      TS = TS {X ::= a};
   //b) optional
      For each rule 'Y ::= αXβ' of G, do
         TS = TS - {Y ::= αXβ} ∪ {Y ::= αXZβ, Z ::= α || ϵ}
   //c) zero or more
      For each 'X ::= αβ' of G, do
         TS = TS - {X ::= alphaβ}∪ {X ::= αβX || ϵ}
   //d) one or more
      For each 'X ::= aβ' of G, do
         TS = TS - {X ::= aβ} ∪ {X ::= aZ, Z ::= βX || ϵ}
   Let F :=  ; // F is a set of sets of terminal symbols
   If ϵ ∈ First(X) then
```

```
17:    F := First(X) ∪ Follow(X);
18:  Else
19:    F := First(X);
20:  For each x F, do
21:    For each 'Y ::= αxβ' of G, do
22:      //e) concatenation
23:        TS = TS - {Y ::= αxβ} ∪ {Y ::= αZ, Z ::= xβ ‖ αxβ}
24:      //f) or TS = TS {X ::= αZ, Z ::= xβ ‖ αβ}
25:  Return TS;
```

SILLA Called Functions

```
1: Create_Initial_Population(set of positive samples P): returns a set of grammars;
2:  result := ; // a set of sets of production rules
3:  For each sample w P, such that | w | = n, do
4:    result := result   S ::= AB, A ::= w[1], B ::= CD,
5:      C ::= w[2], ... X ::= YZ, Y ::= w[n-1], Z ::= w[n]
6:  Let Avg, Deep and i be integers;
7:  Add to result a grammar with the following form:
8:  For Avg := 1 to average size of production rules, do
9:      Deep := logAvg ( | w |) ;
10:     For i = 1 to Deep, generate the production rules:
11:     - 'S ::= A1A2...AAvg'
12:     - 'A1 ::= B1B2...BAvg'
13:     - 'A2 ::= C1C2...CAvg'
14:     - (...)
15:     For every non-terminal symbol X in right side of the production rules generated
    in last iteration of the previous loop, generate 'X ::= w[i]' until consuming all the
    symbols of w. For the remaining non-terminal symbols, generate 'X ::= e';
```

```
1: Select_Next_Population(set of grammars S1, set of grammars S2): returns a set of
   grammars;
2:  U := S1 S2;
3:  Fill F_Weight % of result with the grammars of U that have the best fitness, and
   remove them from U;
4:  Fill P_Weight % of result with the grammars of U that have the best correct_positive,
   and remove them from U;
5:  Fill N_Weight % of result with the grammars of U that have the best correct_negative,
   and remove them from U;
6:  Fill the rest of result with grammars chosen randomly in U;
```

```
1: Evaluate_Fitness(a set of grammars S, a set of positive samples P, and a set of
   negative samples N);
2:  For each grammar G of S,
3:    Let cp be the number of positive samples correctly classified by G;
4:    Let cn be the number of negative samples correctly classified by G;
5:    Let tp be the total of positive samples;
```

```
6:  Let tn be the total of negative samples;
7:  G.fitness := ((cp / tp) + (cn / tn)) / 2;
8:  G.correct_positive := cp;
9:  G.correct_negative := cn;
```

REFERENCES

[1] Aho, A. V.—Sethi, R.—Ullman, J. D.: Compilers, Principles, Techniques, and Tools. Addison-Wesley 1942, reprinted with corrections in March 1998.

[2] Ahonen, H.: Automatic Generation of SGML Content Models. Electronic Publishing Origination, Dissemination, and Design, Vol. 8, 1995, No. 2/3, pp. 195–206.

[3] Ahonen, H.—Mannila, H.—Nikunen, E.: Generating Grammars for SGML Tagged Texts Lacking DTD. Mathematical and Computer Modeling, Vol. 26, 1997, No. 1, pp. 1–13.

[4] Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Inf. Comput., Vol. 75, 1987, No. 2, pp. 87–106, ISSN 0890-5401, Academic Press, Inc.

[5] Back T.—Urich, H.—Schwefel, H. P.: Evolutionary Computation: Comments on the History and Current State. IEEE Transactions on Software Engineering, Vol. 17, 1991, pp. 3–17.

[6] Basu, M.: Introduction to Biological Sequence Analysis. Tutorial Presentation, World Congress on Computational Intelligence, IEEE, Hawaii, May 12–17, 2002.

[7] Bex, G. J.—Gelade, W.—Neven, F.—Vansummeren, S.: Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data. Proc. of WWW 2008 – 17th International Conference on World Wide Web, Beijing 2008, pp. 825–834.

[8] Bouchou, B.—Duarte, D.—Halfeld Ferrari, M.—Laurent, D.—Musicante, M. A.: Schema Evolution for XML: A Consistency-Preserving Approach. Lecture Notes in Computer Science, Vol. 3153, 2004, pp. 876–888.

[9] Caron, P.—Ziadi, D.: Characterization of Glushkov Automata. Theoretical Computer Science, Vol. 233, 2000.

[10] Chabin, J.—Halfeld-Ferrari, M.—Musicante, M. A.—Rety, P.: Minimal Tree Language Extensions: A Keystone of XML Type Compatibility and Evolution. Proceedings of ICTAC 2010 – 7th International Colloquium on Theoretical Aspects of Computing, LNCS 6255, Springer 2010, pp. 60–75.

[11] Choi, B.: What are Real DTDs Like? Proc. of WebDB 2002 – Fifth International Workshop on the Web and Databases, in conjunction with ACM PODS/SIGMOD 2002. Informal proceedings, Madison 2002, pp. 43–48.

[12] Da Luz, R.—Halfeld Ferrari, M.—Musicante, M. A.: Regular Expression Transformations to Extend Regular Languages (With Application to a Datalog XML Schema Validator). Journal of Algorithms, Vol. 62, 2007, No. 3-4, pp. 148–167.

[13] DTD Tutorial. W3Schools Online Web Tutorials: `http://www.w3schools.com/dtd/`, visited on 26/02/2006.

[14] Durbin, R.—Eddy, S.—Krogh, a.—Mitchison, G.: Biological Sequence Analysis. Cambridge University Press, New York 2000.

[15] Garofalakis, M. N.—Gionis, A.—Rastogi, R.—Seshadri, S.—Shim, K.: XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In Proceedings of ACM SIGMOD Conference on Management of Data, pp. 165–176, May 2000, Dallas, Texas, `http://citeseer.ist.psu.edu/garofalakis00xtract.html`.

[16] Gold, E. M.: Language Identification in the Limit. Information and Control, Vol. 10, 1967, pp. 447–474.

[17] Gold, E. M.: Complexity of Automaton Identification from Given Data. Information and Control, Vol. 37, 1978, No. 3, pp. 302–320.

[18] Goldman, R.—Mchugh, J.—Widom, J.: From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. Workshop on the Web and Databases (WebDB '99),Philadelphia, pp. 25–30, `citeseer.ist.psu.edu/article/goldman99from.html`, 1999.

[19] Goldman, R.—Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB '97, Proceedings of 23rd International Conference on Very Large Data Bases, Athens 1997, pp. 436–445, `citeseer.ist.psu.edu/126680.html`, 1997.

[20] Harman, M.: Search-Based Software Engineering for Maintenance and Reengineering. Conference on Software Maintenance and Reengineering, Bari 2006, p. 311, IEEE Press 2006.

[21] Hopcroft, J. E.—Ullman, J. D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley 1979.

[22] Koza, R. J.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge 1992.

[23] Luke, S.—Hamahashi, S.—Kitano, H.: Genetic Programming. In GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, Banzhaf, W. et al. (Eds.), San Francisco: Morgan Kaufmann 1999.

[24] Mala, D. J.—Ruby, S.—Mohan, V.: A Hybrid Test Optimization Framework – Coupling Genetic Algorithm with Local Search Technique. Computing and Informatics, Vol. 29, 2010, No. 1, pp. 133–164.

[25] McMinn, P.: Search-Based Software Test Data Generation: A Survey. Software Testing, Verification and Reliability, Vol. 14, 2004, No. 2, pp. 105–156.

[26] Mernik, M.—Crepinsekj, M.—Gerlic, G.—Zumer, V.—Bryant, B.—Sprague, A.: Learning Context-Free Grammars Using an Evolutionary Approach. Technical Report, University of Maribor and University of Alabama at Birmingham, 2003.

[27] Namakura, K.—Ishiwata, Y.: Synthesizing Context Free Grammars from Sample Strings Based on Inductive CYK Algorithm. Fifth International Colloquium of Grammatical Inference, Springer-Verlag 2000.

[28] Namakura, K.—Matsumoto, M.: Incremental Learning of Context Free Grammar. LNCS 2484, pp. 749–753, Springer, Berlin 2002.

[29] Nestorov, S.—Abiteboul, S.—Motwani, R.: Inferring Structure in Semistructured Data. SIGMOD Record 1997.

[30] Păun, G. (Ed.): Mathematical Aspects of Natural and Formal Languages. World Scientific Series In Computer Science, Vol. 43, World Scientific, Singapore 1994.

[31] Silva, J. C. T.—Musicante, M. A.—Pozo, A. R. T.—Vergilio, S. R.: XML Schema Evolution by Context Free Grammars. International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 444–449, Software Knowledge Institute, Boston 2007.

[32] Widom, J.: Data Management for XML: Research Directions. IEEE Data Engineering Bulletin, Vol. 22, 1999, No. 3, 1999.

[33] Wu, Y.—Offutt, J.: Modeling and Testing Web-Based Applications. `citeseer.ist.psu;edu/551504.html`, visited on 07/2004.

[34] Xhafa, F.—Carretero, J.—Dorronsoro, B.—Alba, E.: A Tabu Search Algorithm for Scheduling Independent Jobs in Computational Grids. Computing and Informatics, Vol. 28, 2009, No. 2, pp. 251–275.

[35] XML Schema Tutorial. W3Schools Online Web Tutorials: `http://www.w3schools.com/schema/`, visited on 02/26/2006.

[36] XML Tutorial. W3Schools Online Web Tutorials: `http://www.w3schools.com/xml/`, visited on 02/26/2006.

Julio Cesar Teodoro Silva received his M. Sc. degree in informatics from Federal University of Paraná (UFPR), Brazil, in 2006. His current research interests include: software testing and development methodologies.

Aurora Trinidad Ramirez Pozo is an Associate Professor at Federal University of Paraná, Brazil, since 1997. She received her M. Sc. and Ph. D. in electrical engineering from Federal University of Santa Catarina, Brazil (1991 and 1996, respectively). Hers research interests include evolutionary computation, data mining and software engineering.

Silvia Regina Vergilio received her M. Sc. (1991) and D. Sc. (1997) degrees from University of Campinas, UNICAMP, Brazil. She is currently with the Computer Science Department at Federal University of Paraná, Brazil.

Martin A. Musicante received his B. Sc. in computer science from Escuela Superior Latino-Americana de Informtica (1988), M. Sc. and Dr. Sc. in computer science from Universidade Federal de Pernambuco (1990 and 1996, respectively). He is currently Associate Professor at Universidade Federal do Rio Grande do Norte. His research interests are in computer science, focusing on programming languages semantics, XML and languages for web service description.