Inferring a Relax NG Schema from XML Documents

Guen-Hae Kim, Sang-Ki Ko, and Yo-Sub Han(✉)

Department of Computer Science, Yonsei University, 50, Yonsei-Ro, Seodaemun-Gu, Seoul 120-749, Republic of Korea
{guenhaekim,narame7,emmous}@cs.yonsei.ac.kr

Abstract. An XML schema specifies the structural properties of XML documents generated from the schema and, thus, is useful to manage XML data efficiently. However, there are often XML documents without a valid schema or with an incorrect schema in practice. This leads us to study the problem of inferring a Relax NG schema from a set of XML documents that are presumably generated from a specific XML schema. Relax NG is an XML schema language developed for the next generation of XML schema languages such as document type definitions (DTDs) and XML Schema Definitions (XSDs). Regular hedge grammars accept regular tree languages and the design of Relax NG is closely related with regular hedge grammars. We develop an XML schema inference system using hedge grammars. We employ a genetic algorithm and state elimination heuristics in the process of retrieving a concise Relax NG schema. We present experimental results using real-world benchmark.

Keywords: XML schema inference · Regular hedge grammar · Relax NG · Genetic algorithm

1 Introduction

Most information in the real-world has structured with linear ordering and hierarchies. Structural information is often represented in XML (Extensible Markup Language) format, which is both human-readable and machine-readable. An XML schema describes properties and constraints about XML documents. We can manipulate XML data efficiently if we know the corresponding XML schema for the input XML data [11,16,20]. Many researchers demonstrated several advantages when the corresponding XML schema exists [14,21].

All valid XML documents should conform to a DTD or a XML schema. However, in practice, we may not have a valid schema or have an incorrect schema or have an incorrect schema for an input XML data [2,17]. For these reasons, there were several attempts to infer a valid XML schema from a given XML data [3–5]. Bex et al. [3,4] presented an idea for learning deterministic regular expressions for inferring *document type definitions* (DTDs) concisely from XML documents. Bex et al. [5] designed an algorithm for inferring *XML Schema Definitions* (XSDs), which are more powerful than DTDs, from XML documents.

A.-H. Dediu et al. (Eds.): LATA 2016, LNCS 9618, pp. 400–411, 2016.
DOI: 10.1007/978-3-319-30000-9_31

We consider a schema language called *Relax NG*, which is more powerful than both DTDs and XSDs [19]—due to its expressive power, researchers proposed several applications based on Relax NG schema language [1,15]. League and Eng [15] proposed a compression technique of XML data with a Relax NG schema and showed its effectiveness, especially, for highly tagged and nested data.

We study the problem of inferring a concise Relax NG schema from XML documents based on the genetic algorithm approach. Note that a XML document can be described as an ordered tree. Then a Relax NG schema is a regular tree language. Therefore, we use normalized regular hedge grammars (NRHGs) [18] as theoretical tools for representing Relax NG schema since NRHGs exactly captures the class of regular tree languages. We employ a genetic algorithm for learning NRHGs from a set of trees and design a conversion algorithm for obtaining a concise Relax NG schema from NRHGs.

The main idea of inferring a Relax NG schema consists of the following three steps:

1. construct an initial NRHG that only generates all positive instances,
2. reduce the size of the NRHG using genetic algorithm while considering all negative examples, and
3. convert the obtained NRHG into a concise Relax NG schema with the help of state elimination algorithm and variable dependency computation.

We present experimental results with three benchmark schemas and show the preciseness and conciseness of our approach. Our Relax NG inference system successfully infers three benchmark schemas with instances randomly generated by a Java library called xmlgen. We measure the accuracy of our inference algorithm by validating XML documents generated from the original schema against the inferred schema. The inferred schema accepts about 90 % of positive examples and rejects about 80 % of negative examples.

We give some basic notations and definitions in Sect. 2, and present a technique for inferring an NRHG from a set of positive examples in Sect. 3. In Sect. 4, we present a conversion algorithm that converts an NRHG into a Relax NG schema. Experimental results are presented in Sect. 5.

2 Preliminaries

Let Σ be a finite alphabet and Σ^* be the set of all strings over the alphabet Σ including the empty string λ. The size $|\Sigma|$ of Σ is the number of characters in Σ. For a string $w \in \Sigma^*$, we denote the length of w by $|w|$ and the ith character of w by w_i. A language over Σ is any subset of Σ^*.

Let t be a tree, t_n be the number of nodes in t and t_e be the number of edges in t. We define the size $|t|$ of a tree t to be $t_n + t_e$, which is the number of nodes and edges in t. The root node of t is denoted by t_{root}. Let v be a node of a tree. Then, we denote the parent node of v by v_{parent}, the left sibling of v by v_{sibling}, and the ith child of v by $v[i]$. We also denote the number of children of the node v by $|v_{\mathsf{child}}|$ and the label of a node v by $\mathsf{label}(v)$.

A *regular expression* over Σ is $\emptyset, \lambda$, or $a \in \Sigma$, or is obtained by applying the following rules finitely many times. For two regular expressions R_1 and R_2, the union $R_1 + R_2$, the catenation $R_1 \cdot \mathcal{R}_2$, and the star R_1^* are regular expressions.

A *nondeterministic finite-state automaton* (NFA) A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a multi-valued transition function, $s \in Q$ is the initial state and $F \subseteq Q$ is a set of final states. The transition function δ can be extended to a function $Q \times \Sigma^* \to 2^Q$ that reflects sequences of inputs. A string w over Σ is accepted by A if there is a labeled path from s to a state in F such that this path spells out the string w; namely, $\delta(s, w) \cap F \neq \emptyset$. The language $L(A)$ recognized by A is the set of all strings that are spelled out by paths from s to a final state in F: $L(A) = \{w \in \Sigma^* \mid \delta(s, w) \cap F \neq \emptyset\}$.

A *normalized regular hedge grammar* (NRHG) G is specified by a quintuple (Σ, V_T, V_F, P, s), where Σ is an alphabet, V_T is a finite set of tree variables, V_F is a finite set of forest variables, $s \in V_T$ is the starting symbol, and P is a set of production rules, each of which takes one of the following four forms:

(a) $v_t \to x$, where v_t is a tree variable in V_T, and x is a terminal in Σ,
(b) $v_t \to x\langle v_f \rangle$, where v_t is a tree variable in V_T, x is a terminal in Σ and v_f is a forest variable in V_F,
(c) $v_f \to v_t$, where v_f is a forest variable and v_t is a tree variable,
(d) $v_f \to v_t v_f'$, where v_f and v_f' are forest variables and v_t is a tree variable.

We consider a *derivation* of NRHGs. Given a sequence of variables, we repeatedly replace variables with productions on the right-hand side.

1. for a production rule $v_t \to x$, a node labeled by $x \in \Sigma$ is derived from the tree variable v_t,
2. for a production rule $v_t \to x\langle v_f \rangle$, a tree with a root node labeled by x and its child node v_f is derived from v_t,
3. for a production rule $v_f \to v_t$, a node v_t is derived from v_f, and
4. for a production $v_f \to v_t v_f'$, a sequence of nodes v_t and v_f' is derived from v_f.

The language generated by G is the set of trees derived from s. Given an NRHG $G = (\Sigma, V_T, V_F, P, s)$, the size $|G|$ of G to be $|V_T| + |V_F| + |P|$. Note that the NRHGs generate *regular tree languages* of unranked trees. Thus an NRHG can be converted into an unranked tree automaton and vice versa [6].

For more knowledge in automata and formal language theory, the reader may refer to the textbooks [12, 22].

3 Inference of an NRHG from Trees

A XML document is useful to store a structured information and, often, represented as a labeled and ordered tree. Given a set of positive examples (trees) and a set of negative examples, we aim at learning an NRHG that generates all positive examples and does not generate all negative examples.

Let T be a given set of examples, $T_+ \subseteq T$ denote the set of positive examples and $T_- \subseteq T$ be a set of negative examples. First, we construct *primitive NRHGs* from positive examples; For each tree $t \in T_+$, we construct an NRHG G_t that only accepts the tree t. Namely, $L(G_t) = \{t\}$. Then, we take the union of all primitive NRHGs and construct an NRHG G_{T_+} that only accepts all positive examples, that is,

$$L(G_{T_+}) = \bigcup_{t \in T_+} L(G_t).$$

We reduce the size of the resulting NRHG by merging variables using a genetic algorithm. Note that the problem of minimizing the NRHG is at least as hard as any PSPACE-complete problem as the problem of minimizing NFAs is PSPACE-complete [13]. Therefore, we rely on the genetic algorithm approach to reduce the size of very large primitive NRHGs as much as possible.

3.1 Primitive NRHG Construction

We present a linear-time algorithm for constructing a primitive NRHG that only accepts a given tree t. Our algorithm runs recursively from the root node of t to the leaf nodes.

For each node of t, we recursively construct an NRHG according to the number of children of t. If a node v labeled by x has no child, then we create a production rule $T \rightarrow x$. If a node v labeled by x has more than one child, then we create a production rule $T \rightarrow x\langle F\rangle$. Then, we make a forest variable F to generate the sequence of subtrees of v. Let us assume that v has n subtrees from t_1 to t_n and a tree variable T_i generates a subtree t_i. Then, by creating the following variables and production rules, we let F generate the sequence of subtrees: $F \rightarrow T_1F_1, \;\; F_1 \rightarrow T_2F_2, \;\; \cdots \;\; F_{n-2} \rightarrow T_{n-1}F_{n-1}, \;\; F_{n-1} \rightarrow T_n$. Since each node in t is represented by a tree variable, each edge in t is represented by a forest variable and left side of each production rule is a tree variable or a forest variable, $|V_T| = t_n$, $|V_F| = t_e$ and $|P| = |t| = t_n + t_e$. Therefore, it is easy to verify that the algorithm produces an NRHG $G = (\Sigma, V_T, V_F, P, s)$ in $O(|t|)$ time such that $L(G) = t$, where $|V_T| = t_n$, $|V_F| = t_e$, and $|P| = |t|$.

Suppose that we have n positive examples from t_1 to t_n. We first construct n primitive NRHGs called G_1 to G_n, where $L(G_i) = \{t_i\}, 1 \leq i \leq n$. From the n primitive NRHGs, we take the union of the NRHGs and obtain a single NRHG G such that $L(G) = \cup_{i=1}^{n} G_i$. Let $G_i = (\Sigma_i, V_{T_i}, V_{F_i}, P_i, s_i)$ be an NRHG that accepts t_i. Then, we construct a new NRHG $G = (\Sigma, V_T, V_F, P, s)$ that accepts all instances as follows :

$$\Sigma = \bigcup_{i=1}^{n} \Sigma_i, \;\; V_T = \bigcup_{i=1}^{n} V_{T_i}, \;\; V_F = \bigcup_{i=1}^{n} V_{F_i}, \text{ and } P = \bigcup_{i=1}^{n} P_i.$$

Namely, $L(G) = \{t_1, t_2, \ldots, t_n\}$. Then, we merge all occurrences of starting symbols from s_1 to s_n and denote the merge symbol by s.

3.2 Grammar Optimization by Merging Indistinguishable Variables

Now we have an NRHG $G = (\Sigma, V_T, V_F, P, s)$ that only generates all positive examples. Since we aim at retrieving a concise Relax NG schema, we need to reduce the size of G as much as possible. As a first step, we find redundant structures from G by identifying *indistinguishable* variables. Given a variable v, which is either in V_T or V_F, we define $RS(v)$ to be the set of variables that appear in the right-hand side of production rules, where v appears in the left-hand side. Similarly, we define $LS(v)$ to be the set of variables that appear in the left-hand side of production rules,where v appears in the right-hand side.

$$RS(v) = \begin{cases} \{x \mid v \to x \in P\} \cup \{(x, v_f) \mid v \to x\langle v_f\rangle \in P\}, & \text{if } v \in V_T, \\ \{v_t \mid v \to v_t \in P\} \cup \{(v_t, v_f) \mid v \to v_t v_f \in P\}, & \text{if } v \in V_F. \end{cases}$$

We say that two variables v_1 and v_2, where $v_1, v_2 \in V_T$ or $v_1, v_2 \in V_F$ are *right-indistinguishable* if and only if $RS(v_1) = RS(v_2)$. It is easy to see that we do not change the language of G if we replace the occurrences of v_1 and v_2 with a new variable v' since two variables generate exactly the same structures.

$$LS(v) = \begin{cases} \{v_f \mid v_f \to v \in P\} \cup \{(v_{f_1}, v_{f_2}) \mid v_{f_1} \to v v_{f_2} \in P\}, & \text{if } v \in V_T, \\ \{(v_t, x) \mid v_t \to x\langle v\rangle \in P\}, & \text{if } v \in V_F. \end{cases}$$

We also define two variables v_1 and v_2 to be *left-indistinguishable* if and only if $LS(v_1) = LS(v_2)$. We say that two variables v_1 and v_2 are *indistinguishable* if they are right-indistinguishable or left-indistinguishable. Here we show that the language of the resulting NRHG does not change when we merge indistinguishable variables into a single variable.

Theorem 1. *Given two indistinguishable variables v_1 and v_2 of an NRHG $G = (\Sigma, V_T, V_F, P, s)$, let G' be a new NRHG where all indistinguishable variables are merged according to indistinguishable classes of G. Then, $L(G) = L(G')$.*

We repeat the merging process until there is no pair of indistinguishable variables in the NRHG. Empirically, we have obtained about 80 % size reduction by merging indistinguishable variables in primitive NRHGs. We show the experimental evidence in Sect. 5.

3.3 Genetic Algorithm

We now have a primitive NRHG from a set of positive examples without any indistinguishable variables. Next, we need to make it as small as possible while considering a set of negative examples. Recall that our main goal is to compute a concise NRHG that generates all positive examples and does not generate all negative examples.

We employ a genetic algorithm (GA), which involves an evolutionary process, to find a small NRHG from a given primitive NRHG. In a genetic algorithm, we first make a population of candidate solutions called *individuals* and make it evolve to the population of better solutions by the help of genetic operators such as *structural crossover* and *structural mutation.*

We explain how these genetic operators work in our schema inference algorithm. We assume that every individual in the first population is indeed an NRHG with seven variables and thus encoded as a string of length 7.

– *structural crossover*: In the population, we randomly select two encoded individuals p_1 and p_2 as parents. We use $p_1 = 1324133$ and $p_2 = 2234144$ as a running example for explaining our approach.

$$p_1 = 1324133 \text{ and } p_2 = 2234144.$$

These strings encode two partitions π_{p_1} and π_{p_2} as follows:

$$\pi_{p_1} : \{1,5\},\{3\},\{2,6,7\},\{4\} \text{ and } \pi_{p_2} : \{5\},\{1,2\},\{3\},\{4,6,7\}.$$

Namely, the ith number of p_1 implies the index of the block where the ith variable of p_1 belongs. For example, the third number of p_1 is 2, and therefore, the variable 3 belongs to the second block #2 of π_{p_1}. Now we randomly select two blocks, say #2 and #4. The #2 block is copied from p_1 to p_2 by taking the union of #2 blocks in p_1 and p_2 and #2 block to be moved from p_1, and #4 block is copied from p_2 to p_1 in the same way. We obtain the following results:

$$\pi_{p'_1} : \{1,5\},\{3\},\{2\},\{4,6,7\} \ \pi_{p'_2} : \{5\},\{1,2\},\emptyset,\{3,4,6,7\}$$

In this way, the number of blocks of each partition can diminish by merging randomly selected blocks.

– *structural mutation*: We randomly select an individual $p = 1324133$ and replace a character in the string by some random number. For example, if we replace the second character by 4, then the following offspring is produced:

$$\pi_{p'} : \{1,5\},\{3\},\{2,6,7\},\{2,4\}.$$

We employ the GA approach for inferring a concise NRHG from a set of positive examples and a set of negative examples as follows:

1. Initialize the population of candidate solutions. Here we set the population size to, 1000. The initial candidate solutions are the primitive NRHGs reduced by merging indistinguishable variables.
2. Select some pairs of individuals according to the crossover rate (0.4) and construct new pairs of individuals by applying the crossover operator to the selected pairs.
3. Select some individuals according to the mutation rate (0.03) and modify the selected individuals by applying the mutation operator.
4. Calculate a fitness value $f(p)$ for each individual p by the fitness function. Let p be an individual encoding for an NRHG $G = (\Sigma, V_T, V_F, P, s)$. Then, the fitness value $f(p)$ of p is defined as follows:

$$f(p) = \begin{cases} \dfrac{1}{|V_F| + |V_T|} + \dfrac{1}{|P|} + \dfrac{|\{w \in U_+ \mid w \in L(G)\}|}{|U_+|}, & \text{if } U_- \cap L(G) = \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

where U_+ is the set of positive examples and U_- is the set of negative examples.

5. Generate a next generation by *roulette-wheel selection* from the current population of solutions. Note that we retain the individuals from the best 10 % of the current population unchanged in the next generation and select only the remaining 90 % by roulette-wheel selection.
6. Iterate 1–5 steps until the fitness value of the best individual reaches the given threshold.

4 Converting NRHG into Relax NG Schema

Here we present a conversion algorithm from an NRHG into a corresponding Relax NG schema. A Relax NG schema uses references to *named pattern* using the `define` elements. Now tree or forest variables in NRHGs can be directly converted into the `define` elements for being used as references.

4.1 Horizontal NFA Construction

Given an NRHG $G = (\Sigma, V_T, V_F, P, s)$, let us consider the starting symbol s. Without loss of generality, we assume that there is a production rule $s \rightarrow x\langle v_f \rangle \in P$, where $x \in \Sigma$ and $v_f \in V_F$. Then, we convert s into the corresponding `define` element in the resulting Relax NG schema. Now it remains to convert the forest variable v_f into the corresponding element in the schema.

We construct an NFA that accepts all possible sequences of tree variables that can be generated by v_f. We call this procedure the *horizontal NFA construction* for v_f. For each forest variable $v_f \in V_F$, where $v_t \rightarrow x\langle v_f \rangle \in P$, we construct a horizontal NFA $A_{v_f} = (Q, \Sigma, \delta, q_0, q_f)$ as follows:

1. $Q = V_F \cup \{q_f\}$ is a finite set of states,
2. $\Sigma = V_T$ is an input alphabet,
3. $q_0 = v_f$ is the initial state, and
4. the transition function δ is defined as follows:
   (a) $q' \in \delta(q, v_t)$ for each $q \rightarrow v_t q' \in P$, and
   (b) $q_f \in \delta(q, v_t)$ for each $q \rightarrow v_t \in P$.

For example, Fig. 1 shows how we construct horizontal NFAs from a simple NRHG. From the first two production rules, we know that F_0 generates a sequences of sub-elements of `records` and F_1 generates a sequences of sub-elements of `car`. Therefore, we generate two horizontal NFAs A_{F_0} and A_{F_1} with the initial states are F_0 and F_1, respectively.

4.2 State Elimination for Obtaining Regular Expressions

Recall that a Relax NG schema can specify a regular tree language when we consider the tree structures of XML documents captured by the schema. When a Relax NG schema describes a set of possible sequences of trees, we use several elements such as `choice`, `group`, `zeroOrMore`, and so on.

$T_0 \rightarrow \texttt{records}\langle F_0 \rangle,\ \ T_1 \rightarrow \texttt{car}\langle F_1 \rangle,$

$F_1 \rightarrow T_2 F_2,\ \ F_0 \rightarrow T_1 F_0,$

$F_0 \rightarrow T_1,\ F_2 \rightarrow T_3,$

$T_3 \rightarrow \texttt{record},\ T_2 \rightarrow \texttt{country}$

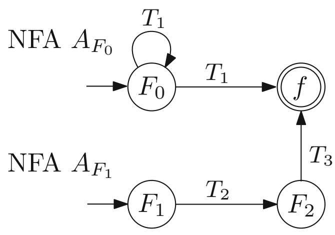

Fig. 1. Since two forest variables F_0 and F_1 are used for generating sequences of sub-elements of `records` and `car`, respectively, we construct two horizontal NFAs A_{F_0} and A_{F_1}. Note that the final states of A_{F_0} and A_{F_1} are merged into a single final state f.

The `choice` element implies that one of the sub-elements inside the element can be chosen. Therefore, we can say that the role of the `choice` element in Relax NG corresponds to the role of the union operator in regular expression. Similarly, the `group` element implies that the sub-elements inside the element should appear in exactly the same order. Thus, the `group` element corresponds to the catenation operator in regular expression. For this reason, we need to convert the obtained horizontal NFAs into the corresponding regular expressions since regular expressions are described in a very similar manner with the Relax NG schema.

State elimination is an intuitive algorithm that computes a regular expression from a finite-state automaton (FA) [9]. There are several heuristics of state elimination for obtaining shorter regular expressions from FAs [7–10]. Delgado et al. [7] observed that an order in eliminating states is crucial for obtaining a shorter regular expression. They defined the weight of a state to be the size of new transition labels that are created as a result of eliminating the state. We borrow their idea and define the weight of a state q in an FA $A = (Q, \Sigma, \delta, s, F)$ as follows:

$$\sum_{i=1}^{\mathsf{IN}} (\mathbb{W}_{\mathsf{in}}(i) \times \mathsf{OUT}) + \sum_{i=1}^{\mathsf{OUT}} (\mathbb{W}_{\mathsf{out}}(i) \times \mathsf{IN}) + \mathbb{W}_{\mathsf{loop}} \times (\mathsf{IN} \times \mathsf{OUT}),$$

where IN is the number of in-transitions excluding self-loops, OUT is the number of out-transitions excluding self-loops, $\mathbb{W}_{\mathsf{in}}(i)$ is the size of the transition label on the ith in-transition, $\mathbb{W}_{\mathsf{out}}(i)$ is the size of the transition label on the ith out-transition, and $\mathbb{W}_{\mathsf{loop}}$ is the size of the self-loop label. After calculating the weights of all states, we eliminate the state with the smallest weight and calculate the state weights again. We repeat this procedure until there are only the initial state and the single final state.

4.3 Schema Refinement

Now we have regular expressions for forest variables of an NRHG and are ready to convert these regular expressions into the form of Relax NG schema. We also use several additional techniques for converting an NRHG into a Relax NG schema.

– Replacing `zeroOrMore` elements by `oneOrMore` elements: The `zeroOrMore` element is used when there is optionally a repetition of a certain pattern like the Kleene star * operator in regular expression. For example, a regular expression a^* can be converted using a `zeroOrMore` element and an element named a inside the `zeroOrMore` element. In some cases such as $a^* \cdot a$ or $a \cdot a^*$, we can describe the pattern using the Kleene plus operator a^+. Since Relax NG schema supports the `oneOrMore` element that corresponds to the Kleene plus in regular expression, we can make the resulting schema smaller in some cases.
– Checking the dependency of forest variables: Consider two forest variables v_1 and v_2, where v_2 can be derived from v_1, but v_1 cannot be derived from v_2. If we construct regular expressions for two forest variables, then there should be some redundancy in two regular expression since the pattern described for v_2 is already contained in the pattern described for v_1. Therefore, we can write the pattern for v_2 by just referring to the pattern for v_1 instead of writing two redundant patterns.

5 Experimental Results

We conduct experiments for inferring a Relax NG schema from a given XML data. For the experiments, we use a Java library xmlgen developed as a part of Sun Multi-Schema XML Validator[1] from randomly generating positive and negative XML instances for an input Relax NG schema.

5.1 Experimental Setup

We use three benchmark Relax NG schemas—XENC, XML-DSig, IBTWSH—to evaluate the performance of our Relax NG inference algorithm.

We aim at inferring only the relationship between elements from XML data. Therefore, we ignore the descriptions for attributes of XML data from benchmark schemas. Moreover, we manually replace the elements defined by the `anyName` name by the elements with the name `anyName` since otherwise randomly generated instances may have too many elements with arbitrary names. We also limit the maximum number of appearing sub-elements in the `zeroOrMore` elements to 2 since otherwise we may have very large XML instances compare to the size of input schema.

5.2 Size Reduction of NRHGs by Optimization

We show that the optimization process helps to reduce the size of primitive NRHGs generated by positive instances.

Table 1 exhibits that the optimization process reduce 84.15 %, 82.13 %, 77.04 % of redundancy from the primitive NRHGs constructed from the positive instances of XENC, XML-DSig and IBTWSH, respectively. Note that the optimization process contributes to the speedup of the genetic process as the size of initial population also decreases substantially.

[1] The Oracle Multi-Schema XML Validator (MSV). https://msv.java.net/.

Table 1. The compression ratio ([size of NRHG before optimization]/[size of NRHG before optimization]) achieved by merging indistinguishable variables from primitive NRHGs.

Benchmark schema	XENC		XML-DSig		IBTWSH	
	Before	After	Before	After	Before	After
# of tree variables	755.17	123.67	1048.73	123.20	1222.86	162.57
# of forest variables	705.17	187.17	998.73	176.93	1122.86	208.29
# of production rules	1460.33	359.83	2047.47	349.13	2345.71	467.29
Average size of NRHG	4094.93	649.27	4691.43	838.14	2920.67	670.67
Compression ratio (%)	15.85		17.87		22.96	

5.3 Precision of Inferred Schema

For three benchmark schemas, we generate 50 positive instances and 25 negative instances by xmlgen. Note that we set the error rate to 1/100 when generating negative instances. Then we run our inference algorithm to infer Relax NG schemas for the instances. We repeat the process 100 times and calculate the average value.

We evaluate the precision of our inference system in two directions. First, we validate the inferred schema against 1,000 positive instances generated from the original benchmark schema. Second, we validate the inferred schema against 1,000 negative instances generated from the original benchmark schema. We expect that the inferred schema should generate the positive instances and not generate the negative instances if they are inferred closely to the original schemas.

Table 2. The precision of our Relax NG schema inference system

Benchmark schema	XENC	XML-DSig	IBTWSH (50/25)	IBTWSH (100/50)
Precision for positive instances (%)	91.98	93.50	46.05	96.87
Precision for negative instances (%)	82.43	83.85	83.45	74.67

Table 2 shows the precision of our inference algorithm. Note that the fourth column shows the results for the benchmark IBTWSH with 100 positive and 50 negative instances. Speaking of the first two results, precisions for positive and negative instances are very similar. The inferred schemas generate more than 90 % of positive instances of the original schemas and do not generate more than 80 % of negative instances. The performance is good considering that we infer schemas with only a small number of instances.

For the benchmark IBTWSH, only 46.05 % of positive instances can be generated by the inferred schema. The inference precision improves significantly when we double the number of instances to 100 positive and 50 negative instances. We suspect that the reason why the precision for the benchmark IBTWSH is especially low is because IBTWSH schema has several `interleave` elements that allow the sub-elements to occur in any order. Since our inference system does not support the inference of `interleave` elements, it seems more difficult to infer schemas with `interleave` elements.

6 Conclusions

XML schemas are formal languages that describe structures and constraints about XML instances. They are crucial to maintain and manipulate XML documents efficiently—an XML document should conform a specific XML schema. However, in practice, we may not have a valid schema or have an incorrect schema. This has led many researchers to design an efficient schema inference algorithm.

We have presented an Relax NG schema—one of the most powerful XML schema languages—inference algorithm based on 1) a genetic algorithm for learning process and 2) state elimination heuristics for retrieving a concise Relax NG schema. We have implemented the proposed algorithm and measured the preciseness and conciseness of the algorithm using well-know benchmark schemas. Our experiments have showed that the proposed algorithm has 90 % preciseness for accepting positive examples and 80 % preciseness for rejecting negative examples. In future, we plan to consider other learning approaches for better performance and more Relax NG specifications such as interleave.

References

1. Athan, T., Boley, H.: Design and implementation of highly modular schemas for XML: customization of RuleML in relax NG. In: Palmirani, M. (ed.) RuleML - America 2011. LNCS, vol. 7018, pp. 17–32. Springer, Heidelberg (2011)
2. Barbosa, D., Mignet, L., Veltri, P.: Studying the XML web: gathering statistics from an XML sample. World Wide Web **8**(4), 413–438 (2005)
3. Bex, G.J., Gelade, W., Neven, F., Vansummeren, S.: Learning deterministic regular expressions for the inference of schemas from XML data. ACM Trans. Web **4**(4), 14 (2010)
4. Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of concise DTDs from XML data. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 115–126. VLDB Endowment (2006)
5. Bex, G.J., Neven, F., Vansummeren, S.: Inferring XML schema definitions from XML data. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 998–1009. VLDB Endowment (2007)
6. Comon, H., Dauchet, M., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2007). http://www.tata.gforge.inria.fr
7. Delgado, M., Morais, J.J.: Approximation to the smallest regular expression for a given regular language. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 312–314. Springer, Heidelberg (2005)

8. Gruber, H., Holzer, M.: Provably shorter regular expressions from deterministic finite automata. In: Ito, M., Toyama, M. (eds.) DLT 2008. LNCS, vol. 5257, pp. 383–395. Springer, Heidelberg (2008)
9. Han, Y.S.: State elimination heuristics for short regular expressions. Fundam. Inf. **128**(4), 445–462 (2013)
10. Han, Y.S., Wood, D.: Obtaining shorter regular expressions from finite-state automata. Theor. Comput. Sci. **370**(1), 110–120 (2007)
11. He, B., Tao, T., Chang, K.C.-C.: Clustering structured web sources: a schema-based, model-differentiation approach. In: Lindner, W., Fischer, F., Türker, C., Tzitzikas, Y., Vakali, A.I. (eds.) EDBT 2004. LNCS, vol. 3268, pp. 536–546. Springer, Heidelberg (2004)
12. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston (1979)
13. Jiang, T., Ravikumar, B.: Minimal nfa problems are hard. SIAM J. Comput. **22**(6), 1117–1141 (1993)
14. Koch, C., Scherzinger, S., Schweikardt, N., Stegmaier, B.: Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In: Proceedings of the 30th International Conference on Very Large Data Bases, pp. 228–239. VLDB Endowment (2004)
15. League, C., Eng, K.: Schema-based compression of XML data with RELAX NG. J. Comput. **2**(10), 9–17 (2007)
16. Löser, A., Siberski, W., Wolpers, M., Nejdl, W.: Information integration in schema-based peer-to-peer networks. In: Eder, J., Missikoff, M. (eds.) CAiSE 2003. LNCS, vol. 2681, pp. 258–272. Springer, Heidelberg (2003)
17. Mignet, L., Barbosa, D., Veltri, P.: The XML web: a first study. In: Proceedings of the 12th International Conference on World Wide Web, pp. 500–510 (2003)
18. Murata, M.: Hedge automata: a formal model for XML schemata (1999)
19. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
20. Shvaiko, P.: A classification of schema-based matching approaches (2004)
21. Wang, G., Liu, M., Yu, G., Sun, B., Yu, G., Lv, J., Lu, H.: Effective schema-based XML query optimization techniques. In: Proceedings of the 7th International Symposium on Database Engineering and Applications, pp. 230–235 (2003)
22. Wood, D.: Theory of Computation. Harper & Row, New York (1987)