Even an Ant Can Create an XSD[⋆]

Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný

Charles University, Faculty of Mathematics and Physics,
Department of Software Engineering,
Malostranské nám. 25, 118 00 Prague 1, Czech Republic
`ondra.vosta@centrum.cz,irena.mlynkova@mff.cuni.cz,`
`jaroslav.pokorny@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 its 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. In particular, we focus on new features of XML Schema language and we propose an algorithm which is an improvement of a combination of verified approaches that is, at the same time, enough general and can be further enhanced. Using a set of experiments we illustrate the behavior of the algorithm on both real-world and artificial XML data.

1 Introduction

Without any doubt the XML [17] 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 [17] and XML Schema [16, 32]. 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 (such as user-defined data types, inheritance, substitutability, etc.), and reusability of parts of a schema or whole schemes. On the other hand, statistical analyses of real-world XML data show that a significant portion of real XML documents (52% [26] of randomly crawled or 7.4% [27] 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% [26] of randomly crawled or 38% [27] of semi-automatically collected XML documents) and even if they are used, they often (in 85% of cases [14]) define so-called *local tree grammars* [29], i.e. languages that can be defined using DTD as well.

⋆ This work was supported in part by Czech Science Foundation (GAČR), grant number 201/06/0756.

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

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

In reaction to this situation a new research area of automatic construction 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. It means that the set of document instances of the inferred schema is not too broad in comparison with the provided set of sample data but, also, it is not equivalent to the sample set. Currently there are several proposals of respective algorithms (see Section 2), but there is still a space for further improvements. Primarily, almost all of the existing approaches focus on construction of regular expressions of DTD (though sometimes expressed in XML Schema language) which are relatively simple. Hence, our key aim is to focus on new constructs of XML Schema language, such as, e.g., unordered sequences of elements or elements having the same name but different structure, that enable to create more realistic schemes. For this purpose we propose an algorithm which is an improvement of a combination of verified approaches that is, at the same time, enough general and can be easily further enhanced. Such algorithm will enable to increase popularity and exploitation of XML Schema which fails especially in complexity of schema definition. Having an automatic generator of XSDs, a user is not forced to specify the whole schema manually, since the inferred one can serve, at least, as a good initial draft and thus eases the definition.

The paper is structured as follows: Section 2 overviews existing works focussing on automatic construction of XML schemes. Section 3 introduces the proposed algorithm in detail and Section 4 discusses the results of experimental testing. Finally, Section 5 provides conclusions and outlines possible future work.

2 Related Work

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

Heuristic approaches [21, 28, 33] are based on experience with manual construction of schemes. Their results do not belong to any class of grammars and they are based on generalization of a trivial schema (similar to the classical *dataguide* [23]) 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 trivial schema until a satisfactory solution is reached [28, 33] and methods which generate a huge number of candidates and then choose the best one [21]. While in the first case the methods are threatened by a wrong step which can cause generation of a suboptimal schema, in the latter case they have to cope with space overhead and specify a reasonable function for evaluation quality of the candidates. A special type of heuristic methods are so-called *merging state algorithms* [33]. They are based on the idea of searching a space of all possible generalizations, i.e. XML schemes, of the given XML documents represented using a prefix tree automaton. By merging its states they construct the optimal solution. In fact, since the space is theoretically infinite, only a reasonable subspace of possible solutions is searched using various heuristics.

On the other hand, methods based on *inferring of a grammar* [11, 19] are based on theory of languages and grammars and thus ensure a certain degree of quality of the result. They are based on the idea that we can view an XML schema as a grammar and an XML document valid against the schema as a word generated by the grammar. Although grammars accepting XML documents are in general context-free [13], the problem can be reduced to inferring of a set of regular expressions, each for a single element (and its subelements). But, since according to Gold's theorem [22] regular languages are not identifiable only from positive examples (i.e. sample XML documents which should conform to the resulting schema), the existing methods exploit various other information such as, e.g., predefined maximum number of nodes of the target automaton representing the schema, user interaction specifying (in)correctness of a partial result, restriction to an identifiable subclass of regular languages, etc.

Finally, let us mention that probably the first approach which focuses directly on true XML Schema constructs has only recently been published in [15]. The authors focus on the importance of context of elements for inferring of XSDs. They define a subclass of XSDs which can be learned from positive examples and focus especially on constructs which are used in real-world XML schemes.

3 Proposed Algorithm

As we have mentioned, almost all the described papers focus on constructs of DTD which are quite restricted. All the elements are defined on the same level thus it is impossible to define two elements having the same name but different structure. But this requirement can have quite reasonable usage due to homonymy of element names. For instance, we can have an XSD of a library where each book as well as author have name. In the former case it can be only a simple string, while in the latter case the name of a human can consist of a couple of elements each having its own semantics – see Fig. 1.

```
<book>
  <name>Sherlock Holmes</name>
</book>
```

```
<author>
  <name>
    <first>Arthur</first>
    <middle>Conan</middle>
    <last>Doyle</last>
  </name>
</author>
```

Fig. 1. Elements with the same name but different structure

Another interesting XSD feature is element `all`, i.e. unordered sequence of subelements, which allows to use an arbitrary permutation of the specified elements in respective document instances. This operator does not increase the expressive power of DTD operators, but it significantly simplifies the notation of used regular expressions, in particular in case of data-centric documents.

Our proposal focuses mainly on these two constructs. It is inspired especially by papers [33] and [21]. From the former one we "borrow" the idea of the Ant Colony Optimization (ACO) heuristic [18] for searching the space of possible generalizations, from the latter one we use a modification of the MDL principle [24] of their evaluation. The ACO heuristic enables to join several approaches into a single one and thus to infer more natural XML schemes. The MDL (Minimum Description Length) principle enables their uniform evaluation.

The main body of the algorithm can be described as follows: Firstly, each document D from the input set I_D is transformed into tree T. The trees hence form a set of input *document trees* I_T. (Note that for simplicity we omit attributes that can be considered as special types of elements.)

Definition 1. *A* document tree *of a document D is a directed graph $T = (V_T, E_T)$, where V_T is a set of nodes corresponding to element instances in D, and E_T is a set of edges s.t. $\langle a, b\rangle \in E_T$ if b is a (direct) subelement of a in D.*

Now, for each distinct element name in I_T, we perform clustering of respective element instances (see Section 3.1) with regard to the above described XSD feature. Then, for each cluster we generalize the trivial schema accepting purely the items in the cluster (see Section 3.2). And finally, we rewrite all the generalized schemes into XSD syntax (see Section 3.3). All the three steps involve a kind of improvement of current approaches. In the first case the existing works simply group elements on the basis of their names, elements with the same name but different structure are not supported and, hence, the result can be misleading. We solve this problem using clustering of elements on the basis of their context and structure. In the second case we propose a general combination of the best of the existing approaches which exploit some of the pure XML Schema constructs and can be easily extended. And in the final step we are able to output XSDs involving not only DTD constructs but also some of the purely XML Schema ones.

3.1 Clustering of Elements

To cluster elements having the same name but different structure we need to specify the similarity measure and the clustering algorithm.

Similarity of Elements As an XML element e can be viewed as a subtree T_e (in the following text denoted as an *element tree*) of corresponding document tree T, we use a modified idea of *tree edit distance*, where the similarity of trees T_e and T_f is expressed using the number of edit operations necessary to transform T_e into T_f (or vice versa). The key aspect is obviously the set of allowed edit operations. Consider two simple operations – adding and removal of a leaf node (and respective edge). As depicted in Fig. 2, such similarity is not suitable, e.g., for recursive elements. The example depicts two element trees of element a

having subelement i having subelement j having subelement k which contains either subelement z or again i. With the two simple edit operations the edit distance would be 4, but since the elements have the same XML schema we would expect the optimal distance of 0.

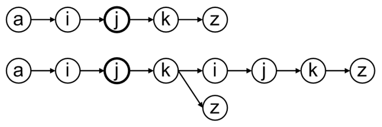

Fig. 2. Tree edit distance of recursive elements

Therefore, for our purpose we exploit a similarity measure defined in [30] which specifies more complex XML-aware tree edit operations involving operations on whole subtrees, each having its cost, as follows:

- *Insert* – a single node n is inserted to the position given by parent node p and ordinal number expressing its position among subelements of p
- *Delete* – a leaf node n is deleted
- *Relabel* – a node n is relabeled
- *InsertTree* – a whole subtree T is inserted to the position given by parent node p and ordinal number expressing position of its root node among subelements of p
- *DeleteTree* – a whole subtree rooted at node n is deleted

As it is obvious, for given trees T_e and T_f there are usually several possible transformation sequences for transforming T_e into T_f. A natural approach is to evaluate all the possibilities and to choose the one with the lowest cost. But such approach can be quite inefficient. Thus authors of the approach propose so-called *allowable sequences* of edit operations, which significantly reduce the set of possibilities and, at the same time, speed up their cost evaluation.

Definition 2. *A sequence of edit operations is* allowable *if it satisfies the following two conditions:*

1. *A tree T may be inserted only if T already occurs in the source tree T_e. A tree T may be deleted only if it occurs in the destination tree T_f.*
2. *A tree that has been inserted via the InsertTree operation may not subsequently have additional nodes inserted. A tree that has been deleted via the DeleteTree operation may not previously have had children nodes deleted.*

The first restriction forbids undesirable operations like, e.g., deleting whole T_e and inserting whole T_f, etc., whereas the second one enables to compute the costs of the operations efficiently. The evaluating algorithm is based on the idea of determining the minimum cost of each required insert of every subtree of T_e and delete of every subtree of T_f using a simple bottom-up procedure.

Hence, in the following text we assume that we have a function $dist(T_e, T_f)$ which expresses the edit distance of XML trees T_e and T_f.

Clustering Algorithm For the purpose of clustering elements we use a modification of *mutual neighborhood clustering (MNC) algorithm* [25]. We start with initial clusters $c_1, c_2, ...c_K$ of elements given by the equivalence of their context.

Definition 3. *A* context *of an element e in a document tree $T = (V_T, E_T)$ is a concatenation of element names $e_0 e_1 ... e_N$, where e_0 is the root node of T, $e_N = e$, and $\forall i \in (1, N) : \langle e_{i-1}, e_i \rangle \in E_T$.*

Thus, the initial clustering is based on a natural assumption that elements having the same context (and element name) are likely to have the same schema definition. The initial clusters are then merged on the basis of element structure using the tree edit distance *dist*. Firstly, $\forall i \in (1, K)$ we determine a representative element r_i of cluster c_i. Then, for each pair of $\langle r_i, r_j \rangle$ s.t. $i \neq j; i, j \in (1, K)$ we determine tree edit distance $dist(T_{r_i}, T_{r_j})$ of the respective trees. The MNC algorithm is parameterized by three parameters – minimum distance $dist_{MIN}$, maximum distance $dist_{MAX}$, and factor F – and exploits the definition of *mutual neighborhood*:

Definition 4. *Let T_e and T_f be two element trees, where T_e is i-th closest neighbor of T_f and T_f is j-th closest neighbor of T_e. Then* mutual neighborhood *of T_e and T_f is defined as $MN(T_e, T_f) = i + j$.*

The MNC algorithm places two element trees T_{r_i} and T_{r_j} into the same group if $dist(T_{r_i}, T_{r_j}) \leqslant dist_{MIN}$ or ($dist(T_{r_i}, T_{r_j}) \leqslant dist_{MAX}$ and $MN(T_{r_i}, T_{r_j}) \leqslant F$), resulting is a set of clusters $c_1, c_2, ..., c_L$ (where $L \leqslant K$) of elements grouped on the basis of their context and structure.

3.2 Schema Generalization

Now, for each cluster of elements $c_i; i \in (1, L)$ we infer an XML schema which "covers" all the instances in c_i and, at the same time, is still reasonably general. We speak about *generalization* of a trivial schema accepting purely the given set of elements. We can view the problem as a kind of optimization problem [12].

Definition 5. *A model $P = (S, \Omega, f)$ of a* combinatorial optimization problem *consists of a search space S of possible solutions to the problem (so-called* feasible region*), a set Ω of constraints over the solutions, and an* objective function *$f : S \to \mathbb{R}_0^+$ to be minimized.*

In our case S consists of all possible generalizations of the trivial schema. As it is obvious, S is theoretically infinite and thus, in fact, we can search only for a reasonable suboptimum. Hence, we use a modification of ACO heuristic [18]. Ω is given by the features of XML Schema language we are focussing on. In particular, apart from DTD concatenation (","), exclusive selection ("|"), and iteration ("?", "+", and "*"), we want to use also the `all` operator of XML Schema representing unordered concatenation (for simplicity denoted as "&"). And finally, to define f we exploit the MDL principle [24].

Ant Colony Optimization The ACO heuristic is based on observations of nature, in particular the way ants exchange information they have learnt. A set of artificial "ants" $A = \{a_1, a_2, ..., a_{card(A)}\}$ search the space S trying to find the optimal solution $s_{opt} \in S$ s.t. $f(s_{opt}) \leqslant f(s); \forall s \in S$. In i-th iteration each $a \in A$ searches a subspace of S 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.

We modify the algorithm in two key aspects. Firstly, we change the heuristic on whose basis the ants search S to produce more natural schemes with the focus on XML Schema constructs. And secondly, we add a temporary negative feedback which enables to search a larger subspace of S. The idea is relatively simple – whenever an ant performs a single step, it spreads a reasonable negative feedback. The difference is that the positive feedback is assigned after i-th iteration is completed, i.e. all ants die, to influence the behavior of ants from $(i + 1)$-st iteration. The negative feedback is assigned immediately after a step is performed, i.e. it influences behavior of ants in i-th iteration and at the end of the iteration it is zeroed. The algorithm terminates either after a specified number of iterations N_{iter} or if $s'_{opt} \in S$ is reached s.t. $f(s'_{opt}) \leqslant T_{max}$, 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 movements, their evaluation using f, and execution of one of the candidate steps. The executed step is selected randomly on the basis of probability given by f.

Generating a Set of Possible Movements Each element in cluster c_i can be viewed as a simple *grammar production rule*, where the left-hand side contains element name and right-hand side contains the sequence of names of its (direct) subelements. For instance the sample elements `name` in Fig. 1 can be viewed as

```
name → #PCDATA
name → first middle last
```

Hence, we can represent the trivial schema accepting purely elements from c_i as a *prefix-tree automaton* and by merging its states we create its generalizations. Each possible merging of states represents a single step of an ant. To generate a set of possible movements, i.e. possible generalizations, we combine two existing methods – k,h-context and s,k-string – together with our own method for inferring of & operators.

The *k,h-context method* [11] specifies an identifiable subclass of regular languages which assumes that the context of elements is limited. Then merging states of an automaton is based on an assumption that two states t_x and t_y of the automaton are identical (and can be merged) if there exist two identical paths of length k terminating in t_x and t_y. In addition, also h preceding states in these paths are then identical.

The *s,k-string method* [33] is based on Nerod's equivalency of states of an automaton assuming that two states t_x and t_y are equivalent if sets of all paths leading from t_x and t_y to terminal state(s) are equivalent. But as such condition is hardly checked, we can restrain to k-strings, i.e. only paths of length of k or paths terminating in a terminal state. The respective equivalency of states then depends on equivalency of sets of outgoing k-strings. In addition, for easier processing we can consider only s most probable paths, i.e. we can ignore singular special cases.

Finally, the idea of *inferring of* & *operators*, i.e. identification of unordered sequences of elements, can be considered as a special kind of merging states of an automaton. It enables to replace a set of ordered sequences of elements with a single unordered sequence represented by the & operator. We describe the process of inferring the unordered sequences in Section 3.3 in detail.

Evaluation of Movements The evaluation of moving from schema s_x to s_y, where $s_x, s_y \in S$ (i.e. the evaluation of merging states of the respective automaton) is defined as

$$mov(s_x, s_y) = f(s_x) - f(s_y) + pos(s_x, s_y) + neg(s_x, s_y)$$

where f is the objective function, $pos(s_x, s_y) \geqslant 0$ is the positive feedback of this step from previous iterations and $neg(s_x, s_y) \leqslant 0$ is the respective negative feedback. For the purpose of specification of f we exploit a modification of the MDL principle based on two observations [21]: A good schema should be enough general which is related to the low number of states of the automaton. On the other hand, it should preserve details which means that it enables to express document instances using short codes, since most of the information is carried by the schema itself and does not need to be encoded. Hence, we express the quality of a schema $s \in S$ described using a set of production rules $R_s = \{r_1, r_2, ..., r_{card(R_s)}\}$ as the sum of the size (in bits) of s and the size of codes of instances in cluster $c_i = \{e_1, e_2, ..., e_{card(c_i)}\}$ used for inferring of s.

Let O be the set of allowed operators and E the set of distinct element symbols in c_i. Then we can view right-hand side of each $r \in R_s$ as a word over $O \cup E$ and its code can be expressed as $|r| \cdot \lceil \log_2(card(O) + card(E)) \rceil$, where $|r|$ denotes length of word r. The size of code of a single element instance $e \in c_i$ is defined as the size of code of a sequence of production rules $R_e = \langle g_1, g_2, ..., g_{card(R_e)} \rangle$ necessary to convert the initial nonterminal to e using rules from R_s. Since we can represent the sequence R_e as a sequence of ordinal numbers of the rules in R_s, the size of the code of e is $card(R_e) \cdot \lceil \log_2(card(R_s)) \rceil$.

Note that since the ACO heuristic enables to use any inferring method to produce possible movements, i.e. schema generalizations, whereas the MDL principle does not consider the way they were inferred, the algorithm can easily be extended to other inferring methods as well as new constructs.

3.3 Inferring of Regular Expressions

The remaining open issue is how to infer the XSD, i.e. the set of regular expressions, from the given automaton. We exploit this information thrice – for evaluation of the objective function f, for generation of set of possible movements of an ant, and to output the resulting XSD. The automaton A can be represented as a directed graph, whose nodes correspond to states and edges labeled with symbols from the input alphabet E represent the transition function, i.e. an edge $\langle v_x, v_y \rangle$ labeled with symbol $e \in E$ denotes that the transition function from state v_x to state v_y on symbol e is defined. The task is to convert A to an equivalent regular expression. As for the DTD operators we use the well-known rules for transforming an automaton to a regular expression, similarly to [28, 33]. A brand new one approach we propose to identify a subgraph of A representing the `all` operator (& operator for simplicity).

In general the & operator can express the unordered sequence of regular expressions of any complexity such as, e.g., $(e_1|e_2) * \&e_3?\&(e_4, e_5, e_6)$, where $e_1, e_2, ..., e_6$ are element names. But, in general, the W3C recommendation of XML Schema language does not allow to specify so-called *nondeterministic data model*, i.e. a data model which cannot be matched without looking ahead. A simple example can be a regular expression $(e_1, e_2)|(e_1, e_3)$, where while reading the element e_1 we are not able to decide which of the alternatives to choose unless we read the following element. Hence, also the allowed complexity of unordered sequences is restricted. The former and currently recommended version 1.0 of XML Schema specification [16,32] allows to specify an unordered sequence of elements, each with the allowed occurrence of $(0, 1)$, whereas the allowed occurrence of the unordered sequence itself is of $(0, 1)$ too. The latter version 1.1 [20, 31], currently in the status of a working draft, is similar but the allowed number of occurrence of items of the sequence is $(0, \infty)$. In our approach we focus on the more general possibility, since it will probable soon become a true recommendation.

First-Level Candidates For the purpose of identification of subgraphs representing the allowed type of unordered sequences, we first define so-called *common ancestors* and *common descendants.*

Definition 6. *Let $G = (V, E)$ be a directed graph. A* common descendant *of a node $v \in V$ is a descendant $d \in V$ of v s.t. all paths traversing v traverse also d.*

Definition 7. *Let $G = (V, E)$ be a directed graph. A* common ancestor *of a node $v \in V$ is an ancestor $a \in V$ of v s.t. all paths traversing v traverse also a.*

Considering the example in Fig. 3 we can see that the common descendants of node 3 are nodes 6 and 7, whereas node 1 has no common descendants, since paths traversing node 1 terminate in nodes 7 and 9. Similarly, the common ancestor of node 6 is node 1. Note that in the former case there can exist paths which traverse d but not v (see node 3 and its common descendant 6), whereas in the latter case there can exist paths which traverse a but not v.

For the purpose of searching the unordered sequences we need to further restrict the Definition 7.

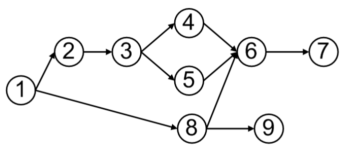

Fig. 3. An example of common ancestors and descendants

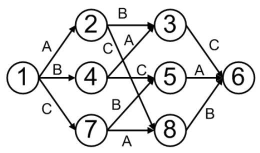

Fig. 4. Automaton P_3 – permutation of three items

Definition 8. *Let $G = (V, E)$ be a directed graph. A* common ancestor of a node $v \in V$ with regard to a node $u \in V$ *is an ancestor $a \in V$ of v s.t. a is a common ancestor of each direct ancestor of v occurring on path from u to v.*

For example, considering Fig. 3, the common ancestors of node 6 with regard to node 2 are nodes 2 and 3.

We denote the node v from Definition 6 or the node a from Definitions 7 and 8 as *input nodes* and their counterparts as *output nodes*. The set of nodes occurring on paths starting in an input node and terminating in an output node are called a *block*. Using the definitions we can now identify subgraphs which are considered as *first-level candidates* for unordered sequences. A node n_{in} is an input node of block representing a first-level candidate if

1. its out-degree is higher that 1,
2. the set of its common descendants is not empty, and
3. at least one of its common descendants, denoted as n_{out}, whose set of common ancestors with regard to n_{in} contains n_{in}.

The three conditions ensure that there are at least two paths leading from n_{in} representing at least two alternatives and that the block is *complete* meaning that there are no paths entering or leaving the block otherwise than using n_{in} or n_{out}. For example, considering Fig. 3, the only first-level candidate is subgraph consisting od nodes 3, 4, 5, 6.

Second-Level Candidates Having a first-level candidate we need to check it for fulfilling conditions of an unordered sequence and hence being a *second-level candidate*. As we know from the specification of XML Schema, such unordered sequence can consist only of simple elements which can repeat arbitrarily. Hence, firstly, we can skip processing of first-level candidates which contain subgraphs representing other operators or repetitions of more complex expressions. For the purpose of further checking we exploit the idea of similarity of graphs: For each $n \in \mathbb{N}$ we know the structure of the automaton P_n which accepts each permutation of n items having all the states fully merged. (An example of automaton P_3 is depicted in Fig. 4 for three items `A`, `B`, `C`.) Thus the idea is to compare the similarity of the first-level candidates with P_n automatons. But the situation is more complicated, since the W3C recommendation allows optional and repeatable elements in the unordered sequences, i.e. the allowed number of occurrences can be also 0 or greater than 1. Together with the fact that the input elements on whose basis the automaton was built do not need to contain all possible permutations, the candidate graph can have much different structure than any P_n. And naturally, we cannot check the similarity with all P_n automatons.

To solve the problem of multiple occurrence of an element, we temporarily modify the candidate graphs by replacing each repeatable occurrence of an element e with auxiliary element e' with single occurrence. For the purpose of similarity evaluation, we can consider the modified graph without repetitions. The repetitions will influence the resulting regular expression, i.e. the respective operator will be added. Then, we can denote the maximum path length l_{max} in the candidate graph and hence denote the size of the permutation. And finally, since both the modified candidate graph and $P_{l_{max}}$ graph are always acyclic, we can evaluate their similarity using a classical edit-distance algorithm.

We can also observe, that the candidate graph must be always a subgraph of $P_{l_{max}}$, otherwise we can skip its processing. Hence, the problem of edit distance is highly simplified. We use the following types of edit operations:

– Adding an edge between two existing nodes, and
– Splitting an existing edge into two edges, i.e. adding a new node and an edge.

The first operation is obvious and corresponds to the operation of adding paths that represent permutations which were not present in the source data and its cost is > 0. In the latter case the operation corresponds to adding an item of a permutation which was not present in the source data and its cost is 0, but it influences the resulting regular expression similarly to the above described multiple occurrence. Naturally, only nodes and edges which are present in $P_{l_{max}}$ and not in the candidate graph or edges from n_{in} to n_{out} can be added. From all the possible edit sequences we choose the one with the lowest total cost [30].

Extension of Candidates As we can see from the third condition which describes a first-level candidate, there can exist more candidates for the output node n_{out}. Hence, the remaining question is which of the candidates ought to be selected. In our proposal we use an approach which is, at the same time, a kind of a heuristic. The idea is based on the simple observation that each permutation of n items contains permutations of $n-1$ items. It can be seen in Fig. 4, which contains not only the P_3 automaton for the three items `A`, `B`, and `C`, but also all three P_2 automatons for pairs `A` and `B` (see subgraphs on nodes 1, 2, 3, 4), `B` and `C` (see subgraph on nodes 2, 3, 6, 8), `C` and `A` (see subgraph on nodes 1, 4, 5, 7).

Hence, for each node of the graph we firstly determine the set of all nodes which fulfill the condition of first-level candidate and we sort them totally in ascending order using the size of the respective block. The ordering determines the order in which we check the conditions of second-level candidates and enables to exploit the knowledge of previously determined second-level candidates, i.e. subgraphs corresponding to permutations for subsets of the given items.

4 Experimental Implementation

For the purpose of experimental analysis of the proposed algorithm we have created a prototype implementation called *SchemaMiner*. Since our approach results from existing verified approaches [21, 33] and focuses on XSD constructs which were not considered yet, a comparison with any other method would either lead to similar results or would be quite unfair. Thus we rather compare our approach with existing real-world XSDs. From the available real-world XML documents we have selected subsets having an XSD and classified them on the basis of their structure. For each collection of documents we have inferred the schema using *SchemaMiner* and analyzed the results.

Since the real-world XML documents did not cover all the features of the proposed algorithm, we have also generated a set of artificial documents which enable to illustrate the remaining properties. Hence, a natural question may arise whether these constructs are worth considering since they do not occur in real applications quite often. As we have mentioned in the Introduction, the general problem of real-world XML data is that they exploit only a small part of all constructs allowed by the W3C specifications. On one hand, XML data processing approaches can exploit this knowledge and focus only on these common constructs. But, on the other hand, it is also necessary to propose methods which help users to use also more complex tools which enable to describe and process XML data more efficiently. In our case it means to find an XSD which describes the selected situation more precisely.

4.1 Real-World XML Documents

We have classified the used XML documents according to their structure into the following categories:

– **Category 1**: Documents having very simple and general structure of type $(e_1|e_2|...|e_n)*$. They do not exploit optional elements, deeper hierarchy of exclusive selections, or sequences of elements. (e.g. [1])
– **Category 2**: These documents exploit purely optional elements, sequences of elements, and repetitions. (e.g. [2, 3])
– **Category 3**: These documents exploit all constructs of DTD, i.e. exclusive selections, repetitions, optional elements, and sequences of elements. (e.g. [4–6])
– **Category 4**: These documents have fairly regular structure suitable for storing into relational databases. The root element typically contains repetition of an element corresponding to n-tuple of simple data in fixed order. (e.g. [2])
– **Category 5**: These documents exploit the `all` construct of XSDs. (e.g. [7])
– **Category 6**: Since each XSD is at the same time an XML document and there exists an XSD of XML Schema language, we have included also XSDs, i.e. XML documents describing structure of other XML documents. (e.g. [8–10])

4.2 Results of Experiments

Comparing the difference between inferred and real-world XML schemes we have found several interesting observations. The most striking difference was inferring of less general schema in all the cases. This finding corresponds to the results of statistical analyses which show that XML schemes are usually too general [27]. Obviously, if a structural aspect is not involved in the sample XML documents, it can hardly be automatically generated. We can even consider this feature as an advantage to specify a more precise schema. On the other hand, another interesting difference was generalization of the inferred schema by setting the interval of repetition to $(0, \infty)$ instead of $(1, \infty)$, i.e. enabling to omit an element in case it should not be omitted. This situation occurs usually in case of repetition of a simple sequence of elements, in particular in category 4. But in this category such behavior is rather natural than harmful. And another significant difference was "inefficient" notation of the regular expressions, e.g., in case of expressions of the form $((e_1, e_2, e_3)|e_2)$ that could be rewritten to $(e_1?, e_2, e_3?)$, though this expression is not equivalent. This property is given by the features of the algorithm which directly transforms an automaton to a regular expression and indicates that additional optimization or user interaction would be useful.

Using category 5 we wanted to focus especially on the unordered sequences. But, unfortunately, the amount of real-world XML data was so small that the results were not usable. That is why we have decided to use artificial data (see Section 4.3). And a similar problem occurred in case of elements with the same name but different structure.

In case of the category 6, i.e. XSDs, we have divided the source data into two groups – a set of XSDs randomly downloaded from various sources and a set of XSDs inferred by the algorithm. The result in case of the first set was highly influenced by the variety of input data and the fact that XML Schema provides plenty of constructs which enable to specify the same structure in various ways. Nevertheless, the resulting schema was still recognizable as the XML Schema XSD. In case of the latter set of XSDs the result was naturally much better since the data came from the same source. An example is depicted in Fig. 5, where we can see the inferred XSD fragment of element `choice` of XML Schema.

4.3 Artificial XML Documents

As we have mentioned, to analyze all the features of the algorithm we have prepared a set of artificial XML documents. In particular we have focused on the occurrence of unordered sequences of elements and the occurrence of elements with the same name but different structure.

Permutated Set We have generated several sets of XML documents which differentiate in two aspects – the size of the permutated set and the percentage of permutations represented in the documents, i.e. having a set of n elements, the percentage of $n!$ of their possible orders.

```
<xs:complexType name="Txs:choice1" >
 <xs:choice minOccurs="1" maxOccurs="1">
  <xs:sequence minOccurs="1" maxOccurs="1">
   <xs:element name="xs:element" type="Txs:element1" minOccurs="1" maxOccurs="1"/>
   <xs:element name="xs:element" type="Txs:element1" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:sequence minOccurs="1" maxOccurs="1">
   <xs:element name="xs:sequence" type="Txs:sequence1" minOccurs="1" maxOccurs="1"/>
   <xs:element name="xs:sequence" type="Txs:sequence1" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:sequence minOccurs="1" maxOccurs="1">
   <xs:element name="xs:element"   type="Txs:element1"   minOccurs="1" maxOccurs="1"/>
   <xs:element name="xs:element"   type="Txs:element1"   minOccurs="0" maxOccurs="unbounded"/>
   <xs:element name="xs:sequence" type="Txs:sequence1"  minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
 </xs:choice>
</xs:complexType>
```

Fig. 5. An example of generated XSD

According to our results the size of set has almost no impact on the resulting schema, whereas the percentage of permutations is crucial. Particular results are depicted in Table 1 for the size of the set of 3, 4, and 5 (for more than 5 items the results were almost the same) and the percentage of permutations of 10, 20, ..., 100%. Value `no` denotes that the permutation operator did not occur in the resulting schema; value `partly` denotes that it occurred, but not for the whole set; value `yes` denotes that it occurred correctly.

Size of	**Percentage of permutations**									
the set	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
3	`no`	`partly`	`no`	`partly`	`no`	`no`	`partly`	`yes`	`yes`	`yes`
4	`no`	`yes`	`partly`	`yes`	`partly`	`partly`	`partly`	`yes`	`yes`	`yes`
5	`no`	`no`	`partly`	`partly`	`partly`	`partly`	`partly`	`yes`	`yes`	`yes`

Table 1. Influence of percentage of permutations

Elements with Different Content Similarly we have created XML documents containing an element with the same name but different structure in various contexts. The key parameter of the testing sets was the percentage of the same subelements within the element. The experiments showed that the algorithm behaves according to the expectations and joins two elements if they "overlap" in more than 50% of their content.

5 Conclusion

The aim of this paper was to propose an algorithm for automatic construction of an XML schema for a given sample set of XML documents exploiting new constructs of XML Schema language. In particular we have focussed on unordered sequences allowed by the `all` element and the ability to specify elements having the same name but different content. We have proposed a hybrid algorithm that combines several approaches and can be easily further enhanced. Our main motivation was to increase the exploitation of XSDs providing a reasonable draft of an XML schema that can be further improved manually by a user, if necessary.

Our future work will focus on further exploitation of other XML Schema constructs, such as, e.g., element groups, attribute groups, or inheritance, i.e. reusability of parts of a schema that can increase naturalness of the result. This idea is highly connected with our second future improvement that will focus on user interaction. This way we can ensure more realistic and suitable results than using a purely automatic reverse engineering. Even our current approach can be extended using user interaction in several steps. The first one is the process of clustering, where a user can specify new clusters, e.g., on the basis of semantics of elements and attributes. A second example of exploitation of user-provided information can be a set of negative examples, i.e. XML documents which do not conform to the target XML schema. Such documents would influence steps of ants, in particular the objective function and, hence, enable to create better result. And, finally, the user interaction can be exploited directly for specifying steps of ants, enabling to find the optimal solution more efficiently.

References

1. Available at: `http://arthursclassicnovels.com/`.
2. Available at: `http://www.cs.wisc.edu/niagara/data.html`.
3. Available at: `http://research.imb.uq.edu.au/rnadb/`.
4. Available at: `http://www.assortedthoughts.com/downloads.php`.
5. Available at: `http://www.ibiblio.org/bosak/`.
6. Available at: `http://oval.mitre.org/oval/download/datafiles.html`.
7. Available at: `http://www.rcsb.org/pdb/uniformity/`.
8. Available at: `http://www.eecs.umich.edu/db/mbench/`.
9. Available at: `http://arabidopsis.info/bioinformatics/narraysxml/`.
10. Available at: `http://db.uwaterloo.ca/ddbms/projects/xbench/index.html`.
11. H. Ahonen. *Generating Grammars for Structured Documents Using Grammatical Inference Methods.* Report A-1996-4, Dep. of Computer Science, University of Helsinki, 1996.
12. R. Bartak. *On-Line Guide to Constraint Programming.* 1998. `http://kti.mff.cuni.cz/~bartak/constraints/`.
13. J. Berstel and L. Boasson. XML Grammars. In *Mathematical Foundations of Computer Science*, volume 1893 of *Lecture Notes in Computer Science*, pages 182–191. Springer, 2000.
14. G. J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: a Practical Study. In *WebDB '04: Proc. of the 7th Int. Workshop on the Web and Databases*, pages 79–84, New York, NY, USA, 2004. ACM Press.
15. G. J. Bex, F. Neven, and S. Vansummeren. XML Schema Definitions from XML Data. In *VLDB '07: Proc. of the 33rd Int. Conf. on Very Large Data Bases*, pages 998–1009, Vienna, Austria, 2007. ACM Press.
16. P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes (Second Edition).* W3C, 2004. `http://www.w3.org/TR/xmlschema-2/`.

17. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, 2006.
18. M. Dorigo, M. Birattari, and T. Stutzle. *Ant Colony Optimization – Artificial Ants as a Computational Intelligence Technique*. Technical Report TR/IRIDIA/2006-023, IRIDIA, Bruxelles, Belgium, 2006.
19. H. Fernau. Learning XML Grammars. In *MLDM '01: Proc. of the 2nd Int. Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 73–87, London, UK, 2001. Springer-Verlag.
20. S. Gao, C. M. Sperberg-McQueen, and H. S. Thompson. *XML Schema Definition Language (XSDL) 1.1 Part 1: Structures*. W3C, 2007. `http://www.w3.org/TR/xmlschema11-1/`.
21. M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: a System for Extracting Document Type Descriptors from XML Documents. In *SIGMOD '00: Proc. of the 2000 ACM SIGMOD Int. Conf. on Management of Data*, pages 165–176, New York, NY, USA, 2000. ACM Press.
22. E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.
23. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB '97: Proc. of the 23rd Int. Conf. on Very Large Data Bases*, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
24. P.D. Grunwald. *A Tutorial Introduction to the Minimum Description Principle*. 2005. `http://homepages.cwi.nl/~pdg/ftp/mdlintro.pdf`.
25. A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall College Div, 1988.
26. L. Mignet, D. Barbosa, and P. Veltri. The XML Web: a First Study. In *WWW '03: Proc. of the 12th Int. Conf. on World Wide Web, Volume 2*, pages 500–510, New York, NY, USA, 2003. ACM Press.
27. I. Mlynkova, K. Toman, and J. Pokorny. Statistical Analysis of Real XML Data Collections. In *COMAD '06: Proc. of the 13th Int. Conf. on Management of Data*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill Publishing Company Limited.
28. C.-H. Moh, E.-P. Lim, and W.-K. Ng. Re-engineering Structures from Web Documents. In *DL '00: Proc. of the 5th ACM Conf. on Digital Libraries*, pages 67–76, New York, NY, USA, 2000. ACM Press.
29. M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, 2005.
30. A. Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *WebDB '02: Proc. of the 5th Int. Workshop on the Web and Databases*, pages 61–66, Madison, Wisconsin, USA, 2002. ACM Press.
31. D. Peterson, P. V. Biron, A. Malhotra, and C. M. Sperberg-McQueen. *XML Schema 1.1 Part 2: Datatypes*. W3C, 2006. `http://www.w3.org/TR/xmlschema11-2/`.
32. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, 2004. `http://www.w3.org/TR/xmlschema-1/`.
33. R. K. Wong and J. Sankey. *On Structural Inference for XML Data*. Technical Report UNSW-CSE-TR-0313, School of Computer Science, The University of New South Wales, 2003.