On Inference of XML Schema with the Knowledge of an Obsolete One

Irena Mlýnková

Charles University, Faculty of Mathematics and Physics, Department of Software Engineering
Malostranské nám. 25, 118 00 Prague 1, Czech Republic
Email: irena.mlynkova@mff.cuni.cz

Abstract

The XML has undoubtedly become a standard for data representation and manipulation. But most of XML documents are still created without the respective description of their structure, i.e. an XML schema. Hence, in this paper we focus on the problem of automatic inferring of an XML schema for a given sample set of XML documents. Contrary to existing approaches we propose an algorithm that exploits additional input information – an obsolete XML schema. Consequently, we are able to exploit the information which was correct once and to infer the schema more efficiently.

Keywords: XML Schema, validity, schema inference, schema correction.

1 Introduction

Without any doubt XML (Bray et al. 2006) is currently a de-facto standard for data representation. Its popularity is given by the fact that it is well-defined, easy-to-use and, at the same time, enough powerful. To enable users to specify own allowed structure of XML documents, so-called *XML schema*, the W3C[1] has proposed two languages – DTD (Bray et al. 2006) and XML Schema (Thompson et al. 2004, Biron & Malhotra 2004). The former one is directly part of XML specification and due to its simplicity it is one of the most popular formats for schema specification. The latter language was proposed later, in reaction to the lack of constructs of DTD. The key emphasis is put on simple types, object-oriented features and reusability of parts of a schema or whole schemas.

On the other hand, statistical analyses of real-world XML data show that a significant portion of XML documents (in particular, 52% (Mignet et al. 2003) of randomly crawled or 7.4% (Mlynkova et al. 2006) of semi-automatically collected[2]) still have no schema at all. What is more, XML Schema definitions (XSDs) are used even less (only for 0.09% (Mignet et al. 2003) of randomly crawled or 38% (Mlynkova et al. 2006) of semi-automatically collected XML documents) and even if they are used, they often (in 85% of cases (Bex et al. 2004)) define so-called *local tree grammars*, i.e. languages that can be defined using DTD as well.

In reaction to this situation a new research area of automatic inference of an XML schema has opened. The key aim is to create an XML schema for the given sample set of XML documents that is neither too general, nor too restrictive. Currently there are several proposals of respective algorithms (see Section 2), but there is still a space for further improvements. In this paper we focus on inferring of a schema from a sample set of XML documents in a special situation when we are provided with the original, but already obsolete schema. According to statistical analyses of real-world XML data (Mlynkova et al. 2006) it is quite a common case, since the XML schema is usually considered as a kind of data documentation. Since the schema is not used as it is supposed to be, i.e. for checking the correct structure of XML documents, it is usually not updated in case the respective data are. Hence, in this paper we propose an algorithm which infers a correct schema on the basis of the knowledge of the outdated one. Contrary to existing approaches that would infer a correct schema regardless the existing one, we are able to exploit the information which was correct once and to infer the schema more efficiently.

The paper is structured as follows: Section 2 overviews existing papers on automatic inference of XML schemas as well as issues related to our stated problem. Section 3 provides background information on languages for XML schema definition. Section 4 describes their relation to the theory of automata and grammars and introduces the problem of schema inference. Section 5 describes the proposed solution in detail. And, finally, Section 6 provides conclusions and outlines possible future work.

2 Related Work

The existing solutions to the problem of automatic inference of an XML schema can be classified according to several criteria. Probably the most interesting one is the type of the result (i.e. DTD or XSD) and the way it is constructed, where we can distinguish heuristic methods and methods based on inferring of a grammar.

Heuristic approaches (Moh et al. 2000, Wong & Sankey 2003, Garofalakis et al. 2000) are based on experience with manual construction of schemas. Their output does not belong to any special class of grammars and, hence, we cannot say anything about its features. They are based on generalization of a trivial schema using a set of predefined heuristic rules, such as, e.g., "if there are more than three occurrences of an element, it is probable that it can occur arbitrary times". These techniques can be further divided into methods which generalize the initial schema until a satisfactory solution is reached (e.g. (Moh et al. 2000, Wong & Sankey 2003)) and methods which generate a number of candidates and then choose the optimal one (e.g. (Garofalakis et al. 2000)). While in the first case the methods are threatened by a wrong step which can cause generation of a suboptimal result, in the latter

This work was supported in part by the Czech Science Foundation (GAČR), grants number 201/09/P364 and 201/09/0990.

[1] http://www.w3.org/

[2] Data collected with the interference of a human operator.

case they have to cope with space overhead and specifying a reasonable function for evaluation of quality of the candidates.

On the other hand, methods based on *inferring of a grammar* (Ahonen 1996, Bex et al. 2007) output a particular class of languages with specific characteristics. Although grammars accepting XML documents are context-free, the problem can be reduced to inferring of a set of regular expressions, each for a single element. But, since according to Gold's theorem (Gold 1967) regular languages are not identifiable in the limit only from positive examples (in our case sample XML documents which should conform to the resulting schema), the existing methods exploit restriction to an *identifiable* subclass of regular languages.

A set of approaches related to our stated problem, i.e. the problem of correcting an incorrect XML schema, are so-called *XML schema evolution* or *XML schema versioning* algorithms (Su et al. 2001, Tan & Goh 2005, Guerrini et al. 2007). However, their aim is opposite to ours. XML schema evolution means that the original schema is replaced by an updated schema and, hence, the effects of the update on its instances need to be solved. In particular, the approaches deal with the problem how document adaptation according to the evolved schema can be (eventually automatically) performed to make them valid again. Schema versioning means that the original documents and schemas should be preserved and a new updated version of the schema is created. Document adaptation is not an issue, but the problem of handling different versions of the same data arises.

Instead of adapting the set of XML documents according to the modified schema we have the opposite task – we want to adapt the given schema according to the set of XML documents. Among the existing works there seems to be only one approach with an aim similar to ours. In (Bertino et al. 2002) the authors propose an approach to evolving a set of DTDs to obtain structures that are correct and precise with regard to a set of XML documents. The approach is intended for a kind of dynamic repository of XML data and DTDs. It is based on exploitation of similarity of XML documents and DTDs and a set of data mining heuristics.

3 XML Schema Languages

The simplest and most popular language for description of the allowed structure of XML documents is currently the Document Type Definition (DTD) (Bray et al. 2006). It enables one to specify allowed elements, attributes and their mutual relationships, order and number of occurrences of subelements, data types and allowed occurrences of attributes. A simple example describing a database of employees is depicted in Figure 1.

```
<!ELEMENT employees (person)+>
<!ELEMENT person (name, email*, relationships?)>
  <!ATTLIST person id ID #REQUIRED>
  <!ATTLIST person note CDATA #IMPLIED>
  <!ATTLIST person holiday (yes|no) "no">
<!ELEMENT name ((first, surname)|(surname, first))>
<!ELEMENT first (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT relationships EMPTY>
  <!ATTLIST relationships superior IDREF #IMPLIED
                          inferior IDREFS #IMPLIED>
```

Figure 1: An example of a DTD of employees

At first glance it seems that the specification of the allowed structure is sufficient. Nevertheless, even in this simple example we can find several problems. For instance, we are not able to specify the correct structure of an e-mail address. Similarly, we cannot simply specify that a person can have four e-mail addresses at maximum. And, as we can see, the fact that the order of elements `first` and `surname` is not significant cannot be expressed easily as well. Therefore, the W3C proposed a more powerful tool – the XML Schema language (Thompson et al. 2004, Biron & Malhotra 2004).

The XML Schema language has a number of advantages. The main advantages are that:

- each XSD is a well-formed and valid XML document,
- it has a strong support of data types, both simple and complex and both built-in and user-defined,
- it enables one to re-use and re-define existing schemes or selected parts,
- it enables one to specify the allowed structure using more precise constraints (e.g. minimum and maximum allowed occurrences, ordered/unordered sequences, integrity constraints etc.) and
- it enables one to specify equivalent schemes using distinct constructs.

For example an XSD equivalent[3] to the example of a DTD in Figure 1 is depicted in Figure 2.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="employees">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="person" minOccurs="1"
                maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="person">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="name"/>
    <xs:element name="email" type="xs:string"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="relationships" minOccurs="0"
                maxOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="id" type="xs:ID" use="required"/>
   <xs:attribute name="note" type="xs:string"/>
   <xs:attribute name="holiday" default="no">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="yes"/>
      <xs:enumeration value="no"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
  </xs:complexType>
 </xs:element>

 <xs:element name="name">
  <xs:complexType>
   <xs:all>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="surname" type="xs:string"/>
   </xs:all>
  </xs:complexType>
 </xs:element>

 <xs:element name="relationships">
  <xs:complexType>
   <xs:attribute name="superior" type="xs:IDREF"/>
   <xs:attribute name="inferior" type="xs:IDREFS"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 2: An example of an XSD of employees

[3] Having the same set of document instances.

4 Relation to Automata and Grammars

An XML schema describing the allowed structure of XML documents is an extended context-free grammar (Berstel & Boasson 2000), i.e. a grammar where nonterminals can be rewritten regardless the context in which they occur. The extension is given by the fact that on right hand sides of productions occur regular expressions.

Definition 1 *Given the alphabet Σ,* a regular expression *(RE) over Σ is inductively defined as follows:*

- $\emptyset$ (empty set) *and* ϵ (empty string) *are REs.*
- $\forall a \in \Sigma$ *: a is a RE.*
- *If r and r' are REs of Σ, then* (rr') (concatenation), $(r|r')$ (alternation) *and* (r^*) Kleene closure) *are REs.*

The DTD language adds two abbreviations: $(r|\epsilon) = (r?)$ and $(rr^*) = (r^+)$. Also the concatenation is expressed via the '`,`' operator. The XML Schema language adds (among other extensions) another one, so-called *unordered sequence* of REs $r_1, r_2, ..., r_k$, i.e. an alternation of all possible ordered sequences of $r_1, r_2, ..., r_k$. The DTD syntax is often extended with respective '`&`' operator.

Definition 2 An extended context-free grammar *is a quadruple $G = (N, T, P, S)$, where N and T are finite sets of* nonterminals *and* terminals*, P is a finite set of* productions *and S is a non terminal called a* start symbol. *Each production is of the form $A \rightarrow r$, where $A \in N$ and r is a regular expression over alphabet $N \cup T$.*

The language generated by grammar G is denoted by $L(G)$.

A language generated by a grammar can be accepted by an automaton, in our case a finite state automaton.

Definition 3 A finite state automaton (FSA) *is a quintuple $A = (Q, \Sigma, \delta, S, F)$, where Q is a set of states, Σ is a set of input symbols (alphabet), $\delta : Q \times \Sigma^* \rightarrow Q$ is the transition function, $S \in Q$ is the start state and $F \subseteq Q$ is the set of final states.*

The language accepted by an automaton A is denoted by $L(A)$.

For each RE we can construct a FSA and vice versa.

4.1 Problem Statement

The studied problem can be described as follows: Being given a set of XML documents $D = \{d_1, d_2, ..., d_n\}$ (i.e. words over an alphabet T_D), we search for an XML schema s_D (i.e. a grammar $G_D = (N_D, T_D, P_D, S_D)$) s.t. $\forall i \in [1, n] : d_i$ is valid against s_D (i.e. $D \subseteq L(G_D)$). In particular, we are searching for s_D that is "enough" concise, precise and, at the same time, general.

Most of the existing approaches use the following strategy: For each occurrence of an element $e \in D$ and its subelements $e_1, e_2, ..., e_k$ we construct a production $\vec{p}_e$ of the form $e \rightarrow e_1\ e_2\ ...\ e_k$.[4] The left hand side is called *element type* and denoted $type(\vec{p}_e)$, the right hand side is called a *content model* of the element type and denoted $model(\vec{p}_e)$. The productions form so-called *initial grammar (IG)*. For each element type the productions are then merged, simplified and generalized using various methods and criteria. A common approach is so-called *merging state algorithm*, where a *prefix tree automaton (PTA)* is built from the productions of the same element type and then generalized via merging of its states. Finally, the generalized automata are expressed in syntax of the selected XML schema language.

An example of a IG and PTA for element `person` is depicted in Figure 3.

[4] Attributes are often omitted for simplicity.

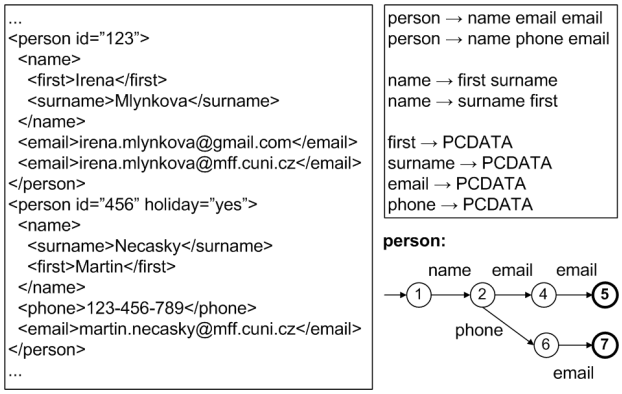


Figure 3: An example of a IG and a PTA

In the existing works the rules for merging the states of an automaton differ, but they have a common aim to create a concise and precise XML schema. While the heuristic approaches exploit a set of various heuristic rules, the approaches based on inference of a grammar utilize the rules so that the result fulfills the conditions the selected subclass of regular languages states.

The problem we are dealing with is a schema inference task with a special condition – knowledge of an obsolete, i.e. incorrect and/or too general, schema s_{orig}. Our aim is to exploit this additional information in order to speed up the inference process and to make the resulting schema more precise.

5 Proposed Approach

In general the given problem can be divided into checking and correction/adaptation of the following subsets:

1. Simple data types
2. Element/attribute names
3. REs

The first two sets can be solved relatively easily. In case of simple data types we simply check whether the selected data types are not too general or too restrictive and, if necessary, we make the respective corrections. In fact, even most of the existing schema inference methods do not deal with simple types at all.

In case of element/attribute names we can select from two approaches. On one hand, we can consider and distinguish either the same or distinct names. On the other hand, we can take into account their semantics and consider that only an element/attribute name can be modified. However, it is only a question of finding the respective mapping between the names, whereas we can find such mapping only in case the changes are semantically related, such as, e.g., changing `name` into `title`. Consequently, it is only minor aspect of the problem and we will not deal with it in the following text as well.

The most important task of the given problem is to check and correct REs. In general we can encounter the following cases:

1. The original XML schema s_{orig} does not need to be corrected. The XML documents in D and valid against it and it is enough concise and precise.
2. The XML documents in D are valid against s_{orig}, however it is too general. In particular, there can occur the following cases:
   (a) Too high upper limit of occurrences (see Example 1)

(b) Too low lower limit of occurrences (see Example 2)

(c) Occurrence of redundant data (see Example 3)

3. The XML documents in D are not valid against s_{orig} anymore. In particular, there can occur two situations:

   (a) s_{orig} does not involve items that XML documents in D do (see Example 4).

   (b) The XML documents do not involve items that are in s_{orig} mandatory (see Example 5).

Example 1 *Consider the following set of productions extracted from XML documents:*

```
E → A B C C C
E → A B C C
```

and the following production taken from s_{orig}:

```
E → A B C+
```

The production should be corrected, since the + operator should be used only in case more than 5 occurrences of an element.

Example 2 *Consider the following set of productions extracted from XML documents:*

```
E → A B C C C
E → A B C C
```

and the following production taken from s_{orig}:

```
E → A B? C
```

The production should be corrected, since the element `B` *is present in all document instances.*

Example 3 *Consider the following set of productions extracted from XML documents:*

```
E → A B C C C
E → A B C C
```

The following productions from s_{orig} need to be corrected since they contain redundant data with regard to the given documents:

```
E → A B X? C+
E → A (B | X) C+
```

Example 4 *Consider the following set of productions extracted from XML documents:*

```
E → A B C C C
E → A B C C
```

The following production from s_{orig} needs to be corrected since it does not involve element `B` *present in the documents:*

```
E → A C+
```

Example 5 *Consider the following set of productions extracted from XML documents:*

```
E → A B C C C
E → A B C C
```

The following production from s_{orig} needs to be corrected since it involves compulsory element `X` *not present in the documents:*

```
E → A B C+ X
```

5.1 Possible Solutions

The first possible solution is to simply ignore s_{orig} and infer a correct schema purely on the basis of D. The advantage of this approach is obvious – we use a verified approach that provides a correct result. However, we do not exploit an available and apparently useful information. Hence, our aim is to exploit this information when appropriate and, thus, to speed up the inference process and provide a more precise schema.

The second natural approach can be based on the following simple observation: The existing inference methods produce plenty of possible solutions, that are evaluated and the (sub)optimal one is selected as the result. It is caused by the fact that the approaches are based on heuristic rules that generalize the IG. The amount of generalizations is high and we do not know which is the optimal one unless we combine it with the rest of the schema. Hence, a natural idea may be that we will exploit the knowledge of s_{orig} in situations when there are multiple generalization possibilities. However, the problem is that this approach can be exploited only in case of simple REs. Otherwise, the inclusion, equivalence and intersection problem of REs cannot be solved in reasonable time and, consequently, we cannot easily find the related schema fragments.

Consequently, the solution we propose is a relaxed version of the two described approaches. We do exploit s_{orig}, however, we do not stick to it 100%. In addition, we are able to find its suboptimal correction/adaptation with reasonable complexity.

5.2 Proposed Solution

In the approach we propose we firstly divide the given problem into two independent and optional steps:

1. Correction of the input schema
2. Specialization of the input schema

In the fist step we assume there exists at least one document $d \in D$ s.t. d is not valid against s_{orig}. Hence, we need to find schema $s_{correct}$, i.e. the correction of s_{orig}, s.t. for $\forall d \in D : d$ is valid against $s_{correct}$. In addition, let $\Sigma_{correct}$ be the set of all possible corrections of s_{orig}. Then we want to find a correction $s_{correct}$ s.t. $dist(s_{orig}, s_{correct}) \leqslant dist(s_{orig}, s)$ for $\forall s \in \Sigma_{correct}$, where $dist(s, s')$ is the edit distance, i.e. the sequence of operations for transforming s to s'. In other words, we want to find a correction that requires the least modifications of s_{orig}.

In the second step we assume that we have a schema $s_{correct}$, s.t. $\forall d \in D : d$ is valid against $s_{correct}$. However, we want to specialize the REs involved in the schema with regard to the data in D, resulting in more precise and readable schema $s'_{correct}$.

Note that any of the steps can be used separately. On one hand, we may require only the correction step without any unnecessary schema modifications. On the other hand, we may have a correct schema but we want to make it more precise, since we know that the data are more specific.

5.2.1 Schema Correction

First of all, let us mention the fact that each content model of an XML document must be so-called *deterministic* or *1-unambiguous*.

Example 6 *Consider the following content model:*

```
(A B) | (A C)
```

It is non-deterministic, because while reading `A`*, the XML processor cannot know which* `A` *in the model is being matched without looking ahead to see which element follows. On the other hand, an equivalent content model:*

```
A (B | C)
```

is deterministic. The processor does not need to look ahead to see what follows; either `B` *or* `C` *will be accepted.*

This requirement is stated directly in the W3C specification of XML (Bray et al. 2006) and ensures that an XML processor can match the schema with the data efficiently. And, consequently, we are able to determine the validity of the documents in D efficiently as well.

The correction algorithm consists of the following steps:

1. We divide the set D into sets D_{valid} and $D_{invalid}$, i.e. valid and invalid documents, s.t. $D_{valid} \cup D_{invalid} = D$ and $D_{valid} \cap D_{invalid} = \emptyset$.
2. For $\forall d \in D_{invalid}$ we create the respective set of productions and merge them with s_{orig}.

The key step of the approach is merging a single production $\vec{p}_e$ created from an element e and its subelements in XML document $d \in D_{invalid}$ with productions of s_{orig}. The merging algorithm can be described as follows: Firstly, we identify production $\vec{q}_e$ from s_{orig} to be merged with. For this purpose we can use any of the strategies used in the existing works for grouping the productions. In most of them the productions are simply grouped according to equivalence of element types, more sophisticated approaches take into account also greater context. Since this is not the key aspect of our proposal, we will further assume the former approach.

Having the two productions $\vec{p}_e$ and $\vec{q}_e$ to be merged, we parse the $model(\vec{p}_e)$. Similarly to the approach of merging productions into a PTA, we match the elements of $model(\vec{p}_e)$ with $model(\vec{q}_e)$ until the parsing does not fail. Whenever we reach an element $e' \in model(\vec{p}_e)$ that invokes invalidity, we create a separate branch of automaton for $\vec{q}_e$ consisting of the rest of the content model staring with e'.

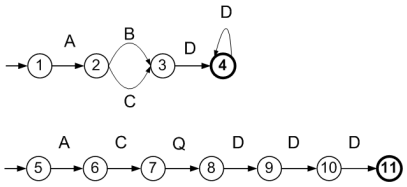
Example 7 *Consider the following example of schema production $\vec{q}_E$:*

`E → A (B | C) D+`

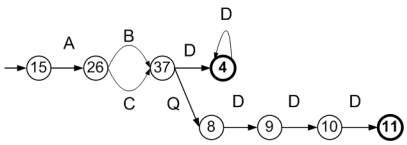
and the following example of document production $\vec{p}_E$:

`E → A C Q D D D`

The automata describing the productions are depicted as follows:



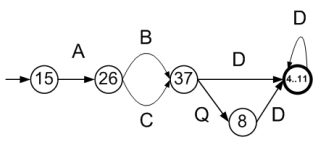
Using the above described algorithm, they are merged into the following automaton:



(Note that if we merge s_{orig} with productions of $d \in D_{valid}$, the automata of s_{orig} do not change, since there occurs no element that would violate creating of a new branch.)

After merging each of the automata, the newly created schema $s_{correct}$ ensures that each $d \in D$ is valid against $s_{correct}$. However, the respective automata, i.e. REs, are not very concise and precise. Therefore, we need to apply an approach that would merge the newly added branches more precisely.

Example 8 *Consider the merged automaton in Example 7. After more elaborate merging of states of the new branch, we get the following more concise result:*



Since there can exist multiple ways how to merge the newly added states with the original ones, we exploit a modification of existing general approach to schema inference that can cope with all the possible cases. In particular, we utilize an approach from (Vosta et al. 2008) since it is one of the recent approaches that combines most of the previously proposed and verified methods.

Firstly, note that the problem of generalization of an automaton is viewed as a kind of optimization problem.

Definition 4 *A model $M = (\Theta, \Omega, \sigma)$ of a* combinatorial optimization problem *consists of a search space Θ of possible solutions to the problem (so-called* feasible region*), a set Ω of constraints over the solutions and an* objective function $\sigma : \Theta \to \mathbb{R}_0^+$ *to be minimized.*

In our case Θ consists of all possible generalizations of an automaton. As it is obvious, Θ is theoretically infinite and thus, in fact, we can search only for a reasonable suboptimum. Therefore, we use a modification of *ACO heuristics* (Dorigo et al. 2006). Ω is given by the features of XML schema language we are focussing on. And finally, to define σ we exploit a modification of the *MDL principle* (Grunwald 2005).

Ant Colony Optimization (ACO) The ACO heuristics is based on observations of nature, in particular the way ants exchange information they have learnt. A set of artificial "ants" $\Lambda = \{a_1, a_2, ..., a_{card(\Lambda)}\}$ search the space Θ trying to find the optimal solution $s_{opt} \in \Theta$ s.t. $\sigma(s_{opt}) \leqslant \sigma(s); \forall s \in \Theta$. In i-th iteration each $a \in A$ searches a subspace of Θ for a local suboptimum until it "dies" after performing a predefined amount of steps N_{ant}. While searching, an ant a spreads a certain amount of "pheromone", i.e. a positive feedback which denotes how good solution it has found so far. This information is exploited by ants from the following iterations to choose better search steps. The search terminates either after a specified number of iterations N_{iter} or if $s'_{opt} \in \Theta$ is reached s.t. $\sigma(s'_{opt}) \leqslant T_{max}$, where T_{max} is a required threshold.

The obvious key aspect of the algorithm is one step of an ant. Each step consist of generating of a set of possible continuations, their evaluation using σ and execution of one of the candidate steps. The executed step is selected randomly with probability given by σ. And this is the biggest strength of the ACO heuristics. Contrary to greedy search strategy which can get stuck in local suboptimum, ACO is able to search greater subspace of Θ due to random selection of continuations and possible temporal moving to a worse case.

Generating a Set of Possible Continuations A single step of an ant is represented using a modification of the current automaton. As we have mentioned, most of the existing approaches exploit the merging state strategy, i.e. reduction of the set of states of the automaton on the basis of various rules, such as k, h*-context* (Ahonen 1996) which merges states with same contexts (prefixes) or s, k*-string* (Wong & Sankey 2003) which merges states with same suffixes.

We will preserve the same merging strategies, the key difference is in the set of states that can be merged. In the original algorithm, any of the states of the automaton that fulfills any of the merging conditions can be merged. In our case we do not want to modify the original automaton, since we want to preserve the information it carries. Therefore, we restrict the merging only to cases when the set of merged states involves at least one of the states of the new branch. Consequently, we can encounter the following two situations:

1. We merge the states within the new branch, i.e. we truncate the new branch.

2. We merge a state of the new branch with an original one, i.e. we reduce the number of states of the whole automaton.

Evaluation of Continuations The evaluation of moving from schema s_x to s_y, where $s_x, s_y \in \Theta$, remains in our case the same. In particular, it is defined as:

$$mov(s_x, s_y) = \sigma(s_x) - \sigma(s_y) + pos(s_x, s_y)$$

where σ is the objective function and $pos(s_x, s_y) \geqslant 0$ is the positive feedback of this step from previous iterations. For the purpose of specification of σ, most of the existing works exploit the MDL principle (Garofalakis et al. 2000). It is based on two observations: A good schema should be enough general which is related to the low number of states of the automata. On the other hand, it should preserve details which means that it enables one to express document instances in D using short codes. In other words, most of the information is carried by the schema itself and, thus, it does not need to be encoded. Hence, the quality of a schema $s \in \Theta$ described using a set of productions $R_s = \{\vec{p}_1, \vec{p}_2, ..., \vec{p}_{card(R_s)}\}$ is expressed using:

- the size (in bits) of R_s and
- the size (in bits) of codes of document instances in D expressed using R_s.

Let O be the set of allowed operators and E the set of distinct element names in D. Then we can view $model(\vec{p})$ of $\forall \vec{p} \in R_s$ as a word over $O \cup E$ and its code can be expressed as $|model(\vec{p})| \cdot \lceil \log_2(card(O) + card(E)) \rceil$, where $|model(\vec{p})|$ denotes length of word $model(\vec{p})$. The size of code of a single instance $d \in D$ is defined as the size of code of an inferring sequence of productions $R_d = \langle \vec{g}_1, \vec{g}_2, ..., \vec{g}_{card(R_d)} \rangle$ necessary to convert the initial nonterminal to d using productions from R_s. Since we can represent the sequence R_d as a sequence of ordinal numbers of the productions in R_s, the size of the code of d is $card(R_d) \cdot \lceil \log_2(card(R_s)) \rceil$.

5.2.2 Schema Specialization

In the second step of the proposed algorithm we assume that we are provided with a correct schema $s_{correct}$. Our current aim is to specify the schema using a more precise schema $s'_{correct}$. And naturally, we want to preserve the validity condition for all documents in D.

The problem of schema specialization can be divided into several steps:

1. Pruning of unused schema fragments
2. Correction of lower and upper bounds of occurrences of schema fragments
3. Correction of operators
4. Refactorization

According to user requirements, selected steps can be omitted depending on the respective application. For instance a user may omit step 1. requiring that unused schema fragments should be preserved. In fact, even the whole specialization process can be omitted in case we want to preserve the information from the original schema s_{orig} as much as possible.

Unused Schema Fragments The aim of this step is to identify schema fragments that are not used in the sample XML documents D. The approach can be described as follows: For each element e in the given schema $s_{correct}$ we preserve a *usage flag* $\varphi_{used}(e)$ that carries the information about its usage in D. At the beginning of the algorithm we set $\varphi_{used}(e) = F$ (false) for $\forall e \in s_{correct}$. Using an XML parser we parse each $d \in D$, we check usage of particular elements of $s_{correct}$ and set $\varphi_{used}(e) = T$ (true) whenever $e \in s_{correct}$ is used. After parsing the whose set D we check the flag φ_{used}. All elements $e \in s_{correct}$ s.t. $\varphi_{used}(e) = F$ together with the associated operators can be eliminated since they are not used in the sample data.

Example 9 *Consider the following set of productions extracted from XML documents:*

```
E → A C
E → A B B B C
E → A B C
E → A B B B B
```

The following production from $s_{correct}$ involves fragment `Q?` not used in the documents.

```
E → A B* C? Q?
```

Parsing the content model of the first production we set $\varphi_{used}(A) = T$ and $\varphi_{used}(C) = T$, i.e.

```
E → A  B*  C?  Q?
    T  F   T   F
```

Parsing the second production we set $\varphi_{used}(A) = T$, $\varphi_{used}(B) = T$, $\varphi_{used}(B) = T$ and $\varphi_{used}(C) = T$, i.e.

```
E → A  B*  C?  Q?
    T  T   T   F
```

Similarly we process the remaining productions which do not change the current settings. Finally, we can see that schema fragment `Q?` is not used in any of the input documents and, hence, we can specialize the schema to:

```
E → A B* C?
```

Note that since we assume that each $d \in D$ is valid against $s_{correct}$, the elimination of unused schema fragments is a correct application which preserves correctness of the content models, as well as validity of the data in D. It can be proven as follows: Since the data are valid, a candidate for elimination must be an optional schema fragment, i.e. an item of a sequence associated with either `?` or $\star$ operator or an item of a choice. Hence the elimination causes either truncating of the sequence or reduction of options of the choice. The schema fragment can be either a single element or a sequence of elements. In the latter case, again due to the assumption of validity, all the elements in the sequence have φ_{used} of `F` and, hence should be eliminated.

Finally, note that this simple strategy can be applied on both DTDs and XSDs, since their treatment in case of used and unused schema fragments is the same.

Occurrences In the second step we want to correct the allowed numbers of occurrences of schema fragments, i.e. operators `?`, + and $\star$ in case of DTD or `minOccurs` and `maxOccurs` attribute values in case of XSD. Similarly to the previous case for each fragment f in the given schema $s_{correct}$ we preserve *minimum repetition flag* $\varphi_{min}(f)$ and *maximum repetition flag* $\varphi_{max}(f)$ that carry the information about its minimum and maximum amount of successive occurrences in D.

At the beginning of the algorithm we set $\varphi_{min}(f) = \infty$ and $\varphi_{max}(f) = 0$ for each fragment $f \in s_{correct}$. Using an XML parser we again parse each $d \in D$. For each repeating sequence of a schema fragment f we determine its length l_f, i.e. the amount of repetitions. If $l_f < \varphi_{min}(f)$, we set $\varphi_{min}(f) = l_f$. If $l_f > \varphi_{max}(f)$, we set $\varphi_{max}(f) = l_f$.

Example 10 *Consider the following set of productions extracted from XML documents:*

```
E → A
E → B
E → A A
```

The following production from $s_{correct}$ should be specialized.

```
E → A+ | B | (C D)
```

Parsing the content models of the productions we set φ_{min} and φ_{max} as follows:

		`E → A+`	`|`	`B`	`|`	`(C D)`
Start:	φ_{min}	∞		∞		∞
	φ_{max}	0		0		0
`E → A`	φ_{min}	1		0		0
	φ_{max}	1		0		0
`E → B`	φ_{min}	0		0		0
	φ_{max}	1		1		0
`E → A A`	φ_{min}	0		0		0
	φ_{max}	2		1		0

The resulting values of φ_{min} and φ_{max} carry information about minimum and maximum occurrences of the respective schema fragments. In particular:

- If $\varphi_{min} = 0$, the respective schema fragment has optional occurrence.
- If $\varphi_{min} > 0$, the respective schema fragment has compulsory occurrence.
- If $\varphi_{max} > 1$, the respective schema fragment has multiple occurrence.

In case we correct an XSD, we can use φ_{min} and φ_{max} as new values for `minOccurs` and `maxOccurs` attributes of respective schema fragments. In case we correct a DTD, we transform the values of φ_{min} and φ_{max} to respective DTD operators – see Table 1, where rep_{min} is the minimal occurrence which induces generalization to arbitrary occurrences.

φ_{min}	φ_{max}	**DTD operator**
0	1	`?`
0	$> rep_{min}$	`*`
> 0	$> rep_{min}$	`+`

Table 1: Transformation of φ_{min} and φ_{max} to DTD operators

In addition, note that the values of φ_{max} also carry the same information as φ_{used}. In particular:

- If $\varphi_{max} = 0$, then $\varphi_{used} = F$.
- If $\varphi_{max} \neq 0$, then $\varphi_{used} = T$.

Consequently, using φ_{min} and φ_{max} we can also identify the unused schema fragments.

Operators Apart from unused schema fragments and imprecise minimum and maximum occurrences, there can occur also too general combinations of operators and allowed occurrences. In particular, we will deal with various combinations of '`|`' (`choice`), '`,`' (`sequence`) and '`(`', '`)`' (`group`) constructs of DTD (XSD).

In general, there can occur two situations, so-called *grouping* and *degrouping*. We will depict them by simple rules listed in Figure 4.

$a?, b? \rightarrow (a, b)?$
$a?, b^* \rightarrow (a, b^+)?$
$a?, b? \rightarrow a\|b$
$a?, b^* \rightarrow a\|b^*$

Figure 4: Grouping and degrouping rules

In general, both types of rules transform the content models to more restrictive ones. Hence, naturally, we can apply the rules only in case the input data are valid also against the more restrictive version.

The algorithm for finding candidates for grouping and degrouping is similar to the previous case: For each of the REs that conform to the left hand sides of the rules in Figure 4, we need to check that the input data conform to their right hand sides as well. While parsing the documents in D, for each of the candidate schema fragment f and for each grouping/degrouping rule $r_1, r_2, ..., r_k$ we preserve the respective flags $\varphi_{r_1}(f), \varphi_{r_2}(f), ..., \varphi_{r_k}(f)$ carrying the information whether or not the instances of f in D conform to right hand side of the respective rule. At the beginning of the algorithm we set the flags $\varphi_{r_1}(f) = T, \varphi_{r_2}(f) = T, ..., \varphi_{r_k}(f) = T$. While parsing the documents, whenever we encounter a document instance that does not fulfill the right hand side of a rule r_i, we set the respective $\varphi_{r_i}(f) = F$. After parsing each $d \in D$ we can apply grouping and degrouping rules only in case the respective flag remains positive, i.e. there occurs no document instance in D that would not be valid against it.

Example 11 *Consider the following schema production:*

`E → A? B? C D*`

In case the document productions are as follows:

`E → A B C D D D D`
`E → A B C`
`E → C D D`

we can apply operation grouping resulting in the following schema production:

`E → (A B)? C D*`

On the other hand, if the document productions are as follows:

`E → A C D D D D`
`E → B C`
`E → A C D D`

we can apply operation degrouping resulting in the following schema production:

`E → (A | B) C D*`

Note that we can use a much wider set of grouping and degrouping rules, depending on user requirements on schema correction. However, if the set of rules is too wide, it can generate too large set of possible combinations and, hence, we should use the classical merging state algorithm instead, since it would enable one to find the suboptimal solution efficiently. The decision remains in hands of a user and his/hers requirements for schema specialization.

Refactorization A natural last step of each of schema inference method is *refactoring*, i.e. improving readability and simplifying structure while preserving the functionality of the resulting schema. A demonstrative set of rules is depicted in Figure 5.

$a?? \rightarrow a?$
$a^{++} \rightarrow a^+$
$a^{**} \rightarrow a^*$
$a^*? \rightarrow a^*$
$a?^* \rightarrow a^*$
$a^{+*} \rightarrow a^*$
$a^{*+} \rightarrow a^*$
$a?^+ \rightarrow a^*$
$a^+? \rightarrow a^*$
$aa^* \rightarrow a^+$
$a^+a^* \rightarrow a^+$
$a?a^+ \rightarrow a^*$
$(ab)\|(ac) \rightarrow a(b\|c)$

Figure 5: Merging of operators

The specified rules enable one to remove duplicate occurrence operators, to merge sequences of distinct occurrence operators into a single one, to merge sequences of the same fragments, to avoid nondeterministic content models etc. Naturally, there can exist various other sets of refactorization rules depending on the requirements of respective applications.

5.3 Complexity

The proposed algorithm consists of schema correction and schema specialization. Schema correction is performed using the ACO heuristic whose complexity in the worst case is limited by the allowed number of iterations, number of steps of an ant and number of ants, i.e. $O(N_{iter} \times N_{ant} \times card(\Lambda))$. On the other hand, schema specialization is based on linear parsing of XML documents in D, i.e. $O(|D| \times max_{i=1}^{card(D)}(|d_i|))$, where $|d_i|$ denotes the number of elements and attributes in document d_i. Naturally, if we decide to perform the schema specialization using the classical ACO heuristic, it will have the same complexity as the correction algorithm. Consequently, the ACO heuristic enables one to search a greater space of possible solutions and, hence to find a better solution, but at the cost of efficiency.

In general, even if we use the ACO heuristic instead of the proposed simple heuristic strategy, the inference algorithm will be still more efficient than the original one that does not take into account the original schema. The reason is that we do not begin with simple PTA, but with an XML schema that is at least partly correct and enough generalized. If we consider the worst case, i.e. the case when the input schema s_{orig} is completely incorrect, the proposed approach builds a classical PTA from the given XML documents and merges them. The unused schema fragments of s_{orig} are then simply removed in linear time.

6 Conclusion

The aim of this paper was to propose an algorithm for automatic inference of an XML schema which exploits an additional information – the original, possibly incorrect or too general schema. We have proposed a two-step approach. Firstly, we correct the schema so that the input XML documents are valid against it, whereas the new schema preservers the information carried by the original one as much as possible. Secondly, we propose a heuristic approach that enables one to specify the schema more precisely. In particular, we propose two alternatives which differentiate in efficiency and quality of the result.

Currently, we are dealing with throughout implementation of the proposal, since we intend to apply it on a representative set of real-world XML data as well as to exploit it in existing applications (Dokulil et al. 2007). We assume that similarly to paper (Bex et al. 2007) we will discover that the real-world data need special treatment since they do not involve all the constructs allowed by the W3C specifications.

Our future work we will focus mainly on integrating of user interaction which is the key aspect in case multiple solutions are available and searching the optimum is made only using heuristics. In fact, there seems to be no work, that would deal with this topic in detail, taking into account reasonable requirements for user's skills and amount of decisions to be made. Next, we will deal with inference of further XSD specific features, in particular integrity constraints. And, finally, since for further processing of the respective XML data also constraints that cannot be expressed in XSD may be useful, we will try to get also beyond its expressive power.

References

Ahonen, H. (1996), *Generating Grammars for Structured Documents Using Grammatical Inference Methods*, Technical Report A-1996-4, Dept. of Computer Science, University of Helsinki.

Berstel, J. & Boasson, L. (2000), XML Grammars, *in* 'Mathematical Foundations of Computer Science', LNCS, Springer, pp. 182–191.

Bertino, E., Guerrini, G., Mesiti, M. & Tosetto, L. (2002), Evolving a Set of DTDs According to a Dynamic Set of XML Documents, *in* 'EDBT '02', Springer-Verlag, London, UK, pp. 45–66.

Bex, G. J., Neven, F. & den Bussche, J. V. (2004), DTDs versus XML Schema: a Practical Study, *in* 'WebDB'04', ACM, New York, NY, USA, pp. 79–84.

Bex, G. J., Neven, F. & Vansummeren, S. (2007), Inferring XML Schema Definitions from XML Data, *in* 'VLDB'07', ACM, Vienna, Austria, pp. 998–1009.

Biron, P. V. & Malhotra, A. (2004), *XML Schema Part 2: Datatypes (Second Edition)*, W3C.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. & Yergeau, F. (2006), *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, W3C.

Dokulil, J., Tykal, J., Yaghob, J. & Zavoral, F. (2007), Semantic Web Repository And Interfaces, *in* 'SEMAPRO'07', IEEE Computer Society, Los Alamitos, USA, pp. 223–228.

Dorigo, M., Birattari, M. & Stutzle, T. (2006), *Ant Colony Optimization – Artificial Ants as a Computational Intelligence Technique*, Technical Report 2006-023, IRIDIA, Bruxelles, Belgium.

Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S. & Shim, K. (2000), XTRACT: a System for Extracting Document Type Descriptors from XML Documents, *in* 'SIGMOD'00', ACM, New York, NY, USA, pp. 165–176.

Gold, E. M. (1967), 'Language Identification in the Limit', *Information and Control* **10**(5), 447–474.

Grunwald, P. (2005), *A Tutorial Introduction to the Minimum Description Principle*. `http://homePAGES.cwi.nl/~pdg/ftp/mdlintro.pdf`.

Guerrini, G., Mesiti, M. & Sorrenti, M. A. (2007), XML Schema Evolution: Incremental Validation and Efficient Document Adaptation, *in* 'XSym'07', Springer, Vienna, Austria, pp. 92–106.

Mignet, L., Barbosa, D. & Veltri, P. (2003), The XML Web: a First Study, *in* 'WWW'03', ACM, New York, NY, USA, pp. 500–510.

Mlynkova, I., Toman, K. & Pokorny, J. (2006), Statistical Analysis of Real XML Data Collections, *in* 'COMAD'06', Tata McGraw-Hill, New Delhi, India, pp. 20–31.

Moh, C.-H., Lim, E.-P. & Ng, W.-K. (2000), Re-engineering Structures from Web Documents, *in* 'DL'00', ACM, New York, NY, USA, pp. 67–76.

Su, H., Kramer, D., Chen, L., Claypool, K. & Rundensteiner, E. A. (2001), XEM: Managing the Evolution of XML Documents, *in* 'RIDE '01', IEEE Computer Society, Washington, DC, USA, p. 103.

Tan, M. & Goh, A. (2005), Keeping Pace with Evolving XML-Based Specifications, *in* 'Current Trends in Database Technology – EDBT '04 Workshops', Springer, Heraklion, Crete, Greece, pp. 280–288.

Thompson, H. S., Beech, D., Maloney, M. & Mendelsohn, N. (2004), *XML Schema Part 1: Structures (Second Edition)*, W3C.

Vosta, O., Mlynkova, I. & Pokorny, J. (2008), Even an Ant Can Create an XSD, *in* 'DASFAA'08', LNCS, Springer, pp. 35–50.

Wong, R. K. & Sankey, J. (2003), *On Structural Inference for XML Data*, Technical Report UNSW-CSE-TR-0313, School of Computer Science, University of NSW.