Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 10 (2012) 120 – 127

Procedia Computer Science

The 3rd International Conference on Ambient Systems, Networks and Technologies (ANT)

Optimization and Refinement of XML Schema Inference Approaches[1]

Michal Klempa, Jakub Stárka, Irena Mlýnková

XML and Web Engineering Research Group, Department of Software Engineering, Charles University in Prague, Czech Republic

Abstract

XML is a widely used technology. Although in most real life applications XML data is required to conform to particular schemas, the majority of real-world XML documents does not contain any explicit declaration. To fill the gap, the research area of automatic schema inference from XML documents has emerged. This paper refines and extends recent approaches to the automatic schema inference by exploiting an obsolete schema in the inference process, designing new MDL measures and heuristic excluding of eccentric data inputs. It delivers a ready-to-use implementation integrated into *jInfer* – a framework for XML schema inference. Experimental results are a part of the paper.

Keywords:
XML, schema, inference, minimal description length

1. Introduction

XML [1] is a popular data format and it has become the format of choice for data representation for its simple but powerful design. To enforce a defined structure of XML documents, one can use XML schema definition languages such as DTD [1], or XSD [2]. Although designing an XML schema is a simple task (especially in DTD language), a half of randomly crawled documents does not link any associated schema [3]. In addition, used schemas are very simple compared to the features provided by the languages.

To overcome this problem, the research of *automatic schema inference* from XML documents has emerged. Each element in the XML schema has its content model defined with a regular expression (RE). Thus the problem of learning a regular language from a finite set of positive examples arises. However, it cannot be solved in general. The current solutions either define a subclass of regular languages, that is *identifiable in the limit* [4, 5], or solve the problem *heuristically* [6, 7, 8]. This work belongs to the latter set and provides several optimizations by exploiting an obsolete schema in the inference process, designing new MDL measures and heuristic excluding of eccentric data inputs. The paper delivers a ready-to-use and easy-to-extend implementation integrated into *jInfer* [9] – a framework for XML schema inference.

[1]This work was supported by the Czech Science Foundation, grant number P202/10/0573.
Email addresses: starka@ksi.mff.cuni.cz (Jakub Stárka), mlynkova@ksi.mff.cuni.cz (Irena Mlýnková)

1877-0509
doi:10.1016/j.procs.2012.06.019

The rest of the paper is structured as follows: In Section 2 we analyze related work. In Section 3 we provide the theoretical background necessary for the rest of the text. The proposed optimizations and experimental evaluation are described in Section 4 and in Section 5 respectively. In Section 6 we conclude.

2. Related Work

Recently, several approaches to the problem of schema inference for a set of XML documents have been proposed. Some of them define an *identifiable subclass* of regular languages and develop algorithms to identify the subclass [4, 5, 10]; others propose *ad-hoc heuristics* [8, 7, 6].

Ahonen [4] solves the interference problem using two identifiable subclasses of regular languages: *k-contextual* and (k, h)-*contextual* languages. The inference proceeds as follows: First, a *prefix tree automaton* (PTA) accepting all positive examples is constructed. Then, it is modified by merging its states to obtain a k-contextual or a (k, h)-contextual automaton, which proceeds to a disambiguation procedure. Finally, the disambiguated automaton is converted into a RE. In [7], a PTA is constructed from the given positive examples too. Additionally, the used PTA contains statistical information from the examples, in particular transition use counts are set during PTA construction. The top s percent of state k-strings are computed by ordering k-strings in decreasing order (according to probability) and then taking just enough of them (from the beginning) for which probabilities sum up to s. Apparently, using a greedy merging, one may end up in a too general automaton. Therefore, the authors define a measure to select the best trade-off automaton.

In [6], the previous method is extended and refined. The main improvements are advanced element clustering considering not only element names, but the structure of element contents to identify distinct elements, and inference of `xs:all` particle in XML Schema output. The latter extension shortens the RE in the output schema (in comparison to an equivalent RE naming nearly all possible permutations).

We build our solution also on work [11]. It deals with the problem of schema inference being given not only XML input documents, but with the knowledge of an old schema of these files which seems to be a common case in real-world data [3]. We will show that the existing approaches can be further optimized.

3. Theory

This section contains definitions and other prerequisites used in the rest of the text. Primarily, we define the type of automata we deal with and the *minimum description length* (MDL) principle [12] used to evaluate the quality of the inferred automaton.

Definition 1 (Deterministic Probabilistic Finite Automaton)**.** *A DPFA is a tuple* $A = (Q, \Sigma, \delta, q_0, \lambda, F, P)$*, where:* Q *is a finite set of states, such that* $\lambda \notin Q$*,* λ *is a dummy state indicating immediate halt,* Σ *is an alphabet (finite set of symbols),* $\delta : (Q \times \Sigma) \to (Q \cup \{\lambda\})$ *is a transition function,* $q_0 \in Q$ *is an initial state,* $P : \delta \to \mathbb{N}_{\not\sim}$ *is function of* transition use counts*,* $F : Q \to \mathbb{N}_{\not\sim}$ *is function of* state final counts*.*

Definition 2 (Probabilistic Prefix Tree Automaton)**.** *Let* $A = (Q, \Sigma, \delta, q_0, \lambda, F, P)$ *be a DFPA. Let* (V, E) *be a directed* underlying graph *of A, where* $V = Q$ *is a set of nodes and* $E \subseteq Q \times Q$ *is a set of edges defined as follows:*

$$(q_1, q_2) \in E \quad \textit{iff} \quad \exists a \in \Sigma : \delta(q_1, a) = q_2.$$

A PPTA is a DFPA whose underlying graph is a tree rooted at state q_0*.*

3.1. MDL Principle

Consider the data consisting of points in two-dimensional space. Let the x-axis represent time and y-axis represent the values coming from an unknown data source. We want to predict the future y values. For this purpose, we have to exploit an underlying data regularity (unless the data are produced by fair coin tosses). The MDL principle suggests to view the data as being generated (explained) by a particular *hypothesis*; the set of possible explanations is denoted as *model*. The decision of which hypothesis explains the data best is not a trivial task not to over-fit the data and, at the same time, to capture the underlying regularity.

In this paper, a special version of MDL, called *Crude MDL* [12], is used which basically tells us that the best hypothesis is the one that compresses the data the most, i.e. $L(H) + L(D|H)$ is minimal, where $L(H)$ is the length, in bits, of the description of the hypothesis and $L(D|H)$ is the length (in bits) of the description of the data when encoded with the help of the hypothesis. To define the $L(H)$ we usually design an ad-hoc code of the hypothesis. There are some universal codes, that may help us. The $L(D|H)$ is usually a probabilistic code, given that H is a probabilistic source of the data which emits a particular value x with probability p.

In the following text, C denotes a *code* and L_C the *code-length function*. The code-length function returns the length of code (in bits) of the given data input encoded using code C. In [13, p. 100] a *standard universal code for integer values (SUCI)* is presented. The code should be used for integers which are not to originate from probabilistic source (and values are possibly unbounded). The code-length for integer value n is computed using the following formula (for $c_0 \approx 2.865$):

$$L_{\mathbb{N}}(n) = \log n + \log\log n + \log\log\log n + ... + \log c_0$$

It means "sum logarithms until the first negative value encounters (exclude it), then add $\log c_0$ constant". The motivation and details of the code can be found in [13], a simplified version is described in [12].

4. Proposed Solution

The proposed solution is a composition work built on ideas from [6, 4, 7, 11]. The work provides a complete schema generation environment, incorporated into the *jInfer* framework [9], which makes it ready-to-use for potential users. In this section we describe only the key parts of our approach; all details can be found in [14]. The main improvements are the following:

- exploitation of a (possibly existing) obsolete XML schema,
- new (more accurate) MDL measures of automaton and input data, and
- a possibility to automatically tag selected input grammar rules as invalid, excluding them from inference (e.g. deviations, misspelled words, etc.), i.e.generate more accurate schema for valid inputs

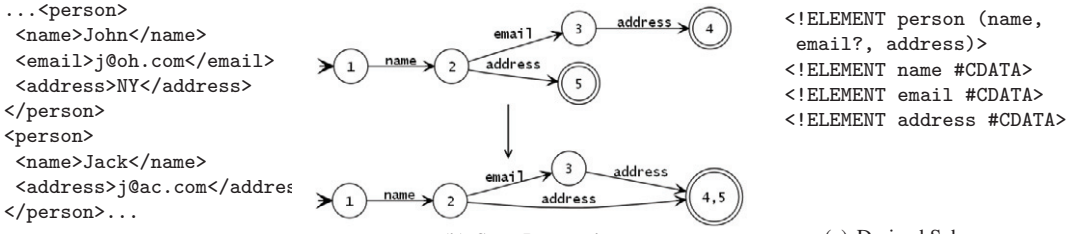
We follow the inference steps proposed in [6]. In **phase I.**, *positive examples* (element instances from input documents) are clustered, grouping the instances corresponding to one element type definition into one cluster. Then, contrary to existing works, we also parse XML schema files (i.e the obsolete ones) into grammar rules. These are then clustered by the element name together with element instances originating from XML documents. Each cluster contains one rule from XML schema input files and zero or more rules from XML documents, all forming the *input grammar*. Since in both XSD and DTD the element content model is basically specified by an RE, we consider positive examples as being generated by some DPFA and try to infer this automaton. If there is a RE from schema input, then a DPFA torso is constructed from it. Starting with an empty automaton or with a torso (when an obsolete schema is processed), we run the same algorithm for building DPFA in the form of PPTA from positive examples (first automaton in Figure 1(b)).

In **phase II.**, the automaton is modified by merging its states. In general, when two or more states are merged, the language generated by the automaton becomes more general (see Figure 1(b)). The merge process is driven by *merge criterion testers*, which search for candidate states for merging.

Finally, in **phase III.**, the inferred automaton is converted into an equivalent RE using a *state removal algorithm* [15] and the RE is added to a list of all XML element definitions (*output grammar*). In the output, an XML schema is generated by naming all element definitions from the output grammar and specifying their content model definitions in the selected schema language (example depicted in Figure 1(c)).

4.1. Merging of States

Merging two states of an automaton proceeds as follows: One state is selected as preserved (usually the first one) and the second one to be removed from the automaton. All in-transitions of the removed state are



(a) XML Fragment (b) State Removal (c) Derived Schema

Fig. 1. Algorithm Overview

reconnected to the preserved state. All out-transitions of the removed state are reconnected to lead from the preserved state, and all loops of the removed state are copied to the preserved state.

With DPFA, care must be taken to preserve an invariant use count and final count properties. After merging, all in-transitions of the preserved state are divided into clusters by the transition source state. For each cluster of in-transitions, a second grouping by the alphabet symbol is performed. Finally, each group is ensured to have exactly one transition: if there are two or more transitions in the group, these are merged into one transition so that the use count is set as the sum of merged transitions. An analogical algorithm is run for out-transitions and loops. We call this process *collapsing state transition and loops* and it preserves the use count property. To preserve the final count property, the preserved state final count value is simply incremented by the final count value of the removed state.

4.2. Selecting States to Merge

To select states to merge in a reasonable way we employ two verified state equivalence criteria: *sk-strings* [7] heuristic criterion and k, h*-context* [4] criterion. Modules responsible for providing candidate alternatives for merging are called *merge criterion testers*. From the available alternatives a *merging state strategy* selects which to merge and which not. In *jInfer* we have implemented several merging state strategies called: *Greedy*, *GreedyMDL*, *HeuristicMDL* and *DefectiveMDL*. The first three are classical approaches, the last one is our own new proposal (see Section 4.4).

Greedy. The *Greedy* strategy simply merges all candidate states provided by merge criterion testers. For example for the k, h*-context* tester it means, that it simply creates a k, h*-context* automaton as defined in [4].

GreedyMDL. The *GreedyMDL* strategy uses the MDL principle to evaluate a DPFA and input strings encoded by the automaton. The precise MDL code (called an *objective quality function*) is described in Section 4.3. For now, it is sufficient to assume the existence of an objective function $mdl(A, S)$ which is given an automaton A and a set of input strings S and returns a non-negative real value, the overall quality of the solution, where a lower value signifies a better solution (remember this is the description length function).

While trying to merge candidate alternatives, the *GreedyMDL* strategy always keeps the currently achieved minimum quality value (and the associated automaton). A space of possible solutions is explored in a greedy way, but some sort of a complete scanning of continuation possibilities is done: all candidate alternatives to merge are evaluated. The algorithm stops when there are no more candidates to merge, or when all alternative candidates returned by merge criterion testers end up in an automaton with higher quality value than the one actually achieved.

HeuristicMDL. The *HeuristicMDL* as a simple heuristic strategy works basically the same way as *GreedyMDL*, but it holds the n best minimal solutions instead of only one. At each iteration, merge criterion testing for one randomly selected automaton of the n best automata is done. All the alternatives returned are attempted to be merged and only the automata with the lower quality value than the current worst solution is stored a in capacity-constrained sorted list (thus, it always holds the best n solutions). The algorithm stops when it is staggering – when the set of the best n solutions is not modified for a whole iteration.

4.3. Objective Quality Function

In the sense of the crude MDL, one has to design a code for a hypothesis and a code for data compressed using the hypothesis. Since in this work a basic assumption is that positive examples were generated by some DPFA, the hypothesis is the DPFA itself. And as described in [13, p. 100], if a hypothesis is of probabilistic character, the best code to use is the complete prefix code with code-lengths equal to $-\log(p)$ for the one option, whose probability of appearance in data equals to p. When generating strings using the DPFA, in each state of the automaton the algorithm decides which transition to follow or whether to output a whole word randomly – driven by a *probabilistic density function* defined by the probabilities of each followed transition, and the actual state that it is final. Given a state q, we compute a unity value:

$$u_q = F(q) + \sum_{q' \in Q, a \in \Sigma} P(q, a, q').$$

Then, function $P'_q : \Sigma \times (Q \cup \{\lambda\}) \to [0, 1]$ is defined as:

$$\begin{aligned} P'_q(a, q') &= \tfrac{P(q,a,q')}{u_q} && (\forall q' \in Q, a \in \Sigma) \\ P'_q(a, q') &= 0 && (q' = \lambda) \end{aligned}$$

The function P'_q together with the value $f'_q = \frac{F(q)}{u_q}$ forms a probabilistic density function of a discrete probability random variable X_q i.e. "what is done next, if we are in state q" defined as:

$$\begin{aligned} P[X_q = (a, q')] &= P'_q(a, q') \\ P[X_q = terminate] &= f'_q \end{aligned}$$

Using the set of random variables X_q (one for each state q), encoding input strings is simple: When the automaton generates a string, the configuration sequence is the same as if it had the input string which the automaton was accepting. Thus, computing a code-length can be done as follows: For each input string, traverse automaton while reading it and record probabilities of transitions along the way. Let us consider input string $s = a_1, \ldots, a_n$. Let probabilities $p_1, \ldots, p_n$ be recorded transition probabilities, and p_{n+1} the probability f'_q of the state, where reading of the string ended. Code-length C of the string s equals to:

$$C(s) = \sum_{i=1}^{n+1} -\log(p_i) = -\log\left(\prod_{i=1}^{n+1} p_i\right) \tag{1}$$

This corresponds with seeing the problem from the other side: probability p_s of the generated string equals the product of probabilities of decisions taken at each configuration. If one designs a complete prefix code over the probability density function of probabilities p_s, the code-length of a single string equals $C(s)$.

However, there is no need to traverse the automaton with each input string to get the total code-length of all input strings. When the DPFA is built, each input string incremented the use count value of each transition passed and incremented the final count value of the state it ended in. When a DPFA is traversed for each input string, each transition is passed exactly its use count-times and traversing ends in each state exactly its final count-times. From this, it is easier to compute the total code-length of input strings S (encoded with the help of DPFA A, where $\{u_q | q \in Q\}$ are pre-computed unity values for each state) as:

$$L(S|A) = \sum_{q \in Q, a \in \Sigma, q' \in Q} \left(P(q, a, q') \cdot -\log\left(\frac{P(q, a, q')}{u_q}\right)\right) \tag{2}$$

The key problem is how to encode the DPFA. In general, there is no universal way, because DPFA is an ad-hoc model to solve a custom ad-hoc problem and we propose an ad-hoc code for it. Let us denote the states of an automaton as $q_1, \ldots, q_{|Q|}$. The proposed code is $< |Q|, |\Sigma|, alphabet, \langle q_1 \rangle, \ldots, \langle q_{|Q|} \rangle >$, where $|Q|$ is the cardinality of the set of states encoded using standard universal code for integers, $|\Sigma|$ is the cardinality of Σ encoded using SUCI. The symbols of an alphabet are named using a uniform code in the table $alphabet$,

which is a translation table from uniform encoding of each symbol into a prefix code. The prefix code for alphabet symbols is established using a histogram of symbol occurrences over all transitions of the automaton. The probability of an individual symbol a for this code is therefore:

$$p_a = \frac{\text{occurences of symbol } a}{\text{occurences of all symbols}}.$$

Each $\langle q_i \rangle$ is a code of one state in the automaton $\langle |Q'_i|, \langle t_1 \rangle, \ldots, \langle t_{|Q'_i|} \rangle \rangle$, where Q'_i is a set of all states immediately reachable from the state q_i, formally $Q'_i = \{q; q \in Q, \exists a \in \Sigma : \delta(q_i, a) = q\}$. Each $\langle t_j \rangle$ is a code of one out-transition of the state q_i, each in a form $\langle q, P(q,a), a \rangle$, where q is a code for a destination state encoded using a uniform code over all states, $P(q,a)$ is a use count value of the transition encoded using SUCI, a is a symbol of alphabet encoded using the prefix code for the alphabet established earlier. Let us denote p_a the probability of each symbol a in the alphabet. Then the code-length of this ad-hoc code of the automaton is:

$$\begin{aligned} L(A) \quad = \quad & suci(|Q|) + suci(|\Sigma|) + |\Sigma| \cdot \log(|\Sigma|) - \log\left(\prod_{a\in\Sigma} p_a\right) + \\ & \left(\sum_{i=1}^{|Q|} suci(|Q'_i|)\right) + \sum_{i=1}^{|Q|} \sum_{j=0}^{|Q'_i|} \left(\log(|Q|) + suci(P(q,a)) - \log(p_a)\right) \end{aligned} \tag{3}$$

where $suci(|Q|)$ is the SUCI length for state-count, $suci(|\Sigma|)$ is the SUCI length for alphabet size, $|\Sigma| \cdot \log(|\Sigma|)$ is the sum of lengths of each left side in the *alphabet translation table* (each symbol with code-length $\log|\Sigma|$), $-\log\left(\prod_{a\in\Sigma} p_a\right)$ is the sum of lengths of each right side in the alphabet translation table (each symbol a with code-length $-\log(p_a)$), $\left(\sum_{i=1}^{|Q|} suci(|Q'_i|)\right)$ is the sum of all SUCI lengths for transition counts of each state, and the last double sum is the sum of the sum of code-length of each transition, destination state with uniform code-length of $\log|Q|$, SUCI length for use count and prefix code-length for a symbol a from the alphabet. The whole code-length function is given as the sum of (2) and (3):

$$mdl(A, S) = L(S|A) + L(A) \tag{4}$$

4.4. DefectiveMDL *Merging State Strategy*

In general, a user can chain merging state strategies arbitrarily. The *DefectiveMDL* strategy should be attached at the end of such a chain, when the automaton is ready to be converted into a RE. It is based on the idea to decide which input strings are so eccentric that they probably are "mistakes" and should be repaired in input documents rather than incorporated into the output schema. In some cases, input documents are selected to cover all expected constructs and thus there is no use for DefectiveMDL strategy, but in case of automatic inference based on "dirty" documents, the identification of "mistakes" can be usefull.

Let T be the set of input strings we suspect as eccentric. We try to remove T from the inference process. If the inferred schema is much simpler, we consider T as eccentric. Apparently, this approach would simply remove all input strings, since no documents fit the simple `EMPTY` construct. Here, a trade-off thinking applies and we exploit MDL again. We try to remove input strings in set T. If the MDL value $mdl(A, S \setminus T)$ is smaller enough than the value $mdl(A, S)$, we consider T as eccentric.

To formalize the "smaller enough", we define a criterion: When the description length of a new automaton and input strings, together with the description length of the removed input strings, is smaller than the description length of an old automaton together with all input strings, the strings are considered eccentric. We define an *error code* for one input string $a_1, \ldots, a_n$ as a sequence of prefix codes for each symbol (specified the same way as in (3)). Thus, the code-length of the error code for one input string equals to:

$$L_{error}(s) = \sum_{i=1}^{n} -\log(p_{a_i}) = -\log\left(\prod_{i=1}^{n} p_{a_i}\right) \tag{5}$$

where p_{a_i} is the probability of symbol a_i in the established prefix code for the alphabet. Since by removing input strings it may occur that we also remove some symbol from the alphabet used in the automaton, the

prefix code for the alphabet is established with a histogram not only over the automaton, but also over removed input strings. Basically, the prefix code for the alphabet remains the same, since it has to encode all input strings, no matter they are used in the automaton or in the error code. So the code-length of the error code of all removed strings is:

$$L_{error}(S) = \sum_{s \in S} L_{error}(s) \tag{6}$$

We denote $mdl(A, S, S_r)$ the MDL code-length of an automaton A, strings S encoded using the automaton A, and strings S_r removed. Then $mdl(A, S, S_r)$ equals to:

$$mdl(A, S, S_r) = mdl(A, S) + L_{error}(S_r) \tag{7}$$

The idea of MDL comparison can be likened to a compression of a text document using zip compression: When the length of the zip-file plus the length of some removed sentences from the document is smaller than the length of the original zip-file with all sentences, it makes sense to deduce from the phenomena that the removed sentences are so eccentric that they corrupt underlying data regularity.

The input strings can be removed from the automaton as follows: The automaton is traversed while reading an input string (remember that the automaton is deterministic) and each transition gets its use count value decremented along the way. The final state gets its final count value decremented. Since only strings that previously formed the automaton are removed, use counts and final counts never reach negative values.

The last question is which input strings to remove. In *jInfer*, we use a program interface called *Suspect*, which returns input strings it is suspecting as eccentric. Checking strings one by one is one simple strategy implemented. If we remove all input strings that pass one transition, the transition is rendered as unused.

5. Experiments

To test the proposed algorithms, we have generated random test files using an XML data generator *ToXGene* [16] and its supplemented templates for the *XMark* benchmark [17]. Three files were generated, with different length: `auction_big.xml` (∼1MB), `auction_small.xml` (∼100kB) and `auction_tiny.xml` (∼30kB). We used the available `auction.dtd` for the benchmark as an input schema S_{old} in tests.

We tested combinations of *Greedy*, *GreedyMDL*, *HeuristicMDL* with alternatives combined from the merge condition testers $(2, 1)$-*context* and *sk/heuristic* ($s = 50\%, k = 2$). The *Greedy* strategy coupled with $(2, 1)$-*context* condition tester served as a simulation of [4], each of them with and without the schema on input. We have also tested *SchemaMiner* [6] and *Trang* [18], which is a converter between various schema formats, both without schema input since they do not support it. (*SchemaMiner* was not able to finish computation within a reasonable time, thus results for the tiny dataset are available.). To shorten the REs, we replace element names using substitutions from Table 1.

element	*shortcut*	*element*	*shortcut*	*element*	*shortcut*		
initial	i	reserve	r	bidder	b	type	t
current	c	privacy	p	itemref	f	interval	l
seller	s	annotation	a	quantity	q		

Table 1. Substitution table of element names into shortcut names

Let us explore the inferred REs for element `open_auction`. The DTD specifies RE $ir?b*cp?fsaqtl$ for this element. The resulting REs for each inference algorithm are depicted in Table 2.

As we can see the (k, h)-*context* method did not perform very well. *GreedyMDL* and *HeuristicMDL* preferred more complex REs for the big dataset, since it better fits the data (lower MDL value) than *Greedy*, which simply merged everything it could. *Trang* is able to learn chain RE and since the DTD expression is a chain RE, being given a big enough dataset, *Trang* is always able to learn exactly the DTD expression. Thus, if the user is expecting only very simple REs on the output of the algorithm, *Trang* is able to satisfy this. *SchemaMiner* probably fell into a common problem of converting an automaton into corresponding RE – when bad state removal order is used, the RE constructed can be such as the one on *SchemaMiner* output.

Method	*Big*	*Small*	*Tiny*
2, 1-context	$i((c\|(rc))\|((b\|(rb))b*c))(f\|(pf))saqtl$		$i(b\|(rb))b*cpfsaqtl$
Greedy	$i(r\|b)*cp*fsaqtl$	$i(r\|b)*cp*fsaqtl$	$i(r\|b)*cpfsaqtl$
GreedyMDL	$i(c\|((b\|r)b*c))(f\|(pf))saqtl$	$i(b\|r)*cp*fsaqtl$	$i(b\|r)*cpfsaqtl$
HeuristicMDL	$i(c\|((b\|r)b*c))(f\|(pf))saqtl$	$i(b\|r)*cp*fsaqtl$	$i(b\|r)*cpfsaqtl$
Trang	$ir?b*cp?fsaqtl$	$ir?b*cp?fsaqtl$	$ir?b+cpfsaqtl$
SchemaMiner	-	-	$i(bb*cpfsaqtl)\|(rbb*cpfsaqtl)$

Table 2. Element `open_auction`

The usage of existing schema leads to enforcing the output REs to be in the form of the schema because DFSA torso is built *before* parsing input strings, thus the DFSA in in the form of RE from the input schema.

6. Conclusion

The aim of this paper was to describe several optimizations we have proposed for the approaches dealing with the problem of XML schema inference. Three main refinements are: exploiting an additional information for the purpose of optimization – the original XML schema, design of a finer MDL measure for usage in heuristic methods, and possibility of generating a simpler schema by repairs in input documents.

By incorporating the schema input, the resulting schema is enforced not to deviate much from the original. It is highly practical for users who have the old schema available and want to infer the schema accurate for the new data, but there is still a space for improvements. The designed MDL measure is more appropriate than measures used in common, as it employs a lot of probability codes, which are code-length optimal. It provides the solution with the superiority feature to prefer simpler DFSA with smaller input dataset.

In our future work we will focus mainly on further optimization using other possibly available input data, such as XML queries, XSLT transformations, or negative examples, and exploitation of user interaction.

References

[1] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, Extensible markup language (xml) 1.0 (2004). URL `http://www.w3.org/TR/2004/REC-xml-20040204`

[2] P. V. Biron, A. Malhotra, XML Schema Part 2: Datatypes Second Edition, `http://www.w3.org/TR/xmlschema-2/` (2004).

[3] I. Mlynkova, K. Toman, J. Pokorny, Statistical Analysis of Real XML Data Collections, in: COMAD'06: Proc. of the 13th Int. Conf. on Management of Data, Tata McGraw-Hill Publishing, New Delhi, India, 2006, pp. 20–31.

[4] H. Ahonen, Generating grammars for structured documents using grammatical inference methods, Ph.D. thesis, Department of Computer Science, University of Helsinki, Series of Publications A, Report A-1996-4 (1996).

[5] G. J. Bex, W. Gelade, F. Neven, S. Vansummeren, Learning deterministic regular expressions for the inference of schemas from xml data, ACM Trans. Web 4 (2010) 14:1–14:32.

[6] O. Vošta, I. Mlýnková, J. Pokorný, Even an ant can create an xsd, in: DASFAA'08: Proceedings of the 13th international conference on Database systems for advanced applications, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 35–50.

[7] A. Raman, J. Patrick, P. North, The sk-strings method for inferring pfsa, in: In Proceedings of the, 1997.

[8] R. K. Wong, J. Sankey, On structural inference for xml data, Tech. rep. (2003).

[9] M. Klempa, M. Mikula, R. Smetana, M. Švirec, M. Vitásek, jinfer xml schema inference framework, `http://jinfer.sourceforge.net/modules/paper.pdf`.

[10] G. J. Bex, F. Neven, T. Schwentick, S. Vansummeren, Inference of concise regular expressions and dtds, ACM Trans. Database Syst. 35 (2010) 11:1–11:47.

[11] I. Mlynkova, On inference of xml schema with the knowledge of an obsolete one, in: A. Bouguettaya, X. Lin (Eds.), Twentieth Australasian Database Conference (ADC 2009), Vol. 92 of CRPIT, ACS, Wellington, New Zealand, 2009, pp. 79–86.

[12] P. Grunwald, A tutorial introduction to the minimum description length principle (2004). URL `http://www.citebase.org/abstract?id=oai:arXiv.org:math/0406077`

[13] P. Grünwald, The minimum description length principle, Mit Press, 2007.

[14] M. Klempa, Optimization and Refinement of XML Schema Inference Approaches, Master Thesis, Charles University, Prague, Czech Republic, 2011.

[15] Y.-S. Han, D. Wood, Obtaining shorter regular expressions from finite-state automata, Theor. Comput. Sci. 370 (1-3).

[16] D. Barbosa, A. O. Mendelzon, J. Keenleyside, K. A. Lyons, Toxgene: An extensible template-based data generator for xml, in: Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002), 2002, pp. 49–54.

[17] Xmark an xml benchmark project, `http://www.xml-benchmark.org/`.

[18] J. Clark, Trang: Multi-format schema converter based on relax ng, `http://www.thaiopensource.com/relaxng/trang.html`.