Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Michal Klempa

Optimization and Refinement of XML Schema Inference Approaches

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Irena Mlýnková Ph.D.
Study programme:	Informatics
Specialization:	Software Systems

Prague, 2011

I would like to thank my supervisor RNDr. Irena Mlýnková Ph.D., for her helpful suggestions, advice and corrections.

Název práce: Optimalizace a zdokonalení stávajících přístupů automatické konstrukce schématu XML dokumentů

Autor: Michal Klempa

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková Ph.D.

Abstrakt: I když je XML obecně užívána technologie, většina XML dokumentů v oběhu neodpovídá žádnému specifikovanému schématu. Za této situace vznikl výzkum automatické konstrukce schémat z XML dokumentů. Tato práce zdokonaluje a rozšiřuje stávající přístupy automatické konstrukce schémat především využitím starého schématu v procesu konstrukce, návrhnem nových MDL metrik a heuristickým vyloučením excentrických datových vstupů. Práce přináší jednoduše rozšířitelnou a k použití připravenou implementaci ve formě pluginu do aplikace jInfer (vyvinuté v rámci softwarového projektu). Součástí práce jsou experimentální výsledky.

Klíčová slova: XML, XML schéma, odvozování schématu, odvozování regulárních výrazů z pozitivních příkladů

Title: Optimization and Refinement of XML Schema Inference Approaches

Author: Michal Klempa

Department: Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková Ph.D.

Abstract: Although XML is a widely used technology, the majority of real-world XML documents does not conform to any particular schema. To fill the gap, the research area of automatic schema inference from XML documents has emerged. This work refines and extends recent approaches to the automatic schema inference mainly by exploiting an obsolete schema in the inference process, designing new MDL measures and heuristic excluding of excentric data inputs. The work delivers a ready-to-use and easy-to-extend implementation integrated into the jInfer framework (developed as a software project). Experimental results are a part of the work.

Keywords: XML, XML schema, schema inference, inference of regular expressions from positive examples

Contents

1 Introduction

The XML is a popular data format and it has become the format of choice for data representation for its simple but powerful design. To enforce a defined structure of XML documents, one can use XML schema definition languages (for example DTD [41], or XML Schema [36, 14]).

Although designing an XML schema is a simple task (especially in DTD language), a half of randomly crawled documents [7, 30] does not link any associated schema. In addition, schemas used are "quite simple, compared to the features provided by these languages" [7].

To overcome this problem, the research of automatic schema inference from XML documents has emerged. Each element in the XML schema has its content model defined with a regular expression [8]. Thus the problem of learning a regular language from a finite set of positive examples arises and this can not be solved in general [20]. Current solutions either define a subclass of regular languages, that is learnable in the limit [5, 10], or solve the problem heuristically [38, 34, 40]. This work belongs to the latter.

To infer a regular expression heuristic solutions usually construct a *prefix tree automaton* (PTA) that accepts all positive examples given and then merges *equivalent states* of the automaton to generalize the language accepted by it. When the automaton is considered small enough, it is converted to an equivalent regular expression. As there are infinitely many automatons that can be inferred, and each of them can be converted into infinitely many equivalent regular expression one has to design clever criteria and algorithms. The solutions differ in state equivalence definitions, the decision of when to stop merging and how to convert the automaton into regular expression.

But not only XML documents can be used to infer the schema. "In real-world XML data the XML schema is usually considered as a kind of data documentation. Since the schema is not used as it is supposed to be, i.e. for checking the correct structure of XML documents, it is usually not updated in case the respective data are." [31] It is quite common that there is an obsolete schema available which can be exploited for the inference. To our knowledge, only [31] deals with XML schema inference with help of an obsolete one, but it lacks implementation and experimental results.

Even though relatively many solutions proposed, none of them is implemented in a user friendly and easy-to-use environment publicly available. The potential users have only commercial solutions available.

1.1 Aim of the Thesis

The aim of the thesis is to analyze recent approaches and find the aspects to improve. We choose the previous work [38] as a starting point for our own solution and refine it by employing new metrics (new MDL design) and schema input. By using the additional information obtained from an obsolete schema we improve the inference performance and satisfy the user expectations, that the inferred schema should be similar to the old one. The key point is an implementation, which should be ready-to-use and publicly available. We have integrated this work into the jInfer framework [28] developed earlier as a software project, so it is ready-to-use in a friendly environment and easy to extend in future work.

1.2 Structure of the Thesis

We define basic terms in Section 2; the precise problem formulation is in Section 3. An analysis of existing solutions can be found in Section 4, which is followed by the core of the work, the proposed solution in Section 5. A brief description of implementation details is in Section 6 and we complete the work with experimental results in Section 7. Section 8 concludes the work done and discusses the work left for future research. The attachment contains a source code of this work, available under the GNU GPL.

2 Basic Definitions

This section contains definitions and other prerequisites used in the rest of the text.

2.1 Formal Languages Theory

2.1.1 Languages, grammars, etc.

Definition 1 (Alphabet)**.** An *alphabet* is any finite, non-empty set of symbols. Usually denoted as Σ. $\Box$

Definition 2 (Word)**.** A *word* over the alphabet Σ is any finite sequence of symbols from set Σ. Empty sequence (empty word) is denoted by ε. $\Box$

Definition 3 (Language)**.** A *language* over the alphabet Σ is any set of words over Σ. Languages are usually denoted L with an index. $\Box$

Definition 4. The notation Σ^* denotes set of all words over the alphabet Σ. $\Box$

Definition 5 (Word concatenation)**.** Let $u = u_1u_2u_3 \dots u_n, v = v_1v_2v_3 \dots v_m$ be two words, then the word $u \cdot v = u_1u_2u_3 \dots u_nv_1v_2v_3 \dots v_m$ is a *concatenation* of u, v. The $\cdot$ concatenation operator is usually omitted. $\Box$

Definition 6 (Language concatenation)**.** Let L_1, L_2 be languages, then the language $L_1 \cdot L_2 = \{uv | u \in L_1 \land v \in L_2\}$ is a *concatenation of languages* L_1, L_2. The $\cdot$ operator is usually omitted. $\Box$

Definition 7 (Language iteration)**.** Let L be a language, let $n \in \mathbb{N} \setminus \{0\}$, let $L^0 = \{\varepsilon\}$, then the L^n is defined recursively: $L^n = L^{n-1} \cdot L$. Furthermore $L^* = \bigcup_{i=0}^{\infty} L^i$. $\Box$

Definition 8 (Grammar)**.** A *grammar* G is a tuple (N, T, P, σ), where N is a set of non-terminal symbols, T is a set of terminal symbols, such that $N \cap T = \emptyset$, $\sigma \in N$ is an initial non-terminal and

$$P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$$

is a finite set of production rules. Rule $(u, v) \in P$ is usually denoted as $u \xrightarrow[G]{} v$ or $u \longrightarrow v \in P$. $\Box$

Definition 9 (Production step). Let $G = (N, T, P, \sigma)$ be a grammar. Binary relation $\underset{G}{\Longrightarrow} \subseteq (N \cup T)^* \times (N \cup T)^*$ defined as

$$x \underset{G}{\Longrightarrow} y \quad \text{iff} \quad \exists w_1, w_2 \in T^* \exists u \longrightarrow v \in P \text{ such that } x = w_i u w_2 \text{ and } y = w_1 v w_2$$

is called a *production step.* $\square$

Definition 10 (Regular grammar). A *regular grammar* $G = (N, T, P, \sigma)$ is a grammar where

$$P \subseteq (N \times (T^*(N \cup \{\varepsilon\})))$$

$\square$

Definition 11 (Context free grammar). A *context free grammar* $G = (N, T, P, \sigma)$ is a grammar where

$$P \subseteq (N \times (N \cup T)^*)$$

$\square$

Definition 12 (Language generated by a grammar). Let $G = (N, T, P, \sigma)$ be a grammar. A *language generated by* G is $L(G) = \{w | w \in T^* \wedge \sigma \underset{G}{\Longrightarrow}^* w\}$. Where $\underset{G}{\Longrightarrow}^*$ is a transitive closure of binary relation $\underset{G}{\Longrightarrow}$. $\square$

Definition 13 (Regular language). Language $L(G)$, where G is a regular grammar, is called *regular.* $\square$

Definition 14 (Regular Expression (RE)). A *regular expression (RE)* $\overline{R}$ is defined recursively as follows:

- $\overline{\varepsilon}$ is regular expression corresponding to language $\emptyset$ (called the ε regular expression),
- let $a \in \Sigma$, then $\overline{a}$ is regular expression corresponding to language $\{a\}$ (token regular expression),
- let $\overline{R_1}, \overline{R_2}$ be regular expressions corresponding to languages L_1, L_2, then $\overline{R_1} \cdot \overline{R_2}$ is regular expression corresponding to language $L_1 \cdot L_2$ (concatenation regular expression),
- let $\overline{R_1}, \overline{R_2}$ be regular expressions corresponding to languages L_1, L_2, then $\overline{R_1} | \overline{R_2}$ is regular expression corresponding to language $L_1 \cup L_2$ (alternation regular expression),

- let $\overline{R}$ be regular expression corresponding to language L, then $\overline{R}*$ is regular expression corresponding to language L^*.

$\square$

Theorem 1. A regular expression R corresponds to a regular language $L(R)$. $\square$

Definition 15 (Extended context free grammar)**.** An *extended context free grammar* $G = (N, T, P, \sigma)$ is a grammar where

$$P \subseteq (N \times \overline{R})$$

where $\overline{R}$ is a regular expression over the alphabet $N \cup T \setminus \{\varepsilon\}$. $\square$

Definition 16 (Deterministic Finite-State Automaton (DFA))**.** A *deterministic finite-state automaton* A is a tuple $A = (Q, \Sigma, \delta, q_0, \lambda, F)$, where:

- Q is a finite set of states, such that $\lambda \notin Q$,
- λ is a dummy state indicating immediate halt,
- Σ is an alphabet (finite set of symbols),
- $\delta : (Q \times \Sigma) \to Q \cup \{\lambda\}$ is a transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

$\square$

Definition 17 (Configuration)**.** A *configuration* of DFA A is a tuple $(q, w) \in Q \times \Sigma^*$, where q is the current state of the automaton, and w is the word generated so far. $\square$

Definition 18 (Computational step)**.** Binary relation $\underset{A}{\vdash} \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ defined as

$$(q, w) \underset{A}{\vdash} (q', wa) \quad \text{iff} \quad \delta(q, a) = q',$$

where $w \in \Sigma^*$ is word generated so far and $a \in \Sigma$ is the symbol generated in this step, is called a *computational step.* $\square$

Definition 19 (Language generated by DFA)**.** A *language $L(A)$ generated by DFA A* is a set of words defined as follows

$$L(A) = \{w | w \in \Sigma^* \wedge \exists q_F \in F : (q_0, \varepsilon) \underset{A}{\vdash}^* (q_F, w)\}$$

Where $\underset{A}{\vdash}^*$ is a transitive closure of binary relation $\underset{A}{\vdash}$. □

We define the DFA as generative, starting with ε as output word. At each computational step, one symbol is generated concatenated with the output word of the automaton. Once a computation gets into λ state, it can not end, since there is no way out of it.

Definition 20 (Prefix Tree Automaton (PTA))**.** Let $A = (Q, \Sigma, \delta, q_0, \lambda, F)$ be a DFA. Let (V, E) be a directed *underlying graph* of A, where $V = Q$ is a set of nodes and $E \subseteq Q \times Q$ is a set of edges defined as follows

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

A PTA is a DFA whose underlying graph is a tree rooted at the initial state q_0. □

2.1.2 Deterministic Probabilistic Finite-State Automaton

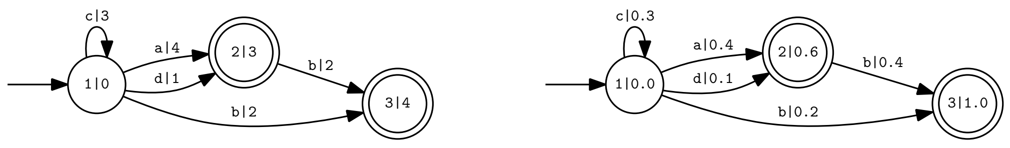
Deterministic Probabilistic Finite-State Automaton (DPFA) is DFA extended with probabilities

- at each transition: that this transition is followed,
- at each state: that the generation stops at this state.

Formal definition which follows is a restricted version of [37, def. 4], since this version fits better programming purposes of this thesis.

Definition 21 (Deterministic Probabilistic Finite Automaton)**.** A DPFA is a tuple $A = (Q, \Sigma, \delta, q_0, \lambda, F, P)$, where:

- Q is a finite set of states, such that $\lambda \notin Q$,
- λ is a dummy state indicating immediate halt,
- Σ is an alphabet (finite set of symbols),
- $\delta : (Q \times \Sigma) \rightarrow (Q \cup \{\lambda\})$ is a transition function,



(a) An example of DPFA, state λ and transitions leading to it are omitted

(b) Corresponding DPFA with probabilities

Figure 1: Example DPFA

- $q_0 \in Q$ is an initial state,
- $P : \delta \to \mathbb{N}$ is function of transition use counts,
- $F : Q \to \mathbb{N}$ is function of state final counts.

$\square$

Use counts and final counts can be null ($0 \in \mathbb{N}$). State λ serves as a dummy state indicating immediate halting (or transition non-existence in other words). Use counts and final counts can be easily converted into corresponding probabilities of transitions and states, thus every DPFA is also DPFA in the sense of [37, def. 4].

Example 1. Let $A = (Q, \Sigma, \delta, q_0, \lambda, F, P)$, where $Q = \{1, 2, 3\}$, $\Sigma = \{a, b, c, d\}$, $q_0 = 1$, $F(1) = 0, F(2) = 3, F(3) = 4$ and

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\delta((1,a)) & = & 2 & P((1,a)) & = & 4 \\
\delta((1,b)) & = & 3 & P((1,b)) & = & 2 \\
\delta((1,c)) & = & 1 & P((1,c)) & = & 3 \\
\delta((1,d)) & = & 2 & P((1,d)) & = & 1 \\
\delta((2,a)) & = & \lambda & P((2,a)) & = & 0 \\
\delta((2,b)) & = & 3 & P((2,b)) & = & 2 \\
\delta((2,c)) & = & \lambda & P((2,c)) & = & 0 \\
\delta((2,d)) & = & \lambda & P((2,d)) & = & 0 \\
\delta((3,a)) & = & \lambda & P((3,a)) & = & 0 \\
\delta((3,b)) & = & \lambda & P((3,b)) & = & 0 \\
\delta((3,c)) & = & \lambda & P((3,c)) & = & 0 \\
\delta((3,d)) & = & \lambda & P((3,d)) & = & 0
\end{array}$$

The automaton is depicted in Fig. 1. Each state has its label in form `name|final count` and each transition in form `symbol|use count`. $\square$

Definition 22 (Probabilistic Prefix Tree Automaton)**.** Let

$$A = (Q, \Sigma, \delta, q_0, \lambda, F, P)$$

be a DFPA. Let (V, E) be a directed *underlying graph* of A, where $V = Q$ is a set of nodes and $E \subseteq Q \times Q$ is a set of edges defined as follows

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

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

2.1.3 Interval Extended Regular Expression

For the purpose of this work, a interval extended regular expression is defined as follows:

Definition 23 (Interval Extended Regular Expression)**.** An *interval extended regular expression (IERE)* R is a tuple $R = (C, m, n)$, where C is a IERE defined recursively (see below) and where numbers $m \in \mathbb{N}, n \in \mathbb{N} \cup \{\infty\}, m \leq n$ represent minimal and maximal repeat counts of C. C is defined recursively as follows:

- let $\overline{R}$ be a RE corresponding to language L, then $(R, 1, 1)$ is IERE corresponding to language L,

- let $(R_1, m_1, n_1), (R_2, m_2, n_2)$ be IEREs corresponding to languages L_1, L_2, then $C = (R_1, m_1, n_1) \cdot (R_2, m_2, n_2)$ is IERE corresponding to language $L_1 \cdot L_2$ (concatenation IERE),

- let $(R_1, m_1, n_1), (R_2, m_2, n_2)$ be IEREs corresponding to languages L_1, L_2, then $C = (R_1, m_1, n_1)|(R_2, m_2, n_2)$ is IERE corresponding to language $L_1 \cup L_2$ (alternation IERE),

- let $(C, 1, 1)$ be IERE corresponding to language L, then (C, m, n); $m, n \in \mathbb{N}$ is IERE corresponding to language

  $$L^m \cdot \left(\bigcup_{i=0}^{m-n} L^i \right).$$

  Moreover, the (C, m, ∞) is IERE corresponding to language

  $$L^m \cdot L^*.$$

□

This definition is equivalent with Def. 14, but it establishes syntactic shorthands for any kind of regular expression repeat specification in form of (m, n). Well known shorthands $C?$; $C+$; $C*$ are represented as $(C, 0, 1)$; $(C, 1, \infty)$; $(C, 0, \infty)$ respectively.

Example 2. Expression $C = ((a, 1, 1)|(b, 1, 1), 0, \infty) \cdot (c, 1, 1)$ corresponds to language $L = \{w = x_1, \ldots, x_n | (\forall i \leq n - 1 : x_i \in \{a, b\}) \wedge (x_n = c)\}$. An IERE can be visualized as a tree, see Fig. 2. □

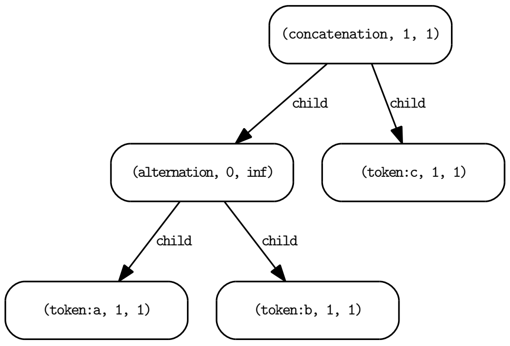


Figure 2: Example IERE

When writing IEREs, we use syntactic shorthands: we omit m, n if they are both equal to 1, well known shorthands $? + *$ represent m, n values $(0, 1)$; $(1, \infty)$; $(0, \infty)$ respectively, we omit () where possible, as priority of $? * +$ operators is higher then priority of $|\cdot$ operations and finally we omit $\cdot$ operator completely. A shorthand notation of IERE from Example 2 is $(a|b) * c$.

2.2 XML Documents and Schemas

An *alphabet of element names* Σ_E is the set of all names appearing in XML documents and schema files on input. All text nodes are named the same, e.g. "textnode", if it does not conflict with another element name.

An *XML schema* is ECFG S over the alphabet of element names [8]. The XML schema consists of production rules called *element type definitions.* The left-hand side of an element type definition is an *element type* and the right-hand side is an *element content model.* An element content model is basically a regular expression over the alphabet of element names. The XML schema can be written in various schema definition languages, such as DTD [41] or XML Schema [36, 14]. An *element instance* is literally the element and its contents as it is found in an XML document. An element instance *content* is a word over the alphabet of element names.

The element instance in the XML document is *valid against* its element type definition in the schema, if the element instance content is a word from a language $L(R)$, corresponding to a regular expression R - the right-hand side of the element type definition. The XML document is said to be valid against the schema, if all element instances in the document are valid against their appropriate element type definitions.

Example 3. Consider the following XML document and its schema

```
<person>          element instance
  <info>          element instance
    Some text     element instance
  </info>
  <note/>
</person>
```

$$\begin{array}{rcl} person & \longrightarrow & (info+, note) \\ info & \longrightarrow & (textnode?) \\ note & \longrightarrow & (\varepsilon) \end{array}$$

The alphabet of element names in this example is

$$\Sigma_E = \{person, info, textnode, note\}.$$

Each line in the sample schema is an element type definition, the left-hand side is element type, the right-hand side is a content model. The content of element instance `person` from the example is a word $info, note$ over alphabet Σ_E. The same schema represented in DTD language would be as follows

```
<!ELEMENT person (info+,note)>
<!ELEMENT info #CDATA>
<!ELEMENT note EMPTY>
```

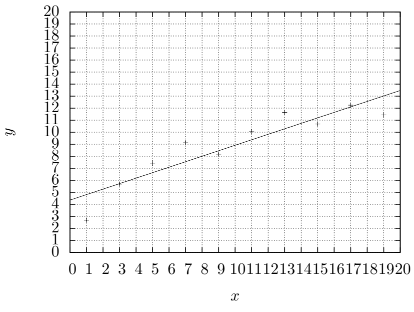
□

2.3 MDL Principle

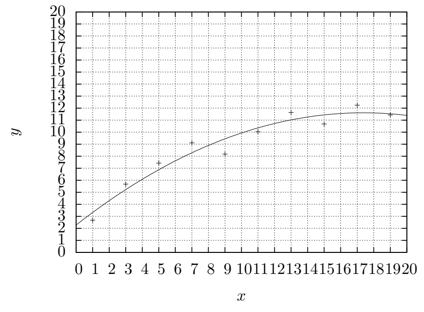
For the purpose of this work we utilize the *minimum description length* principle (MDL), which is described in [23].

Example 4. Consider the data consisting of points in Figure 3(a). Consider the x-axis as time and y-axis as the values coming from an unknown data source. We want to predict the future y values. For this purpose, we have to exploit an underlying data regularity (there must be at least minimal regularity unless the data are produced by fair coin tosses).

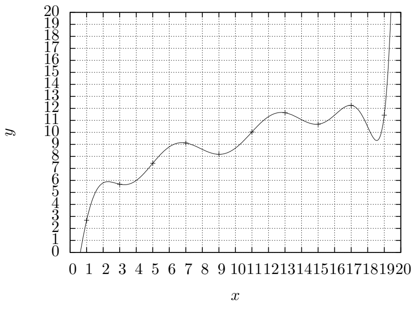
If we hope that the data are generated using a polynomial and some random noise. We can fit the data using a simple linear regression and clearly, this does not capture the data regularity well. Looking for a 2nd and 10th degree



(a) Points and 1st degree polynomial



(b) Trade-off 2nd degree polynomial



(c) Polynomial of degree 10 overfits the data

Figure 3: An example of overfitting

polynomial with least error will result in data fitting as depicted in Fig. 3(b) and 3(c).

How to select the "best" data explanation? Instead of selecting too simple (1st degree) or too complex (10th degree) polynomial, it makes sense to select the trade-off polynomial as the best data explanation.

This is where MDL principle can be employed. It is a general method to select the best data explanation among possible ones. □

The possible explanation is also called a *hypothesis* explaining the data and the set of possible explanations is usually denoted as *model.* The model has to be designed artificially. Then the MDL suggests us to view the data as being generated by a particular hypothesis. As we have seen in Example 4, the decision of which hypothesis explains the data best, is not a trivial task. Care must be taken to not over-fit the data and at the same time to capture the underlying regularity.

2.3.1 MDL - The Basic Idea

Following the tutorial [23], learning data can be viewed from two points of view: data prediction based on the data seen until now and data compression. Both views point to the regularity underlying the data. Let us quote the basic idea from [23]:

> **MDL: The basic idea**
> The goal of statistical inference may be cast as trying to find regularity in the data. ‘Regularity’ may be identified with ‘ability to compress’. MDL combines these two insights by *viewing learning as data compression*: it tells us that, for a given set of hypotheses $\mathcal{H}$ and data set D, we should try to find the hypothesis or combination of hypotheses in $\mathcal{H}$ that compresses D most.

2.3.2 Definition of Crude MDL

In this work, special version of MDL called “Crude MDL” as defined in [23] is used:

> **Crude, Two-part Version of MDL Principle (Informally Stated)** Let $\mathcal{H}^{(1)}, \mathcal{H}^{(2)}, \ldots$ be a list of candidate models (e.g., $\mathcal{H}^{(k)}$ is the set of k-th degree polynomials), each containing a set of point hypotheses (e.g., individual polynomials). The best point hypothesis $H \in \mathcal{H}^{(1)} \cup \mathcal{H}^{(2)} \cup \ldots$ to explain the data D is the one which minimizes the sum $L(H) + L(D|H)$, where
>
> - $L(H)$ is the length, in bits, of the description of the hypothesis; and
> - $L(D|H)$ is the length, in bits, of the description of the data when encoded with the help of the hypothesis.
>
> The best model to explain D is the smallest model containing the selected H.

This basically tells us, that the best hypothesis is the one, that compresses the data most. To define the $L(H)$ we usually design an ad-hoc code of the hypothesis. There are some universal codes, that may help us (for example the one from Section 2.3.4 for integers). The $L(D|H)$ is usually a probabilistic code, given the H is a probabilistic source of the data which emits a particular value x with probability p.

2.3.3 Probabilities and Codelengths

In the following text, the C denotes a code and L_C the codelength function. The codelength function returns the length of code (in bits) of the given data input encoded using code C.

Two most important observations from the tutorial are interesting in the scope of this work:

> **Probability Mass Functions correspond to Codelength Functions** Let $\mathcal{Z}$ be a finite or countable set and let P be a probability distribution on $\mathcal{Z}$. Then there exists a prefix code C for $\mathcal{Z}$ such that for all $z \in \mathcal{Z}$, $L_C(z) = \lceil -\log P(z) \rceil$. C is called the code corresponding to P. Similarly, let C' be a prefix code for $\mathcal{Z}$. Then there exists a (possibly defective) probability distribution P' such that for all $z \in \mathcal{Z}$, $-\log P'(z) = L_{C'}(z)$. P' is called the probability distribution corresponding to C'.
>
> Moreover C' is a complete prefix code iff P is proper ($\sum_z P(z) = 1$).
>
> Thus, large probability according to P means small code length according to the code corresponding to P and vice versa. We are typically concerned with cases where $\mathcal{Z}$ represents sequences of n outcomes; that is, $\mathcal{Z} = \mathcal{X}^n (n \geq 1)$ where $\mathcal{X}$ is the sample space for one observation.
>
> **The P that corresponds to $\mathcal{L}$ minimizes expected codelength** Let P be a distribution on (finite, countable or continuous-valued) $\mathcal{Z}$ and let L be defined by
>
> $$L := \arg \min_{L \in \mathcal{L}_{\mathcal{Z}}} E_P[L(Z)].$$
>
> Then L exists, is unique, and is identical to the codelength function corresponding to P, with lengths $L(z) = -\log P(z)$.

These two observations describe a relationship between probabilistic source of data and minimal codelength achievable when compressing the data. When the data is believed to originate from probabilistic source, the only meaningful encoding to use is the code C from Sec. 2.3.3.

2.3.4 Standard Universal Code for Integers

In [22, p. 100] a *standard universal code for integer values* is presented. The code should be used for integers which are believed not to originate from probabilistic source (and values are possibly unbounded). The codelength for integer value n

is computed using formula:

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

where $c_0 \approx 2.865$. It means "sum logarithms until the first negative value encounters (exclude it), then add $\log c_0$ constant". The motivation and details of the code can be found in [22], simplified version is described in [23], see also [35].

2.3.5 Uniform Code

When probabilistic source of data has a uniform distribution, that is: data consists of n equally probable options, each with probability of appearance equal to $\frac{1}{n}$, then the codelength of each option equals

$$-\log(\frac{1}{n}).$$

Which equals

$$\log(n),$$

as one would expect.

2.3.6 Noninteger Codelengths

Some readers may be concerned why particular codelengths defined in next sections are not rounded to the next integer value. As this topic is far beyond the scope of this work, let us just quote [22, p. 99] once again:

> **Summary: Integers Don't Matter**
> There is NO practical application of MDL in which we worry about the integer requirement for codelengths; we always allow for codelength functions to take noninteger lengths. Instead, we call a function a "code- length function" if and only if it satisfies the Kraft inequality, that is, iff it corresponds to some defective probability distribution.

3 Problem Statement

The basic problem studied in this work can be formulated as follows:

Problem 1 (XML schema inference)**.** Given a set of input XML documents D, we want to infer a schema S_D, such that each input document $d \in D$ is valid against schema S_D. It should be concise and precise enough, while, at the same time, general. $\square$

A common way to solve the problem is to extract all element instances E from input documents, cluster them into groups $E_1, \ldots, E_n$, assuming that in each group element instances corresponding to the same element type definition reside. The set E is often called the *initial grammar*, as each element $e \in E$ can be viewed as a production rule $e \longrightarrow e_{\text{contents}}$ of respective ECFG.

Example 5. Consider the two input XML documents as depicted in Fig. 4. The initial grammar rules extracted from these two documents is depicted in Table 1 where the right-hand sides are IEREs (see Def. 23) and horizontal lines divide the rules into groups according to element names. As right-hand sides are always concatenation IEREs, they can be also viewed as words over alphabet of element names. $\square$

```
<person>
  <info>
    Some text
  <note/>
  </info>
```

```
</person> <person>
  <more/>
  <more/>
  <more/>
</person>
```

Figure 4: Example XML documents

$person$	$\longrightarrow$	$(info, 1, 1)$	E_1
$person$	$\longrightarrow$	$(more, 1, 1) \cdot (more, 1, 1) \cdot (more, 1, 1)$	
$info$	$\longrightarrow$	$(textnode, 1, 1) \cdot (note, 1, 1)$	E_2
$note$	$\longrightarrow$	$(\varepsilon, 1, 1)$	E_3
$more$	$\longrightarrow$	$(\varepsilon, 1, 1)$	
$more$	$\longrightarrow$	$(\varepsilon, 1, 1)$	E_4
$more$	$\longrightarrow$	$(\varepsilon, 1, 1)$	

Table 1: Initial grammar rules extracted from documents in Fig. 4

For every group E_i we extract the right-hand sides of rules into set S_i of *positive examples* - input strings (words). Then for every group E_i a regular expression R_i has to be inferred using positive examples S_i from a regular language. The problem then boils down to the problem of

Problem 2 (Regular Expression Inference)**.** Given a set S of words (positive examples) over the alphabet of element names, we want to find a regular expression R, such that $S \subseteq L(R)$. □

The solution to this problem is limited by the solution of a subproblem incorporated in it, and that is the problem of learning a regular language from positive examples. In [20] has been shown, that no algorithm solving the latter problem exists. Although regular expressions allowed in DTD/XSD correspond to a subclass of regular languages called one-unambiguous regular languages, in [9] is shown that even for this class no such algorithm exists. Thus Problem 2 has to be solved either by defining a subclass of regular languages for which an algorithm capable of learning it from positive examples exists, or by ad-hoc heuristic measures.

A variation of Problem 1 is also studied in this work (in Sections 5.2.1, 5.2.3):

Problem 3 (XML schema inference using an old schema)**.** Given a set of input XML documents D and an old schema S_{old} of these documents (possibly not all documents are valid against the schema), we want to infer a new schema S_D for input documents, such that each document is valid against this new schema. It should be concise and precise enough, while at the same time, general. We should make it similar to the old schema or at least utilize the information provided by it. □

Problem 3 is solved using ideas from [31]. We extend this work by trying to solve a third problem:

Problem 4 (XML schema inference using an old schema and input document invalidation)**.** Given a set of input XML documents D and an old schema S_{old} for these documents (possibly not all documents are valid against the schema), we want to infer a new schema S_D for input documents, such that the majority of documents are valid against S_D. It should be concise and precise enough, while at the same time, general. If the old schema S_{old} is available on the input, we should make the new schema S_D similar to it or at least utilize the information provided by it. Some element instances $e \in E$ in input documents may be tagged as invalid, if they are excentric and it is believed that they should be repaired to be valid against the new schema, rather than making the new schema too complex by incorporating them into inference. □

The Problem 4 is faced mainly in Section 5.2.8 by newly developed heuristic measures.

4 Related Work

Several approaches to the problem of schema inference for a set of XML documents (see Problem 1) have been proposed. Some of them define an identifiable subclass of regular languages and develop algorithm to identify the subclass [5, 10, 12], others propose an ad-hoc heuristic, which infers no special subclass of regular languages [40, 34, 38].

4.1 Generating Grammars for Structured Documents Using Grammatical Inference Methods

Paper [5] solves Problem 1 using two identifiable subclasses of regular languages: k-contextual and (k, h)-contextual languages. "Our assumption behind is that the grammars used in structured documents have only limited context in the following sense. If a sufficiently long sequence of elements occurs in two places in the examples, the elements that can follow this sequence are independent of the position of the sequence in the document structure." [5] Let us quote [5, lemma 5.5, p. 39] as the definition of a k-contextual language:

> **Definition 24** (k-contextual language)**.** A regular language L is k-contextual if and only if there is a finite automaton M such that $L = L(M)$, and for any two states p_0 and q_0 of M and any string v with $|v| = k$ we have: if there are states p_k and q_k of M such that $\delta(p_0, v) = p_k$ and $\delta(q_0, v) = q_k$, then $p_k = q_k$ □
>
> An automaton M is said to be k-contextual, if it fulfills the conditions from Def. 24.

and [5, lemma 5.10, p. 44] as the definition of the (k, h)-contextual language:

> **Definition 25** ((k, h)-contextual language)**.** A regular language L is (k, h)-contextual if and only if there is a finite automaton M such that $L = L(M)$, and for any two states p_0 and q_0 of M, and all input symbols $a_1 a_2 \ldots a_k$ we have: if there are states $p_1, \ldots, p_k$ and $q_1, \ldots, q_k$ such that $\delta(p_0, a_1) = p_1, \delta(p_1, a_2) = p_2, \ldots, \delta(p_{k-1}, a_k) = p_k$ and $\delta(q_0, a_1) = q_1, \delta(q_1, a_2) = q_2, \ldots, \delta(q_{k-1}, a_k) = q_k$, then $p_i = q_i$, for every i with $0 < h \leq i \leq k$. □
>
> An automaton M is said to be (k, h)-contextual, if it fulfills the conditions of Def. 25.

Authors define both in another way following the previous work [32], and then prove equivalence with definitions based on automatons.

The inference proceeds as follows: first a PTA accepting all positive examples is constructed, then it is modified by merging states to obtain a k-contextual or a (k,h)-contextual automaton, which proceeds to a disambiguation procedure and, finally, the disambiguated automaton is converted to a regular expression.

Example 6. We take the example from [5]. Consider the following initial grammar rules for element Entry:

$$\begin{aligned}
Entry &\longrightarrow Headword, Inflection, Example, Example\\
Entry &\longrightarrow Headword, Inflection, Parallel\_form, Example, Example, Example\\
Entry &\longrightarrow Headword, Parallel\_form, Example, Example\\
Entry &\longrightarrow Headword, Preferred\_form, Example\\
Entry &\longrightarrow Headword, Inflection, Preferred\_form, Example, Example
\end{aligned}$$

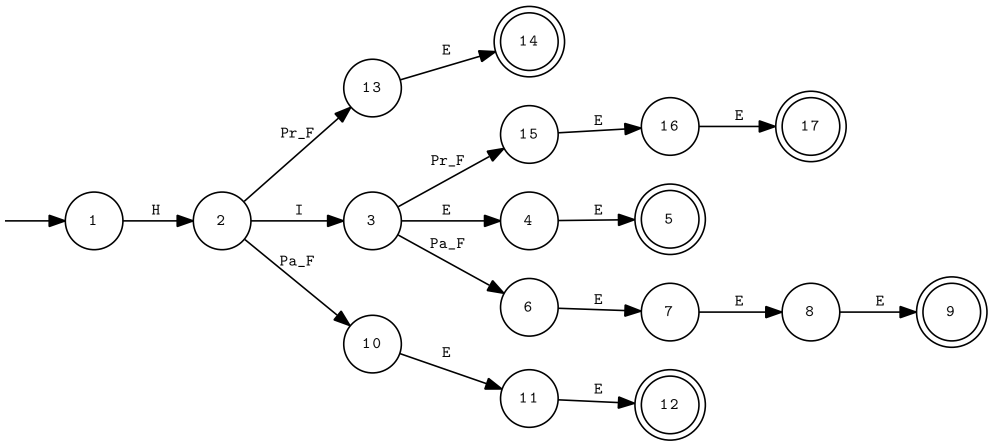
The prefix tree automaton built from these rules is depicted in Fig. 5(a), a 2-contextual automaton created by merging states $(14,16),(17,5,8,9,12),(7,11)$ is depicted in Fig. 5(b), and a $(2,1)$-contextual automaton created from the 2-contextual automaton by further merging states $(15,13),(4,16,7,5),(6,10)$ is depicted in Fig. 5(c). □

Furthermore, authors propose an algorithm for converting PTA into a (k,h)-contextual automaton in linear time in number of states. Large part of the work is dedicated to disambiguation of content model into 1-unambiguous content model (see [15] for 1-unambiguity).

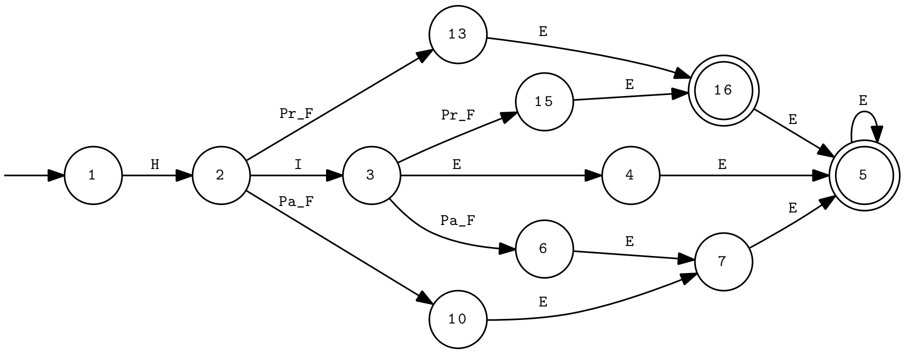
4.2 Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data

Works [10, 12, 9] are implemented into the SchemaScope [13] environment and are basically based on the following observation [10]:

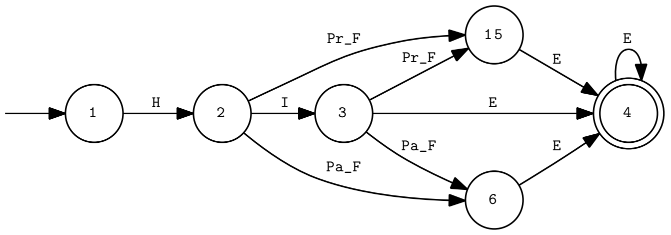
> In practice, regular expressions occurring in DTDs and XSDs are concise rather than arbitrarily complex. Indeed, a study of 819 DTDs and XSDs gathered from the Cover Pages (xml.coverpages.org) (including many high-quality XML standards) as well as from the Web at large, reveals that regular expressions occurring in practical schemas are such that every alphabet symbol occurs only a small number of times



(a) PTA constructed from the initial grammar



(b) 2-context automaton created by merging states of the PTA



(c) $(2,1)$-context automaton created by merging states of the PTA

Figure 5: An example for (k,h)-context

[29]. In practice, therefore, it suffices to learn the subclass of deterministic regular expressions in which each alphabet symbol occurs at most k times, for some small k. We refer to such expressions as *k-occurrence regular expressions.*

In this quotation references have been adjusted to guide the reader properly. As the authors state, their work focuses on inference of k-occurrence regular expressions, which are defined as follows

Definition 26 (k-occurrence regular expression (k-ORE)). A regular expression is k-occurrence if every alphabet symbol occurs at most k times in it. □

When $k = 1$, authors call such a regular expression as single occurrence regular expression (SORE). Inference of SOREs is studied in [9]. Authors prove that the class of 1-unambiguous regular expressions is not learnable in the limit [9] and prove that k-ORE is learnable in the limit. The authors then propose an algorithm for learning k-OREs and estimating the best k using various metrics.

4.3 The sk-strings method for inferring PFSA

In the work [34] first the PTA is constructed from given positive examples. Additionaly, used PTA contains statistical information from examples, in particular transition use counts are set during PTA construction. State final counts are represented as a special out-transition of the final state with the meaning "end of computation" (using a delimiter symbol "/" on such transitions). The automaton model is equivalent with Def. 21, and it is called PFSA in the work.

Then, the problem of state equivalence is taken literally from the other side - not the context preceding the state, but tails starting at the state are inspected. For this purpose, the authors define a k-string:

Definition 27 (k-string). Let $A = (Q, \Sigma, \delta, q_0, F)$ be PFSA, let $q \in Q$ be some state of A. The set of k-strings of state q (denoted as k-strings(q)) is defined as:

$$\text{k-strings}(q) = \{z | (z \in \Sigma^*, |z| = k \wedge (\delta(q, z) \subset Q \vee |z| < k \wedge \delta(q, z) \cap F \neq \emptyset)\}$$

That is, the set of all words over the alphabet Σ, where

- each word is either of length k and starting at state q, following transitions and reading symbols of the word, the automaton must not halt,
- or the word is shorter, and starting at state q, following transitions and after reading all symbols of the word, one must end in a final state.

Furthermore, the probability of the k-string is defined as a product of all probabilities of each transition traversed in generating that string. □

Several state sk-equivalence heuristics are proposed:

sk-OR The top $s\%$ of the k-strings of state 1 are k-strings at state 2, or vice-versa,

sk-AND The top $s\%$ of the k-strings of state 1 are k-strings at state 2, and vice-versa,

sk-LAX The top $s\%$ of the k-strings of state 1 is the same set as the top $s\%$ of the k-strings of state 2, in the same order, but perhaps with a different probability distribution,

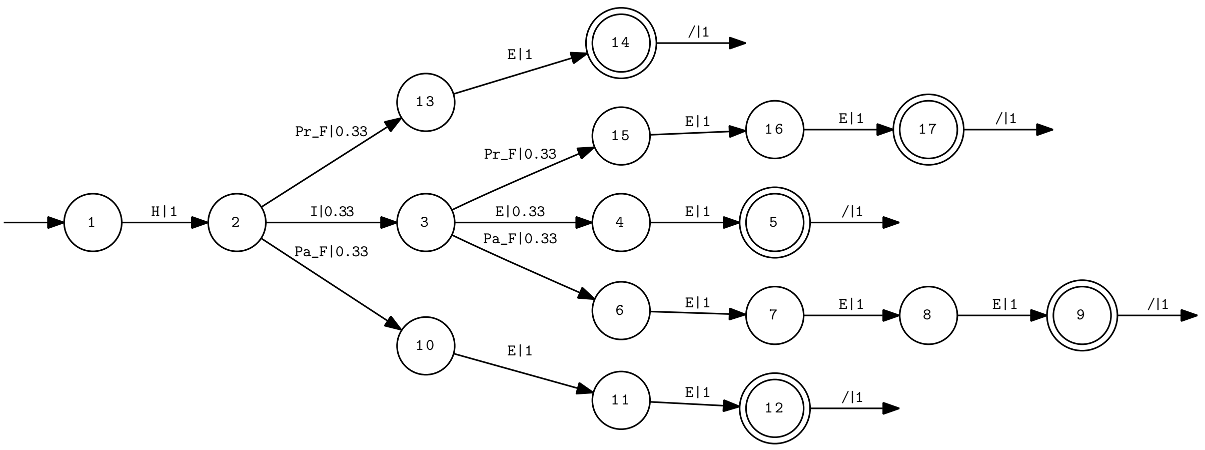
sk-STRICT The top $s\%$ of the k-strings of state 1 is the same set as the top $s\%$ of the k-strings of state 2 in the same order, with the same probability distribution.

The top s percent of state k-strings are computed by ordering k-strings in decreasing order (according to probability), then taking just enough of them (from the beginning) for which probabilities sum up to s. It is unclear what to do, when some strings have the same probability and one must decide which of them take into the top $s\%$ set.

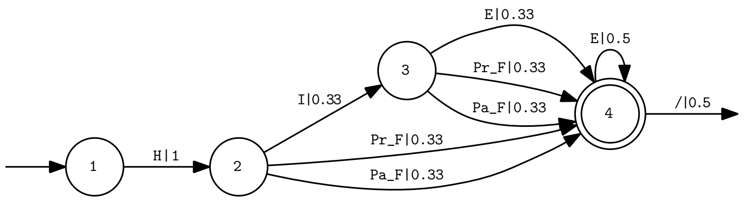
Example 7. Merging based on sk-equivalence is depicted in Fig. 6. The starting PTA is depicted in Fig. 6(a), then given $s = 100\%, k = 2$, sk-AND heuristics, merging of states $(6, 7)$ is performed, which results in a non-determinism at state 6, so it is solved by merging states $(6, 7, 8, 9)$. States $(13, 16, 4, 6, 11)$ satisfy the merge criterion and are merged. Once again a non-determinism appears at state 4, it is solved by merging states $(4, 14, 17, 5, 12)$. States $(10, 15)$ satisfy the criterion, so they are merged. The result is depicted in Fig. 6(b). □

Clearly, using greedy merging, one may end up in a too general automaton. Therefore, the authors define a measure to select the best trade-off automaton from many options. First, using different parameters of the sk-equivalence, possible automatons are generated, and then they are evaluated using the quoted MML [19] formula [39]:

$$\sum_{j=1}^{N} \left(m_j + \log \frac{(t_j - 1)!}{(m_j - 1)! \prod_{i=1}^{m_j} (n_{ij} - 1)!} + m_j \log V + m'_j \log N \right) - \log(N - 1)! \tag{1}$$



(a) PTA constructed from initial grammar, with respective probabilities on transitions



(b) An automaton created by merging states of the PTA, $s = 100\%, k = 2$, sk-AND heuristics

Figure 6: An example for sk-strings

where N is the number of states in the PFSA, t_j is the number of times the jth state is visited, V is the cardinality of the alphabet including the delimiter symbol, n_{ij} the frequency of the ith arc from the jth state, m_j is the number of different arcs from the jth state and m'_j is the number of different arcs on non-delimiter symbols from the jth state. The logs are to the base 2 and the MML is in bits.

The automaton with minimal MML value is selected.

4.4 On Structural Inference for XML Data

[40] builts on the previous work [34]. To search a wider space of solutions, authors implement the Ant Colony Optimization (ACO, see [18]) coupled with the previously defined MML measure (1). The ACO works in iterations. In each iteration, several artificial ants search the possible automaton generalizations using the sk-strings heuristic, after the whole iteration is done, each automaton inferred is measured by a MML and the trail followed by the corresponding ant is tagged with positive pheromone. The amount of pheromone is weighted with respect to the MML measure of the automaton inferred. Each ant has only a limited life -

a number of merges it can do until it dies.

The ant is selecting states to merge using the sk-strings heuristic alternatives. Each alternative has its *value* calculated as a combination of an immediate MML change h when alternative would be merged and the positive pheromone p left from previous iteration:

$$value = p^\alpha + h^\beta$$

where α, β are mixing parameters of the algorithm. From this *value*, the probability of selecting the alternative x is calculated:

$$P = \frac{value}{\sum_{\text{all alternatives}} value}$$

To compute the heuristic measure h, the authors use an immediate MML value change:

$$h = \frac{-\delta MML(A, merge) + MML(A)}{MML(A)}$$

where $\delta MML(A, merge)$ is the immediate change of MML value for the automaton A, when the *merge* alternative is applied. The value h is therefore a ratio of the new automaton MML value to the old automaton MML value. The pheromone spread by an ant is calculated by weighting an individual ant performance compared with average of the iteration:

$$p = \frac{averageMML}{mmlOf(a.solution)}$$

The authors finally propose a hybrid heuristic called sk-ANT, which combines the ACO and the sk-equivalence in such a way that for small automatons more merging alternatives are considered, thus the heuristic is more driven by ants searching the space. For large automatons, a stricter sk-equivalence applies and the heuristic is more driven by sk-equivalence criteria.

4.5 Even an Ant Can Create an XSD

In [38], the previous method is extended and refined. The main improvements are as follows: advanced element clustering considering not only element name, but the structure of element contents to identify distinct elements, and inference of a `xs:all` particle in XML Schema output. The XML document can be represented as a tree, then the element content is represented as a subtree of the document tree. The metric used to cluster elements according to their contents is a modified *tree edit distance* based on [33]. The clustering algorithm used is a a modification of mutual neighborhood clustering (MNC) algorithm [25]. The clustering of elements is a big improvement, since it is common that two elements with same element name, but with a different semantic appear in XML files. For example element named "name" is commonly used, and in a book archive may be used as a name of the author with subelements surname, etc. or as a name of the book, where the schema should define only string content model. These two versions of the element name can be exploited using the element clustering proposed in [38].

The second big addition of the work is an ability to infer `xs:all` particle and thus shortening the regular expression in the output schema (in comparison to a complicated regular expression naming nearly all possible permutations).

Minor improvements were added to the ACO heuristic, particularly, the negative feedback, which is assigned to merge alternative immediately (not after whole iteration) forces ants to explore even wider space of possible solutions. The difference is also in the evaluation of a solution. In [40] the PFSA is evaluated using MML, but in [38], each PFSA is first converted into a corresponding regular grammar and that grammar is evaluated using a simple MDL measure.

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

We build our solution also on work [31]. It deals with the problem of schema inference given not only XML input documents, but with the knowledge of an old schema of these files. Not all input documents have to be valid against the old schema. The work proposes merging of grammar rules from the initial grammar with rules extracted from the schema into one common automaton. This concept helps with schema generalization, all input documents are valid against newly inferred schema, while the schema is kept as similar with the old one as possible.

The authors propose the concept of schema specialization, where schema constructs unused in input documents should be removed from the schema. Inference of tight bounds of `minOccurs`, `maxOccurs` are proposed in the [31].

The majority of the proposed solutions from [31] were implemented in this work.

5 Proposed Solution

The proposed solution is a composition work built on ideas from [38, 5, 34, 31]. The work provides a complete schema generation environment, incorporated into the jInfer framework [28], which makes it ready-to-use for potential users. The main improvements introduced in the proposed solution are as follows:

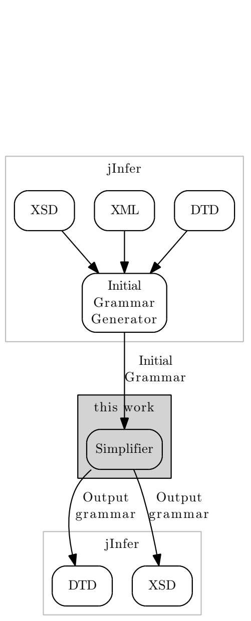
- schema files input - possibly an old schema of XML documents on input (implementation of [31]) - the user can select whether to use a schema inferredon the basis of the old one or a schema inferred purely from input XML documents,
- new (more accurate) MDL measures of automaton and input data,
- possibility to automatically tag selected input grammar rules as invalid, excluding them from inference (deviations, misspelled words or too excentric data) and thus generate more accurate schema for valid inputs (and letting the user repair invalid parts of input documents),
- great and easy extensibility of proposed algorithms and ready to use environment.

. Thorough description of whole solution together with the proposed improvements follows. Sections 5.2.7, 5.2.1 and 5.2.8 contain descriptions of main improvements.

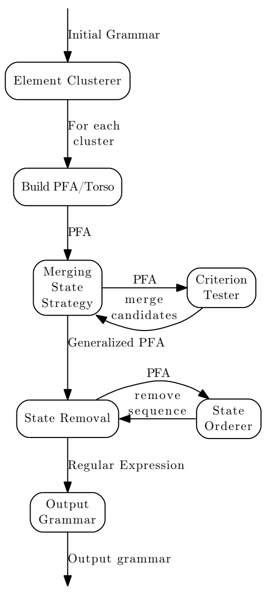
5.1 Solution Overview

We follow a two step design proposed in [38]. First, positive examples (element instances from input documents) are clustered, grouping instances of XML elements corresponding to one element type definition into one cluster. Input modules of jInfer framework ([28]) parse XML schema files into grammar rules. These are clustered together with element instances originating from XML documents. Clustering of both is done based on the element name. After the clustering, each cluster can contain one rule from schema input files and zero or more rules from XML input documents.

Since in both XSD and DTD the element content model is basically specified by a regular expression, we consider positive examples (actual contents of XML elements in input XML files) as being generated by some DPFA (as defined in Def. 21) and try to infer this automaton. If there is a regular expression from schema input, then a DPFA torso is constructed (see Section 5.2.1) from it.



(a) Solution in context of jInfer framework



(b) Solution in detail

Figure 7: Solution overview

Starting with an empty automaton or with a torso, we run the same algorithm for building DPFA in the form of PPTA from positive examples (see Section 5.2.2).

The automaton is then modified by merging its states. When states are merged, the language generated by the automaton becomes more general. Merging states process is driven by *merge criterion testers*, which search for candidate states for merging.

The inferred automaton is then converted into an equivalent regular expression using the state removal algorithm (see [24]) and the regular expression is added into a list of all XML element definitions (output grammar). On the output, schema is generated by naming all element definitions from output grammar and specifying their content model definitions in the selected schema language (DTD or XML Schema). Converting of output grammar into the schema language is done using output modules of the jInfer framework (see [28]). The whole process is depicted in Fig. 7.

5.2 Automaton Inference

As stated before, element instances of one cluster, found in input XML documents are considered to be generated by DPFA. If there is a regular expression extracted from the schema input file, automaton torso is built at first. Then the algorithm of building probabilistic prefix tree automaton takes place.

5.2.1 Building DPFA Torso from Regular Expression

A grammar rule of each element definition are extracted from input schema files. The right-hand side of the rule is a regular expression - content model of the element defined in the schema.

To build an DPFA torso from this regular expression, Algorithm 1 and Alg. 2 are used. The transitions are set up by this algorithm with use counts set to 0. As schema is assumed to be the model of data, use counts are to be incremented when traversing these transitions in Algorithm 4 to extend this automaton torso to generate all input strings. Analogical explanation holds for final count values of states. The process of building the DPFA torso is depicted in Fig. 8.

Algorithm 1 Build DPFA Torso

```
Input:
    $R$ - A set of initial grammar rules extracted from schema files
    $A$ - an empty DPFA
Output:
    Modified automaton $A$, such that it would generate a language corresponding
    to regular expressions in $R$, if it had correct probabilities set up
1: for each $r \in R$ do
2:     $buildTorsoOnRegex(A, q_0, r.getRightHandSide())$
3: end for
```

5.2.2 Building DPFA as Probabilistic Prefix Tree Automaton

Algorithm to build a PPTA is simple: for each input string (positive example), read its symbols and follow transitions of the automaton while it is possible (transition exists). Then create new transitions and states while reading symbols of the string. The last state created in this process gets incremented its final count value, also increment use count values of transitions along the way. After reading one input string, one is sure, that the automaton is extended to generate the string. The algorithm is depicted in Algorithms 4, 5, and the building process is depicted in Figures 9, 10.

If there is an automaton torso on input of this algorithm, everything works the same way, but instead of building the automaton as a prefix tree, it is a mixture

Function 2 *buildTorsoOnRegex*(A, q, S)

Input:

A - Automaton

q - State to start building from

S - Regular expression, denote its components as $(R, m, n) = S$

Output:

Creates a structure corresponding to S starting at state q recursively. Returns the state, in which building ended.

```
1: if $(R, m, n) = (\varepsilon, 0, 0)$ then
2:     $F(q) \leftarrow F(q) + 1$
3:     return $q$
4: else if $(R, m, n)$ is in the form of token: $(R, m, n) = (a, 1, n); a \in \Sigma$ then
5:     $q' \leftarrow \delta(q, E)$                      if there is a transition, follow it
6:     if $q' = \lambda$ then                          there is no transition, create it
7:         $q' \leftarrow createNewState(A)$          create a new state
8:         $\delta(q, a) \leftarrow q'$                  add transition
9:         $P'(q, a, q') \leftarrow 0$                   new transition is not used
10:    end if
11: else if $(R, m, n)$ is in the form of concatenation: $R = (S_1, 1, n_1) \cdot (S_2, 1, n_2) \cdot \ldots \cdot (S_i, 1, n_i)$
    then
12:    $q' = q$
13:    for each $(S_k, 1, n_k) \in R$ do
14:        $q' = buildTorsoOnRegex(q', (S_k, 1, n_k))$ recurse, keep the last state
15:    end for
16: else if $(R, m, n)$ is in form of alternation: $R = (S_1, 1, n_1)|(S_2, 1, n_2)| \ldots |(S_i, 1, n_i)$ then
17:    $E$ = new list
18:    for each $(S_k, 1, n_k) \in R$ do
19:        $E.addLast(buildTorsoOnRegex(q, S_k))$ build alternations
20:    end for
21:    $mergeStates(A, E)$                    merge ending states into the first state from the list
22:    $q' = E.getFirst()$
23: else
24:    $error$                                   permutation is not supported
25: end if
26: if $n = \infty$ then                          occurrence is unbounded (*)
27:    $mergeStates(A, q, q')$                use merging of states to produce cycle
28:    return $q$                                 building should continue from here
29: end if
30: return $q'$
```

Function 3 *createNewState*(A)

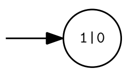
Input:

A - Automaton

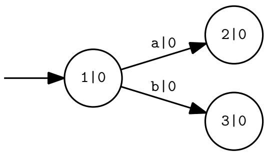
Output:

Creates new state q', with no out transition and returns it.

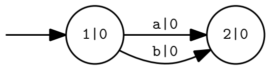
```
1: $q' \leftarrow$ new state                      create new state
2: $Q \leftarrow Q \cup \{q'\}$                    add new state to the set of states
3: $F(q') \leftarrow 0$                          new state final count set to 0
4: for each $x \in \Sigma$ do
5:     $\delta(q', x) \leftarrow \lambda$                no transition out of new state
6: end for
7: return $q'$                                  return the new state
```



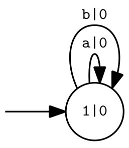
(a) At the beginning, empty automaton



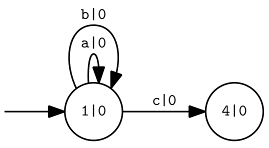
(b) After recursing once to solve alternation, before line 21 is executed. State q is the state 1|0



(c) After line 21 is executed, at the next line, q' is set to the state 2|0



(d) After line 27 is executed, the Kleene closure is solved by merging of states q and q'



(e) After the building is complete. The automaton implementation never denotes two states with same name (that is why the name 4 is used for the new state)

Figure 8: Building of a DPFA torso from regular expression $(a|b) * c$

of the given torso (remember that the algorithm follows transitions if they exist) and some branches to generate strings not represented by the regular expression from schema input.

Algorithm 4 Build DPFA as PPTA

```
Input:
    $R$ - A set of initial grammar rules
    $A$ - An empty automaton
Output:
    Automaton $A$ which generates strings from right-hand sides of $R$.
1: for each $r \in R$ do
2:     $buildPTAonString(A, r.getRightHandSide())$
3: end for
```

5.2.3 Automaton Minimization

Remember the Fig. 9, where, after reading all input strings, transitions with use counts set to zero remained. Such transitions originating from DPFA torso and never used in real-worl XML data are useless in the schema, so they are removed by procedure called automaton minimization (see [24]). When minimizing an automaton, first, all transitions with 0 use count are removed. Then the automaton is searched for states that remained unreachable from the initial state and states,

Function 5 *buildPTAonString*(A, S)

Input:

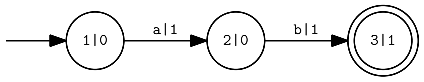
S - Input string (sequence of alphabet characters)

A - Automaton

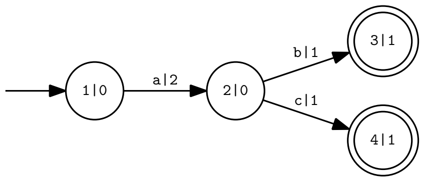
Output:

Modifies automaton A to generate S

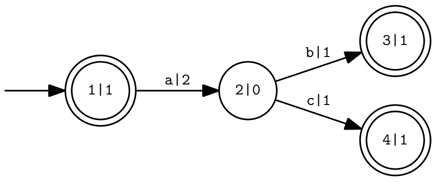
```
1:  $q = q_0$
2:  for each $a \in S$ do                      traverse symbols of S
3:      $q' \leftarrow \delta(q, a)$
4:      if $q' = \lambda$ then                 no transition exists
5:          $q' \leftarrow createNewState(A)$  create new state
6:          $\delta(q, a) \leftarrow q'$       add transition
7:          $P(q, a, q') \leftarrow 0$         new transition is not used
8:      end if
9:      $P(q, a, q') \leftarrow P(q, a, q') + 1$   increment transition use count along the way
10:     $q \leftarrow q'$                      move on to the next state
11: end for
12: $F(q) \leftarrow F(q) + 1$                 increment the final count
```



(a) After parsing input string ab

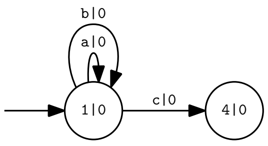


(b) After parsing input strings ab, ac

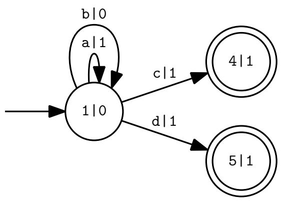


(c) After parsing input strings ab, ac, ε

Figure 9: An example of building DPFA as PPTA

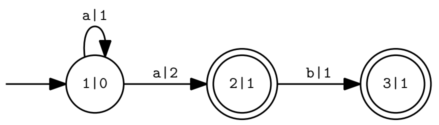


(a) DPFA torso from Fig. 8(e)

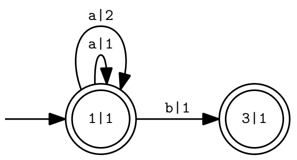


(b) After parsing input strings ac, d

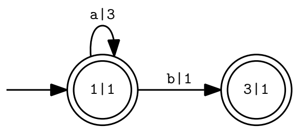
Figure 10: An example of building DPFA as PPTA starting from DPFA torso



(a) An example DPFA



(b) After merging of state 2 into state 1. Final count of state 1 is already incremented



(c) After transition and loop collapsing of state 1

Figure 11: An example of merging states process

from which no final state can be reached and these are removed as well. Minimization is done at the beginning of each Merging State Strategy (see Section 5.2.6).

5.2.4 Merging States

Merging two states proceeds as follows: select one state as preserved (usually the first one) and the second one to be removed from the automaton. All in-transitions of the removed state are reconnected to the preserved state. All out-transitions of the removed state are reconnected to begin from the preserved state, and all loops of the removed state are copied to the preserved state.

With DPFA, a care must be taken, to preserve invariant of use count and final count properties. After the merging, all in-transitions of the preserved state are divided into clusters by the transition source state. For each cluster of in-transitions, second grouping by the alphabet symbol is performed. Finally, each group is ensured to have exactly one transition: if there are 2 or more transitions in the group, these are merged into one transition so that the use count is set as the sum of merged transitions. Analogical algorithm is run for out-transitions and loops. We call this process “collapsing state transition and loops” and it preserves the use count property. To preserve the final count property, the preserved state final count value is simply incremented by the removed state final count value. Merging states and collapsing transitions is depicted in Fig. 11.

5.2.5 Selecting States to Merge

To select states to merge in some clever way we employ two state equivalence criteria: sk-strings [34] heuristic criterion and k, h-context [5] criterion. Modules responsible for providing candidate alternatives for merging are called *Merge Criterion Testers*. From the available alternatives a module called Merging State Strategy can select which to merge and which not (cooperation of these two blocks is depicted in Fig. 7).

5.2.6 Merging State Strategies

We have implemented several Merging State Strategies called: Greedy, Greedy-MDL, HeuristicMDL and DefectiveMDL. First three are no surprise, the last is our new proposal (it is discussed in Sec. 5.2.8).

Greedy The Greedy strategy simply merges all the candidate states provided by Merge Criterion Testers. For example for the k, h-context tester this means, that it simply creates a k, h-context automaton as defined in [5]. Algorithm is depicted in Alg. 6.

Algorithm 6 Greedy Merging State Strategy

```
Input:
    $M$ - A list of Merging State Testers
    $A$ - An automaton
Output:
    Returns an automaton modified by merging all alternatives provided to it
    from testers
 1: repeat
 2:     $L \leftarrow$ empty list
 3:     for each $m \in M$ do
 4:         $L.addAll(m.getAlternatives(A))$
 5:     end for
 6:     for each $alternative \in L$ do
 7:         for each $listOfStates \in alternative$ do
 8:             $mergeStates(A, listOfStates)$
 9:         end for
10:     end for
11: until $L = \emptyset$
12: return $A$
```

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

While trying to merge candidate alternatives, the GreedyMDL strategy always keeps current minimum quality value achieved (and the associated automaton). A space of possible solutions is explored in greedy way, but some sort of a complete scanning of continuation possibilities is done: all candidate alternatives to merge are tried. The algorithm stops when there are no more candidates to merge, or when all alternative candidates returned by merge criterion testers end up in an automaton with higher quality value than actually achieved. The algorithm is depicted in Alg. 7.

Algorithm 7 GreedyMDL Merging State Strategy

```
Input:
    $M$ - A list of Merging State Testers
    $A$ - An automaton
    $S$ - Input strings (positive examples from XML documents)
    $mdl(A, S)$ - Quality measure function
Output:
    An automaton modified by merging only those alternatives provided by
    testers, which lead to lower quality value
 1: $\varphi_m \leftarrow mdl(A, S)$                    Quality of best automaton
 2: repeat
 3:    $staggering \leftarrow true$
 4:    $L \leftarrow$ empty list
 5:    for each $m \in M$ do
 6:       $L.addAll(m.getAlternatives(A))$
 7:    end for
 8:    $A' \leftarrow clone(A)$                          Test it on copy of automaton
 9:    for each $alternative \in L$ do
10:       for each $listOfStates \in alternative$ do
11:          $mergeStates(A', listOfStates)$
12:       end for
13:       $\varphi \leftarrow mdl(A', S)$
14:       if $\varphi < \varphi_m$ then
15:          $\varphi_m \leftarrow \varphi$
16:          $A \leftarrow A'$
17:          $staggering \leftarrow false$
18:       else
19:          $A' \leftarrow clone(A)$
20:       end if
21:    end for
22: until $staggering = true$
23: return $A$
```

HeuristicMDL The HeuristicMDL as a simple heuristic strategy works basically the same way as GreedyMDL, but it holds n best minimal solutions instead of only one. At each iteration, merge criterion testing for one randomly selected automaton of the n best automatons is done. All the alternatives returned are attempted to be merged and only the automatons with lower quality value than the current worst solution are stored in capacity-constrainted sorted list (thus it always holds the best n solutions). The algorithm stops when it is staggering - when the set of best n solutions is not modified for a whole iteration. The algorithm is depicted in Alg. 8.

Algorithm 8 HeuristicMDL Merging State Strategy

Input:

M - A list of Merging State Testers
A - An automaton
S - Input strings (positive examples from XML documents)
$mdl(A, S)$ - Quality measure function
n - How many intermediate solutions to keep

Output:

An automaton modified by merging states with heuristic approach

```
 1: $solutions \leftarrow$ new sorted list of pairs
 2: $\varphi_{old} \leftarrow mdl(A, S)$
 3: $solutions.insert((\varphi_{old}, A))$
 4: $stagger \leftarrow 0$
 5: repeat
 6:     $A' \leftarrow randomSelectFromList(solutions)$
 7:     $L \leftarrow$ empty list
 8:     for each $m \in M$ do
 9:         $L.addAll(m.getAlternatives(A))$
10:     end for
11:     $A'' \leftarrow clone(A')$
12:     for each $alternative \in L$ do
13:         for each $listOfStates \in alternative$ do
14:             $mergeStates(A'', listOfStates)$
15:         end for
16:         $\varphi \leftarrow mdl(A'', S)$
17:         if $\varphi < solutions.last.\varphi$ then
18:             $solutions.insert((\varphi, A''))$
19:         end if
20:         if $solutions.size > n$ then
21:             $solutions.removeLast()$
22:         end if
23:     end for
24:     if $solutions.first.\varphi = \varphi_{old}$ then
25:         $stagger = stagger + 1$
26:     end if
27:     $\varphi_{old} \leftarrow solutions.first.\varphi$
28: until $stagger < 10$
29: return $solutions.first.A$
```

5.2.7 Objective Quality Function (MDL)

Before we describe DefectiveMDL strategy, we stop to describe the MDL designed to evaluate solutions. In the sense of the crude MDL (see Sec. 2.3.2), one has to design a code for a hypothesis and a code for data compressed using the hypothesis. Since in this work a basic assumption is that positive examples were generated by some DPFA, the hypothesis is the DPFA itself. And as described in Sec. 2.3.3, if a hypothesis is of probabilistic character, the best code to use is the complete prefix code with codelengths equal to $-\log(p)$ for the one option, whose probability of appearance in data equals to p.

When generating strings using the DPFA in each state of the automaton the algorithm decides which transition to follow or whether to output a whole word (end generation process) at random. This random choice is always driven by a probability density function defined by probabilities:

- of each transition that it is followed,
- the actual state that it is final.

Given a state q, the computation of these probabilities is straightforward. First we compute a unity value

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

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

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

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

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

Using the set of random variables X_q (one for each state q), encoding input strings is simple. When the automaton generates a string, the configuration sequence is the same, as if it had the string on input and the automaton was accepting the string. Thus computing a codelength of all input strings can be done easily - for each input string, traverse automaton while reading it and record probabilities of transitions along the way.

Let us consider input string $s = a_1, a_2, \ldots, a_n$. Let probabilities $p_1, \ldots, p_n$ be recorded transition probabilities, and p_{n+1} the probability f'_q of the state, where reading of the string ended (accepted). Codelength C of the string s then equals to

$$C(s) = \sum_{i=1}^{n+1} -\log(p_i).$$

By summing logarithms one gets

$$C(s) = -\log\left(\prod_{i=1}^{n+1} p_i\right). \tag{2}$$

This naturally corresponds with seeing the problem from the other side: probability p_s of the whole string generated equals the product of probabilities of decisions taken at each configuration. If one designs a complete prefix code over probability density function of probabilities p_s, the codelength of a single string equals exactly (2).

There is no need to traverse the automaton with each input string to obtain the total codelength of all input strings. When DPFA was built, each input string incremented the use count value of each transition passed and incremented the final count value of the state it ended in. When DPFA is traversed for each input string, each transition is passed exactly its *use count* times and traversing ends in each state exactly its *final count* times. From this, it is easier to compute the total codelength of input strings S (encoded with help of DPFA A) as

$$L(S|A) = \sum_{q \in Q, a \in \Sigma, q' \in Q} \left(P(q,a,q') \cdot -\log\left(\frac{P(q,a,q')}{u_q}\right)\right), \tag{3}$$

where $\{u_q | q \in Q\}$ are precomputed unity values for each state.

The key problem is how to encode the DPFA. There is no universal way, DPFA is an ad-hoc model to solve a custom ad-hoc problem and we propose an ad-hoc code for it. Let us denote the states of an automaton as $q_1, \ldots, q_{|Q|}$. The proposed code is

$$|Q|, |\Sigma|, \mathit{alphabet}, \langle q_1 \rangle, \ldots, \langle q_{|Q|} \rangle. \tag{4}$$

Where $|Q|$ is the cardinality of the set of states encoded using standard universal code for integers (SUCI, see Sec. 2.3.4), $|\Sigma|$ is the cardinality of Σ encoded using SUCI. The symbols of an alphabet are named using a uniform code in the table *alphabet*, which is a translation table from uniform encoding of each symbol into a prefix code. The prefix code for alphabet symbols is established using a histogram of symbol occurrences over all transitions of the automaton. Probability of an individual symbol a for this code is therefore

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

Each $\langle q_i \rangle$ is a code of one state in the automaton:

$$|Q'_i|, \langle t_1 \rangle, \ldots, \langle t_{|Q'_i|} \rangle \tag{5}$$

Where Q'_i is a set of all states immediately reachable from the state q_i, formally $Q'_i = \{q; q \in Q, \exists a \in \Sigma : \delta(q_i, a) = q\}$. Each $\langle t_j \rangle$ is a code of one out-transition of the state q_i . Each in form:

$$q, P(q,a), a \tag{6}$$

Where q is a code for a destination state encoded using a uniform code over all states, $P(q,a)$ is a use count value of the transition encoded using SUCI, a is a symbol of alphabet encoded using the prefix code for the alphabet established earlier.

Let us denote p_a the probability of each symbol a in the alphabet. Then the codelength of this ad-hoc code of the automaton is:

$$\begin{aligned} L(A) \quad = \quad & \mathit{suci}(|Q|) + \mathit{suci}(|\Sigma|) + |\Sigma| \cdot \log(|\Sigma|) - \log\left(\prod_{a\in\Sigma} p_a\right) + \\ & + \left(\sum_{i=1}^{|Q|} \mathit{suci}(|Q'_i|)\right) + \\ & + \sum_{i=1}^{|Q|} \sum_{j=0}^{|Q'_i|} \left(\log(|Q|) + \mathit{suci}(P(q,a)) - \log(p_a)\right), \end{aligned} \tag{7}$$

where

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

The whole codelength function is given as the sum of (3) and (7):

$$mdl(A, S) = L(S|A) + L(A) \tag{8}$$

And since S is not used anywhere in computation of $L(S|A)$, it may be omitted as a parameter of $mdl(A, S)$. Precisely, it should be removed from Algorithms 7, 8.

5.2.8 DefectiveMDL Merging State Strategy

A user can chain merging state strategies and what really makes sense, is to attach this strategy at the end of chain. Now, the automaton is ready to be converted into a regular expression, but before the conversion step, we propose DefectiveMDL strategy. It is an algorithm to decide which input strings are so excentric that they probably are "mistakes" and should be repaired in input documents rather then incorporated into the output schema. The strategy to identify possibly excentric input strings is based on the following ideas. Let T be the set of input strings we suspect as excentric. Try removing T from the inference process if the inferred schema can be much simpler then, we consider T to be excentric. But this would simply remove all input strings, since no documents fit `EMPTY` construct exactly.

Here, a trade-off thinking applies and that is where MDL can help. Try to remove input strings T and if the MDL value $mdl(A, S \setminus T)$ is smaller enough than the value $mdl(A, S)$ consider T as excentric. To formalize the "smaller enough", we define a criterion: When the description length of a new automaton and input strings, together with the description length of the removed input strings (encoded by typing them in alphabet) is smaller than the description length of an old automaton + all input strings, the strings are considered excentric. To formalize this, we define an error code for one input string $a_1, \dots, a_n$ as a sequence of prefix codes for each symbol (established the same way as in the previous code (7)), thus a codelength of the error code for one input string equals to

$$L_{error}(s) = \sum_{i=1}^{n} -\log(p_{a_i}) = -\log\left(\prod_{i=1}^{n} p_{a_i}\right) \tag{9}$$

where p_{a_i} is the probability of symbol a_i in the established prefix code for the alphabet. Since by removing input strings it may occur that we also remove some symbol from the alphabet used in the automaton, the prefix code for the alphabet is established with a histogram not only over the automaton, but also over removed input strings. Basically, the prefix code for the alphabet remains the same, since it has to encode all input strings no matter they are used in the automaton or in the error code. So the codelength of the error code of all removed strings is

$$L_{error}(S) = \sum_{s \in S} L_{error}(s) \tag{10}$$

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

$$mdl(A, S, S_r) = mdl(A, S) + L_{error}(S_r) \tag{11}$$

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

A complete algorithm of DefectiveMDL is depicted in Alg. 9. Input strings are removed from the automaton easily: the automaton is traversed while reading an input string (remember that the automaton is deterministic) and each transition gets its *use count* value decremented along the way. The final state gets its *final count* value decremented. Since only strings that previously builtup the automaton are removed, use counts and final counts can never reach negative values. Function $tryRemoveInputString(A, s)$ is depicted in Alg. 10, function

Algorithm 9 DefectiveMDL Merging State Strategy

```
Input:
    $A$ - An automaton
    $S$ - Input strings (positive examples from XML documents)
    $mdl(A, S, S_r)$ - a quality measure function (see Equation (11))
Output:
    An automaton possibly modified by removing input strings
 1: $S_r \leftarrow \emptyset$                          Strings removed until now
 2: $S' \leftarrow getSuspectedStrings(A, S)$
 3: while $S' \neq \emptyset$ do
 4:     $\varphi \leftarrow mdl(A, S, S_r)$
 5:     for each $s \in S'$ do
 6:         $tryRemoveInputString(A, s))$
 7:     end for
 8:     $\varphi' \leftarrow mdl(A, S \setminus S', S_r \cup S')$
 9:     if $\varphi' \geq \varphi$ then
10:         for each $s \in S'$ do
11:             $undoRemoveInputString(A, s))$
12:         end for
13:     else
14:         $S \leftarrow S \setminus S'$
15:         $S_r \leftarrow S_r \cup S'$
16:     end if
17:     $S' \leftarrow getSuspectedStrings(A, S)$
18: end while
19: return $minimalize(A)$
```

$undoRemoveInputString(A, s)$ works oppositely (incrementing use counts and final count). The remaining question is which input strings to try to remove? We propose a program interface called Suspect, which should return input strings it is suspecting as excentric. Checking strings one by one is one simple strategy implemented. However, with removing one input string, the automaton may not become any simpler. So we propose so-called transition suspecting. If we remove all input strings that pass one transition, the transition is rendered as unused, so it is removed by automaton minimization resulting in a simpler schema.

This merging strategy is called defective, since not all input strings are represented by the output automaton.

5.3 Obtaining Regular Expression

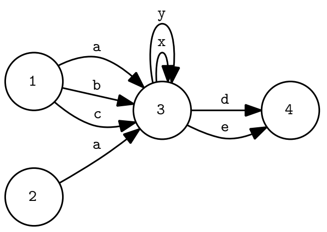
When the merging strategy finishes its work, the automaton is considered to be the best trade-off representation of XML element's content model. The automaton is converted into a regular expression by a state removal algorithm (see [24]). As the order in which states are removed influences the regular expression complexity, we provide an interface to implement various heuristics to select a proper

Algorithm 10 tryRemoveInputString(A, s)

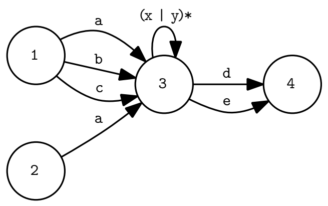
```
Input:
    $A$ - An automaton
    $s$ - An input string
Output:
    An automaton without an input string
1: $q = q_0$
2: for each $a \in s$ do                       traverse symbols of s
3:     $q' \leftarrow \delta(q, a)$
4:     $P(q, a, q') \leftarrow P(q, a, q') - 1$          decrement transition use count along the way
5:     $q \leftarrow q'$                          move on to the next state
6: end for
7: $F(q) \leftarrow F(q) - 1$                     decrement the final count
```

state to remove each time. Two basic strategies of removing states are implemented in this work: ordered and heuristic. Ordered strategy is a modification of the state removal algorithm described in [24]. First, super final and super initial states are created. The super initial state precedes the initial state and has only one ε-transition leading to the initial state. For each state with positive final count, a new ε transition to the super final state is created. The algorithm then removes one state at a time (except for super initial and super final states). The process of removing one state is thoroughly explained in [24] and it is depicted in fig. 12. The decision which state to remove is delegated to an Orderer submodule. After removing all but 2 super-states, the automaton can have several transitions from the super initial to the super final state (no transition can have opposite direction). These transitions are combined into one transition with an alternation regular expression and that regular expression is the result. The state removal ordered strategy is depicted in Alg. 11. Orderer based on state weighting [17] is used as the default (according to [21] it still outperforms other methods). It is possible to replace the submodule of Orderer which measures the regular expression length. Originally, [17] proposes the length of regular expression to be a terminal count. We add two more length functions of regular expression: a number of characters needed to represent the regular expression in DTD syntax (complete with $(), *?$ characters) and a custom bitcode.

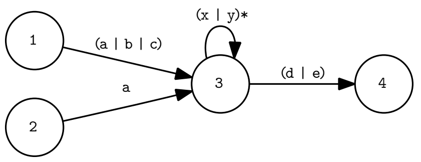
Heuristic strategy is a form of "lookahead", it does basically the same, but tries to remove several different states in one step and keeps best n solutions in each step. It is very similar to Alg. 8 in basic heuristic design.



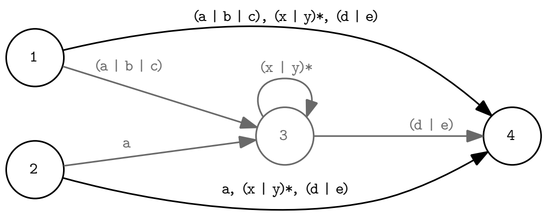
(a) An example automaton, going to remove state 3



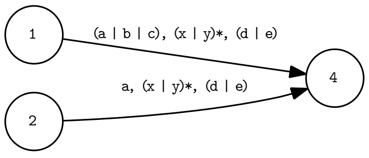
(b) State loops are combined into one loop with an alternation regular expression of original loops



(c) Multiple in/out-transitions ending in the same source/destination state are combined to form one transition with an alternation regular expression of original ones



(d) For each combination of in-transition and out-transition, a new bypass transition is added with a concatenation regular expression



(e) Finally, the state and transitions are removed

Figure 12: An example of removing a state

Algorithm 11 State Removal Ordered

Input:
A - An automaton

Output:
A regular expression representing the same language as the input automaton

```
 1: $q_{si} = createNewState(A)$            create super initial state
 2: $\delta(q_{si}, \varepsilon) \leftarrow q_0$            $\varepsilon$ transition to the original initial state
 3: $q_{sf} = createNewState(A)$            create the super final state
 4: for each $q \in Q$ do
 5:     if $F(q) > 0$ then
 6:         $\delta(q, \varepsilon) \leftarrow q_{sf}$            out transition to super final state
 7:     end if
 8: end for
 9: while $|Q| > 2$ do
10:     $q \leftarrow getStateToRemove(A)$
11:     $removeState(A, q)$
12: end while
13: return $getCombinedRegex(A)$
```

6 Implementation

All features described in Section 5 were implemented in the jInfer framework [28], implying that we use the NetBeans Platform [3] and the Java Platform [2]. Overview of all modules in context of jInfer framework is depicted in Fig. 13.

Packages and classes not mentioned in this section are a part of jInfer framework, not this work. Packages mentioned in this section are part of this work, together with all their subpackages, unless stated otherwise. Packages named `properties` inside packages that are mentioned in this section, are part of this work, but contain properties panels to present the configuration properties of a module to the user, and thus are not very interesting. Nevertheless, we mention them to emphasize the fact that all modules in the work are configurable using configuration panels with a user-friendly interface and so the work is ready to use by potential users (it is not only an experimental implementation, but a ready-to-use solution).

Package `cz.cuni.mff.ksi.jinfer.base.automaton` contains our PFSA implementation. Class `Automaton` is the automaton itself, its method `mergeStates()` merges states exactly as described in Section 5.2.4. The thing to note is that our automaton implementation is always aware of states removed when merging states. Merging criterion testers searching an automaton for states to merge do not have to bother with details which naturally appear, such as: states 2,3 are considered equivalent and states 3,4 are considered equivalent, but when the first merge occurs, the state 3 disappears and the second request would be invalid.

The place to look in the source code of jInfer where the root of this work is located, is package `cz.cuni.mff.ksi.jinfer.twostep.processing.automaton mergingstate`. Here, merging state algorithms are implemented.

The subpackage `conditiontesting` contains implemented merge condition testers: `sk-strings`, (k, h)`-context` and `combined` (in which a user can select a combination of other testers). The `deterministic` tester is used only to make an automaton deterministic when it is needed, it only searches for non-determinisms. New merge condition tester can be added following the tutorial [26], and implementing interfaces `MergeConditionTester(Factory)` defined in this package.

The subpackage `evaluating` contains all MDL-related evaluation classes. The `automatonNaiveAlphabet` and the `automatonNaiveDefective` are part of proposed solution, the former (calculating the value of equation (8)) is proposed to be used with GreedyMDL and HeuristicMDL merging state strategies (see Section 5.2.6), and the latter (calculating the equation (11)) with the DefectiveMDL strategy (see Section 5.2.8). The package `universalCodeForIntegers` contains what it claims, the class for calculating the SUCI (see Section 2.3.4). Packages `regexp*` contain various metrics of regular expressions and are used in the state removal strategy (see Section 5.3) of converting an automaton into a regular expression to weight states of the automaton. Once again, new evaluators can be added easily by implementing interfaces defined in the package `evaluating`.

The subpackage `simplifying` contains strategies for generalizing the PPTA/Torso (see Sections 5.2.2, 5.2.1). Apart from strategies described in Section 5.2.6 which are in appropriate packages, the package contains also the `chained` strategy, which enables user to chain 2 or more strategies using a properties window. In the package `khgrams`, we have implemented the method [5], which creates a (k, h)-context automaton. Same effect can be achieved using the Greedy strategy and the (k, h)-context merge condition tester. The package `userinteractive` is a part of jInfer framework. Extensibility is ensured through `AutomatonSimplifier(Factory)` interfaces.

The subpackage `regexping` contains the state removal method of converting a PFSA into an equivalent regular expression. Package `regexping.stateremoval.ordered.ordering.userinteractive` is a part of jInfer framework, all other packages are part of this work.

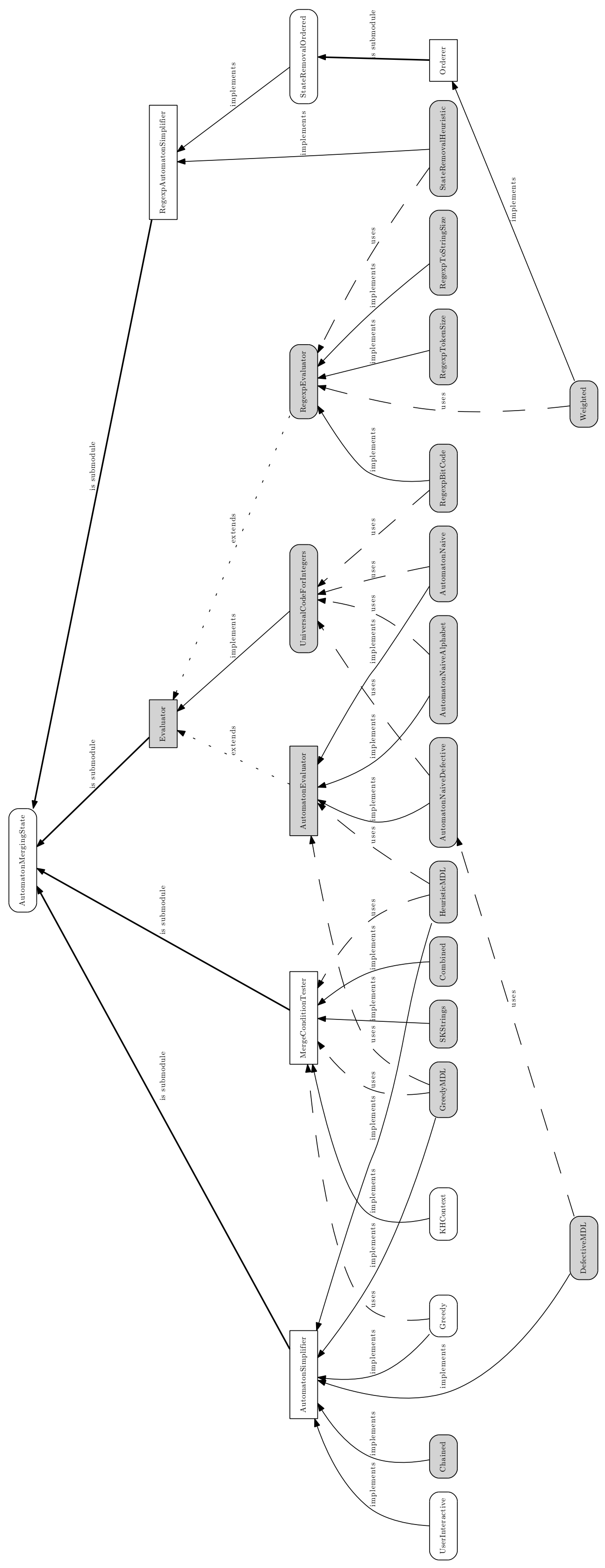


Figure 13: Overview of all modules in context of jInfer framework, filled boxes are part of this work

7 Experimental Results

To test the proposed algorithms, we have generated random testfiles using an XML data generator called ToXGene [6] and its supplemented templates for XMark benchmark [4]. The XMark benchmark is a simulation of an auction website, a typical e-commerce application. Three files were generated, with different length: auction_big.xml (∼1MB), auction_small.xml (∼100kB) and auction_tiny.xml (∼30kB). We used the available auction.dtd [1] (it is listed in Appendix C) for the benchmark as an input schema S_{old} in tests. Sample fragment of the data generated can be found in Appendix C.

In jInfer, we have tested combinations of Greedy, GreedyMDL, HeuristicMDL with alternatives combined from the merge condition testers: $(2, 1)$-context and sk-OR heuristic ($s = 50\%, k = 2$). Greedy strategy coupled with $(2, 1)$-context condition tester served as a simulation of [5]. Each of these were tested with and without the schema on input. We have tested the SchemaMiner experimental implementation [38] without schema input (as it infers only from XML documents). The publicly available application Trang [16], which is a converter between various schema formats (Relax-NG, XSD, DTD), is also able to infer a schema from the given set of XML documents. Precisely, it infers the subclass of SOREs, the *chain regular expressions* [10], which are defined in [11].

The SchemaMiner was not able to finish computation within a reasonable time, thus only results for the tiny dataset are available.

7.1 Overall Quality of Inferred Solution

First interesting comparison involves regular expressions generated using various methods. To shorter the expressions printed here, we substitute element names using this substitution Table 2. Let us explore the inferred regular expressions for

element	*shortcut*	*element*	*shortcut*	*element*	*shortcut*
initial	i	reserve	r	bidder	b
current	c	privacy	p	itemref	f
seller	s	annotation	a	quantity	q
type	t	interval	l		
#PCDATA	c	bold	b	keyword	k
emph	e				
interest	i	business	b	education	e
age	a				

Table 2: Substitution table of element names into shortcut names

element `open_auction`. The DTD specifies regular expression $ir?b*cp?fsaqtl$ for

Method	*Regular expression*
$2, 1$-context	$i((c\|(rc))\|((b\|(rb))b*c))(f\|(pf))saqtl$
Greedy	$i(r\|b)*cp*fsaqtl$
GreedyMDL	$i(c\|((b\|r)b*c))(f\|(pf))saqtl$
HeuristicMDL	$i(c\|((b\|r)b*c))(f\|(pf))saqtl$
Trang	$ir?b*cp?fsaqtl$

Table 3: Element `open_auction`, dataset auction_big.xml

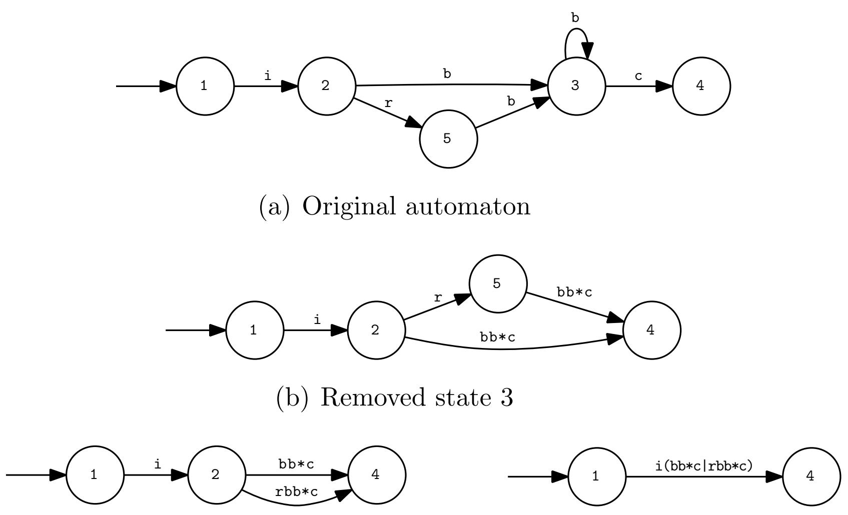
Method	*Regular expression*
$2, 1$-context	$i((c\|(rc))\|((b\|(rb))b*c))(f\|(pf))saqtl$
Greedy	$i(r\|b)*cp*fsaqtl$
GreedyMDL	$i(b\|r)*cp*fsaqtl$
HeuristicMDL	$i(b\|r)*cp*fsaqtl$
Trang	$ir?b*cp?fsaqtl$

Table 4: Element `open_auction`, dataset auction_small.xml

this element. The resulting regular expressions for each inference algorithm are depicted in Tables 3, 4, 5 for big, small and tiny input dataset respectively. As we can see the (k, h)-context method did not performed very well. GreedyMDL and HeuristicMDL preferred more complex regular expression for the big dataset, since it better fitted the data (lower MDL value), than Greedy, which simply merged everything it could. Trang is able to learn chain regular expression and since the DTD expression is a chain regular expression, being given any big enough dataset, Trang is always able to learn exactly the DTD expression (as this expression was used to generate the data). Thus if the user is expecting only very sipmle REs on the output of the algorithm, Trang is able to satisfy this user needs. SchemaMiner probably fell into a common problem of converting an automaton into corresponding regular expression. When bad state removal order is used, the regular expression constructed can be such as the one on SchemaMiner output. In Fig. 14 we have depicted the bad ordering causing exactly the error, which is a probable cause of SchemaMiner output.

Method	*Regular expression*
$2, 1$-context	$i(b\|(rb))b*cpfsaqtl$
Greedy	$i(r\|b)*cpfsaqtl$
GreedyMDL	$i(b\|r)*cpfsaqtl$
HeuristicMDL	$i(b\|r)*cpfsaqtl$
Trang	$ir?b+cpfsaqtl$
SchemaMiner	$i(bb*cpfsaqtl)\|(rbb*cpfsaqtl)$

Table 5: Element `open_auction`, dataset auction_tiny.xml



(a) Original automaton (b) Removed state 3 (c) Removed state 5 (d) Removed state 2

Figure 14: Bad state removal ordering ending in longer RE

method	*regular expression*
2, 1-context	too long regular expression (917 characters)
Greedy	$(e\|b\|c\|k)*$
GreedyMDL	$(((c\|k)*\|(e,e*,(c\|k)))*$ $\|((b\|(e,e*,b)),(b*\|(e,e*,b))*$ $,((c\|k)\|(e,e*,(c\|k)))))*,$ $((e,e*)?\|((b\|(e,e*,b)),$ $(b*\|(e,e*,b))*,(e,e*)?))$
HeuristicMDL	$(c,c*)\|(((b\|e\|k)\|(c,c*,(b\|k\|e))),$ $((k\|e\|b)*\|(c,c*,(b\|k\|e)))*,(c,c*)?)$
Trang	$(c\|b\|k\|e)*$

Table 6: Element `text`, dataset auction_big.xml

7.2 Influence of Input Size

Let us explore the inferred regular expressions for element `text`. The DTD specifies regular expression $(c|b|k|e)*$ for this element. The results are in Tables 6, 7, 8. GreedyMDL was able to infer the same RE as defined in the original DTD (i.e. $(c|b|k|e)*$) using the tiny dataset, but the RE inferred using the small one is more complex than RE inferred using the big one, which is peculiar. According to MDL principle and theory behind it, MDL should prefer smaller hypotheses for smaller data input. HeuristicMDL was not any better, however. It was unable to infer the RE $(b|c|k|e)*$ even using the tiny dataset. We investigated this behavior by capturing the PFSA inferred by GreedyMDL, before it was converted to RE. The respective PFSAs are depicted in Fig. 15. One explanation of the phenomena may be, that being given less data, more regularity can be exploited.

method	*regular expression*
2, 1-context	too long regular expression (716 characters)
Greedy	$(c\|b\|k\|e)*$
GreedyMDL	too long regular expression (462 characters)
HeuristicMDL	$((k\|b\|e)*\|(c,(k\|e\|b)))*, c?$
Trang	$(c\|b\|k\|e)*$

Table 7: Element `text`, dataset auction_small.xml

method	*regular expression*
2, 1-context	$(k, k*)\|(e\|(k, k*, e, e))\|$ $((((c\|(b, c))\|(k, k*, c))\|((e\|(k, k*, e, e)),$ $(c\|(b, c)))), (((b, c)*\|(k, k*, c))*\|((e\|(k, k*, e, e)),$ $(c\|(b, c))))*, ((k, k*)?\|(e\|(k, k*, e, e))))$
Greedy	$(b\|c\|k\|e)*$
GreedyMDL	$(e\|b\|c\|k)*$
HeuristicMDL	$((b\|k\|e)*\|(c,(e\|b\|k)))*, c?$
Trang	$(c\|b\|k\|e)*$
SchemaMiner1	$(eb?)\|(bk*)\|(kk*(ee*\|b))$
SchemaMiner2	$(be\|kk*\|e)$
SchemaMiner3	$ee*bb*$

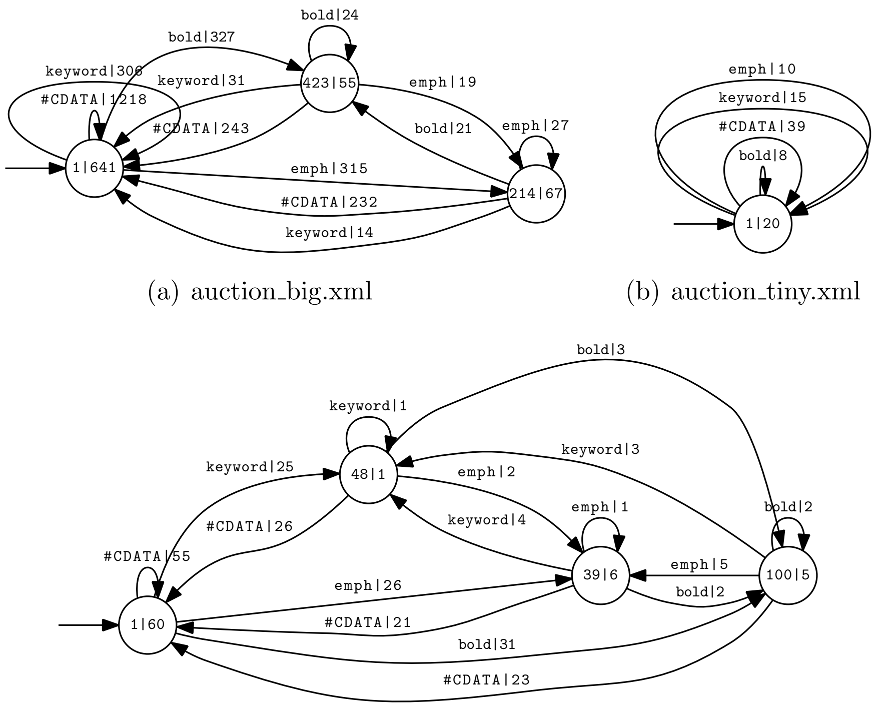
Table 8: Element `text`, dataset auction_tiny.xml

Let us consider fair coin tosses: 10 trials vs. 10000 trials. In the former case, it is more probable that one can exploit regularity, which originally was not intended to be there. Probably the automaton depicted in Fig. 15(c) fits the data better than the more general automaton in Fig. 15(b). The RE may be complex due to the automaton-to-RE conversion procedures, too.

The SchemaMiner output in this test is divided into 3 REs, since SchemaMiner uses advanced element clustering techniques. It has recognized 3 versions of the `text` element in the input data:

- the one used inside `listitem` element (the first RE),
- the one used inside `mail`, `item/description` and `category/description` element contexts (the second RE),
- the one used inside `annotation/description` context (the third RE).

Here, the strong side of SchemaMiner has taken the advantage, the advanced clustering enables to define element `text` using different content models in different contexts. That is exactly the feature XSD provides us, and it had also lead to much simpler REs in each case.



(a) auction_big.xml (b) auction_tiny.xml

(c) auction_small.xml

Figure 15: The PFSA inferred by GreedyMDL for `text` element using various input datasets

7.3 Influence of Schema Input

The regular expressions from the input schema were more general than the ones inferred. In this case, it lead to enforcing the output REs to be in the form of the schema ones, rather than making them more specific. In particular, for element `text`, the RE inferred using the input schema was nearly always $(e|b|c|k)*$, that is the same as in the input schema. The cause is, that DFSA torso is built *before* parsing input strings, thus enforcing the DFSA the form of regular expression from the input schema. Since DFSA torso is built exactly in form depicted in Fig. 15(b), there are no input strings which would create any new branches.

Let us inspect the particular `profile` element. The original DTD specifies RE $i*, e?, g?, b, a?$ for it.

The RE for element `profile`, using GreedyMDL+schema input:

$$i*, ((b|(e,b))|((g|(e,g)),b)), a?$$

Now we see, how useful it would be to have a heuristic rewriting rule (let us denote it the *optionality rule*), that rewrites REs of the form $(a|ba)$ to $b?a$. For now, we rewrite this RE by hand:

$$i*, ((e?, b)|(e?, g, b)), a?$$

One more heuristic rule indicated by this example is to rewrite RE $(abc)|(ade)$ to $a(bc)|(de)$ (let us denote it the *factor rule*). For now, we rewrite this RE by hand:

$$i*, e?, (b)|(g, b), a?$$

Using an optionality rule again, we can rewrite it even further:

$$i*, e?, g?, b, a?$$

Developing these rewriting rules is left for the future work. For now, we can see how useful they would be.

Let us consider the RE for element `profile`, using GreedyMDL (without schema input):

$$(e|i)*, (b|(g, b)), a?$$

Let us rewrite it by hand using an optionality rule:

$$(e|i)*, g?, b, a?$$

Even though the schema input improved the inferred RE in term of not enabling unlimited occurrence of `education` element, other parts of the overall algorithm perform so bad, that a heuristic rule rewriting after the inference is needed to obtain a good-quality result.

7.4 Performance

We have tested the implementation using 2GHz Turion CPU coupled with 4GB RAM in NetBeans 7.0 under Ubuntu 11.04 OS. The performance of all but HeuristicMDL strategy was satisfying, inferring a schema in less than minute in the worst case (1MB dataset GreedyMDL) and instantly in most other cases. HeuristicMDL had the running time of approx. 60 minutes (1MB dataset), which is undesirable. The user can not afford to wait such a long time. Compare it to the SchemaMiner, which for tiny dataset (∼30kB) ran instantly (under 10 seconds), for the small dataset (∼100kB) we interrupted the inference after 12 hours, when it was only inferring the content model for element `open_auction`, and had even more complex element `text` in front. According to debug output, the computation did not halt nor cycle, but little progress was made.

Compare it to the Trang, which worked flawlessly for any size of input and as a bonus inferred exactly the REs from original DTD (or sometimes even more accurate). But Trang infers only the subclass of regular expression (CHAREs), that enables it to infer very efficiently.

8 Conclusion

The aim of this thesis was to propose optimization and refinements of classical algorithms for automatic inference of an XML schema. We have proposed three main refinements:

- exploiting an additional information for the purpose of schema inference - the original, possibly incorrect or too general schema,
- designing a finer MDL measure for evaluation of solutions in heuristic methods,
- possibility of generating a simpler schema and guides for user to repair his input documents to be valid against it.

By incorporating the schema input, the resulting schema inferred is enforced not to deviate much from original schema. It is highly practical for users, who have the old schema available and want to infer the schema, accurate for the data. Experimental results confirm that this goal was accomplished, but there is still a space for improvements.

The designed MDL measure is more appropriate than measures used in common, as it employs a lot of probability codes, which are codelength optimal. It provides the solution with the superiority feature to prefer simpler DFSA with smaller input dataset. This behavior is somewhat in coincidence with DFSA to RE conversion routines, which are unsatisfying at the moment, as they increase the regular expression complexity rapidly with increasing number of DFSA states.

The possibility of generating a simpler schema by invalidating input element instances was implemented and tested, but did not find its use case. Only a minimum of element instances were invalidated, if any. Maybe the MDL code used to decide input excentricity, although consistent in theory, is not the right decision rule for such a problem. We describe possible improvements to deal with this problem in future work.

Nevertheless,the biggest pros of the solution remains: its incorporation into freely and publicly available jInfer framework, developed earlier as a software project. This gives the algorithms user-friendly interface (not a command-line) and great extension possibilities. Anyone can extend the proposed solution and test new features.

The application performance is satisfying and it can be used for instant schema inference without any prior theoretic knowledge of the user. In default configuration, the best of algorithms available are run to produce the best available output.

8.1 Future Work

By experimental testing, we have identified possible places to improve our solution. Let us name the top priority ones:

- We would like to implement a set of heuristic rules for output regular expression refinement based on practical observations from Section 7. Since the automaton merging state algorithm has some common pitfall constructs it produces, these should be refined after the main inference is run.
- As the state removal algorithm [24] behaves poorly, even when coupled with best heuristic available now [17, 21], we should rethink the automaton-to RE conversion step. This would need testing of recent conversion approaches and selecting the best one for our purposes, but this is a topic for one whole thesis.
- A simple, but maybe powerful change is to add a possibility to build the automaton torso from input schema *after* the PPTA is constructed. This would enable better schema specialization at low cost of changes made.
- We should also design a new DefectiveMDLuser module which would be more driven by user opinion on how many element instances (s)he is willing to repair by hand to gain a profit of better schema. Of course, the possibility of automatic repairs would be also a tremendous improvement.

Appendix A Content of CD

Attachment 1 (Binary Distribution)**.** A binary distribution of this thesis implementation consists of NetBeans module files (.nbm), located on CD included in this thesis printed version in folder `bin/`. The distribution can be also built from source files (see Attachment 3). The modules have to be installed into working NetBeans installation as plugins. See jInfer Tutorial [27] for step-by-step instructions.

Attachment 2 (Testing Projects)**.** Testing projects are located in folder `test_projects/`. These are the projects we created when testing thesis implementation (the results can be found in Section 7). Project files have to be opened in a working jInfer installation. Each project has input XML files and DTD schema already set up, the experimental results are in the project output folder. Nevertheless, one can test the inference process with various options set up in project properties (see Appendix B).

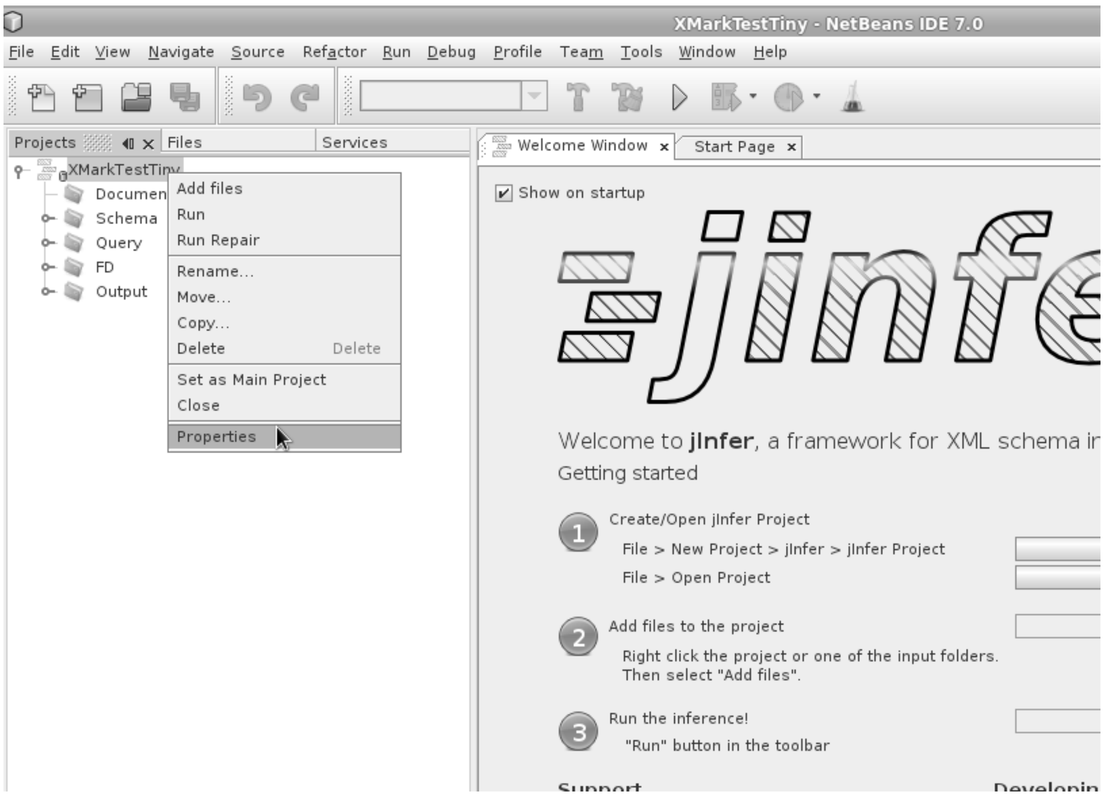
Attachment 3 (Source code)**.** The source code of this thesis is located in folder `src/`. The source code is a checkout of public jInfer svn repository `trunk/` folder[1] to the date 2.8.2011. The same source code can be obtained by issuing command

```
svn co -r 1758 \
  https://jinfer.svn.sourceforge.net/svnroot/jinfer/jinfer/trunk/
```

[1]`https://jinfer.svn.sourceforge.net/svnroot/jinfer/jinfer/trunk/`

Appendix B User Guide

In this section, we briefly describe the steps needed to use the thesis implementation. The implementation is Attachment 1 and can also be built from sources Attachment 3. After opening a sample project from the Attachment 2 (see Appendix A) or creating a new project, one may be interested in changing the modules in inference chain. To access project properties, right click on project and Properties, as depicted in Fig. 16. Then, the Merging State Strategy module can be selected by selecting "Automaton Merging State" branch of the left pane and then using the combo boxes on the right pane as depicted in Fig. 17. For a particular Merging State Strategy, the Merge Condition Tester can be set as depicted in Fig. 18. To run the inference, right click on project and select Run, or use the green arrow to run the default project as depicted in Fig. 19. The result of the inference is opened in editor, as depicted in Fig. 20.



Welcome to **jInfer**, a framework for XML schema ir
Getting started

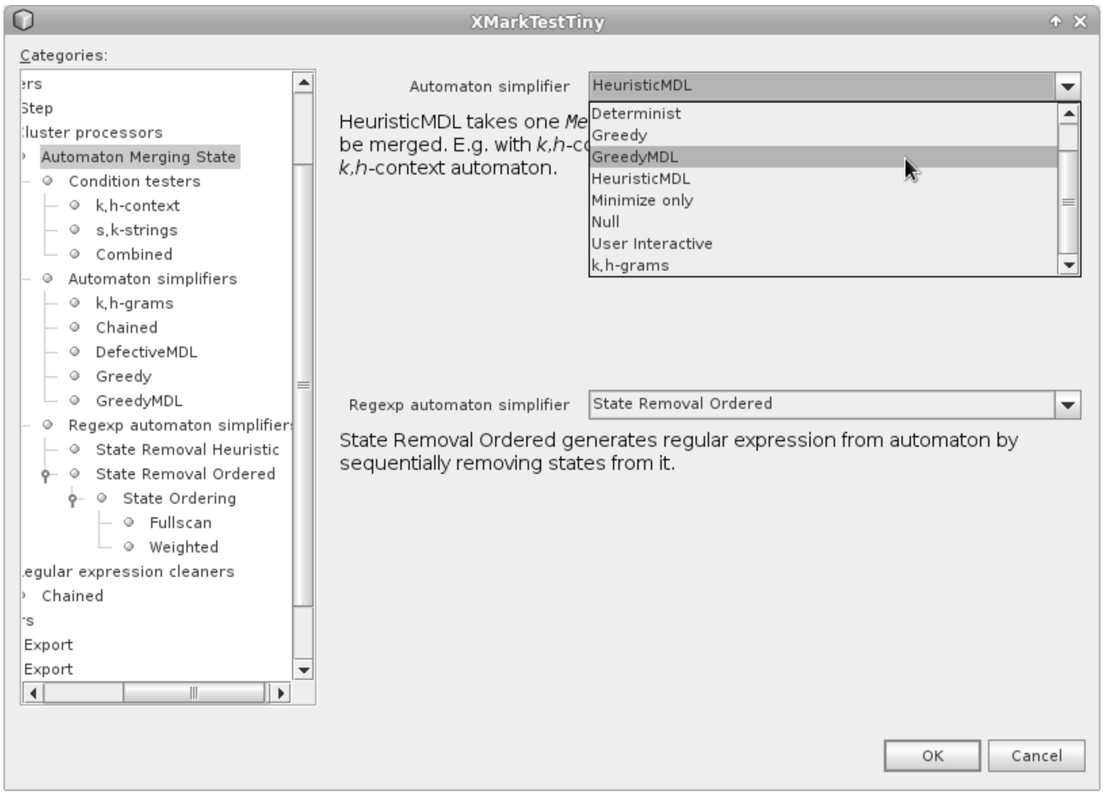
1 Create/Open jInfer Project
File > New Project > jInfer > jInfer Project
File > Open Project

2 Add files to the project
Right click the project or one of the input folders.
Then select "Add files".

3 Run the inference!
"Run" button in the toolbar

Support Developin

Figure 16: How to open project Properties



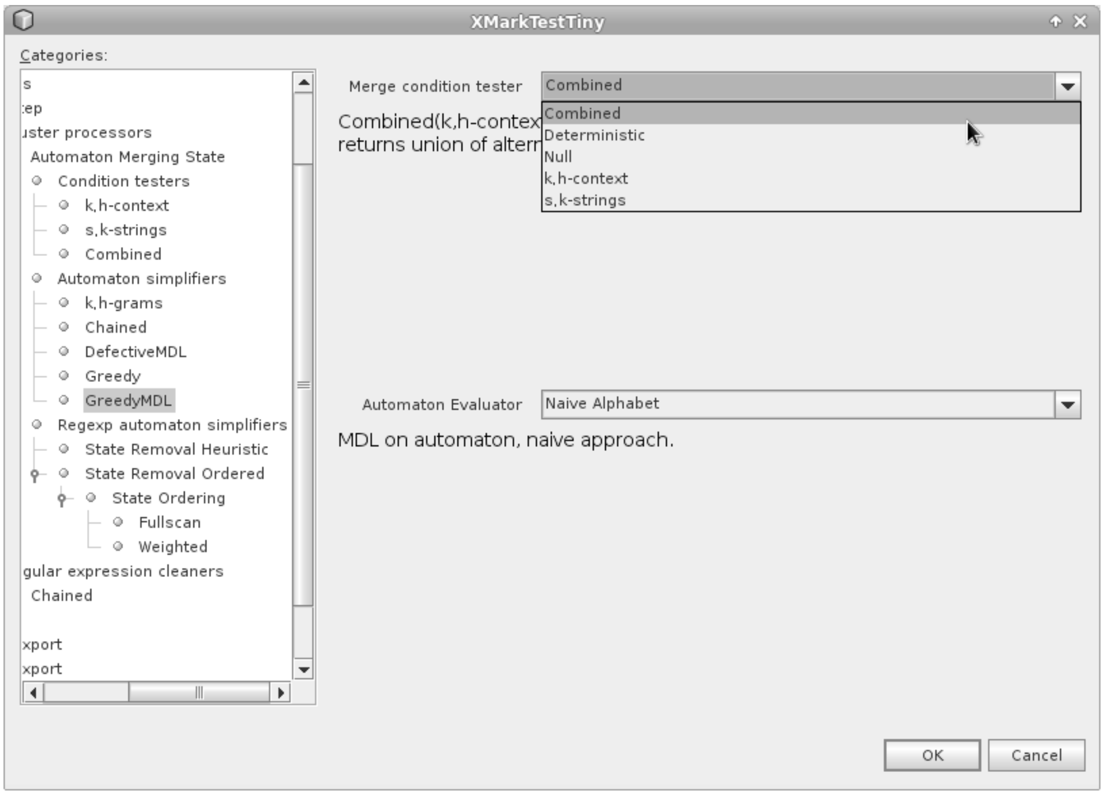
Automaton simplifier: HeuristicMDL

HeuristicMDL takes one *Me*
be merged. E.g. with *k,h*-c
k,h-context automaton.

Regexp automaton simplifier: State Removal Ordered

State Removal Ordered generates regular expression from automaton by sequentially removing states from it.

Figure 17: Selecting the Merging State Strategy



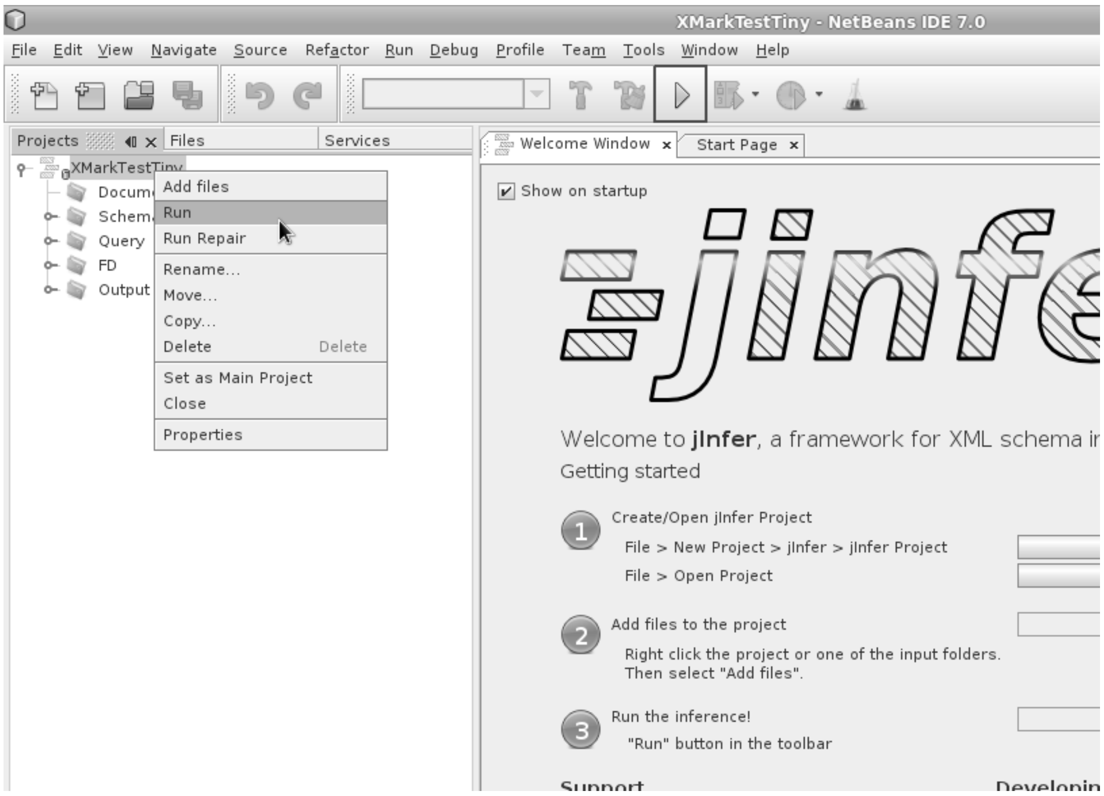
Merge condition tester: Combined

Combined(k,h-contex
returns union of altern

Automaton Evaluator: Naive Alphabet

MDL on automaton, naive approach.

Figure 18: Setting Merge Criterion Tester for GreedyMDL



Welcome to **jInfer**, a framework for XML schema i
Getting started

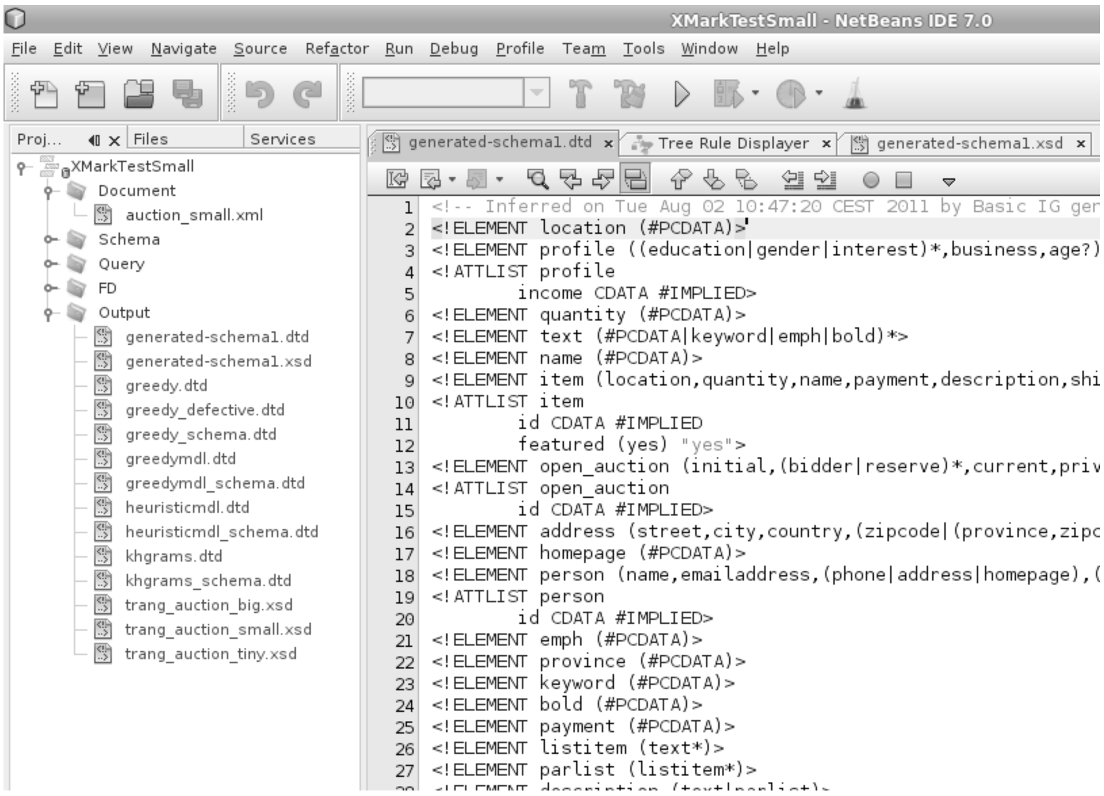
1 Create/Open jInfer Project
File > New Project > jInfer > jInfer Project
File > Open Project

2 Add files to the project
Right click the project or one of the input folders.
Then select "Add files".

3 Run the inference!
"Run" button in the toolbar

Support Developin

Figure 19: Running the inference



```
<!-- Inferred on Tue Aug 02 10:47:20 CEST 2011 by Basic IG ger
<!ELEMENT location (#PCDATA)>
<!ELEMENT profile ((education|gender|interest)*,business,age?)
<!ATTLIST profile
        income CDATA #IMPLIED>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT text (#PCDATA|keyword|emph|bold)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT item (location,quantity,name,payment,description,shi
<!ATTLIST item
        id CDATA #IMPLIED
        featured (yes) "yes">
<!ELEMENT open_auction (initial,(bidder|reserve)*,current,priv
<!ATTLIST open_auction
        id CDATA #IMPLIED>
<!ELEMENT address (street,city,country,(zipcode|(province,zipc
<!ELEMENT homepage (#PCDATA)>
<!ELEMENT person (name,emailaddress,(phone|address|homepage),(
<!ATTLIST person
        id CDATA #IMPLIED>
<!ELEMENT emph (#PCDATA)>
<!ELEMENT province (#PCDATA)>
<!ELEMENT keyword (#PCDATA)>
<!ELEMENT bold (#PCDATA)>
<!ELEMENT payment (#PCDATA)>
<!ELEMENT listitem (text*)>
<!ELEMENT parlist (listitem*)>
```

Figure 20: Inference result

Appendix C Experimental Data

The DTD for the XMark benchmark [1] is depicted in Fig. 22. An example of XML fragment of data generated is depicted in Fig. 21

```
<open_auction id="open_auction1">
  <initial>20.90</initial>
  <reserve>33.44</reserve>
  <bidder>
    <date>10/23/00</date>
    <time>01:41:33</time>
    <personref person="person172"></personref>
    <increase>12.90</increase>
  </bidder>
  <current>33.80</current>
  <itemref item="item206"></itemref>
  <seller person="person195"></seller>
  <annotation>
    <author person="person183"></author>
    <description>
      <text>nicel ver a orswor uiltiness
        saf aurenc alic efug ecur ompeius
        decei as urpose hereo njus
        <keyword>brav ui etter orse aid
          bristl un ortune aptain
        </keyword>
        odd hought eep oa houlde bhors
        win an niqu aintl uttin eonato
      </text>
    </description>
    <happiness>3</happiness>
  </annotation>
  <quantity>1</quantity>
  <type>Featured</type>
  <interval>
    <start>09/06/01</start>
    <end>04/19/00</end>
  </interval>
</open_auction>
```

Figure 21: An example of data generated, one instance of `open_auction` element

```
<!ELEMENT site          (regions, categories, catgraph, people, open_auctions, closed_auctions)>
<!ELEMENT regions       (africa, asia, australia, europe, namerica, samerica)>
<!ELEMENT item          (location, quantity, name, payment, description,\
                        shipping, incategory+, mailbox)>
<!ATTLIST item          id ID #REQUIRED
                        featured CDATA #IMPLIED>
<!ELEMENT person        (name, emailaddress, phone?, address?, homepage?,\
                        creditcard?, profile?, watches?)>
<!ATTLIST person        id ID #REQUIRED>
<!ELEMENT open_auctions (open_auction*)>
<!ELEMENT open_auction  (initial, reserve?, bidder*, current, privacy?,\
<!ATTLIST open_auction  id ID #REQUIRED>
                        itemref, seller, annotation, quantity, type, interval)>
<!ELEMENT profile       (interest*, education?, gender?, business, age?)>
<!ATTLIST profile       income CDATA #IMPLIED>
<!ELEMENT address       (street, city, country, province?, zipcode)>
<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction  (seller, buyer, itemref, price, date, quantity, type, annotation?)>
<!ELEMENT text          (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold          (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword       (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph          (#PCDATA | bold | keyword | emph)*>
<!ELEMENT categories    (category+)>              <!ELEMENT province      (#PCDATA)>
<!ELEMENT category      (name, description)>      <!ELEMENT zipcode       (#PCDATA)>
<!ATTLIST category      id ID #REQUIRED>          <!ELEMENT country       (#PCDATA)>
<!ELEMENT name          (#PCDATA)>                <!ELEMENT homepage      (#PCDATA)>
<!ELEMENT description   (text | parlist)>         <!ELEMENT creditcard    (#PCDATA)>
<!ELEMENT parlist       (listitem)*>              <!ELEMENT interest      EMPTY>
<!ELEMENT listitem      (text | parlist)*>        <!ATTLIST interest      category IDREF #REQUIRED>
<!ELEMENT catgraph      (edge*)>                  <!ELEMENT education     (#PCDATA)>
<!ELEMENT africa        (item*)>                  <!ELEMENT income        (#PCDATA)>
<!ELEMENT asia          (item*)>                  <!ELEMENT gender        (#PCDATA)>
<!ELEMENT australia     (item*)>                  <!ELEMENT business      (#PCDATA)>
<!ELEMENT namerica      (item*)>                  <!ELEMENT age           (#PCDATA)>
<!ELEMENT samerica      (item*)>                  <!ELEMENT watches       (watch*)>
<!ELEMENT europe        (item*)>                  <!ELEMENT watch         EMPTY>
<!ELEMENT location      (#PCDATA)>                <!ATTLIST watch         open_auction IDREF #REQUIRED>
<!ELEMENT quantity      (#PCDATA)>                <!ELEMENT privacy       (#PCDATA)>
<!ELEMENT payment       (#PCDATA)>                <!ELEMENT initial       (#PCDATA)>
<!ELEMENT shipping      (#PCDATA)>                <!ELEMENT bidder        (date, time, personref,\
<!ELEMENT reserve       (#PCDATA)>                                        increase)>
<!ELEMENT edge          EMPTY>                    <!ELEMENT seller        EMPTY>
<!ATTLIST edge          from IDREF #REQUIRED\     <!ATTLIST seller        person IDREF #REQUIRED>
                        to IDREF #REQUIRED>       <!ELEMENT current       (#PCDATA)>
<!ELEMENT incategory    EMPTY>                    <!ELEMENT increase      (#PCDATA)>
<!ATTLIST incategory    category IDREF #REQUIRED> <!ELEMENT type          (#PCDATA)>
<!ELEMENT mailbox       (mail*)>                  <!ELEMENT interval      (start, end)>
<!ELEMENT mail          (from, to, date, text)>   <!ELEMENT start         (#PCDATA)>
<!ELEMENT from          (#PCDATA)>                <!ELEMENT end           (#PCDATA)>
<!ELEMENT to            (#PCDATA)>                <!ELEMENT time          (#PCDATA)>
<!ELEMENT date          (#PCDATA)>                <!ELEMENT status        (#PCDATA)>
<!ELEMENT itemref       EMPTY>                    <!ELEMENT amount        (#PCDATA)>
<!ATTLIST itemref       item IDREF #REQUIRED>     <!ELEMENT buyer         EMPTY>
<!ELEMENT personref     EMPTY>                    <!ATTLIST buyer         person IDREF #REQUIRED>
<!ATTLIST personref     person IDREF #REQUIRED>   <!ELEMENT price         (#PCDATA)>
<!ELEMENT people        (person*)>                <!ELEMENT annotation    (author, description?,\
<!ELEMENT emailaddress  (#PCDATA)>                                        happiness)>
<!ELEMENT phone         (#PCDATA)>                <!ELEMENT author        EMPTY>
<!ELEMENT street        (#PCDATA)>                <!ATTLIST author        person IDREF #REQUIRED>
<!ELEMENT city          (#PCDATA)>                <!ELEMENT happiness     (#PCDATA)>
```

Figure 22: DTD for the XMark benchmark: auction.dtd

Bibliography

[1] http://www.ins.cwi.nl/projects/xmark/Assets/auction.dtd.

[2] Java platform, standard edition. http://www.oracle.com/technetwork/java/javase/overview/index.html.

[3] The netbeans platform. http://netbeans.org/features/platform/index.html.

[4] Xmark — an xml benchmark project. http://www.xml-benchmark.org/.

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

[6] Denilson Barbosa, Alberto O. Mendelzon, John Keenleyside, and Kelly A. Lyons. Toxgene: An extensible template-based data generator for xml. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, pages 49–54, 2002.

[7] Denilson Barbosa, Laurent Mignet, and Pierangelo Veltri. Studying the xml web: Gathering statistics from an xml sample. *World Wide Web*, 8:413–438, December 2005.

[8] Jean Berstel and Luc Boasson. Xml grammars. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, MFCS '00, pages 182–191, London, UK, 2000. Springer-Verlag.

[9] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from xml data. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 825–834, New York, NY, USA, 2008. ACM.

[10] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from xml data. *ACM Trans. Web*, 4:14:1–14:32, September 2010.

[11] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise dtds from xml data. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 115–126. VLDB Endowment, 2006.

[12] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and dtds. *ACM Trans. Database Syst.*, 35:11:1–11:47, May 2010.

[13] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Schemascope: a system for inferring and cleaning xml schemas. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1259–1262, New York, NY, USA, 2008. ACM.

[14] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition, W3C Recommendation. October 2004.

[15] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Information and computation*, 142:182–206, 1997.

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

[17] Manuel Delgado and José Morais. Approximation to the smallest regular expression for a given regular language. In *CIAA*, pages 312–314, 2004.

[18] M. Dorigo, A. Colorni, and V. Maniezzo. Positive feedback as a search strategy. Technical Report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1991.

[19] Michael P. Georgeff and Chris S. Wallace. A general selection criterion for inductive inference. In *ECAI*, pages 219–228, 1984.

[20] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[21] Hermann Gruber, Markus Holzer, and Michael Tautschnig. Short regular expressions from finite automata: Empirical results. In *CIAA 2009. LNCS*, pages 188–197. Springer, 2009.

[22] P.D. Grünwald. *The minimum description length principle.* Mit Press, 2007.

[23] Peter Grunwald. A tutorial introduction to the minimum description length principle, 2004.

[24] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.

[25] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall College Div, 1988.

[26] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jinfer module developer's tutorial. `http://jinfer.sourceforge.net/doc_tutorial_dev.html`.

[27] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jinfer tutorial. `http://jinfer.sourceforge.net/doc_tutorial.html`.

[28] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jinfer xml schema inference framework. `http://jinfer.sourceforge.net/modules/paper.pdf`.

[29] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of xml schema. *ACM Trans. Database Syst.*, 31:770–813, September 2006.

[30] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The xml web: a first study. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 500–510, New York, NY, USA, 2003. ACM.

[31] Irena Mlynkova. On inference of xml schema with the knowledge of an obsolete one. In Athman Bouguettaya and Xuemin Lin, editors, *Twentieth Australasian Database Conference (ADC 2009)*, volume 92 of *CRPIT*, pages 79–86, Wellington, New Zealand, 2009. ACS.

[32] Stephen Muggleton. *Inductive acquisition of expert knowledge.* Turing Institute Pr., 1990.

[33] Andrew Nierman and H. V. Jagadish. Evaluating structural similarity in xml documents. In *WebDB*, pages 61–66, 2002.

[34] Anand Raman, Jon Patrick, and Palmerston North. The sk-strings method for inferring pfsa. In *In Proceedings of the*, 1997.

[35] Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 11(2):416–431, 1983.

[36] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Xml schema part 1: Structures second edition. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004.

[37] Enrique Vidal, Franck Thollard, Colin De La Higuera, Francisco Casacuberta, and Rafael C. Carrasco. Probabilistic finite-state machines – part i. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 2005.

[38] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an xsd. In *DASFAA'08: Proceedings of the 13th international conference on Database systems for advanced applications*, pages 35–50, Berlin, Heidelberg, 2008. Springer-Verlag.

[39] C. S. Wallace and M. P. Georgeff. A general objective for inductive inference. Technical Report 32, Dept. Computer Science, Monash University, http://www.csse.monash.edu.au/˜lloyd/tildeMML/Structured/TR32/, March 1983.

[40] Raymond K. Wong and Jason Sankey. On structural inference for xml data. Technical report, 2003.

[41] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (xml) 1.0. W3c recommendation, XML Core Working Group, World Wide Web Consortium, 2004.

List of Tables

List of Figures