Towards Inference of More Realistic XSDs

Irena Mlýnková, Martin Nečaský
Department of Software Engineering, Charles University in Prague, Czech Republic
{irena.mlynkova,martin.necasky}@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 works, whose aim is to infer as concise schema as possible, we focus on inferring of a more realistic result, i.e. a schema that is closer to human-written ones and bears more precise information. For this purpose we extend and combine the existing verified techniques (such as ACO heuristics or MDL principle) with a set of heuristics exploiting semantics of element/attribute names, thesauri or statistical analysis of input data. Using a set of examples we show and discuss advantages of our proposal.

Categories and Subject Descriptors

I.7.1 [**Document and Text Processing**]: Document and Text Editing – languages, document management

General Terms

Measurement, Algorithms

Keywords

XML Schema, XSD constructs, schema inference.

1. INTRODUCTION

Without any doubt the XML [6] 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 [6] and XML Schema [15, 5]. 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% [12] of randomly crawled or 7.4% [13] 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% [12] of randomly crawled or 38% [13] of semi-automatically collected XML documents) and even if they are used, they often (in 85% of cases [3]) 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 will focus on the persisting problem of inferring of realistic schemas, in particular XSDs. In all the existing papers the authors focus on inference of concise and precise regular expressions. This approach is sufficient for DTDs, but since an XSD can be expressed in various ways having different semantics, we can go even further. To achieve more realistic results we utilize and extend existing verified techniques, such as ACO heuristics or MDL principle, with exploitation of various additional information, such as semantics of element/attribute names, thesauri or statistical analysis of input data. Besides proposal of a brand new approach we also discuss advantages and disadvantages of other possible solutions that may come to mind. Using a set of examples we demonstrate the behavior and advantages of the proposal.

The paper is structured as follows: Section 2 overviews existing papers on automatic inference of XML schemas. Section 3 provides background information on XML Schema and states related classes of equivalence. Section 4 introduces the problem of schema inference, describes the unsolved aspects and discusses possible solutions. Section 5 describes the proposed solution in detail. And, finally, Section 6 provides conclusions and outlines possible future work.

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

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

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 [14, 17, 9] are based on experience with manual construction of schemas. Their result 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 grammar until a satisfactory solution is reached (e.g. [14, 17]) and methods which generate a number of candidates and then choose the optimal one (e.g. [9]). 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 specifying a reasonable function for evaluation of quality of the candidates.

On the other hand, methods based on *inferring of a grammar* [1, 4] 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 [10] regular languages are not identifiable only from positive examples (i.e. sample XML documents which should conform to the resulting schema), the existing methods need to exploit restriction to an *identifiable* subclass of regular languages.

3. XML SCHEMA

The constructs of XML Schema can be divided into *basic*, *advanced* and *auxiliary*. The basic constructs involve simple data types (`simpleType`), complex data types (`complexType`), elements (`element`), attributes (`attribute`), groups of elements (`group`) and groups of attributes (`attributeGroup`). Simple data types involve both built-in data types (except for `ID`, `IDREF`, `IDREFS`), such as, e.g., `string`, `integer`, `date` etc., as well as user-defined data types derived from existing simple types using `simpleType` construct. Complex data types enable one to specify both content models of elements and their sets of attributes. The content models can involve ordered sequences (`sequence`), choices (`choice`), unordered sequences (`all`), groups of elements (`group`) or their allowable combinations. Similarly, they enable one to derive new complex types from existing ones. Elements simply join simple/complex types with respective element names and, similarly, attributes join simple types with attribute names. And, finally, groups of elements and attributes enable one to globally mark selected schema fragments and exploit them repeatedly in various parts using so-called *references.* In general, basic constructs are present in almost all XSDs.

The set of *advanced* constructs involves type substitutability and substitution groups, identity constraints (`unique`, `key`, `keyref`) as well as related simple data types (`ID`, `IDREF`, `IDREFS`) and assertions (`assert`, `report`). Type substitutability and substitution groups enable one to change data types or allowed location of elements. Identity constraints enable one to restrict allowed values of elemets/attributes to unique/key values within a specified area and to specify references to them. Similarly, assertions specify additional conditions that the values of elements/attributes need to satisfy, i.e. they can be considered as an extension of simple types.

The set of *auxiliary* constructs involves wildcards (`any`, `anyAttribute`), external schemas (`include`, `import`, `redefine`), notations (`notation`) and annotations (`annotation`). Since they do not have a key impact on schema structure or semantics, we will not deal with them in the rest of the text.

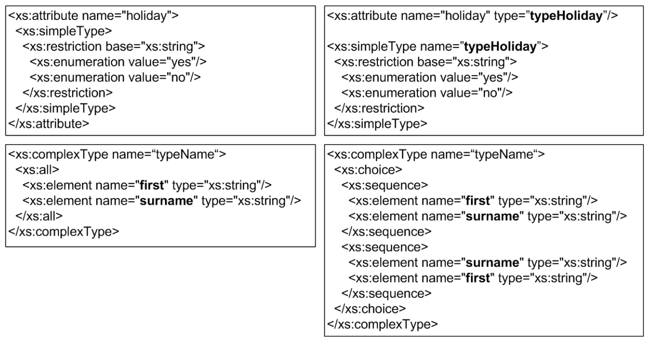
3.1 Equivalence of XSD Constructs

The XML Schema language contains plenty of "syntactic sugar", i.e. constructs that enable one to generate XSDs that have different structure but are *equivalent.*

Definition 1. *Let s_x and s_y be two XSD fragments. Let $\eta(s) = \{d$ s.t. d is an XML document fragment valid against $s\}$. Then s_x and s_y are* equivalent, *$s_x \sim s_y$, if $\eta(s_x) = \eta(s_y)$.*

Consequently, having a set χ of all XSD constructs, we can specify the quotient set $\chi/\sim$ of χ by $\sim$ and respective equivalence classes – see Table 1.

Example 1. *As depicted in Figure 1, for instance class C_{ST} specifies that there is no difference if a simple type is defined locally or globally, whereas class C_{Seq} expresses the equivalence between an unordered sequence of elements e_1, e_2, ..., e_l and a choice of their possible ordered permutations.*



```
<xs:attribute name="holiday">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="yes"/>
      <xs:enumeration value="no"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
```

```
<xs:attribute name="holiday" type="typeHoliday"/>

<xs:simpleType name="typeHoliday">
  <xs:restriction base="xs:string">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
  </xs:restriction>
</xs:simpleType>
```

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

```
<xs:complexType name="typeName">
  <xs:choice>
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="surname" type="xs:string"/>
    </xs:sequence>
    <xs:sequence>
      <xs:element name="surname" type="xs:string"/>
      <xs:element name="first" type="xs:string"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Figure 1: Examples for classes C_{ST} and C_{Seq}

In addition, each of the previously defined classes of $\sim$ equivalence can be represented using a selected *canonical representative* as listed in Table 1 as well. Note that each of the constructs not mentioned there forms a single class $C_1, C_2, ..., C_n$.

3.2 Relation to Automata and Grammars

An XML schema describing the allowed structure of XML documents is a context-free grammar [2], i.e. a grammar where nonterminals can be rewritten without regard to the context in which they occur.

Definition 2. A 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 $L(G)$.

Table 1: XSD equivalence classes of $\chi/\sim$

Class	Constructs	Canonical representative
C_{ST}	globally defined simple type, locally defined simple type	locally defined simple type
C_{CT}	globally defined complex type, locally defined complex type	locally defined complex type
C_{El}	referenced element, locally defined element	locally defined element
C_{At}	referenced attribute, locally defined attribute, attribute referenced via an attribute group	locally defined attribute
C_{ElGr}	content model referenced via an element group, locally defined content model	locally defined content model
C_{Seq}	unordered sequence of elements $e_1, e_2, ..., e_l$, choice of all possible ordered sequences of $e_1, e_2, ..., e_l$	choice of all possible ordered sequences of e_1, e_2, ..., e_l
C_{CTDer}	derived complex type, newly defined complex type	newly defined complex type
C_{SubGr}	elements in a substitution group γ, choice of elements in γ	choice of elements in γ
C_{Sub}	data types $\tau_1, \tau_2, ..., \tau_k$ derived from type τ, choice of content models defined in $\tau_1, \tau_2, ..., \tau_k, \tau$	choice of content models defined in $\tau_1, \tau_2, ..., \tau_k, \tau$

Definition 3. *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.

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

Definition 4. 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),* $\delta : Q \times \Sigma^* \to Q$ *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* $L(A)$*.*

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

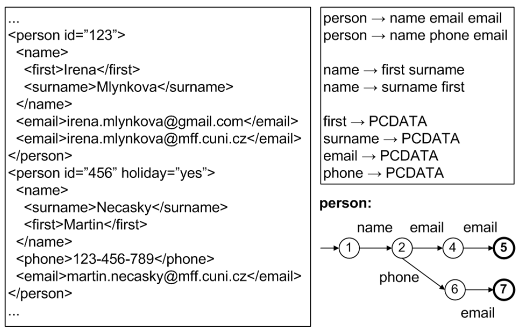
4. PROBLEM STATEMENT

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

Most of the existing approaches use the following strategy: For each occurrence of element $e \in I$ and its subelements $e_1, e_2, ..., e_k$ we construct a production $\vec{p}_e$ of the form $e \to e_1\ e_2\ ...\ e_k$.[3] 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.

[3]Attributes are often omitted for simplicity.

Example 2. *An example of an IG and PTA for element* `person` *is depicted in Figure 2.*



```
...
<person id="123">
  <name>
    <first>Irena</first>
    <surname>Mlynkova</surname>
  </name>
  <email>irena.mlynkova@gmail.com</email>
  <email>irena.mlynkova@mff.cuni.cz</email>
</person>
<person id="456" holiday="yes">
  <name>
    <surname>Necasky</surname>
    <first>Martin</first>
  </name>
  <phone>123-456-789</phone>
  <email>martin.necasky@mff.cuni.cz</email>
</person>
...
```

```
person → name email email
person → name phone email

name → first surname
name → surname first

first → PCDATA
surname → PCDATA
email → PCDATA
phone → PCDATA
```

Figure 2: An example of an 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. In particular, all the resulting schemas exploit only canonical representatives of classes of $\sim$, except for paper [16] which deals with class C_{seq}. Our aim is to enhance the strategies to generate more realistic XSDs that involve advanced constructs, such as inheritance (classes C_{CT}, C_{CTDer}) or globally defined fragments (classes C_{El}, C_{ElGr}, C_{At}), i.e. not only the canonical representatives, but also constructs a man would use. On one hand, these constructs can be characterized as "syntactic sugar", since a structurally equivalent schema can be expressed using canonical representatives. But, on the other hand, the schema is much more lucid and can bear additional semantic information that can be exploited in various schema-based approaches such as schema-driven XML-to-relational storage strategies, query optimization etc.

4.1 Possible Solutions

The naive approach towards our aim is to identify all possible subschemas that could be specified in an equivalent manner and to offer the options to a user. (S)he can either determine the selected constructs individually or specify that, e.g., globally defined complex types should be preferred where possible. However, this simple and straightforward approach has a big disadvantage. It requires a sophisticated user that is able to understand all the aspects of the processed XML data and is willing to make all the decisions. This requirement is possible for simple schemas involving a small amount of elements and attributes. But for more complex ones it is inappropriate, especially when the user is not acquainted with the semantics of the data.

Shared Fragments.

An improvement to the naive approach can be various heuristic rules that reduce the set of options. In particular, we can exploit the observation that there is only a small usage of global definition of schema fragments that are not *shared* among others. For instance, we can find out that elements e_x and e_y have a common subset of attributes $A = \{a_1, a_2, ..., a_m\}$. Hence, we can *outline* these attributes to a common global `attributeGroup`. And a similar observation can be done for single attributes, single elements, sets of elements as well as sets of both attributes and elements.

However, there can still be multiple choices for describing the outlined schema fragment. For instance, the set A of attributes could be outlined also using separate globally defined `attribute` constructs, or a combination of `attributeGroup` and `attribute`, as well as a globally defined `complexType`. Hence, we can further restrict this rule and always use as the globally defined fragment only the smallest schema construct that covers the shared items.

Although both the ideas are correct, the problem is that it can *merge* also schema fragments that have nothing in common but a subset of subelements and/or attributes. For instance, let us have an element `person` with attribute `id` and subelements `name`, `address` and `phone` and an element `book` with attribute `id` and subelements `name`, `address` and `authors`. The approach would lead to creating three complex types – `typeCommon` involving the common items and two derived types `typePerson` and `typeBook`, each adding the respective subelement. Since the two elements have nothing in common, such schema would obviously be semantically wrong. In fact it is only a structural "coincidence" that these types seem to be similar, therefore we need to exploit further information than only simple analysis of structure.

Element/Attribute Names' Semantics.

The solution to the problem is taking into account also the semantics of the data. In particular, we use a kind of a thesaurus that enables to discover semantic relations. Hence, we discover that the semantic similarity of words `person` and `book` is not high, but similarity of, e.g., `person`, `author` and `editor` is sufficient. And it also brings another advantage. Let us have two candidates for outlining – `employee` and `manager`. The thesaurus not only determines that they are related, but also that `employee` is a *broader term* to `manager`. Hence, instead of creating `typeCommon` and derived types `typeEmployee` and `typeManager`, we will create `typeEmployee` and a derived type `typeManager`.

However, even though this solution is nearly optimal, we can encounter several cases that need special treatment.

Firstly, it is reasonable to perform the merging only in case the merged sets are reasonably big. Naturally, we could merge even singletons, but it does not seem to have much usability. We can either let a user to specify the minimum size of the merged set or use a value derived from the average fan-out of the currently processed data set.

Secondly, we have to consider that there are several element/attribute names that occur quite commonly, such as `id`, `comment` etc. Hence, we should create a kind of a *stop list*, i.e. a list of element/attribute names that have only minor influence on determining the merged sets.

And, finally, there can occur situations when there are two elements that fulfill the above described conditions, but they still do not have nothing in common. Such situation occurs in case of homonymy of element names or too general terms, such as `article`, `item`, `part` etc. Consequently, we should consider also context of the elements, i.e. their ancestors and/or siblings.

Data Statistics.

If we want to go even further, we can even exploit the given XML documents more deeply. At first sight this idea seems to be pointless, since all the existing schema inference methods result from the given XML data. However, since their common aim is to infer the most concise and precise schema, whereas ours is to infer more realistic one, we can view the data from different points.

The first idea that could come to mind is to use the same approach as in case of exploitation of thesaurus, i.e. exploit similarity of element/attribute values. We can assume that since the textual values bear the information, they are even more reliable than the structural parts. But, contrary to the number of distinct element/attribute names, the assessment on the basis of data values is not as straightforward, since the value domains can be large and their intersections and similarities can be misleading.

Hence, our motivation results from ideas used in adaptive schema-driven XML-to-relational mapping strategies (e.g. [8]). They apply various XML-to-XML transformations on the input XML schema, evaluate them and select the optimal one. For this purpose, we can exploit almost any equation known for regular expressions. However, since the amount of options is again large and we would get to the same problem as described above, we need to limit ourselves to those that can be assessed as relevant. But, since we do not have an etalon in a set of XML queries like the adaptive mapping methods do, we will exploit etalons relevant to schema inference.

The existing inference methods apply merging rules ensuring conciseness and reasonable generalization, such as, e.g.

$$a, a, a, a, a = a^+$$
$$(a, b)|(a, c) = a, (b|c)$$

However, there are also other rules, that enable to provide less concise, but more realistic schemas. An example of such rule can be so-called *splitting repetitions*:

$$a^+ = a, a^*$$

or, in general, not performing selected merging states of an automaton.

For instance we can find out that in 95% of cases element `person` have two subelements `phone` at maximum and only in 5% of cases there occur elements `person` having more than five subelements `phone`. In the existing works the schema would be generalized to `phone+` or `phone*` although for most of the input XML documents it is too general. Consequently, if we exploit the above described equation we could preserve the first two occurrences of element `phone` and provide more realistic schema, i.e. `phone phone phone*`, bearing more precise information.

5. PROPOSED ALGORITHM

All the above discussed improvements can be applied on any of the existing schema inference approaches. Nevertheless, we will describe the extension of approach from [16] 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 PTA is viewed as a kind of optimization problem.

Definition 5. *A model* $M = (\Theta, \Omega, f)$ *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 $f : \Theta \rightarrow \mathbb{R}_0^+$ *to be minimized.*

In our case Θ consists of all possible generalizations of PTA. 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 [7]. Ω is given by the features of XML schema language we are focussing on, i.e. XSD constructs. And finally, to define f we will exploit a modification of the MDL principle [11].

5.1 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. $f(s_{opt}) \leqslant f(s); \forall s \in \Theta$. 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. $f(s'_{opt}) \leqslant T_{max}$, 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 continuations, their evaluation using f and execution of one of the candidate steps. The executed step is selected randomly with probability given by f. 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.

5.2 Generating a Set of Possible Continuations

A single step of an ant is represented using a modification of the current automaton starting with PTA. 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* [1] which merges states with same contexts (prefixes) or s, k-*string* [17] which merges states with same suffixes.

Example 3. *Two examples of merging are depicted in Figure 3. In the former case we can merge states 4 and 5, because they have the same prefixes of length 1 (i.e. edge* `address`*). In the latter case, we can merge states 3 and 5 because they have the same suffixes of length 1.*

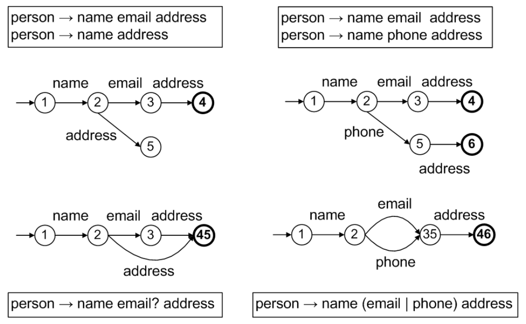

person → name email address
person → name address
person → name email address
person → name phone address
name
email
address
address
phone
address
name
email
address
address
name
email
address
phone
person → name email? address
person → name (email | phone) address

Figure 3: Merging states of an automaton

For the purpose of our improvements we need to modify the steps of ants, i.e. rules for modification of the inferred automata/productions. In particular, we need the following:

1. to add new merging rules,
2. to add splitting an automaton into multiple ones and
3. to make automaton modifications globally, i.e. with regard to other automata.

The first case is related to the exploitation on statistics of the given XML data. Contrary to existing works, while merging we want to take into account also additional information that influence the process.

Example 4. *An example of exploitation of data statistics is depicted in Figure 4. The numbers above the particular elements depict the amount of data instances that induce this production. If we do not consider the statistics, we would infer the schema on the left. But, with regard to the data, it would be too general, since most of the* `person` *elements have right three subelements* `phone`*, whereas only in few cases there are persons with more phone numbers. Consequently, from the point of view of preciseness of information on the data, the schema on the right is much realistic and bears more precise information.*

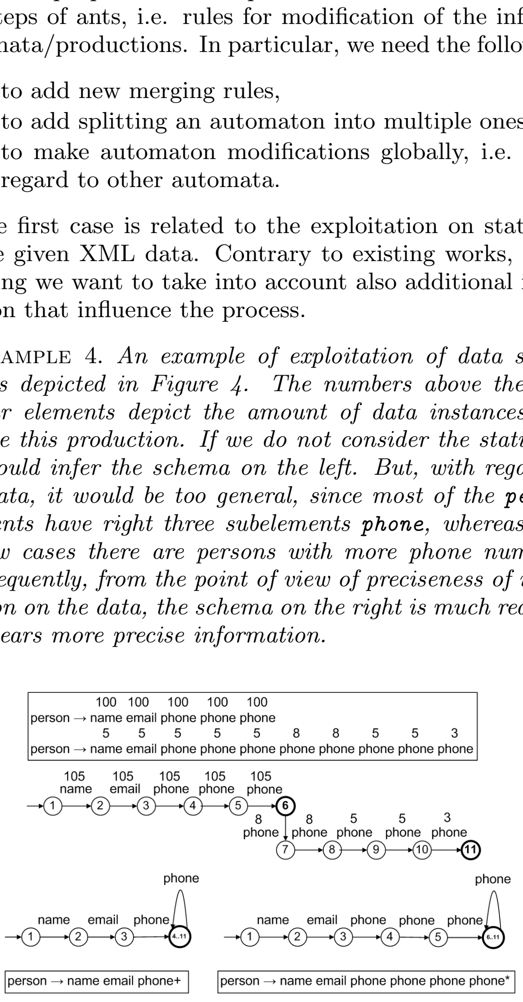

100 100 100 100 100
person → name email phone phone phone
5 5 5 5 5 8 8 5 5 3
person → name email phone phone phone phone phone phone phone phone
105 105 105 105 105
name email phone phone phone
8 8 5 5 3
phone phone phone phone phone
name email phone
phone
name email phone phone phone
phone
person → name email phone+
person → name email phone phone phone phone*

Figure 4: Splitting repetitions

The second point of splitting automata/productions is important for outlining sets of elements/attributes into globally defined items. And, similarly, the third one is related to the fact that we do not want to outline any schema fragment, but only those that occur in multiple contexts.

EXAMPLE 5. *An example of splitting an automaton/grammar is depicted in Figure 5. In this case we exploit the fact that elements* `person` *and* `manager` *have common items that can be outlined into a globally defined schema fragment represented using a separate production/automaton. Since "person" and "manager" are semantically related, the sets are enough large and the items are not too general, the result again bears more information and it is more realistic.*

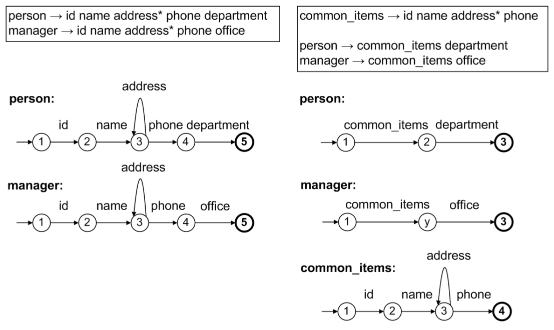

person → id name address* phone department
manager → id name address* phone office
common_items → id name address* phone
person → common_items department
manager → common_items office
person:
address
id
name
phone
department
manager:
office
common_items:
common_items
1
2
3
4
5
y

Figure 5: Splitting an automaton/production

5.3 Evaluation of Continuations

The evaluation of moving from schema s_x to s_y, where $s_x, s_y \in \Theta$ is defined as

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

where f 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 f, most of the existing works exploit the MDL principle [9]. 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 to express document instances in I 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 I expressed using R_s.

Let O be the set of allowed operators and E the set of distinct element names in I. 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 $d \in I$ 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 $card(R_d) \cdot \lceil \log_2(card(R_s)) \rceil$.

The problem is that since our primary aim is not the conciseness, we need to modify this approach. Otherwise, it would disadvantage all the productions that outline schema fragments as well as those splitting repetitions. For the purpose of the following explanation, let us denote R'_s as the original productions and R''_s as the outlined productions, $R'_s \cup R''_s = R_s$ and $R'_s \cap R''_s = \emptyset$.

Evaluation of Outlining.

In general, if we outline a reasonably big schema fragment from at least two productions, we obviously decrease the size of R_s. But, on the other hand, if there exists $d \in I$ s.t. R_d involves a production $\vec{p} \in R_s$ that was split into multiple productions, the length of R_d increases. To solve this problem, we modify the evaluation as follows:

- Each $\vec{p} \in R_s$ is provided with a weight $\alpha(\vec{p}) \in [0, 1]$ which influences (multiplies) the size of code of $\vec{p}$ expressing its appropriateness with regard to the previously described outlining heuristics.
- The productions from R''_s are not involved in counting the size of R_d for $\forall d \in I$.

Although outlining of a production always increases the amount productions to infer the instances in I, each of the outlining operation always produces a schema that is equivalent to the original one. Hence, we can omit these productions in counting the size of inferring sequences. On the other hand, the quality of the outlined production will influence the resulting size (i.e. cost) of the schema – the more appropriate the outlined production is, the lower its weight α is and, hence, the smaller contribution it makes to the overall size of R_s.

The strategy of assigning α is as follows: At the beginning of the search algorithm (i.e. when $R_s = R'_s$) for $\forall \vec{p} \in R'_s$: $\alpha(\vec{p}) = 1$ representing the fact that each contributes to the total cost 100% as in the original metric. If two productions $\vec{p}, \vec{q} \in R_s$ are split into productions $\vec{o} \in R''_s$ and $\vec{p'}, \vec{q'} \in R_s$, s.t. $type(\vec{o}) \in model(\vec{p'})$ and $type(\vec{o}) \in model(\vec{q'})$, each of them is assigned α as follows:

$$\begin{array}{rcll} \alpha(\vec{o'}) & = & \text{newly counted} & \\ \alpha(\vec{p'}) & = & \alpha(\vec{p}) & ; \vec{p} \in R'_s \\ & = & \text{newly counted} & ; \vec{p} \in R''_s \\ \alpha(\vec{q'}) & = & \alpha(\vec{q}) & ; \vec{q} \in R'_s \\ & = & \text{newly counted} & ; \vec{q} \in R''_s \end{array}$$

Let $context(\vec{x}) = \{\vec{q} \in R'_s$ s.t. $type(\vec{x}) \in model(\vec{q}) \vee (\exists \vec{p} \in R''_s$ s.t. $type(\vec{x}) \in model(\vec{p}) \wedge type(\vec{q}) \in context(\vec{p}))\}$ and let $model'(\vec{x})$ be $model(\vec{x})$, where $\forall \vec{p} \in R''_s$ s.t. $type(\vec{p}) \in model(\vec{x})$ is recursively replaced with $model(\vec{p})$. The weight $\alpha(\vec{x})$ is evaluated as follows:

$$\begin{array}{rl} \alpha(\vec{x}) = 1 - (& \alpha_1 \times sim_{i=1}^{|context(\vec{x})|}(q_i \in context(\vec{x})) + \\ & \alpha_2 \times \frac{|context(\vec{x})|}{|R'_s|} + \\ & \alpha_3 \times \frac{|model'(\vec{x})| - |stoplist(model'(\vec{x}))|}{avg_{i=1}^{k}(|model'(\vec{q}_i)|)} \;) \end{array}$$

where $\sum_{i=1}^{3} \alpha_i = 1$ and $\forall i : \alpha_i \in [0, 1]$, $sim()$ evaluates semantic similarity of the given words and $stoplist()$ returns the set of terminals of the given content model that occur in the stoplist. In other words, the weight of the outlined production is expressed as a combination of similarity of elements it influences (the higher, the better), the amount of elements it influences (the more, the better) and its size except for words in the stoplist (the bigger, the better).

EXAMPLE 6. *An example of counting the respective parameters for evaluation α is depicted in the following table.*

In step A we start with three productions without parameters, since their weight α remains 1. In step B we outline production $\vec{P}$. It occurs in context of 2 original productions and its length is of 6. In step C we outline production $\vec{Q}$ which occurs in context of $\vec{c}$, but also $\vec{P}$, resulting in overall context $\vec{a}$, $\vec{b}$ and $\vec{c}$. The length of $\vec{Q}$ is of 5. Note that the length of $model'(\vec{P})$ does not change though $model(\vec{P})$ does.

	$\vec{x}$	$context(\vec{x})$	$\lvert model'(\vec{x})\rvert$
A	`a -> u w (b | c)`	-	-
	`b -> x y w (b | c)`	-	-
	`c -> d (b | c)`	-	-
B	`a -> u P`	-	-
	`b -> x y P`	-	-
	`c -> d (b | c)`	-	-
	`P -> w (b | c)`	$\vec{a}, \vec{b}$	6
C	`a -> u P`	-	-
	`b -> x y P`	-	-
	`c -> d Q`	-	-
	`P -> w Q`	$\vec{a}, \vec{b}$	6
	`Q -> (b | c)`	$\vec{a}, \vec{b}, \vec{c}$	5

Evaluation of Splitting Repetitions.

In case of splitting repetitions we encounter an opposite problem. The length of the preferred production is always longer than the length of the most concise one, because (regarding the data statistics) we do not include all the repeating fragments into the repetition. Hence, in this case using an appropriate weight we modify the contribution of such production to the size of codes of instances. The more such preferred production is used in the instances, the lower the weight is and, hence, the smaller contribution it makes.

We again provide the productions with a weight, β, expressing their appropriateness with regard to splitting repetitions. The weight is assigned as follows: At the beginning of the search algorithm, each edge of the PTA is assigned a number of instances it is induced by. At the same time, each of the productions is assigned weight β of 1 representing the fact that each contributes to the total cost 100% as in the original metric. If the selected merging rule does not induce merging repetitions, β remains the same and we only sum up the numbers of instances of the merged edges (as depicted in Figure 4). If a merging rule that induces repetitions is applied on a production $\vec{x}$, i.e. there is a sufficient amount of a repeating pattern r (a sequence $r_{(1)}r_{(2)}...r_{(l)}$), the weight$\beta(\vec{x'})$ of resulting production $\vec{x'}$ is counted as follows: Let rep_{min} be the minimum amount of occurrences a pattern must fulfill to be merged into a repetition, let i, j, s.t. $1 \leqslant i \leqslant j \leqslant l; j - i + 1 \geqslant rep_{min}$, determine the borders of the selected subsequence of repeating patterns to be merged and let $ind(r_{(i)})$ denote the number of inducing instances of $r_{(i)}$. Then:

$$\begin{aligned}
\beta(\vec{x'}) &= 1 - \tfrac{score(\vec{x'})}{score(\vec{x})} \\
score(\vec{x}) &= \textstyle\sum_{k=1}^{l} ind(r_{(k)}) \\
score(\vec{x'}) &= \textstyle\sum_{k=1}^{i-1} ind(r_{(k)}) + \\
&\quad rep_{min} * max_{k=i}^{j}(ind(r_{(k)})) + \\
&\quad \textstyle\sum_{k=j+1}^{l} ind(r_{(k)})
\end{aligned}$$

EXAMPLE 7. *Consider the example in Figure 4. If we do apply the first merging, assuming that $rep_{min} = 3$, then $score(\vec{x}) = 105+3\times 105 = 420$. If apply the second merging, then $score(\vec{x}) = 105 + 105 + 105 + 3 \times 8 = 339$, i.e. it is a better candidate.*

5.4 Rewriting Productions into XSD Syntax

The last step in the inference process is rewriting the inferred productions into the syntax of the respective XML schema language. In the existing works it is a straightforward process of rewriting an automaton into a RE. In our case we have to solve the question of outlined productions, i.e. globally defined schema fragments.

In general in all cases we can exploit classes C_{El}, C_{ElGr} and C_{At}. Depending on the type of the outlined production we simply define respective globally defined element, group of elements, attribute or group of attributes. The problem is when we want use also classes C_{CT} and C_{CTDer}, i.e. globally defined complex types and their mutual derivation. Firstly, we naturally need to follow the W3C specifications [15]. In particular, we have two choices – extension and restriction.

DEFINITION 6. *A complex type τ_y is* an extension *of a complex type τ_x if $attr(\tau_y) \supseteq attr(\tau_x)$ and $model(\tau_y) = model(\tau_x)m$, where $attr(\tau)$ is the set of attributes of type τ and m is a content model.*

A complex type τ_y is a restriction *of a complex type τ_x if $\eta(\tau_y) \subset \eta(\tau_x)$.*

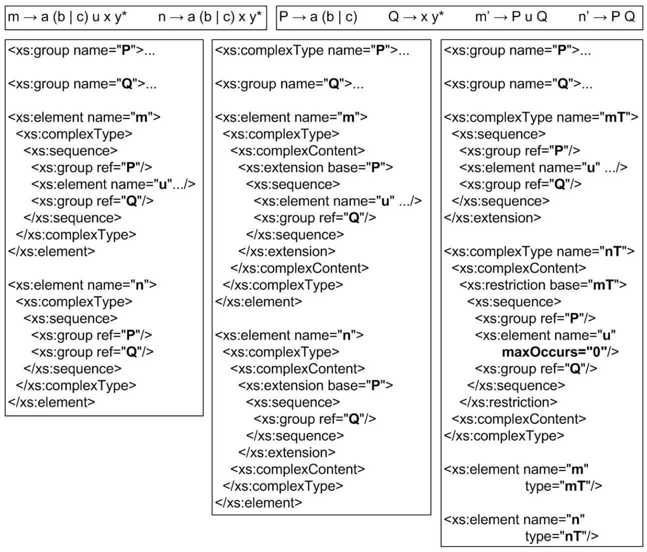
Consequently, we can exploit outlining of complex types and their derivation only in case it can be specified in either of the two ways. Otherwise we must use its combination with classes C_{El}, C_{ElGr} and C_{At}.

EXAMPLE 8. *Consider the example in Figure 6, where $\vec{P}$ and $\vec{Q}$ denote productions outlined from original productions $\vec{m}$ and $\vec{n}$. In the first case we exploit class C_{ElGr} and define groups `P` and `Q` that are referenced from element definitions of `m` and `n`. In the second case we combine classes C_{ElGr}, C_{CT} and C_{CTDer} and define complex type `P` to be the ancestor of complex types of both elements `m` and `n`. Similarly, in the third case we define content model of `m`, i.e. type `mT`, to be the ancestor of type `nT` derived by restriction.*

In general, we usually have multiple options how to define the type hierarchy. For selecting the optimal one we again exploit the thesaurus, but this time we do not determine semantic similarity of element names, but their mutual hierarchy. The rules are relatively simple:

- Relationships *Broader Term (BT) / Narrower Term (NT)*, which specify more/less general terms, determine the broader term to be the candidate for ancestor.
- Relationships *Use (USE) / Used For (UF)*, which specify authorized/unauthorized terms, determine any of the terms to be the candidate for ancestor.
- Relationship *Related Term (RT)* determines related terms, for whom a common ancestor is created.
- In other cases, i.e. when the terms are not related or we cannot say anything about their semantic relationship, we do not exploit derived complex types.

EXAMPLE 9. *Consider again the example in Figure 6. If we knew that "m" is broader term of "n" (e.g. "employee" and "director"), we would choose the third option. If we knew*



```
m → a (b | c) u x y*      n → a (b | c) x y*
```

```
P → a (b | c)      Q → x y*      m' → P u Q      n' → P Q
```

```
<xs:group name="P">...

<xs:group name="Q">...

<xs:element name="m">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="P"/>
      <xs:element name="u".../>
      <xs:group ref="Q"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="n">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="P"/>
      <xs:group ref="Q"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<xs:complexType name="P">...

<xs:group name="Q">...

<xs:element name="m">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="P">
        <xs:sequence>
          <xs:element name="u" .../>
          <xs:group ref="Q"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="n">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="P">
        <xs:sequence>
          <xs:group ref="Q"/>
        </xs:sequence>
      </xs:extension>
    <xs:complexContent>
  </xs:complexType>
</xs:element>
```

```
<xs:group name="P">...

<xs:group name="Q">...

<xs:complexType name="mT">
  <xs:sequence>
    <xs:group ref="P"/>
    <xs:element name="u" .../>
    <xs:group ref="Q"/>
  </xs:sequence>
</xs:extension>

<xs:complexType name="nT">
  <xs:complexContent>
    <xs:restriction base="mT">
      <xs:sequence>
        <xs:group ref="P"/>
        <xs:element name="u"
               maxOccurs="0"/>
        <xs:group ref="Q"/>
      </xs:sequence>
    </xs:restriction>
  <xs:complexContent>
</xs:complexType>

<xs:element name="m"
               type="mT"/>

<xs:element name="n"
               type="nT"/>
```

Figure 6: Options of rewriting into XSD syntax

that "m" and "n" are related terms (e.g. "cat" and "dog"), we would choose the second option. And if we knew that "m" and "n" are not related, but their content models are so similar that they were selected for merging, we would choose the first option.

The simple example depicts that a set of XML documents can be expressed using distinct schemas. The proposed approach is able to analyze additional information on the data and provide an output that reflects them. If we consider the resulting schema of existing approaches, i.e. two element definitions that involve separate local specifications of their content models, we can see that any of the three results is much more informative and precise.

6. CONCLUSION

The aim of this paper was to propose an algorithm for automatic inference of an XML schema which is more realistic and bears more precise information than results of existing inference methods. For this purpose we exploited a set of heuristics based on semantics of element/attribute names and data statistics and utilized several verified approaches such as the ACO heuristics or MDL principle. Currently we are dealing with extending the implementation of [16] with the proposed improvements, since we intend to apply it on a representative set of real-world XML data. We assume that similarly to paper [4] 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 the searching 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.

7. ACKNOWLEDGEMENT

This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838).

8. REFERENCES

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

[2] J. Berstel and L. Boasson. XML Grammars. In *Mathematical Foundations of Computer Science*, LNCS, pages 182–191. Springer, 2000.

[3] G. J. Bex, F. Neven, and J. V. den Bussche. DTDs versus XML Schema: a Practical Study. In *WebDB'04*, pages 79–84, New York, NY, USA, 2004. ACM.

[4] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema Definitions from XML Data. In *VLDB'07*, pages 998–1009, Vienna, Austria, 2007. ACM.

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

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

[7] M. Dorigo, M. Birattari, and T. Stutzle. *Ant Colony Optimization – Artificial Ants as a Computational Intelligence Technique*. TechReport 2006-023, IRIDIA, Bruxelles, Belgium, 2006.

[8] F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML Documents in Relational Databases. In *VLDB'04*, pages 1297–1300, Toronto, ON, Canada, 2004. Morgan Kaufmann.

[9] 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*, pages 165–176, New York, NY, USA, 2000. ACM.

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

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

[12] L. Mignet, D. Barbosa, and P. Veltri. The XML Web: a First Study. In *WWW'03*, pages 500–510, New York, NY, USA, 2003. ACM.

[13] I. Mlynkova, K. Toman, and J. Pokorny. Statistical Analysis of Real XML Data Collections. In *COMAD'06*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill.

[14] C.-H. Moh, E.-P. Lim, and W.-K. Ng. Re-engineering Structures from Web Documents. In *DL'00*, pages 67–76, New York, NY, USA, 2000. ACM.

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

[16] O. Vosta, I. Mlynkova, and J. Pokorny. Even an Ant Can Create an XSD. In *DASFAA'08*, LNCS, pages 35–50. Springer, 2008.

[17] R. K. Wong and J. Sankey. *On Structural Inference for XML Data*. TechReport UNSW-CSE-TR-0313, School of Computer Science, University of NSW, 2003.