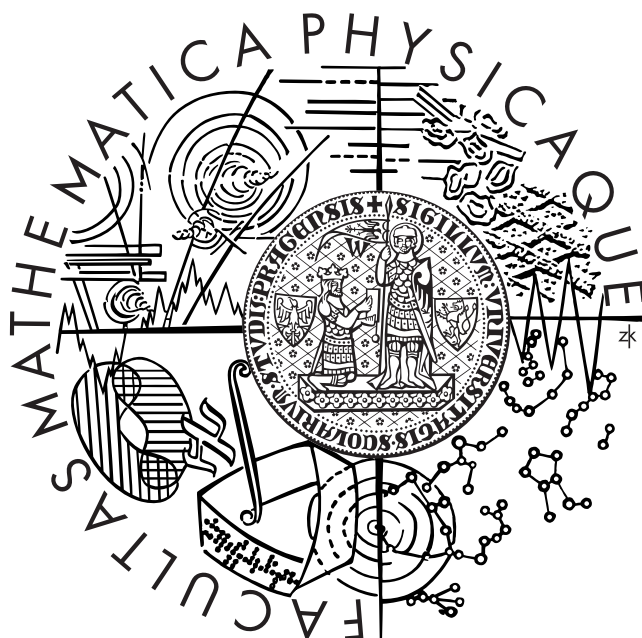


Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Julie Vyhnanovská

Automatic Construction of an XML Schema for a Given Set of XML Documents

Department of Software Engineering
Supervisor: RNDr. Irena Mlýnková, Ph.D.
Study Program: Information Systems

I would like to thank to my supervisor RNDr. Irena Mlýnková, Ph.D., for her helpful suggestions and advices, her corrections of my English and provided sets of testing data. It helped me a lot.

Prohlašuji, že jsem svou diplomovou práci vypracovala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on April 16, 2009

Julie Vyhnanovská

Title: Automatic Construction of an XML Schema for a Given Set of XML Documents

Author: Julie Vyhnánovská

Department: Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková, Ph.D.

Supervisor's e-mail address: irena.mlynkova@mff.cuni.cz

Abstract:

XML language is a popular communication and data exchange format. But there are still a lot of documents that have no XML schema definition. This thesis is focused on an automatic construction of an XML schema for such documents. As an XML schema description language XML Schema is used. This thesis is mainly focused on the advanced XML Schema constructs such as inheritance and on the ways of how user interaction can help in the schema inference process.

Keywords: XML, XML Schema, automatic construction of an XML Schema, user interaction

Název práce: Automatic Construction of an XML Schema for a Given Set of XML Documents

Autor: Julie Vyhnánovská

Katedra (ústav): Katedra Softwarového Inženýrství

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

e-mail vedoucího: irena.mlynkova@mff.cuni.cz

Abstrakt:

Jazyk XML je populární formát pro komunikaci a výměnu dat. Stále ale existuje mnoho dokumentů bez popisu schématu. Tato práce je zaměřena na automatickou konstrukci XML schématu pro takové dokumenty. Pro popis schématu je použit jazyk XML Schema. Tato práce je zaměřena zejména na pokročilejší konstrukce jazyka XML Schema jako například dědičnost a na možnosti, jak může pomoci interakce s uživatelem v procesu odvozování schématu.

Klíčová slova: XML, XML Schema, automatická konstrukce XML schématu, uživatelská interakce

Contents

1	Introduction	6
1.1	Motivation	6
1.2	Aim of the Thesis	7
1.3	Structure of the Thesis	7
2	Basic Definitions	8
2.1	Formal Languages Theory	8
2.1.1	Grammars	8
2.1.2	Regular Expressions	10
2.2	Automata Theory	10
3	XML and XML Schema	12
3.1	XML Syntax	12
3.2	XML Validation	14
3.3	XML Schema Language	15
3.3.1	Type Definitions	16
3.3.2	Particles	17
3.3.3	Inheritance	19
4	Related Work	21
4.1	DTD Based Solutions	21
4.1.1	DTD-Miner	21
4.1.2	XTRACT	24
4.1.3	sk-ANT	30
4.2	XSD Based Solutions	35
4.2.1	<i>i</i> XSD	35
4.2.2	Schema Miner	43
4.3	Conclusion	48
5	Proposed Algorithm	50
5.1	Motivation	50

5.2	Inferring Algorithm	51
5.2.1	User Interaction and Participation Measure	51
5.2.2	Clustering of Elements	52
5.2.3	Type Inheritance	57
5.2.4	Schema Generalisation	59
5.2.5	XML Schema Inference	67
6	Implementation	70
6.1	Architecture	70
6.1.1	Third Party Packages	70
6.1.2	Code Structure	70
6.2	Implemented Fragments	71
6.3	Building and Executing	71
7	Experimental Results	72
7.1	Testing Sets	72
7.1.1	Conclusion	73
7.2	Types Inference	74
8	Conclusion	76
8.1	Future Work	77
A	Content of CD	81
B	SchemaBuilder Guide	82

Chapter 1

Introduction

1.1 Motivation

The internet is a world of communication and data exchange. But if two parts want to communicate with each other, they must use the same language. One of the popular languages used for a communication and a data exchange on the internet is the Extensible Markup Language (XML) [20]. XML is a markup language, that specifies only a syntax of markups. The structure of the XML document must be described explicitly in a so-called *XML schema document*. Thus, if two parts want to communicate using the XML language, they must have the same XML schema document to know the XML document structure.

The two most popular languages, that are both proposed by the W3C [16], are DTD [20] and XML Schema [17, 18, 19]. The former one is a part of the XML specification and it is a basic XML schema description language. But there are situations, where DTD constructs are insufficient. The latter language was created for purposes, where DTD is not strong enough. When the object-oriented approaches were spreaded in the computer science, there were requirements for an object-oriented description of the XML document structure. XML Schema is the product. It provides inheritance between types, substitution groups or user-defined data types.

But specifying a schema for an XML document is not mandatory. Thus there can be XML documents that are not associated with a schema. It has been found, that 52 % of randomly crawled XML documents have no schema defined [10]. This observation yields to the study of automatic inference of XML schema for a given set of XML documents. A number of works is concerned with the automatic schema construction. Most of it is focused on inferring DTD [4, 11, 12, 22]. Some later works are focused on additional

XML Schema constructs [14, 15, 3, 2] especially on inferring different types for elements with the same name but different structure. But there is still space for further improvements.

1.2 Aim of the Thesis

The aim of this thesis is to propose an algorithm of construction of XML Schema for a given set of XML samples. This thesis analyses existing solutions and discuss their advantages and disadvantages. It mainly focuses on ways of how can user interaction help in the XML Schema inference. Especially in the inference of constructs that cannot be expressed by DTD such as inheritance or groups. Part of the thesis is an experimental implementation of the proposed algorithm and it includes experimental results.

1.3 Structure of the Thesis

This chapter presented a motivation and the aim of the thesis. Second Chapter contains basic definitions from automata theory and formal languages theory needed for the next chapters. In Chapter 3 XML and XML Schema languages are described and some definitions are presented. Related work is described and discussed in Chapter 4. In Chapter 5 possibilities of the user interaction are discussed and the proposed algorithm is outlined. Chapter 6 contains some details of the experimental implementation. Experimental results are presented in Chapter 7. Finally, Chapter 8 contains a conclusion and some suggestions for the future work. Part of this thesis is also a brief user guide for the experimental implementation located in Appendix B.

Chapter 2

Basic Definitions

In this chapter basic definitions from the theoretical informatics are placed. The definitions necessary in the following chapters are from the scope of the formal languages theory and the automata theory [6].

2.1 Formal Languages Theory

An **Alphabet** Σ in the context of formal languages can be any finite or infinite set, although usually a set of symbols or characters is used. The elements of an alphabet are called **letters**. A **word** over an alphabet can be any finite sequence of letters. The set of all words over an alphabet Σ is denoted by Σ^* . An empty word is denoted by λ .

Definition 2.1.1 *A formal language L over an alphabet Σ is a subset of Σ^* .*

A formal language can be represented simply by a set of words or can be described using various types of formalisms. For example, a formal language can be a set of words generated by some formal grammar, a set of words that match by a particular regular expression or a set of words accepted by some automaton.

2.1.1 Grammars

Definition 2.1.2 *A formal grammar G is a quaternion (S, N, Σ, P) , where S is the start symbol, $S \in N$, N is a set of nonterminal symbols, Σ is a set of terminal symbols (an alphabet) and P is a set of production rules of the form*

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

Definition 2.1.3 Given a grammar $G = (S, N, \Sigma, P)$, the binary relation \Rightarrow_G on strings in $(\Sigma \cup N)^*$ is defined by:

$$x \Rightarrow_G y \iff \exists u, v, w \in \Sigma^*, X \in N : x = uXv \wedge y = uvw \wedge X \rightarrow w \in P$$

Definition 2.1.4 The language of G denoted by $L(G)$ is defined as a set $w \in \Sigma^* | S \Rightarrow_G^* w$ where \Rightarrow_G^* is the transitive closure of $(\Sigma \cup N)^*$.

In other words, a string in the language $L(G)$ is generated by applying repetitively any production rule from P starting with the start symbol S , until no nonterminal symbols is present in the string. The language consists of all the strings that can be generated in this manner.

Chomsky Hierarchy

The Chomsky hierarchy defines classes of formal grammars according to their production rules complexity.

Level 0 - unrestricted grammars include all formal grammars as defined in Definition 2.1.2. They generate exactly all languages that can be recognised by a *Turing machine*.

Level 1 - context-sensitive grammars include grammars that have only rules of the form $uXv \rightarrow uvw$ with X a nonterminal, u, v, w sequences of terminals and nonterminals and the sequence w nonempty. The rule $S \rightarrow \lambda$ is allowed only if S does not appear on the right side of any other rule. These grammars generate so called *context-sensitive languages*.

Level 2 - context-free grammars generate so called *context-free languages*. These grammars can have only rules of the form $X \rightarrow w$ where X is a nonterminal and w is a sequence of terminals and nonterminals.

Level 3 - regular grammars generate the *regular languages*. These languages are exactly all languages that can be decided by a *finite state automaton* or obtained by *regular expressions*. Regular grammars allow only rules of the form $X \rightarrow xY$ or $X \rightarrow x$ where X, Y are nonterminals and x is a terminal. The rule $S \rightarrow \lambda$ is allowed only if S does not appear on the right side of any other rule.

2.1.2 Regular Expressions

Regular expressions have the same expressive power as regular grammars. It means, that every languages, that can be denoted by regular grammars, can be also described by regular expressions. Regular expressions consist of letters from an alphabet Σ and operators.

Definition 2.1.5 *A regular expression R is a sequence over an alphabet $\Sigma \cup \{?, +, *, |, (,)\}$, where $?$ is an zero-or-one operator, $+$ is one-or-more operator, $*$ is zero-or-more operator, $|$ is choice operator and $(,)$ are grouping parentheses.*

- *zero-or-one operator $?$ makes the symbol or group optional. For example a regular expression $abc?$ accepts a set $\{ab, abc\}$*
- *one-or-more operator $+$ allows repetition of symbol or group. For example a regular expression $abc+$ accepts a set $\{abc, abcc, abccc, \dots\}$*
- *zero-or-more operator $*$ has the similar usage as $+$, but allows optionality. For example a regular expression $abc*$ accepts a set $\{ab, abc, abcc, abccc, \dots\}$*
- *choice operator $|$ accepts one of the symbol or group. For example a regular expression $a|b|c$ accepts a set $\{a, b, c\}$*
- *parentheses group symbols or fragments of regular expression. All operators can be applied on a group. For example a regular expression $(ab)+$ accepts a set $\{ab, abab, ababab, \dots\}$*

Thanks to the parentheses, regular expressions can be nested together. All the following examples are valid regular expressions.

Example: Valid regular expressions

$(a b)*$	$(a b)c$
$(a bc?)*$	$(ab?c) + c$
$((ab*) * ac)+$	$((ab) + (bc) + (abc + c)+)$

2.2 Automata Theory

The Automata theory is closely related to the Formal languages theory. In the Section 2.1, classes of grammars were defined. Each automata is classified by the class of formal languages that it is able to recognise. For the purpose of this thesis, only a *finite state automaton* will be defined, that recognizes class of *regular languages*.

Definition 2.2.1 A deterministic finite state automaton A is a quintuple $A = (Q, q_0, \Sigma, \delta, F)$ where Q is a set of states, q_0 is the initial state, $q_0 \in Q$, Σ is an alphabet of the language the automaton recognizes, δ is a transition function $\delta : Q \times \Sigma \rightarrow Q$ and F is a set of final states, $F \subseteq Q$.

Simply said, an automaton consists of a set of states and a set of transitions. Each transition defines an outgoing state, an incoming state and a letter. The outgoing state is a state, where transition “starts”. The incoming state is a state, where transition “ends”. Similarly a state defines a set of outgoing transitions and a set of incoming transitions. The outgoing transition is a transition, that “leaves” the state. The incoming transition is a transition, that “comes in” to the state.

Definition 2.2.2 A finite state automaton $A = (Q, q_0, \Sigma, \delta, F)$ recognizes or accepts a word $w = w_0, w_1, w_2 \dots w_n$ if $\delta(q_0, w_0) = q_1 \wedge \delta(q_1, w_1) = q_2 \wedge \delta(q_2, w_2) = q_3 \dots \delta(q_n, w_n) = q_{n+1} \wedge q_{n+1} \in F$. A path $p(w)$ is a sequence $q_0, q_1 \dots q_{n+1}$.

Definition 2.2.3 A language of A denoted by $L(A)$ is defined as a set $\{w \in \Sigma^* \mid \widehat{\delta}(q_0, w) \in F\}$ where $\widehat{\delta} : Q \times \Sigma^* \rightarrow Q$.

A function $\widehat{\delta}$ is in other words a concatenation of particular δ operations over the word w .

Definition 2.2.4 Having a set of words W , a prefix tree automaton T for W is a deterministic finite state automaton that accepts exactly each $w \in W$ and for each pair $w_1, w_2 \in W$, paths $p(w_1)$ and $p(w_2)$ have in common exactly the states corresponding to the common prefix of the words.

Chapter 3

XML and XML Schema

XML (eXtensible Markup Language) [20] standardised by the W3C consortium [16] is a standard specifying a structured data description. XML is a simplified subset of SGML (Standard Generalised Markup Language) [21] designed to make the XML parser much easier than an SGML parser. The XML syntax is simpler but it keeps the whole expressive power of SGML. XML is used in various places for various purposes. For example for a data exchange, as a data storage or as a communication protocol (XML-RPC).

3.1 XML Syntax

Each XML document contains one or more elements. An element may refer to other elements to cause the inclusion in the document. Actually, an XML document can be viewed as a kind of a tree.

Definition 3.1.1 *A root element or a document element is an element which is not a part of a content of any other element in the document. There can be only one such element.*

For example in the XML document from Figure 3.1, the root element is a `mobile` element.

Definition 3.1.2 *An XML element or simply an element is a fragment of an XML document which is either delimited by a start-tag and end-tag or by an empty-element-tag. Each element has a type, it is identified by name and may have a content and a set of attributes. Each attribute has a name and a value.*

```

<xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<mobile>
  <manufacturer>Nokia</manufacturer>
  <model>
    <no>6300</no>
    <type>bar</type>
    <weight>91 g</weight>
    <dimensions>106x43x11 mm</dimensions>
  </model>
  <model>
    <no>N81</no>
    <type>slide</type>
    <weight>140 g</weight>
    <dimensions>102x50x17.8 mm</dimensions>
    <description>Symbian S60, WIFI</description>
  </model>
</mobile>

```

Figure 3.1: XML Examples: Nokia.xml

```

<xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<mobile>
  <manufacturer id="12345">"Sony Ericsson"</manufacturer>
  <model>
    <no>Z750</no>
    <type>Clamshell</type>
    <dimensions>97x49x20.1 mm</dimensions>
    <weight>110 g</weight>
  </model>
</mobile>

```

Figure 3.2: XML Examples: SonyEricsson.xml

Element tags start with a < bracket and end with a > bracket. An end-tag follows the open bracket with a / character, an empty-element-tag places a / character before the close bracket.

Each non-empty element starts with the start-tag and ends with the end-tag with the same name as the start-tag. An element content is placed between start-tag and end-tag. For example non-empty element `mobile` has a start-tag `<mobile>`, end tag `</mobile>` and all between these tags is a content.

Each empty element consists only from an empty-element-tag and has no content. For example `<wheel/>` is an empty element.

A start-tag or an empty-element-tag may have a set of attributes. Attributes are placed after element name delimited by a space character. Each attribute has a name and value and has a structure of `name="value"`. For example `<wheel size="17" type="car"/>` is an empty element with attributes `size` and `type`. The `size` attribute has a value 17 and the `type` attribute has a value `car`.

A content of an element may have the following value:

- a character string
- a list of child elements
- mixed content - child elements interlaced with character strings

Definition 3.1.3 *A child element or subelement of the given element e is an element that occurs in the content of e . For the child element, e is a parent element.*

Each document may be a well-formed XML document and, in addition, a valid XML document.

Definition 3.1.4 *A document is an XML document if it is well-formed, as defined in the XML specification. In addition, the XML document is valid if it meets certain further constraints.*

In brief, an XML document is well-formed, if it has one root element and all elements are of the form defined above in this Section.

This section has briefly described an XML syntax. The description covers only constructs necessary for this thesis. The complete XML specification is placed in [20].

3.2 XML Validation

An XML document can conform to some defined schema. An XML schema is a description of a type of XML document. It defines a type of each element and defines a root element. The type consists of the definition of a content and allowed attributes.

Definition 3.2.1 *An XML document is valid if it conforms to its associated XML schema.*

There are some languages that express XML schemas. The XML specification comes with a Document Type Definition (DTD) language. This language is supported by the XML itself and can be defined in an external source or directly in an XML document. For the DTD syntax see the XML specification [20].

DTD defines a type for each element name occurring in an XML document. Such type definition contains a list of allowed attributes, the allowed

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="mobile" type="MobileType"/>

<xsd:complexType name="MobileType">
  <xsd:sequence>
    <xsd:element name="manufacturer" type="ManufacturerType"/>
    <xsd:element name="model" type="ModelType" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ManufacturerType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="id" type="idType"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:simpleType name="idType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="ModelType">
  <xsd:all>
    <xsd:element name="no" type="xsd:positiveInteger"/>
    <xsd:element name="type" type="xsd:string"/>
    <xsd:element name="weight" type="xsd:string"/>
    <xsd:element name="dimensions" type="xsd:string"/>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
  </xsd:all>
</xsd:complexType>

```

Figure 3.3: XML Schema Examples: mobile.xsd

attribute value, and the allowed element content as a deterministic regular expression, where Σ is a set of all element names in the XML document.

The DTD language has relatively poor expressive power. Thus, other more expressive XML schema languages were developed. The most popular languages are RELAX NG [9] and XML Schema [17]. This thesis is mainly focused on the XML Schema language, since it is widely spreaded and most promising XML schema language in the present.

3.3 XML Schema Language

The XML Schema [17] is a language describing the structure of XML documents. XML Schema was approved as a W3C Recommendation in 2001. Like DTD, XML Schema defines types of elements consisting of allowed attributes definition and the element content definition. But it provides much more con-

structs such as advanced basic types definition, inheritance, model groups, permutation operator and so on. The next advantage of XML Schema is that it has an XML syntax. Each XML Schema document (XSD) is also an XML document.

Definition 3.3.1 *An XML Schema document is an XML document valid against the XML Schema specification.*

In Figure 3.3 an example of an XSD is outlined. This document will be used as an example in the whole section. The schema defines a structure of the XML documents from Figure 3.1 and 3.2. Each of the elements in the schema has a prefix `xsd:` which is associated with the XML Schema namespace through the `xmlns` declaration that appears in the `schema` element.

3.3.1 Type Definitions

Each XML element is assigned with a type. An element type defines attributes of the element and the element content. Types are divided into two groups. Elements that contain any subelements or carry attributes are said to have *complex types*, elements with only a text content are said to have *simple types*.

Definition 3.3.2 *An XSD type or simply a type is defined as a pair of a content type and a set of attribute declarations.*

Simple Types

Definition 3.3.3 *A simple type is an XSD type with empty attribute declaration set and content type of a data type as defined in XML Schema specification.*

Simple types can be assigned to an element or an attribute. If the element is of the simple type, it does not have any attributes and have only text content of the specified data type. Attributes can have only simple types.

Simple types defined as a part of the XML Schema language are called built-in simple types. For example built-in simple types are `string` and `positiveInteger`. But simple types can also be user-defined. A User-defined simple type is usually derived from some built-in or previously defined simple type by restriction, list or union. For example the `idType` simple type is derived from built-in simple type `integer` by a restriction. The value scope of the `idType` is 10000 - 99999 which is a subset of the `integer` value scope.

Complex Types

Definition 3.3.4 *A complex type is an XSD type where the content type is represented by a content model*

Complex Types can be assigned only to elements. Complex types allow to define element attributes or an element content that consists of some elements. A complex type is represented by a `complexType` element. The element usually contains content model declaration followed by attribute declarations, both optional. The example shows the complex type `manufacturerType` with simple content and one attribute `id`. The simple content is derived from the `string` simple type by extension. The extension allows to define additional attributes.

A `complexType` element has an attribute `mixed` which is used to define mixed content. When an element content is mixed, character data are enabled to appear between child elements declared as an element content.

A content model is recursively built from particles `{element, sequence, choice, all, group}`. In the root of the content model only particles `{sequence, choice, all}` can occur. For example `mobileType` type uses the `sequence` particle and `modelType` type uses the `all` particle.

3.3.2 Particles

Particles involve occurrence constraints. By default, each particle may occur exactly once. To overwrite this, `minOccurs` and `maxOccurs` attributes are defined for each particle. For example to mark a particle as optional, `minOccurs` must be set to 0 as showed in case of element `description`. To allow particle to be repeated anytimes, `maxOccurs` must be set to `unbounded` as showed in `model` element declaration.

Sequence Particle

A sequence particle contains a possible empty list of particles `{element, sequence, choice, group}`. The particles must occur in an instance document in the exact order. A sequence particle is represented by a `sequence` element. For example the `mobile` element has two subelements, `manufacturer` and `model`. The elements must occur in the defined order, `model` element may occur more times.

Choice Particle

A choice particle contains a possible empty list of particles {`element`, `sequence`, `choice`, `group`}. In an instance document exactly one particle from the list may occur. A choice particle is represented by a `choice` element.

All Particle

An all particle contains a possible empty list of `element` particles. The `maxOccurs` attribute of this particle and all the subelements must be set to 1 and the `minOccurs` attribute must be set to 0 or 1. In an instance document, the particles may occur in any order. For example `model` element may contain its subelements in any order as showed in the instance documents.

Group Particle

A group particle can be viewed as a wrapper that groups particles for further usage. It contains at most one particle of {`sequence`, `choice`, `all`} and can be defined globally. A globally defined group can be then referenced in complex type declarations. This construct makes particle declarations reusable.

Element Particle

An element particle does not contain any particles at all. It represents subelement which may occur in the element content. Each element must have a type. If no type is assigned to an element, `anyType` is used, which means, that the element may have any content. An element type can be referenced by a `type` attribute or can be declared as anonymous type. Such type has no name and is placed in the element content of the `element` element.

Particles are the main building blocks of an XSD. From particles element types are composed. A content model built from particles can be expressed as a regular expression describing the content of the element. An alphabet Σ is a set of all subelement names present in the element content. Parentheses are represented by start-tags and end-tags of particles, operators {`+`, `?`, `*`} are represented by `minOccurs` and `maxOccurs` attributes and an operator `|` is represented by the choice particle. The sequence particle stands for the usual concatenation. Group particles can be replaced by the referenced particles. And only the last particle remains, an all particle, which can be replaced by the choice particles like follows: $(a\&b\&c) \rightarrow (a(bc|cb)|b(ac|ca)|c(ab|ba))$.

From the above it is clear, that an all particle is only a “syntactic sugar” which only reduces the size of the schema, but has no additional expressive

power. Thus, a content model has the same expressive power as regular expressions, and can describe regular languages. For this purpose regular expressions can be extended by one additional operator `&` which stands for the all particle and have the previously outlined meaning.

The XML Schema specification constraints a content model to be *deterministic*, which means, that also the regular expression describing given content model must be deterministic. Here the determinism means, that in every phase of computation, the next step must be decided, not guessed. For example having a finite state automaton, each state must not have two or more outgoing transitions for the same letter to be deterministic.

3.3.3 Inheritance

XML Schema provides a type inheritance. The two provided approaches of how one type can be inferred from another are an extension and a restriction. Both simple types and complex types can be inferred, however only complex types inheritance will be described, since simple types are not important for this thesis.

Definition 3.3.5 *Having a type T_1 inherited from a type T_2 , T_1 is said to be a subtype and T_2 is said to be a supertype.*

The inheritance of complex types have the following syntax. As the root element `complexType` element is used. It contains one subelement, `simpleContent` or `complexContent`. The `simpleContent` element is mainly used, when the subtype is a simple type with some additional attributes, for example the `ManufacturerType`. The `complexContent` element contains one of the subelements `extension` or `restriction` according to the chosen inheritance type.

Restriction

When a subtype is restricted from a supertype, the value space of the subtype must be a subset of the value space of the supertype. Each instance of the subtype must be also a valid instance of the supertype. More formally, when G_1 is a grammar describing the subtype and G_2 is a grammar describing the supertype, $L(G_1) \subseteq L(G_2)$. Technically, XML Schema allows to restrict one complex type from another by restricting the allowable number of repetition of the particles.

The following example shows the subtype `RestrictedMobileType`. The supertype `MobileType` is defined as a `manufacturer` element followed by at least one `model` element. The subtype restrict the occurrence of the `model` element to at most 10.

```
<xsd:complexType name="RestrictedMobileType">
  <xsd:restriction base="MobileType">
    <xsd:sequence>
      <xsd:element name="manufacturer" type="ManufacturerType"/>
      <xsd:element name="model" type="ModelType" maxOccurs="10"/>
    </xsd:sequence>
  </xsd:restriction>
</xsd:complexType>
```

Extension

The subtype is extended from a supertype by appending additional particles. In the other words, the subtype content model is an concatenation of the supertype content model and the content model specified by the additional particles. Note, that XML Schema does not allow other kinds of extensions than appending. This simplifies the extension processing, but it is a relatively important restriction that users must deal with.

The following example outlines the `ExtendedMobileType` extended from the `MobileType`. The value space of the subtype is defined as a `manufacturer` element followed by at least one `model` element optionally followed by a `description` element.

```
<xsd:complexType name="ExtendedMobileType">
  <xsd:extension base="MobileType">
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
  </xsd:extension>
</xsd:complexType>
```

Chapter 4

Related Work

In this chapter representants of the existing solutions will be described. Most of existing solutions have focused on inference of DTDs [4, 11, 12, 22]. DTD based solutions will be described in Section 4.1. Some later works have focused also on additional XML Schema constructs [14, 15, 3, 2]. Such solutions will be described in Section 4.2.

4.1 DTD Based Solutions

4.1.1 DTD-Miner

DTD-Miner [11] system is a prototype for a structural re-engineering framework proposed in [12]. This framework consists of three consecutive phases.

In the first phase XML documents are mapped into an n-ary tree representation called the Document Tree. Document Tree nodes represent XML elements in related document, edges represent parent-child relationships between elements. Each node in the Document Tree is uniquely identified by a Node-ID (NID). The NID is also globally unique across all Document Trees. In addition, each node contains an element name, a list of attributes and a PCDATA flag (whether the corresponding element contains PCDATA). Example of Document Trees for documents in Figure 3.1 and 3.2 are shown in Figure 4.1.

The second phase, called Structure Discovery phase, creates an overall structure from a set of Document Trees. In DTD-Miner system a Spanning Graph is used as the overall structure.

In the final phase of DTD Construction a set of heuristic rules is applied on the Spanning Graph to obtain the final DTD.

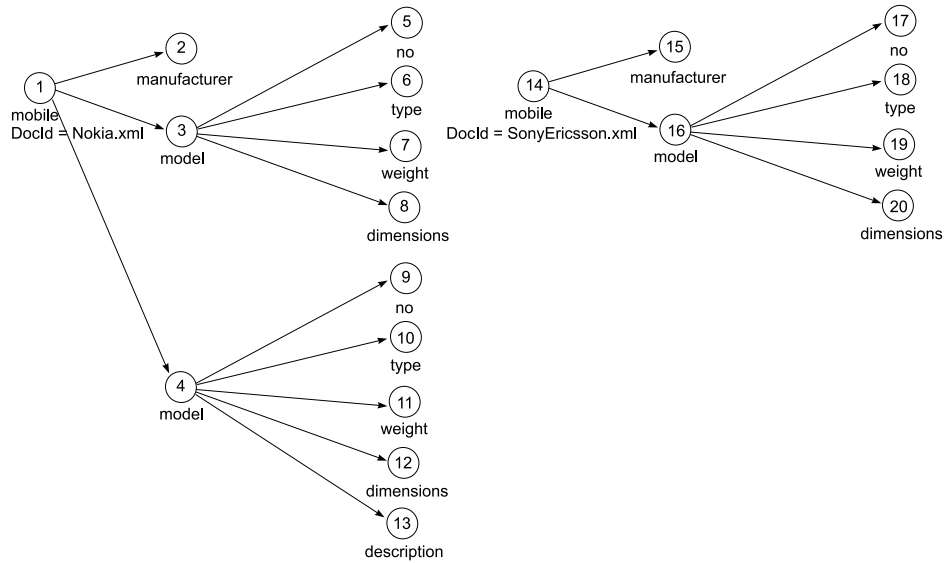


Figure 4.1: Document Trees for Nokia.xml and SonyEricsson.xml

Spanning Graph

The Spanning Graph is an ordered directed acyclic graph. Each node in the graph represents an element and is uniquely identified by a unique GID. Each node also contains a tag name, a list of attributes and a list of NIDs. The list of attributes is a union of all the lists of attributes in the nodes of the Document Trees with the matching element name. The list of NIDs contains all the NIDs of nodes in all the Document Trees with the matching element name. The left-right ordering of sibling nodes denotes the left-right ordering of subelements of a single parent element.

Edges in the Spanning Graph model hierarchical relationships between the elements. The edges are uniquely identified by an EID. Each edge is assigned with an edge list that identifies the parent nodes in the Document Trees where the parent-child relationship exists.

The Spanning Graph construction is an iterative algorithm over a set of Document Trees. At the beginning the Spanning Graph is an empty graph. At each iteration step a Document Tree is merged into the intermediate Spanning Graph. The Final Spanning Graph is obtained when all Document Trees from the given set have been merged into the Spanning Graph.

In a merging step of the algorithm the root node of the Document Tree is merged into the root node of the intermediate Spanning Graph and then all its child nodes are recursively merged with appropriate nodes from the intermediate Spanning Graph. Merging of two nodes is based on determining

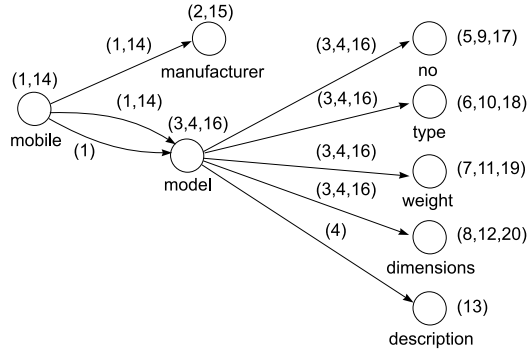


Figure 4.2: Spanning Graph for Document Trees in Figure 4.1

a best order of their child nodes with preserving a primary tree order. For this purpose DTD-Miner uses the longest common subsequence approach. When a Document Tree node is merged with an appropriate Spanning Graph node, the edge list and the node list must be updated. Figure 4.2 shows a Spanning Graph for the Document Trees shown in Figure 4.1.

Heuristic Rules

To obtain a DTD from the Spanning Graph a set of heuristic rules must be applied on the edges of the Spanning Graph. The heuristic rules are as follows:

Define Optionality: The element represented by the node N_c is optional, if (N_p, N_c) edge list $\subset N_p$ node list where N_p is a parent of N_c . For example, element `<description>` in document Nokia.xml in Figure 3.1 is optional for element `<model>`, because in the Spanning Graph in Figure 4.2 $(\text{model}, \text{description})$ edge list = $\{4\} \subset \text{model}$ node list = $\{3, 4, 16\}$. Hence, the edge is marked as *optional* in the Spanning Graph.

Merge Repeat: Element represented by node N_c has zero or more occurrences, if there exists a pair of distinct adjacent sibling edges (N_p, N_c) and (N_p, N_{c+1}) in the Spanning Graph, so that $N_c = N_{c+1}$. In this case the edges are merged into a single edge in the Spanning Graph with the edge list as the union of the edge list of the two edges. If the edge list of the newly created edge equals to the N_p node list, the edge is marked as *oneOrMore*. Otherwise, the edge is marked as *zeroOrMore*. For example, element `<model>` has one or more occurrences within element `<mobile>`, because there are two distinct adjacent sibling edges $(\text{mobile}, \text{model})$ in the Spanning Graph in Figure 4.2 and the union of edge lists of the two edges = $\{1, 14\} = \text{mobile}$ node

list = {1, 14}. Hence, the new edge is marked as *oneOrMore* in the Spanning Graph.

Define Group: Firstly the sequence of child edges is split into groups. A group is a subsequence of adjacent edges, where all edge lists are identical. In the next step the algorithm merges two adjacent groups with common subsequence. Two groups have common subsequence, if the second group starts with the first element and the first group ends with the last element of this subsequence. For example, groups {*a, b, c, d*}, {*b, c, d, e*} have common subsequence {*b, c, d*}. These two groups are merged into a new one {*a, b, c, d, e*} with the corresponding DTD fragment (*a(bcd)e*). Finally, the optionality of the group and each individual element of the group must be specified. The group is marked as *oneOrMore*, if the union of edge lists equals to the node list of parent node. Otherwise, the group is marked as *zeroOrMore*. The optionality of an individual element depends on the optionality of the element or elements from the source groups. An element which has no repetition mark is marked as *required*. If only one of the merged elements has repetition mark or both elements have the same mark, then the merged element has this mark too. If elements have different repetition marks, then the merged element has the *zeroOrMore* mark.

Conclusion

The DTD-Miner method ommits a very useful operator |. XML fragments like $\langle A \rangle \langle B \rangle$ and $\langle B \rangle \langle A \rangle$ lead to the DTD fragment ($A?BA?$) instead of ($AB|BA$). This fragment is too general and does not follow the clear meaning of the source XML fragments. Hence, for more complicated XML documents this method returns too complicated and too general DTD to be further useful.

4.1.2 XTRACT

The XTRACT system described in [4] generates a set of candidate DTDs in a generalization and a factoring module and then chooses the best DTD in an MDL module using a Minimum Description Length (MDL) principle. An input sequence *I* represents one element name from the document collection. *I* contains all subelement sequences of element with the given name from all documents in the collection. Like the DTD-Miner, it uses a set of heuristic rules to generalize and, thereby, simplify the final DTD.

Generalization Module

This module infers a set S_g of generalized DTDs. For each input sequence in the parameter I the GENERALIZE procedure outlined in Algorithm 1 independently infers a number of DTDs and adds them to S_g including the input sequence itself. Therefore, $I \subseteq S_g$.

Algorithm 1 GENERALIZE

Input: A set of subelement sequences I

Output: A set of generalized DTDs S_g

```
for each sequence  $s$  in  $I$  do
  add  $s$  to  $S_g$ 
  for  $r := 2, 3, 4$  do
     $s' := \text{DISCOVERSEQPATTERN}(s, r)$ 
    for  $d := 0.1x|s'|, 0.5x|s'|, |s'|$  do
       $s'' := \text{DISCOVERORPATTERN}(s', d)$ 
      add  $s''$  to  $S_g$ 
    end for
  end for
end for
return  $S_g$ 
```

Algorithm 2 DISCOVERSEQPATTERN

Input: An sequence s and a factor r

Output: The sequence s with replaced symbols

```
repeat
  let  $x$  be a subsequence of  $s$  with the maximum number ( $\geq r$ ) of contiguous repetitions in  $s$ 
  replace all ( $\geq r$ ) contiguous occurrences of  $x$  in  $s$  with a new auxiliary symbol  $A_i = (x)^*$ 
until  $s$  no longer contains  $\geq r$  contiguous occurrences of any subsequence  $x$ 
return  $s$ 
```

The Algorithm 2 DISCOVERSEQPATTERN replaces sequences of the form $xxx\dots x$ in the input sequence s with the regular expression $(x)^*$, more precisely with an auxiliary symbol. There is one-to-one correspondence between the auxiliary symbol and the regular expression, thus, the auxiliary symbol represents this regular expression in every other candidate DTDs. The parameter r determines the minimum number of continuous repetitions of subsequence x in s required for replacing. If there are more replacing candidates, the candidate with most repetitions is chosen.

The Algorithm 3 DISCOVERORPATTERN first uses Algorithm 4 PARTITION to partition the input sequence s . Then it replaces each subsequence

Algorithm 3 DISCOVERORPATTERN

Input: A sequence s and a factor d

Output: The sequence s with replaced symbols

$s_1, s_2, \dots, s_n := \text{PARTITION}(s, d)$

for each subsequence s_j in s_1, s_2, \dots, s_n **do**

let the set of distinct symbols in s_j be a_1, a_2, \dots, a_m

if $m > 1$ **then**

replace subsequence s_j in sequence s with a new auxiliary symbol $A_i = (a_1 | \dots | a_m)^*$

end if

end for

return s

s_i from the PARTITION result with the auxiliary symbol for the regular expression $(a_1 | a_2 | \dots | a_n)$ where symbols a_1, a_2, \dots, a_n are all symbols from s_i . The PARTITION procedure splits the input sequence s into the smallest possible subsequences s_1, s_2, \dots, s_n such that for any occurrence of a symbol a in a subsequence s_i , there does not exist another occurrence of a in some other subsequence s_j within a distance d .

Algorithm 4 PARTITION

Input: A sequence s and a factor d

Output: Partitions s_1, s_2, \dots, s_i of sequence s

$i := \text{start} := \text{end} := 1$

$s_i = s[\text{start}, \text{end}]$

while $\text{end} < |s|$ **do**

while $\text{end} < |s|$ and a symbol in s_i occurs to the right of s_i within a distance d **do**

$\text{end} := \text{end} + 1$

$s_i := s[\text{start}, \text{end}]$

if $\text{end} < |s|$ **then**

$i := i + 1$

$\text{start} := \text{end} + 1$

$\text{end} := \text{end} + 1$

$s_i := s[\text{start}, \text{end}]$

end if

end while

end while

return s_1, s_2, \dots, s_i

Example:
 $I = \{aaaab, abab, acc, ab\}$

 1) $s = aaaab$ each of the r and d parameter leads to the a^*b pattern.

 2) $s = abab$ $r = 2$ leads to the $(ab)^*$ pattern, $r \geq 3$ and $d \geq 2$ leads to the $(a|b)^*$ pattern.

 3) $s = acc$ $r = 2$ leads to the ac^* pattern.

 4) $s = ab$ this sequence is not generalized.

 Finally $S_g = I \cup \{a^*b, (ab)^*, (a|b)^*, ac^*\}$
Factoring Module

The Factoring module factors two or more candidate DTDs in S_g into a new one and inserts it into set S_f . Unlike the generalization, factoring leaves the semantics of candidate DTDs unchanged, but reduces the size of the DTD and, thus, it is a better candidate than the source one. The set of candidate DTDs S_f also contains all the candidate DTDs in S_g .

Example:
 From the set of DTD patterns S_g the Factoring module generates a $a(c^*|b)$ pattern.

 $S_f = S_g \cup \{a(c^*|b)\}$
MDL Module

The MDL module is the most important module in XTRACT system. The MDL module chooses a set of candidate DTDs S from S_f and creates the final DTD D , which is an *or* of candidate DTDs in S . The final DTD D must cover all input sequences in I and must minimize the MDL cost.

The MDL cost is based on a length in bits needed to describe the DTD and a length in bits of DTD encoded sequences. Minimization of the MDL cost leads to the DTD which is simple enough and covers enough details at the same time. The following function computes the number of bits needed to describe the DTD itself:

Definition 4.1.1 *Let Σ be the set of subelement names that appear in sequences in I . Let Ω be a set of metacharacters $|, *, +, ?, (,)$. Let the length of a DTD viewed as a string in $\Sigma \cup \Omega$, be n . Then, the length of the DTD in bits is $n \lceil \log(|\Sigma| + |\Omega|) \rceil$.*

Example: MDL computation

$$\Sigma = \{a, b, c\}; |\Sigma| = 3 \rightarrow \lceil \log(3 + 6) \rceil = 4.$$

Costs of the DTD fragments from S_f :

$aaaab$	$5 \times 4 = 20$	$(ab)^*$	$5 \times 4 = 20$
$abab$	$4 \times 4 = 16$	$(a b)^*$	$6 \times 4 = 24$
acc	$3 \times 4 = 12$	ac^*	$3 \times 4 = 12$
ab	$2 \times 4 = 8$	$a(c^* b)$	$7 \times 4 = 28$
a^*b	$3 \times 4 = 12$		

Functions presented below encode a given sequence according to the given DTD and compute the code length.

A) $seq(D, s) = \varepsilon$ if $D = s$

Note that the DTD D is a sequence of symbols from Σ and does not contain any symbols from Ω .

B) $seq(D_1 \dots D_k, s_1 \dots s_k) = seq(D_1, s_1) \dots seq(D_k, s_k)$

If D can be split into k regular expressions, such that each subsequence s_i matches the corresponding regular expression D_i , then the result of this step is the concatenation of particular $seq(D_i, s_i)$ results.

C) $seq(D_1 | \dots | D_m, s) = iseq(D_i, s)$

If s matches the regular expression D_i , then the result of this step is the concatenation of the index i and the result of $seq(D_i, s)$. Note that the index i must be encoded by $\lceil \log(m) \rceil$ bits.

D) $seq(D^*, s_1 \dots s_k) = \begin{cases} kseq(D, s_1) \dots seq(D, s_k) & \text{if } k > 0 \\ 0 & \text{otherwise} \end{cases}$

If s can be split into k subsequences, each matching the regular expression D , then the result is the concatenation of k and particular $seq(D, s_i)$ results. Note, that since there is no way to determine repetition count during decoding, k must be encoded with its length. In this case k is encoded as size of encoded k ($\lceil \log(k) \rceil$ times 1) followed by 0 followed by binary encoded k .

Example:

Sequences from I encoded by DTD fragments from S_f :

$\{aaaab, abab, acc, ab\}$ DTD fragments from S_f trivially encode themselves with empty string, a^*b :

$$seq(a^*b, aaaab) \stackrel{=B}{=} seq(a^*, aaaa)seq(b, b) \stackrel{=D}{=} 4\epsilon\epsilon\epsilon\epsilon\epsilon = 4 =^{bin} 1110100$$

similarly other sequences:

$$seq(a^*b, ab) = 101$$

$(ab)^*$:

$$seq((ab)^*, abab) = 11010$$

$$seq((ab)^*, ab) = 101$$

$(a|b)^*$:

$$seq((a|b)^*, aaaab) = 111010100001$$

$$seq((a|b)^*, abab) = 11101000101$$

$$seq((a|b)^*, ab) = 1101001$$

ac^* :

$$seq(ac^*, acc) = 11010$$

$a(c^*|b)$:

$$seq(a(c^*|b), acc) = 011010$$

$$seq(a(c^*|b), ab) = 1$$

Finally the set of candidate DTDs S , is found by using expression

$$\min_{S \subset S_f} \left\{ \sum_{s \in S} c(s) + \sum_{i \in I} \min_{s \in S} d(s, i) \right\}$$

where $c(s)$ is code length of the DTD fragment s and $d(s, i)$ is length of $seq(s, i)$. Since finding the subset is an NP-hard problem, XTRACT system uses randomized and approximation algorithms to obtain at least a suboptimal result.

Example:

There are several subsets of S_f to demonstrate MDL cost computing:

$$S_1 = a^*b, abab, acc, ab = 12 + 16 + 12 + 8 + 7 = 48 + 7 = 55$$

$$S_2 = a^*b, (ab)^*, acc = 12 + 20 + 12 + 7 + 5 + 3 = 44 + 15 = 59$$

$$S_3 = (a|b)^*, acc = 24 + 12 + 12 + 11 + 7 = 36 + 30 = 66$$

$$S_4 = aaaab, abab, acc, ab = 20 + 16 + 12 + 8 = 56$$

The best subset is S_1 . This example does not demonstrate that DTD fragments generated by generalization and factoring modules are strong candidates for the final DTD. This feature takes effect only on longer and more complicated input sequences, starting on sequence $aaaab$ which is only sequence represented by DTD fragment different from itself. Aim of this example is to demonstrate and explain computing of XTRACT system.

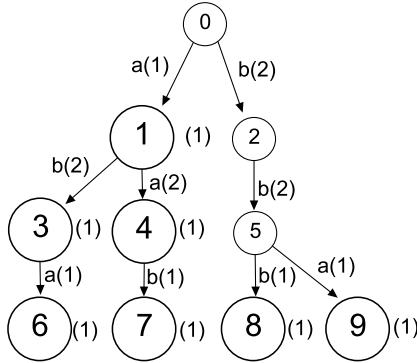


Figure 4.3: An example of PTA for $I = \{a, ab, aa, aba, bba, bbb\}$

4.1.3 sk-ANT

The sk-ANT algorithm presented in [22] by Wong and Sankey is one of grammatical inference methods. This method first constructs a Prefix Tree Automaton (PTA) from a given set of input strings and generalize it by merging states of the automaton. To determine which states should be merged, sk-ANT combines an Ant Colony Optimization (ACO) heuristic with an sk-strings method.

Prefix Tree Automaton

The PTA is constructed as follows. Let I is the set of input strings. With the first string from I , the automaton simply accepts this first string. Then, for each of the rest of the strings in I , the string shares as many states as possible. When a symbol is found, that does not match a valid transition, a new path is inserted into automaton that matches the rest of the string. An example is shown in Figure 4.3. The numbers in the parentheses represent frequencies of the state finalities and transitions. This allows the automaton to be used as a Probabilistic Finite State Automaton (PFSA).

The PTA represents the language that accepts exactly the input strings. Hence the goal is to generalize the language by merging states of the automaton.

Minimum Message Length

A Minimum Message Length (MML) is a measure of an inferred expression quality. Since expressions are encoded in a PFSA during computation, a PFSA must be encoded to determine its code length. The MML is expressed

as follows:

$$MML(A) = \sum_{j=1}^N \left\{ \log_2 \frac{(t_j - 1)!}{(m_j - 1)! \prod_{i=1}^{m_j} (n_{ij} - 1)!} \right\} \\ + M(\log_2 V + 1) + M' \log_2 N - \log_2(N - 1)!$$

where N is the number of states in the PFSA, V is the cardinality of the alphabet plus one, t_j is the number of times the j -th state is visited, m_j is the number of arcs from the j -th state (plus one for final states), m'_j is the number of arcs from the j -th state (no change for final states), n_{ij} is the frequency of the i -th arc from the j -th state, M is the sum of all m_j values and M' is the sum of all m'_j values.

Ant Colony Optimization

In this technique artificial ants browse the search space and leave pheromones on solutions they go through. The pheromones stand for a positive feedback and measure a quality of solution the ant found. The algorithm operates over several iterations to allow the positive feedback of the pheromones to take effect. The pheromone placement is delayed until the end of an iteration. In the next iteration, higher quality solutions are more likely to be chosen by ants. When a certain number of iterations end without improvement to the best solution, the algorithm terminates.

In Algorithm 5 the ACO algorithm is outlined. The parameter *ams* (ant move selector) is the way of how ants select state pairs to merge. An original algorithm showed in Algorithm 6 acts as follows. In loop over all merging possibilities first computes a heuristic value for the given merge at line 3 and at the line 5 the algorithm computes a weighting value from the heuristic value and the pheromones of the given merge. At line 8 a merge is randomly selected according to the weighting value.

The *heuristic*, *weighting* and *pheromones* functions are attributes and, hence, can be customized. However, the following functions are implemented:

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

where $\delta MML(A, merge)$ represents a change in the MML of A that would result from *merge*,

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

where *averageMML* is an average for all ants in the iteration, and

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

Algorithm 5 ACOOptimizer

Input: A PTA PTA_{R+} , a positive integer $stagLimit$, and an ant move selector ams

Output: A generalized form of the input PTA.

```
1:  $bestSolution := PTA_{R+}$ 
2:  $stagCount := 0$ 
3: while  $stagCount < stagLimit$  do
4:    $ants := makeAnts(antCount, PTA_{R+}, ams)$ 
5:    $liveCount := antCount$ 
6:    $improvement := false$ 
7:   while  $liveCount > 0$  do
8:     for  $ant$  in  $ants$  do
9:        $ant.step()$ 
10:      if  $ant.dead()$  then
11:        if  $mmlOf(ant.solution) < mmlOf(bestSolution)$  then
12:           $bestSolution := ant.solution$ 
13:           $improvement := true$ 
14:        end if
15:         $liveCount := liveCount - 1$ 
16:      end if
17:    end for
18:  end while
19:   $updatePheromones(ants)$ 
20:  if not  $improvement$  then
21:     $stagCount := stagCount + 1$ 
22:  else
23:     $stagCount := 0$ 
24:  end if
25: end while
26: return  $bestSolution$ 
```

where α and β are parameters of the function and are used as an exponential weight of heuristic and pheromone values respectively.

The sk -strings Method

This method is used for determining an equivalence of states of PTA. It is based on k -tails heuristic, which is a relaxed form of the Nerode equivalence relation. The Nerode relation says, that a pair of states are equivalent if they are indistinguishable in the strings that follow them. The k -tails relaxes this to only accepting tails up to a length k . The sk -strings method uses in addition PFSA and considers only the top s percent of the most probable k -strings. The k -strings differ from k -tails in that they do not have to end in a final state. The probability of a k -string is the product of probabilities of transitions that k -string must pass to be accepted by the automaton.

Algorithm 6 antMoveSelector

Input: A set of all state pairs *merges*, an ant heuristic *heuristic*, an ant weighting function *weighting* and a pheromone table *pheromones*.

Output: A state pair representing the chosen merge.

```
1: choices := []
2: for merge in merges do
3:   h := heuristic(merge)
4:   p := pheromones[merge]
5:   value := weighting(h, p)
6:   choices.add((value, merge))
7: end for
8: return stochasticChoice(choices)
```

There are five variants implemented (AND, OR, LAX, STRICT and XEN). Since the *sk*-ant algorithm uses the AND heuristic, only this heuristic will be described in this work. The Algorithm 7 provides the description of the AND heuristic.

sk-ANT

Finally the two methods are combined into *sk*-ANT heuristic which uses the best features of both. The *sk*-ANT heuristic is actually the modification of ACO with *ams* algorithm improved. The *sk*-antMoveSelector in Algorithm 8 differs from the Algorithm 6 in using the *skCriterion* function outlined in Algorithm 7. This criterion may be weakened when it becomes too strict (see lines 11-12). Note that if the criterion is weak enough to let all merges pass, the algorithm will behave identically to the original version.

Algorithm 7 *sk*-strings: andCriterion

Input: A real number s in $[0, 1]$, a positive integer k , and a pair of states s_1 and s_2 .

Output: A boolean, true if s_1 and s_2 are *sk*-AND equivalent.

```
1: tails := k-strings of  $s_1$ , sorted by decreasing order of probability
2: total := 0
3: for tail in tails do
4:   total := total + tail.probability
5:   if  $\delta(s_2, \textit{tail.string}) = \emptyset$  then
6:     return false
7:   end if
8:   if total  $\geq s$  then
9:     tails := k-strings of  $s_2$ , sorted by decreasing order of probability
10:    total := 0
11:    for tail in tails do
12:      total := total + tail.probability
13:      if  $\delta(s_1, \textit{tail.string}) = \emptyset$  then
14:        return false
15:      end if
16:      if total  $\geq s$  then
17:        return true
18:      end if
19:    end for
20:    return true
21:  end if
22: end for
23: return false
```

Algorithm 8 *sk*-antMoveSelector

Input: A set of all state pairs *merges*, an ant heuristic *heuristic*, an ant weighting function *weighting*, a pheromone table *pheromones* and an *sk*-strings criterion *skCriterion*.

Output: A state pair representing the chosen merge.

```
1: choices := []
2: while choices.size() = 0 do
3:   for merge in merges do
4:     if skCriterion(merge) then
5:       h := heuristic(merge)
6:       p := pheromones[merge]
7:       value := weighting(h, p)
8:       choices.add((value, merge))
9:     end if
10:  end for
11:  if choices.size = 0 then
12:    skCriterion.weaken()
13:  end if
14: end while
15: return stochasticChoice(choices)
```

4.2 XSD Based Solutions

Some of the existing works have focused on the XSD inference. However, only a couple of them infer schema, that cannot be expressed by a DTD. This section describes two such approaches - *iXSD* and Schema Miner.

4.2.1 *iXSD*

An algorithm *iXSD* has been introduced by Bex et al. in [3]. This algorithm is the first of known approaches, that generates output in XML Schema instead of DTD and goes behind the DTD expressive power by supporting XML types. The *iXSD* is a typical grammar-inferring approach.

In [3] the authors provide several observations important for grammatical inference methods for XML Schema constructs.

Observation 4.2.1 *The class of all XSDs cannot be learned from positive examples only.*

This means that no matter how many examples from a target XSD D are provided, there is no algorithm that will always retrieve D given only the examples. Due to this observation, a subclass of XSDs must be identified that at once can be learned from positive examples only and covers sufficient amount of XSDs occurring in practice.

Observation 4.2.2 *In more than 98 % of the XSDs occurring in practice, the content model of an element depends only on the label of the element itself, the label of its parent and (sometimes) the label of its grandparent.*

An XSD whose content models depend only on labels up to the k -th ancestor is called *k-local*.

Observation 4.2.3 *More than 99 % of the XSDs in practice consist of elements with content models as regular expressions in which each element name occurs only once. Such regular expressions are called Single Occurrence Regular Expressions (SOREs) and can be learned from positive examples only.*

SOREs also satisfy the W3C specification constraint, that all content models in an XSD must be deterministic.

In the light of these observations, Bex et al present an algorithm *iLOCAL* that can infer any k -local and single occurrence XSD from a “sufficiently large” set of XML documents.

An *iXSD* algorithm is a concatenation of the *iLOCAL* algorithm and an algorithm REDUCE that unifies equivalent and “sufficiently similar” types.

```

<accounts>
  <administrators>
    <administrator>
      <name>John Smith</name>
      <email>john.smith@email.com</email>
    </administrator>
  </administrators>
  <users>
    <user>
      <name>Jack Black</name>
      <email>jack.black@email.com</email>
    </user>
    <user>
      <name>John Doe</name>
      <email>john.doe@email.com</email>
      <setting>
        <name>theme</name>
        <value>blue</value>
      </setting>
      <setting>
        <name>pagesize</name>
        <value>100</value>
      </setting>
    </user>
  </users>
  <sessions>
    <user>
      <session_id>VE3DT56gY7HJ4BW23De4</session_id>
      <setting>
        <name>pagesize</name>
        <value>50</value>
      </setting>
    </user>
  </sessions>
</accounts>

```

Figure 4.4: XML Examples: a fragment from users.xml

Basic Definitions

Before *i*LOCAL and REDUCE algorithms may be described in detail, several definitions should be defined.

Definition 4.2.1 *An XML fragment is sequence $\langle a_1 \rangle f_1 \langle /a_1 \rangle \dots \langle a_n \rangle f_n \langle /a_n \rangle$ of elements where a_1, \dots, a_n are element names and f_1, \dots, f_n are XML fragments.*

Definition 4.2.2 *If f is an XML fragment, then $paths(f)$ is the set of all labeled paths starting at a root element in f .*

Example:

For the XML document in Figure 4.4:

$paths(f) = \{\lambda, \text{accounts}, \text{accounts administrators}, \text{accounts users}, \text{accounts sessions}, \text{accounts administrators administrator}, \dots\}$

Definition 4.2.3 $strings(f, p)$ is a set of all strings of element names occurring below an occurrence of path p in f .

Example:

For the XML document in Figure 4.4:

$strings(f, \lambda) = \{\text{accounts}\}$

$strings(f, \text{accounts}) = \{\text{administrators users sessions}\}$

$strings(f, \text{accounts users user}) = \{\text{name email, name email setting setting}\}$

...

Definition 4.2.4 An XSD is a triple $D = (T, \rho, \tau)$ consisting of a finite set of types T ; a mapping ρ from T to regular expressions r as given by the syntax

$$r ::= \lambda | a | r, r | r + r | r^* | r^+ | r?$$

where λ denotes the empty string and a ranges over element names; and a mapping τ that assigns a type to each pair (t, a) with the element name a occurring in $\rho(t)$.

The function $\tau(t, a)$ uniquely identifies an element type for the element a occurring in the type definition of the type t .

Example:

For the XML document in Figure 4.4:

$\tau(\text{root}, \text{accounts}) = \text{accounts}$

$\tau(\text{users}, \text{user}) = \text{user1}$

$\tau(\text{sessions}, \text{user}) = \text{user2}$

The mapping $\rho(t)$ stands for the ordinary regular expression over element names without type definition in the definition of t .

Example:

For the XML document in Figure 4.4:

$\rho(\text{user}) = \text{name, email, setting}^*$

Definition 4.2.5 $F(D, t)$ is a set of all XML fragments of type t in D .

Definition 4.2.6 Let $p|_k$ stand for the path formed by the k last element names of a path p (If $\text{length}(p) \leq k$ then $p|_k = p$). Two paths p and q are k -equivalent if $p|_k = q|_k$.

If $\text{length}(p) < k$, p is k -equivalent to itself only.

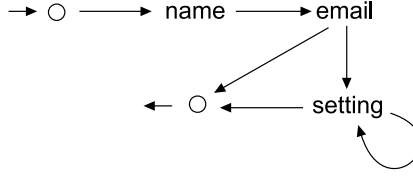


Figure 4.5: Example of SOA - accepting the same language as the SORE `name, email, setting*`

Definition 4.2.7 A pair (D, s) with D being an XSD and s being a type in D is called k -local if for all k -equivalent p and q such that $\tau(s, p) \rightarrow t$ and $\tau(s, q) \rightarrow t' : t = t'$.

For example, if D is an XSD for document in Figure 4.4, $(D, root)$ is not 1-local, because $p = \text{account users user}$ and $q = \text{account sessions user}$ are 1-equivalent, and $\tau(root, p) \rightarrow \text{user1}$ and $\tau(root, q) \rightarrow \text{user2}$. $(D, root)$ is then 2-local.

Definition 4.2.8 A regular expression r is single occurrence (SORE) if every element name occurs at most once in it. An XSD is single occurrence (SOXSD) if it contains only single occurrence regular expressions (SOREs).

Definition 4.2.9 Let in and out be two special symbols, distinct from the element names, that will serve as the initial and final state, respectively. A single occurrence automaton (SOA) is a graph $A = (V, E)$ where all states in $V - \{in, out\}$ are element names, and $E \subseteq (V - \{out\}) \times (V - \{in\})$ is the edge relation.

An example of SOA is outlined in Figure 4.5.

Definition 4.2.10 $L(A)$ is a set of all strings accepted by the SOA A .

The i LOCAL Algorithm

The goal of the i LOCAL algorithm is to infer a k -local SOXSD for a given k and a finite corpus of XML fragments C . The algorithm is outlined in Algorithm 9.

At the first line, i LOCAL constructs a type $p|_k$ for each path p in $paths(C)$.

Algorithm 9 *i*LOCAL

Input: a natural number k and corpus C

Output: a k -local SOXSD (D, t) such that $C \subseteq F(D, t)$

- 1: Let the set of types T consist of all $p|_k$ with $p \in \text{paths}(C)$
 - 2: Initialize the mappings ρ and τ to empty
 - 3: **for** each type $p|_k$ in T **do**
 - 4: add $p|_k \mapsto \text{TOCORE}(i\text{SOA}(k\text{-strings}(C, p|_k)))$ to ρ
 - 5: **end for**
 - 6: **for** each path pa in $\text{paths}(C)$ **do**
 - 7: add $(p|_k, a) \mapsto (pa)|_k$ to τ
 - 8: **end for**
 - 9: **return** (D, t) with $D = (T, \rho, \tau)$ and $t = \lambda$
-

Algorithm 10 The *i*SOA algorithm

Input: a finite set of sample strings S

Output: a SOA A such that $S \subseteq L(A)$

- 1: Let V be the set of states consisting of all element names occurring in S plus the initial state in and final state out.
 - 2: Initialize $E := \emptyset$
 - 3: **for** each string $a_1 \dots a_n$ in S **do**
 - 4: add the edges $(in, a_1), (a_1, a_2), \dots, (a_n, out)$ to E
 - 5: **end for**
 - 6: **return** $A = (V, E)$
-

Example: Type inference

$k = 2$

$C =$ XML fragment from Figure 4.4

T consists of the following types:

$\lambda,$	accounts,
accounts administrators,	administrators administrator,
accounts users,	users user,
accounts sessions,	sessions user,
administrator name,	administrator email,
user name,	user email,
user setting,	user session_id

At lines 3-5 *i*LOCAL constructs the content models for previously computed types. Here $k\text{-strings}(C, p|_k)$ stands for the set of all strings in C that occur below paths that are k -equivalent to p :

$$k\text{-strings}(C, p|_k) := \bigcup \{ \text{strings}(f, q) \mid f \in C, q \in \text{paths}(f), p|_k = q|_k \}.$$

The Algorithm 10 *i*SOA learns a SOA A from a finite set of sample strings S .

Example: SORE inference $k = 2$ $p|_k = \text{accounts users}$ $k\text{-strings}(C, p|_k) := \{\text{name email, name email setting setting}\}$ $i\text{SOA}(k\text{-strings}(C, p|_k)) := \text{SOA in Figure 4.5}$ $\text{ToSORE}(i\text{SOA}(k\text{-strings}(C, p|_k))) := \text{name, email, setting*}$ **Minimization**

Because *iLOCAL* tends to generate more types than necessary, in the worst case, *iLOCAL*(C, k) may return an XSD with $O(n^k)$ types where n is the number of different element names occurring in C , the number of generated types must be reduced. Here minimization is done by Algorithm 11 MINIMIZE. The algorithm minimizes an XSD D by unifying equivalent types in D . Here s is equivalent to t if $F(D, s) = F(D, t)$.

Algorithm 11 MINIMIZE**Input:** an XSD $D = (T, \rho, \tau)$ and type $r \in T$ **Output:** (D, r) with redundant types in D removed

- 1: **while** there are distinct types s and t in T with $t \neq r$ and $F(D, s) = F(D, t)$ **do**
- 2: replace each $(s', a) \mapsto t$ in τ by $(s', a) \mapsto s$
- 3: remove $t \mapsto \rho(t)$ from ρ and t from T
- 4: **end while**

The MINIMIZE algorithm may fail to unify types which should be unified, when *iLOCAL* runs on incomplete corpora. Hence, minimization algorithm is needed, that not only unifies equivalent types, but also unifies 'similar' types. The algorithm REDUCE does so by adapting D such that $F(D, s) = F(D, t)$, for all types s and t that are similar enough.

To define a similarity for types in an inferred XSD, *iSOA* algorithm must be adapted, such that for each edge (a, b) of the automaton A learned for a sample S the support $\text{supp}_A(a, b)$ is kept. The $\text{supp}_A(a, b)$ is the number of strings in S for which (a, b) needed to be added to the edges of A . An example is outlined in Figure 4.6.

The similarity of two types s and t then can be defined as follows. Let $\text{dist}(A, B)$ be the *normalized edit distance* between the support-annotated SOAs $A = (V, E)$ and $B = (W, F)$:

$$\text{dist}(A, B) := \frac{\sum_{(a,b) \in E-F} \text{supp}_A(a, b)}{\sum_{(a,b) \in E} \text{supp}_A(a, b)} + \frac{\sum_{(a,b) \in F-E} \text{supp}_B(a, b)}{\sum_{(a,b) \in F} \text{supp}_B(a, b)}$$

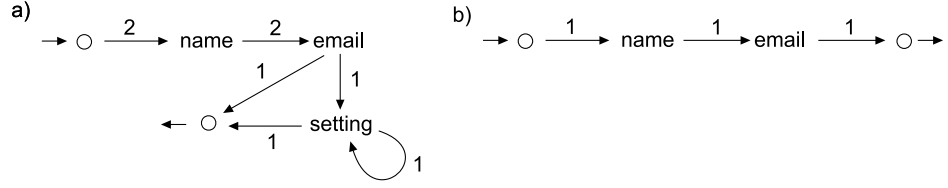


Figure 4.6: The SOAs for a) `accounts users user` and b) `accounts administrators administrator` with support.

Intuitively, $dist(A, B)$ counts the number of edges present in A but not in B and the number of edges present in B but not in A , weighted by the support these edges have in the original sample.

Example:

A = the SOA in Figure 4.6 a)
 B = the SOA in Figure 4.6 b)

$$\begin{aligned} \sum_{(a,b) \in E-F} supp_A(a,b) &= 3 \\ \sum_{(a,b) \in E} supp_A(a,b) &= 8 \\ \sum_{(a,b) \in F-E} supp_B(a,b) &= 0 \\ \sum_{(a,b) \in F} supp_B(a,b) &= 3 \\ dist(A, B) &= \frac{3}{8} + 0 = \frac{3}{8} \end{aligned}$$

The edit distance $dist_D(s, t)$ between the types s and t is then defined as

$$dist_D(s, t) = \max_{(s', t') \in reach_D(s, t)} dist(soa(s'), soa(t'))$$

The algorithm REDUCE is outlined in Algorithm 12. There are two more functions that must be defined.

Definition 4.2.11 For an XSD $D = (T, \rho, \tau)$, let $elems_D(t)$ denote the set of all element names a for which $\tau(t, a)$ is defined. The set $reach_D(s, t)$ of pairs of types jointly reachable from (s, t) is the least set containing (s, t) such that $(s', t') \in reach_D(s, t)$ and $a \in elems_D(s') \cap elems_D(t')$ implies that $(\tau(s', a), \tau(t', a)) \in reach_D(s, t)$.

Intuitively, $reach_D(s, t)$ is the set of all pairs (s', t') for which there exists a path p such that $\tau(s, p) \rightarrow s'$ and $\tau(t, p) \rightarrow t'$.

The algorithm merges types whose edit distance is less than the parameter ε . In the first phase, lines 4 - 15, the selected types s and t are transformed so that $F(D, s) = F(D, t)$ and hence, will be merged in MINIMIZE. It does so by adjunction of $soa(s')$ with $soa(t')$ at line 6. All states and edges in

Algorithm 12 REDUCE

Input: an inferred XSD $(D, r) = iLOCAL(k, C)$ for some k and C , and a similarity threshold ε

Output: (D, r) with similar types in D merged and redundant types removed

- 1: let $(T, \rho, \tau) = D$
- 2: initialize $M := \{(s, t) \in T^2 \mid 0 < dist_D(s, t) < \varepsilon\}$
- 3: **while** M is non-empty **do**
- 4: **for** each $(s, t) \in M$ **do**
- 5: **for** each $(s', t') \in reach_D(s, t)$ **do**
- 6: set $soa(s') := soa(s') \uplus soa(t')$
- 7: set $soa(t') := soa(s')$
- 8: **for** each a in $elems_D(t') - elems_D(s')$ **do**
- 9: add $(s', a) \mapsto \tau(t', a)$ to τ
- 10: **end for**
- 11: **for** each a in $elems_D(s') - elems_D(t')$ **do**
- 12: add $(t', a) \mapsto \tau(s', a)$ to τ
- 13: **end for**
- 14: **end for**
- 15: **end for**
- 16: recompute $M := \{(s, t) \in T^2 \mid 0 < dist_D(s, t) < \varepsilon\}$
- 17: **end while**
- 18: **for** each type t in T **do**
- 19: replace each $t \mapsto \rho(t)$ in ρ by $t \mapsto ToSORE(soa(t))$
- 20: **end for**
- 21: MINIMIZE(D, r)

$soa(t')$ that are not in $soa(s')$ are added to $soa(s')$ and supports are updated with

$$supp_{soa(s') \uplus soa(t')}(a, b) := supp_{soa(s')}(a, b) + supp_{soa(t')}(a, b).$$

The figure 4.7 shows the adjunction of the SOAs in Figure 4.6. Lines 18-20 converts the updated SOAs into SOREs. Finally, the MINIMIZE algorithm is called at line 21.

Conclusion

The $iXSD$ algorithm acts with the power of XSD. It constructs content models for element types instead of element names. The algorithm has two attributes. The attribute k defines size of context in which types are identified, ε determines the sensitivity of REDUCE algorithm. In the next work, Bex et al focuses on finding the best value of k .

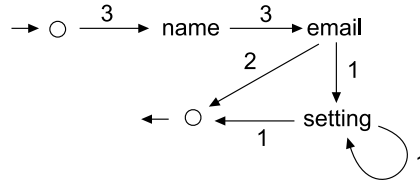


Figure 4.7: Adjunction of the SOA in Figure 4.6 a) with the SOA in Figure 4.6 b)

4.2.2 Schema Miner

Another way how an XSD can be inferred from a given set of XML documents was presented in [14] (in Czech, with shorter English version [15]). In this thesis Vošta presents a merging state algorithm based on the sk-ANT heuristic (see Subsection 4.1.3) with some additional improvements suited for an XSD, especially for inferring unordered sequences `<xs:a11>`.

The main body of the algorithm consists of four steps. Firstly, the Document Tree T is created from each document D from the input set of XML documents I_D . The Document Tree was presented in Subsection 4.1.1 and an example is shown in Figure 4.1. Then, from the set of document trees I_T a Dependency graph G is constructed as outlined in Algorithm 13.

Definition 4.2.12 *A Dependency Graph is a directed graph $G = (V, E)$, where V represents a set of nodes that corresponds to element names in all input documents, and E is a set of edges s.t. $(A, B) \in E$ if and only if there exists an input document, where element B is a direct subelement of A .*

Algorithm 13 createDependencyGraph

Input: A set of Document Trees I_T

Output: A Dependency Graph G

- 1: $G :=$ empty graph
 - 2: **for** each T in I_T **do**
 - 3: $element := T.root$
 - 4: **if** not exists a node for $T.root$ in G **then**
 - 5: create node $element$ in G
 - 6: **end if**
 - 7: $insertDependencies(G, T.root)$
 - 8: **end for**
 - 9: **return** G
-

Algorithm 14 insertDependencies

Input: A Dependency Graph G and an element $element$

Output: An edited Dependency Graph G

```
1:  $node :=$  node for  $element$  in  $G$ 
2: for each direct subelement of  $element$  do
3:   if not exists a node for it in  $G$  then
4:     create node  $subelement$  in  $G$ 
5:   end if
6:    $targetNode :=$  node for  $subelement$  in  $G$ 
7:   create an edge in  $G$  for  $(node, targetNode)$ 
8:    $insertDependencies(G, subelement)$ 
9: end for
```

In the next two steps the main inference algorithm is used. Firstly, the clustering of elements determines element types, then each type is generalized. In the final fourth step, the generalized types are transformed into the XSD syntax.

Clustering of Elements

In this step elements with the same name are clustered according to their similarity. Hence, a similarity measure must be specified.

Definition 4.2.13 *An element tree T_e is a subtree of the Document Tree T where element e is the root of T_e .*

As a similarity measure a modified idea of *tree edit distance* is used. The edit distance of trees T_e and T_f ($dist(T_e, T_f)$) is expressed as the number of edit operations needed to transform T_e into T_f . Since the classical tree edit distance measure, with only *insert node* and *delete node* edit operations allowed, is not suited for recursive trees, the set of allowed edit operations should be extended. The basic example is shown in Figure 4.8. These Document Trees have the same XML schema and the optimal edit distance hence should be 0. But having only operations *insert node* and *delete node*, the edit distance is 4.

The allowed edit operations for recursive trees are thus as follows.

- *Insert* - a single node n is inserted to the position given by parent node p and ordinal number expressing its position among subelement of p
- *Delete* - a leaf node n is deleted
- *Relabel* - a node n is relabeled

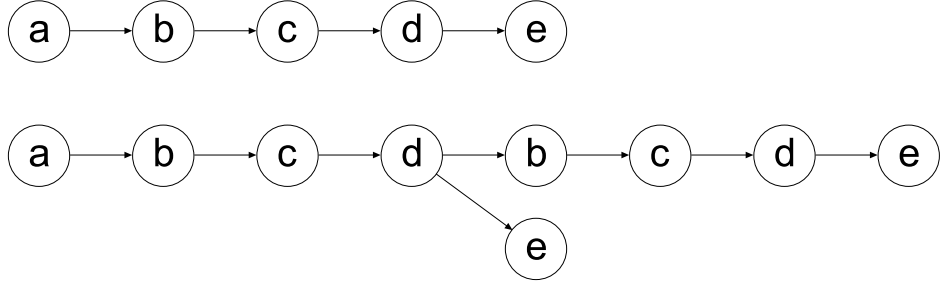


Figure 4.8: Tree edit distance of recursive element trees

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

To determine the Tree edit distance for given trees T_e and T_f , all possible edit sequences have to be evaluated and the best one chosen. Since there can be a lot of such possibilities, a set of constraints was specified.

Definition 4.2.14 *A sequence of edit operation is allowable if it satisfies the following two conditions:*

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

Clustering algorithm itself is in this work a modification of *mutual neighborhood clustering* (MNC) algorithm [7]. Firstly elements are separated into the clusters according to their context (a path from root to the given element). Next, representative elements are chosen for each cluster. The MNC algorithm calculates only with the representative elements. Two elements T_e and T_f are placed into the same group (and their clusters are merged), if

$$dist(T_e, T_f) \leq dist_{MIN} \vee (dist(T_e, T_f) \leq dist_{MAX} \& MN(T_e, T_f) \leq F)$$

where $dist_{MIN}$ represents minimum distance, $dist_{MAX}$ represents maximum distance, F is a factor and the mutual neighborhood $MN(T_e, T_f)$ is defined as follows:

Definition 4.2.15 Let T_e and T_f be two element trees, where T_e is i -th closest neighbor of T_f and T_f is j -th closest neighbor of T_e . Then mutual neighborhood of T_e and T_f is defined as $MN(T_e, T_f) = i + j$.

Schema Generalization

For schema generalization a modification of sk-ANT heuristic (described in Section 4.1.3) is used. Note that sk-ANT heuristic combines the ACO principle with the sk-string method as a selector of states to be merged. Vošta in his thesis modifies ACO heuristic with adding a temporary negative feedback which enables to search a larger subspace of possible generalizations. The negative feedback is assigned after each step of an ant. At the end of each iteration, all negative feedbacks are zeroed.

The next modification of the sk-ANT heuristic is related to the generating of a set of possible movements. Here, apart from the sk-string method, two other methods are applied.

The k, h -context method says, that two states t_x and t_y are identical, if there exist two identical paths of length k terminating in t_x and t_y . In addition, also h preceding states in these paths are then identical. This method hence inserts such states to a set of merge candidates.

The second additional method is focused on inferring unordered sequences. Replacing a set of ordered sequences of elements with a single unordered sequence represented by the & operator can be considered as a special kind of merging states. The currently recommended version 1.0 of XML Schema specification [18, 19] allows to specify an unordered sequence of elements, each with the allowed occurrence of (0,1). The new specification 1.1, currently in the phase of a working draft, allows the number of occurrence of items of the sequence to be (0,∞). In this algorithm Vošta focuses on the latter possibility, since it will probably soon become a recommendation.

First-Level Candidates

For the purpose of identification of subgraphs representing the allowed type of unordered sequences, *common ancestors* and *common descendants* must be defined.

Definition 4.2.16 Let $G = (V, E)$ be a directed graph. A *common descendant* of a node $v \in V$ is a descendant $d \in V$ of v s.t. all paths traversing v traverse also d .

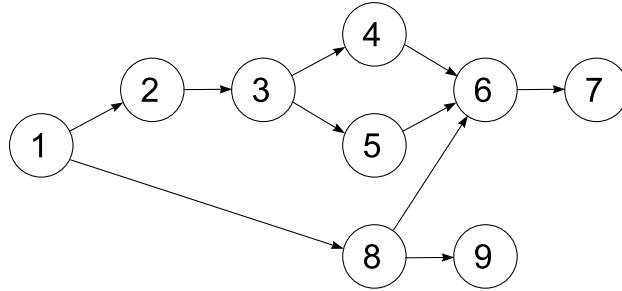


Figure 4.9: An example of common ancestors and descendants

Definition 4.2.17 Let $G = (V, E)$ be a directed graph. A common ancestor of a node $v \in V$ is an ancestor $a \in V$ of v s.t. all paths traversing v traverse also a .

Example:

Considering the graph in Figure 4.9.

Common descendants of node 3 are nodes 6 and 7.

Node 1 has no common descendants since paths traversing node 1 terminates in nodes 7 and 9.

Common ancestor of node 6 is node 1.

Note that there can occur paths which traverse a but not v . Hence the definition of *common ancestor* must be restricted as follows:

Definition 4.2.18 Let $G = (V, E)$ be a directed graph. A common ancestor of a node $v \in V$ with regard to a node $u \in V$ is an ancestor $a \in V$ of v s.t. a is a common ancestor of each direct ancestor of v occurring on path from u to v .

Example:

Considering the graph in Figure 4.9.

Common ancestor of node 6 with regard to node 2 are nodes 2 and 3.

With the definitions, subgraphs can be identified which are considered as first-level candidates for unordered sequences. Using definition of common descendant, the candidate subgraph contains node v as an input node, its common descendant d as an output node and all nodes occurring on paths from v to d . Similarly for common ancestor, where a is an input node and v is an output node. A node n_{in} is an input node of block representing a first-level candidate if

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

The first condition ensures that there are at least two paths leading from n_{in} representing at least two alternatives. The third condition ensures that there are no paths entering or leaving the block otherwise than using n_{in} or n_{out} . For example in Figure 4.9 the only first-level candidate is subgraph consisting of nodes 3, 4, 5, 6.

Second-Level Candidates

Second-level candidates represent an unordered sequence and, hence, they are candidates for merging of states. To determine *Second-Level Candidates* each *First-Level Candidate* must be checked for fulfilling conditions of an unordered sequence. Vořta does so by comparing the similarity of the first-level candidates with P_n automata. P_n automaton accepts each permutation of n items having all the states fully merged. The maximum path length l_{max} in the candidate graph denotes the size of the permutation. The candidate graph is then compared with $P_{l_{max}}$ graph using an edit-distance algorithm. Note, that the candidate graph must be always a subgraph of $P_{l_{max}}$. Hence, the edit operations can be reduced to the following operations:

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

From all the possible edit sequences the one with the lowest total cost is chosen.

4.3 Conclusion

In this section the main schema inference methods were described. The older approaches focused on inference of a DTD [11, 4, 22], latest works are mainly focusing on inferring of an XSD and its additional features [3, 14].

The grammatical inference method *iXSD* [3] described in Subsection 4.2.1 tries to identify subclasses of XSDs that can be learned from positive example

only and present theoretically complete algorithm that can infer any XSD from the subclass being given only positive examples. This method goes behind the expressive power of the DTD by using element types instead of element names in determining element groups. This is a basic way of how to use the power of XML Schema in schema inference.

The same approach of using element types is also used in heuristic method [14] presented by Vošta described in Subsection 4.2.2. This method firstly determines element types and then for each type infers the regular expression describing the content. Here Vošta extended regular expression by the new permutation operator & which corresponds to the unordered sequences of XML Schema.

In the schema inference features of the XML Schema language other than element types and unordered sequences have not been implemented yet. Features waiting for the implementation are mainly inheritance, substitution groups or globally defined groups. This work is focusing on these features in the next chapters. In particular there will be shown how user interaction can help with finding difficult constructs such as inheritance to get more exact XSD.

Chapter 5

Proposed Algorithm

5.1 Motivation

XML Schema is a very complex language. It allows a user to define schema of documents more precisely and in more detail than the DTD. For example XML Schema has large range of built-in data types and allows a user to define his own types. It allows to define an element with the same name but different structure or it uses a sort of inheritance to define relations between element types. However, Some constructs of the XML Schema language are only “syntactic sugar” which do not go behind the expressive power of the DTD and lead only to more concise schema. For example, unordered sequences represented by the *all* element reduce the size of final XML schema by listing elements available in the unordered sequence ($a&b&c$) instead of explicitly specifying all permutations of elements ($a(bc|cb)|b(ac|ca)|c(ab|ba)$). Since the XML Schema language has a large variety of constructs, it is not rare that a regular expression can be expressed by more XML Schema constructs. The easiest example is a use of unordered sequences instead of a choice from a set of permutations. The large scale of XML Schema constructs together with the difficulty of their inference is the cause of the fact that there is still no XML Schema inference method that deals with its full expressive power.

In this chapter possibilities of how to use XML Schema constructs in the process of inference an XSD from a given set of XML samples will be discussed. The main aim of the discussion is to show how user interaction can be integrated into the process of schema inference to obtain a more precise and realistic schema. When a user participates in the decision process in the right place, the algorithm should give better results in a shorter time. The questions are, where the right places are, how a user should interact and

```
<bicycles>
  <bicycle>
    <wheel/>
    <wheel/>
  </bicycle>
  <bicycle>
    <wheel/>
    <wheel/>
  </bicycle>
</bicycles>
```

Figure 5.1: XML Examples: a fragment from bicycles.xml

what knowledge should a user have. Next to the discussion, the implemented inferring algorithm will be described.

5.2 Inferring Algorithm

The proposed algorithm, called *Schema Builder*, is based on the Schema Miner method (see Subsection 4.2.2) presented by Vošta in [14]. This algorithm was chosen because it is heuristic, which suits well with user interaction, it infers XSDs and focuses also on two constructs of XML Schema language - element type and unordered sequences. However, Schema Builder adds ability to define the same type for elements with the different name but similar structure and ability to define inheritance between two types.

The algorithm is split into three steps. In the first step elements from input documents are clustered according to their types. The second step of the algorithm infers a finite state automata accepting all input strings for each cluster from the first step. The last step takes a finite state automata and converts it to the XML Schema language.

5.2.1 User Interaction and Participation Measure

User interaction is a very wide concept. A user can participate in the inferring process at various places and can have different XML and XML Schema knowledge. A user can also participate in every decision to obtain the optimal result, or put up with a suboptimal result to save his work.

The algorithm allows a user to control the participation measure by setting proper initial parameters and, thus, choose a configuration which leads to the optimal relation between user participation measure and the quality of inferred XML schema.

For example having an XML fragment from Figure 5.1, a user can set, that if an element repeats two or more times, the repetition should be un-

```
<xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<mail>
  <from>
    <name>...</name>
    <address>...</address>
  </from>
  <to>
    <name>...</name>
    <address>...</address>
  </to>
  <subject>...</subject>
  <body>...</body>
</mail>
```

Figure 5.2: XML Examples: mail.xml

bounded. But this configuration yields to the bicycles with more than two wheels, which is incorrect. On the other hand, if a user sets the minimal repetition parameter to three, `<bicycles>` element will have only two `<bicycle>` subelements and this is also incorrect. Solution is to mark the repetition of `<bicycle>` subelements manually.

The concrete ways of how a user can direct the participation measure will be discussed later in appropriate subsections.

The next issue is also a fallibility of a user. A user can do mistakes which yield in invalid schema or schema which does not match with the input XML documents. An inferring algorithm should control decisions of the user and warn him if he does something wrong. But sometimes should control mechanisms yield to an unacceptable complexity and thus is better to trust the user.

5.2.2 Clustering of Elements

XML Schema is a type based language. Each element must have a type. Vošta in his thesis focuses only on inferring different types for elements which have the same name but different structure. He clusters elements according to their names and then splits these clusters when there are different structures for elements with the same name. The following algorithm can also define the same type for elements which have different names but similar structure. This yields to a higher complexity, since Vošta does not compare elements with different names. The following example shows, how the computation differs.

Example:

Having XML file from Figure 5.2:

Clusters using Schema Miner:

```
mail mail      from mail.from
to mail.to     subject mail.subject      address
body mail.body name mail.from.name, mail.to.name
mail.from.address, mail.to.address
```

Candidates for further split are name and address. The treeEditDistance for mail.from.name and mail.to.name is 0, cluster remains unchanged. Similarly for the address cluster.

Number of comparisons = 2

Number of clusters = 7

Clusters using Schema Builder:

```
mail          mail.subject
mail.from     mail.to
mail.from.name mail.to.name
mail.from.address mail.to.address
mail.body
```

The algorithm compares clusters each to each. The following cluster pairs have the treeEditDistance equal to the 0 and should be merged:

```
mail.from, mail.to
mail.from.name, mail.to.name
mail.from.address, mail.to.address
```

Number of comparisons = 19

Number of clusters = 6

Definition 5.2.1 A type cluster of the type T is a set C_T of XML elements which have the same XSD type. A string of the XML element E is a sequence of names of subelements of E with preserved order. The type strings S_T is a set of strings of each $E \in C_T$

Algorithm 15 mergeTypes

Input: A set of Clusters C

Output: A set of Clusters C after merge process

```
1: for each pair  $T_1, T_2$  in  $C$  do
2:    $distance := treeEditDistance(T_1.representant, T_2.representant)$ ;
3:   if  $distance < MAX\_DISTANCE$  then
4:      $merge(C, T_1, T_2)$ ;
5:   end if
6: end for
7: return  $C$ 
```

The proposed algorithm, outlined in Algorithm 15, works as follows: In the first step, elements are clustered according to their context. Firstly, elements with the same name and with the same path from root will have the same type, since the XML Schema does not allow different types for element names within the same context. Once elements are clustered according to their context, the algorithm should determine, if there are clusters which are similar enough to have the same type. As a similarity measure, *treeEditDistance* outlined in Algorithm 16 is used. The proposed *treeEditDistance* algorithm is similar to the algorithm described in Section 4.2.2. Some changes will be discussed later.

If no clusters will be merged, algorithm must do $\frac{n^2}{2}$ comparisons, where n is the number of the initial clusters. But when merging will be forced, number of comparisons may rapidly decrease. From the above example, when clusters `mail.from` and `mail.to` are merged, contexts of subelements become equal and, thus, clusters `mail.from.name`, `mail.to.name` and `mail.from.address`, `mail.to.address` should be also merged and cannot be compared anymore.

Algorithm 16 treeEditDistance

Input: a pair of trees T_1, T_2

Output: A tree edit distance as integer in range 0 - 100

1: *distance* := *nodeDistance*($T_1.root, T_2.root, T_1.root, T_2.root, true$);

2: **return** (*distance* * 100) / ($T_1.size + T_2.size$);

Tree Edit Distance

As mentioned before, the *treeEditDistance* algorithm is a modified algorithm described in Section 4.2.2. The main difference is that in Schema Builder two trees with roots with the different names can be compared (for example `mail.from` and `mail.to` elements and their subtrees). When the roots have different names, recursive computation cannot be applied.

Example:

Comparing two clusters `mail.from` and `mail.from.name`:

nodesList1 = {*name*, *address*}

nodesList2 = {}

Let's iterate over the *nodesList1*

name found in the second tree (the root itself), their distance is 0

address not found in the second tree, distance is 1

The result with the recursion on the elements with different names allowed is 1, the result with such recursion disabled is 2.

Algorithm 17 nodeDistance

Input: $node1, node2, root1, root2, recursively$ as boolean

Output: A distance between $node_1$ AND $node_2$ as integer

$nodesList1 := sortChildNodes(node1);$

$nodesList2 := sortChildNodes(node2);$

$distance := 0;$

while $nodesList1.hasMoreElements$ AND $nodesList2.hasMoreElements$ **do**

$child1 = nodesList1.actual;$

$child2 = nodesList2.actual;$

if $child1.name == child2.name$ **then**

$distance+ = nodeDistance(child1, child2, root1, root2, recursively);$

$nodesList1.next;$

$nodesList2.next;$

else if $child1.name < child2.name$ **then**

if $recursively$ AND $node1.name == node2.name$ **then**

$subtreeDistance = subtreeDistance(child1, root1, root2);$

$topDistance = topDistance(child1, node2, root1, root2);$

$distance+ = \min(subtreeDistance, topDistance);$

else

$distance+ = child1.size;$

end if

$nodesList1.next;$

else

if $recursively$ AND $node1.name == node2.name$ **then**

$subtreeDistance = subtreeDistance(child2, root2, root1);$

$topDistance = topDistance(child2, node1, root2, root1);$

$distance+ = \min(subtreeDistance, topDistance);$

else

$distance+ = child2.size;$

end if

$nodesList2.next;$

end if

end while

while $nodesList1.hasMoreElements$ **do**

$child1 = nodesList1.actual;$

if $recursively$ AND $node1.name == node2.name$ **then**

$subtreeDistance = subtreeDistance(child1, root1, root2);$

$topDistance = topDistance(child1, node2, root1, root2);$

$distance+ = \min(subtreeDistance, topDistance);$

else

$distance+ = child1.size;$

end if

$nodesList1.next;$

end while

Algorithm 18 nodeDistance - continue

```
while nodesList2.hasMoreElements do
  child2 = nodesList2.actual;
  if recursively AND node1.name == node2.name then
    subtreeDistance = subtreeDistance(child2, root2, root1);
    topDistance = topDistance(child2, node1, root2, root1);
    distance+ = min(subtreeDistance, topDistance);
  else
    distance+ = child2.size;
  end if
  nodesList2.next;
end while
return distance;
```

Algorithm 19 subtreeDistance

Input: *child* to explore, *root1* child's root, *root2* root to find in subtree

Output: a minimum distance for the recurse

```
1: elementsList := findInSubtree(child, root2);
2: for each element in elementsList do
3:   min := min(min, nodeDistance(element, root2, root1, root2, false));
4: end for
5: return min;
```

The presented example shows that the unchanged algorithm does not work for this purpose, since it decrease a distance by applying the recursion when it should not be applied. The worst case is comparing an element with its child, when the element contains only the child or its repetition. In this case the algorithm computes distance as 0 and thus clusters should be merged.

User Interaction

User interaction suggests itself in the place of determining wheather to merge types or not. There are two possible methods based on the participation measure. The former method is not to compute the *treeEditDistance* but to ask a user for every pair of types. The latter method is to ask a user only if the *treeEditDistance* is under some defined bound. The former method returns exactly the required type clusters and thus the result is optimal. But this approach forces a user to do up to $\frac{n^2}{2}$ decisions, where n is the number of clusters before the merging step. On the other hand, the latter method uses a heuristic rule and, thus, the result is only suboptimal. But, a user must interact only when the pair is a serious candidate for merging. An improvement would be a two-bound approach: If the distance falls below

Algorithm 20 topDistance

Input: *child* to find, *node* to start search in, *root1* child's root, *root2* node's root

Output: a minimum distance for the recurse

```
1: elementsList := findInTopPath(child, node, root2);
2: for EACH element IN elementsList do
3:   min := min(min, nodeDistance(child, element, root1, root2, false));
4: end for
5: return min;
```

the first bound, the types will be merged automatically. If the distance falls between the first and the second bound, a user will be asked.

The next improvement can be a method, that reduces a number of user decisions by using semantic methods. When two element names are semantically close to each other, the elements are merge candidates and user must decide, whether to merge or not. For example, having documents from Figures 5.2 and 5.3, `mail` and `multiple_mail` are semantically close and could have the same type.

From the above, Schema Builder have implemented user decisions according to the *treeEditDistance* result. In the configuration phase, a user will set a *layer₁* and a *layer₂* constants with an integer, where $0 \leq \textit{layer}_1 \leq \textit{layer}_2 \leq 100$ and a *mergeTypesUI* constant with one of the following values:

- *USERS_DECISION* - This means, that every merging step must be decided by the user.
- *TED_ONE_LAYER* - When the *treeEditDistance* $\leq \textit{layer}_1$, the merge must be decided by the user. Otherwise the types will not be merged.
- *TED_TWO_LAYERS* - When the *treeEditDistance* $\leq \textit{layer}_2$, types will be merged. Otherwise as in the *TED_ONE_LAYER* option.
- *BUILDERS_DECISION* - When the *treeEditDistance* $\leq \textit{layer}_2$, types will be merged. Otherwise the types will not be merged.

The merging algorithm then acts as described in the proper option.

5.2.3 Type Inheritance

Determining type inheritance automatically is a very complex issue, which can be easily solved by using user interaction. A user can decide if one type will be restricted or extended from the other. But the problem is how to determine if the given extension or restriction is valid. For example elements

```

<xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<multiple_mail>
  <from>
    <name>...</name>
    <address>...</address>
  </from>
  <to>
    <name>...</name>
    <address>...</address>
  </to>
  <to>
    <name>...</name>
    <address>...</address>
  </to>
  <to>
    <name>...</name>
    <address>...</address>
  </to>
  <subject>...</subject>
  <body>...</body>
</multiple_mail>

```

Figure 5.3: XML Examples: multiple_mail.xml

mail and multiple_mail from Figures 5.2 and 5.3 seem to be good candidates for inheritance. A user can determine, that the multiple_mail element type can be extended from the mail element type by adding a multiple occurrence to the element to. However, extended element type is validated by concatenating a content model of parent element with a content model defined in the extended type. And this condition cannot be satisfied in this case, since additional to elements occur in the middle of an input string.

Solution would be, that in the phase of schema generalisation, if a valid schema cannot be inferred, a user will determine whether to merge the given types or to separate them completely. The further improvement would be analysing merge candidates whether the extension or restriction can be applied.

Another issue is where to place this part of the algorithm. One way is to place it next to the decision whether to merge types or not. This question can provide some additional options. When a user does not want to merge the types, he can mark, that one type is inherited from the other. But there is a problem with determining inheritance of types in the type merging phase. What to do, when inheritance is marked and one of parent and child type should be merged with another type? Considering the situation in the following example:

Example:

Having a set of types:

```
mailType incoming.mails.mail
multipleMailType incoming.mails.multipleMail - extended from
mailType
mailType2 outgoing.mails.mail
multipleMailType2 outgoing.mails.multipleMail - extended from
mailType2
mailType3 trash.mails.mail
multipleMailType3 trash.mails.multipleMail
```

- 1) merge mailType and mailType2
- 2) merge multipleMailType and multipleMailType3
- 3) merge multipleMailType and multipleMailType2

Lets put aside the inheritance validity issues. The first case is clear. The newly created type should be a parent for types multipleMailType and multipleMailType2. In the second case, a type with extension assigned should be merged with a type without inheritance. Here a newly created type can but also can not be extended from the mailType. In the third case two types with a different parent are merging. There are several possibilities. Inheritance can be removed from one or both child types or the parent types must be also merged.

The second way, which solves the problem, is to place the inheritance part after the type merging phase. Here type clusters will never change and the user is prevented from doing complicated decisions.

Implementation

Schema Builder places the type inheritance fragment after the type merging phase, where all type clusters are determined. A user can pair subtypes with supertypes with the proper inheritance type. The algorithm provides the user for every type a set of strings, that the type must accept. This can help the user with his decision.

5.2.4 Schema Generalisation

When type clusters are determined and the inheritance marked, a schema of each element type must be inferred. A content model can be represented by a regular expression or an equivalent deterministic finite state automaton. In Schema Builder, a content model is represented by a schema automaton, which is a special type of FSA. Thus, the problem of schema generalisation

is transformed to the problem of finding the best schema automaton that accepts at least the set of strings S_T for the type T .

Definition 5.2.2 *A Schema automaton (SA) is an extended finite state automaton. It is a hextuple $(\Sigma, S, S_x, s_0, \delta, F)$, where:*

- Σ is the input alphabet (a finite, non-empty set of element names),
- S is a finite set of basic states,
- S_x is a finite set of extended states,
- S_0 is an initial state, $S_0 \in S \cup S_x$,
- δ is the state-transition function $\delta : S \cup S_x \times \Sigma \cup \{\lambda\} \rightarrow S \cup S_x$
- F is a set of final states, $F \subseteq (S \cup S_x)$

Schema automaton behaves similarly like a deterministic FSA. The only difference is that Schema automaton contains in addition a set of extended states. An extended state s_x represents a subautomaton SA_x which accepts a part of the input string. When the s_x state is reached, the SA_x automaton continues in input processing. The subautomaton *consumes* as much symbols as possible and returns processing to the parent automaton. Every extended state has a helper state. Helper state is a basic state with only one transition point to it, the lambda transition from the extended state. When the processing is returned from the subautomaton, the automaton moves into the helper state according to the lambda transition.

Schema automaton has the same expressive power as a FSA. It can be transformed to the FSA with lambda edges as showed in Figure 5.4. Extended state has only one transition - lambda transition. The transition is redirected to the initial state of the subautomaton. All final states of the subautomaton has one additional lambda transition to the destination state of the original lambda transition. The FSA computation now works as follows. If there is a transition for the given letter, use the transition. Otherwise, if there is a lambda transition, use the lambda transition. Otherwise, return true if the actual state is a final state or false if not. The newly created FSA with lambda edges can be in addition transformed to the FSA without lambda edges by merging states connected with a lambda edge.

The following types of extended states can be present in the automaton:

Extension state This state is used, when the type is extended from another. The supertype is represented by a supertype automaton. According to the specification, the content model of the subtype is a concatenation of

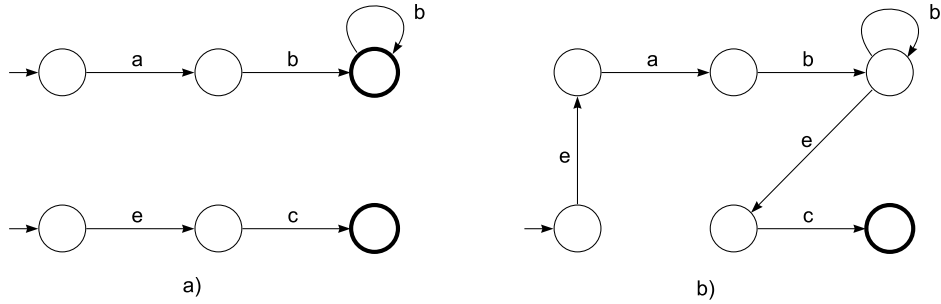


Figure 5.4: Schema automaton transformed to the FSA. a) up - a subautomaton, down - SA. b) transformed FSA with lambda edges. Here e stands for λ

the content model of the supertype and the content model defined in the extension. The extension state stands for the supertype automaton and is placed in the subtype automaton as an initial state. This construction ensures the concatenation of the supertype content model and the content model defined in the rest of the automaton.

Permutation state stands for the *all* particle in the schema. The subautomaton represented by this state models all possible permutations that the particle can accept.

Group state represents an arbitrary automaton. It stands for the *group* particle which can be seen as a reference to the globally defined particle. This state allows automata to reuse automaton fragments in more places, which decrease the number of defined states.

SA Construction

Having a type cluster C_T , the first step is to construct a schema automaton from a given set of input strings S_T . The schema automaton is constructed as PTA (see Figure 4.3). If the type is extended from another, the initial state is an extension state referencing the supertype automaton. Note that each input string is truncated to conform the rest of the string the supertype automaton cannot process. The rest of the SA is constructed as usual PTA.

For example in Figure 5.5, the first automaton was created for the type T having the input string set $\{ab, ac\}$, the second automaton was created for the type S extended from T having the input string set $\{abd\}$.

The generated schema automaton represents the type content model. It accepts all the input strings, but it may not be the best representation for the content model. The appropriate schema may be too big and not user-friendly. Hence, the schema must be generalised in a proper way, which

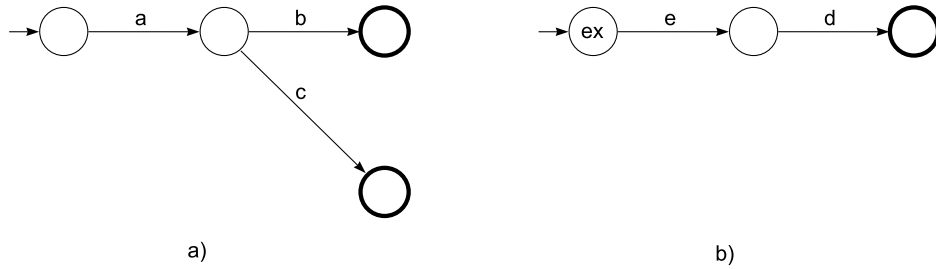


Figure 5.5: Schema automaton examples

allows an automaton to accept more than only given input strings, but also reduces the size of the schema and thus it becomes more user-friendly. To find the optimal schema is the crucial issue. The proposed algorithm is based on the Schema Miner approach 4.2.2 and it shows, how a user can participate to obtain the best possible schema.

SA generalisation

The schema represented by the given SA can be generalised by a sequence of SA modification steps. The sequence is composed from any combination of the following steps.

States merge. In this step two nodes are merged and all the transitions are redirected respectively. Note that for each pair of transitions with the same letter, if the destination nodes are not equal, the destination nodes must be also merged for the preservation of the determinism. The Algorithm 21 outlines the core of the merging step. Figure 5.6 shows an example of a states merge.

Permutation substitution. Here a subgraph is localised, that can be substituted by the permutation state. The Algorithm 22 shows, how the subgraph is substituted. The subgraph is eliminated by one initial state and one final state. All transitions that end in the initial state of the subgraph are redirected to the newly created permutation state. All transitions that start or end in the final state of the subgraph are redirected to start or end in the helper state respectively. An example is shown in Appendix B.

Group substitution In this step a subgraph is localised, which can be extracted for the purpose of the code reuse. The subgraph is substituted in the same way as a permutation. The only difference is that the group state is created instead of the permutation state. The group state holds the reference to the extracted subgraph.

Algorithm 21 mergeStates

Input: two states s_i and s_j to merge
Output: the generalised schema automaton
 $s_j.redirectTransitions(s_i)$
if $s_j.final$ **then**
 $s_i.final = true$
end if
for all $transition_i, transition_j$ **IN** s_i **do**
 if $transition_i.letter == transition_j.letter$ **then**
 if $transition_i.to \neq transition_j.to$ **then**
 $mergeStates(transition_i.to, transition_j.to)$
 else
 $s_i.deleteTransition(transition_j)$
 end if
 end if
end for

Algorithm 22 permutationSubstitution

Input: initial state of the subgraph s_i , final state of the subgraph s_f
Output: the generalised schema automaton
 $permutationState = permutationState(s_i, s_f);$
 $helperState = permutationState.helperState;$
for all $backTransition$ **IN** s_i **do**
 $backTransition.redirectTo(permutationState);$
end for
for all $backTransition$ **IN** s_f **do**
 $backTransition.redirectTo(helperState);$
end for
for all $transition$ **IN** s_f **do**
 $transition.redirectFrom(helperState);$
end for

Having the generalisation steps defined, the issue is how to find the sequence of the steps which yields to the best schema. Here a user can participate in two ways. The first way is that he can select the states and the action without using a heuristic. The second way is that the algorithm helps a user to find good candidates and he will determine whether to do the proposed generalisation steps or not.

The help function should be strong enough to make the decision itself. When the user does not want to accept explicitly all the generalisation steps, the help function can do the steps automatically until no improvement is found.

The help function is based on the sk-ANT heuristic presented in Section 4.1.3 and its modification presented in Section 4.2.2. The function finds the best candidate using the ACO heuristic combined with the sk-strings

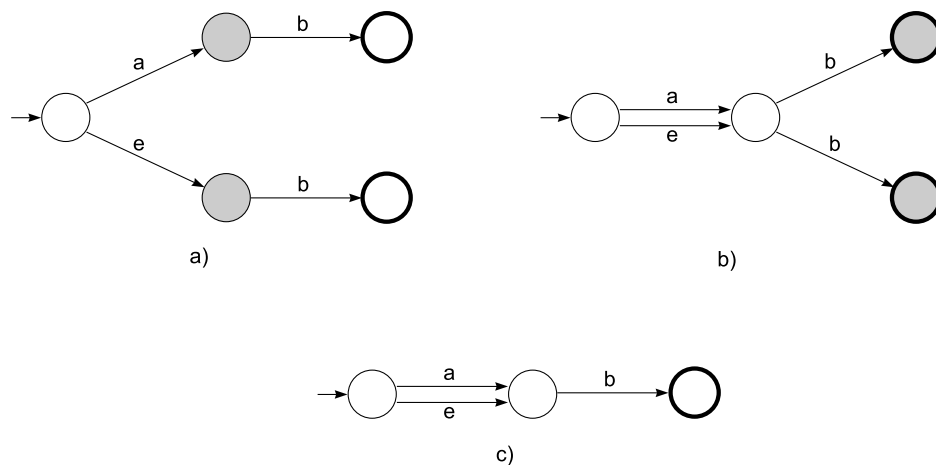


Figure 5.6: States merge example. a) An automaton with two states to merge. b) States merged, but automaton is nondeterministic. The highlighted states must be merged. c) Final automaton.

method, the k-context method and the permutation method as a selector of states and the MDL as a quality measure.

The permutation method can be simplified, when a user must accept the generalisation step. In this case it is sufficient to find the first level candidates. The user then determine if it is the right permutation block and forces the permutation substitution.

The k-context and sk-strings methods provide only candidates for states to merge. The permutation method provides candidates for permutation substitution. The only generalisation step that cannot be determined by the help function is the group substitution.

The group substitution is the perfect example of user interaction advantages. The problem of finding the equivalent subgraphs in two graphs can be a theme for another thesis. But a user can mark such subgraphs easily, especially if he knows what to find.

Inheritance and Validity

When the type C is inherited from the type P , inheritance conditions must be checked. There are two types of inheritance - extension and restriction.

When the type C is extended, the supertype automaton must accept a part of each input string. But this condition may not always be fulfilled. If the condition cannot be satisfied (for example `multiple_mail` extended from `mail`), the supertype automaton can be modified respectively, or the inheritance link can be removed.

When the type C is restricted, input strings of this type must also be accepted by an automaton of the type P . This can be achieved by adding all subtype input strings to a supertype input strings set. And because the generalised schema automaton must accept all the input strings, the supertype automaton also accepts all subtype input strings after the generalisation step. According to the XML Schema specification, a restricted type can be obtained from the supertype only by the occurrence reduction. Thus, the subtype schema automaton is at the beginning a copy of the supertype automaton and only allowed edit operations are such operations that reduces the occurrence. Here, the generalisation does not have any sense.

In both situations the supertype must always been processed before processing of its subtypes so that the inheritance conditions could be checked.

Implementation

Firstly, the schema automaton is created as outlined in Algorithm 23. Given a type T , the schema automaton is initialised. If T is extended, the initial state is created as an extension state and a helper state is added. Then all input strings are added to the schema automaton. The *nextState* method tries to find the proper transition. The extension state here consumes the part of the string that can be accepted by the supertype automaton. If no such part can be accepted, the *createSchemaAutomaton* fails and the type will loose the inheritance mark. The schema automaton will then be created without the extension state.

The proposed algorithm has implemented one optimalization method, that automatically creates the repetition loops if an input string contains a sequence of the length greater than a user defined constant *REPETITION*. This method is placed within the schema automaton construction. This reduces, sometimes significantly, the size of the schema automaton right before the generalisation starts. Note that it is better not to create a loop from all the same letters, but to let one letter to create a transition between two different states. For example in the Figure 5.7, the two schema automatons are shown that are built from strings $\{aaab, aaaab, c\}$ with the *REPETITION* constant set to 3. The first automaton suffers from an *overgeneralisation*, since it accepts also all the strings $\{aaac, aaaac, \dots\}$ which is not desired.

Once the schema automaton is created, the generalisation phase starts. Here, a user can mark two states and merge them, or he can force a help function which finds one solution and presents it to the user. The last option is to force an automatic generalisation step or a chain of maximal possible generalisation steps.

Algorithm 23 createSchemaAutomaton

Input: type T to create schema automaton for

Output: the schema automaton

```
schemaAutomaton = newSchemaAutomaton();
if  $T.isExtended$  then
    schemaAutomaton.initialState = extensionState( $T.parent$ );
    helperState = state();
    schemaAutomaton.initialState.createLambdaTransitionTo(helperState);
else
    schemaAutomaton.initialState = state();
end if
for all  $string$  IN  $T.inputStrings$  do
    actualState = schemaAutomaton.initialState;
    while  $string$  has more letters do
        letter =  $string.next$ ;
        if  $string$  has repetition of letter then
            actualState.createLoopFor(letter);
        else
            nextState = actualState.nextState( $string$ );
            if nextState == null then
                nextState = state();
                actualState.createTransitionTo(nextState, letter);
            end if
            actualState = nextState;
        end if
    end while actualState.final = true;
end for
return schemaAutomaton
```

For the help function the ACO heuristic has been implemented. The ACO heuristic has been described in Section 4.1.3 and the algorithm is outlined in Algorithm 5. But the ant move selector uses two additional criteria to allow a pair of states to be a solution candidate. The k -context criterion is based on the k -contextual method presented by Ahonen in [1]. The k -contextual method says, that two states s_1 and s_2 are identical, if there exist two identical paths of length k terminating in s_1 and s_2 respectively.

The permutation criterion uses the permutation method presented by Vošta in [14]. This method does not produce candidates for merge, but candidates for the permutation substitution.

The MDL method computes a quality of a given solution. It is outlined in Algorithm 24. The automaton description length (ADL) is computed as a number of states plus the number of transitions. The only exception is a permutation state. The length of a permutation state simulates the chain of states and edges from all letters in a given permutation state.

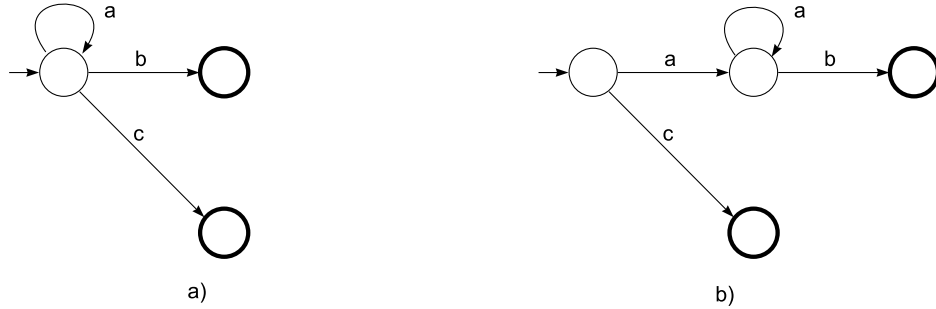


Figure 5.7: Schema automata with repetition loop. a) *overgeneralised* automaton, b) correct automaton.

Algorithm 24 MDL

Input: an automaton A and a set of input strings $strings$

Output: the computed description length

$ADL = 0$

for all $state$ **IN** $A.states$ **do**

if $state$ is a permutation state **then**

$ADL+ = 2 * state.sizeOfPermutation$

else

$ADL+ = 1 + state.transitions.size$

end if

end for

$SDLs = 0;$

for all $string$ **IN** $strings$ **do**

$SDLs+ = computeStringDescriptionLength(string, A);$

end for

return $ADL + (SDLs/strings.size)$

A string s can be represented as a chain of states s_0, s_1, \dots, s_n of the automaton A that are visited during the acceptance process. Note, that s_0 is an initial state and there is a transition from s_0 to s_1 with the first letter of the string s . A code size of the string $size(s) = \sum_{i=1}^n bits(s_i)$ where $bits(s_i)$ denotes a bit count needed to encode a state s_i in the chain. This is $\lceil \log_2(N) \rceil$ where N is a number of transitions of the state s_{i-1} .

5.2.5 XML Schema Inference

Since the schema automaton is generalised, it must be converted to the XML Schema content model definition. As mentioned before, the content model can be represented as a regular expression with one additional operator $\&$ for the *all* particle. Thus, the problem of converting a schema automaton to

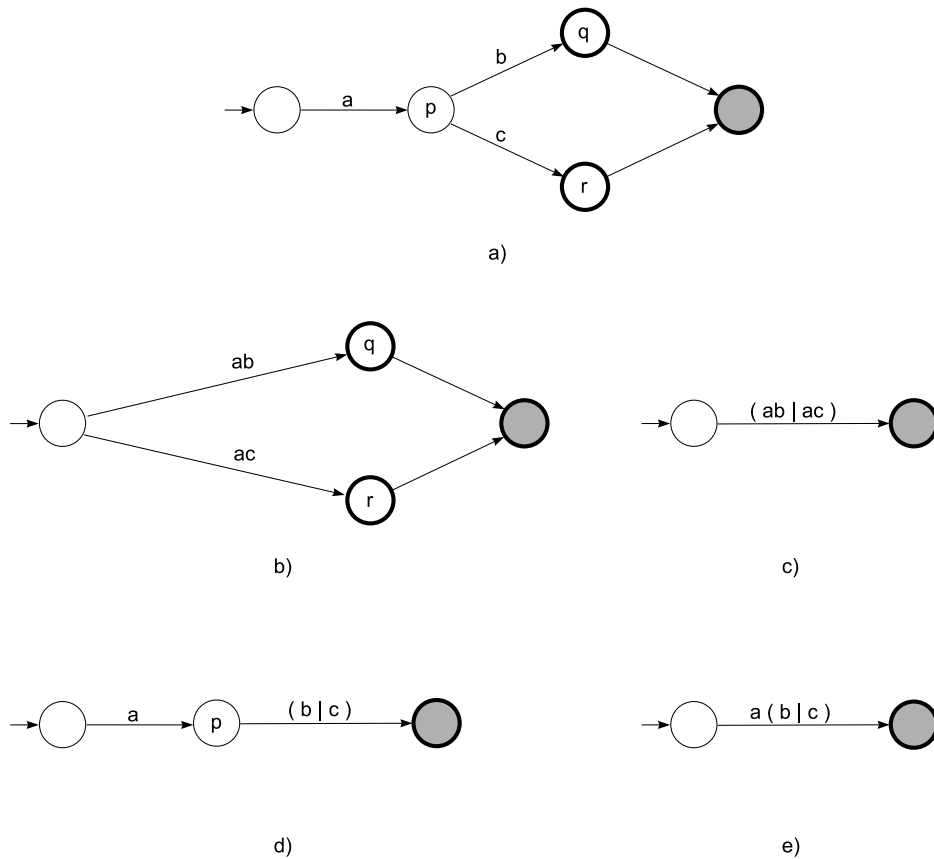


Figure 5.8: An example of two different state removal sequences. a) the initial automaton. The highlighted state is a super-final state b) after remove *p* c) after remove *q,r* d) after remove *q,r* e) after remove *p*

an XML Schema content model can be viewed as a converting a FSA to a regular expression.

A summarisation of techniques for converting a FSA to a regular expression was presented by Neuman in [13]. In this thesis a state removal method is implemented. This method remove states and replaces them and proper transitions with transitions that contain regular expression fragments.

The crucial issue is to select the best order of states to remove. Different removal sequences lead to different regular expressions for the same language. The example is shown in Figure 5.8. One method is proposed by Han and Wood in [5]. They present few heuristics that compute the best state for remove.

The simplest, and also the implemented heuristic orders states by the weight. The weight of a state is a sum of *in-transition* weights plus the sum



Figure 5.9: An example of collapsing transitions. a) an automaton before collapse b) an automaton after collapse

of out-transition weights plus the loop weight. Where a transition weight is a length of regular expression of the given transition. The firstly removed state is the state with the lowest weight. The algorithm is outlined in Algorithm 25. At first all final states must be connected with one super-final state by lambda transitions. Then all states except the initial and the super-final state are removed in the order of the lowest weight. When a state is removed the new transitions are added that represent all combinations of the back-transition, the loop-transition and the front-transition. The appropriate regular expressions are simply concatenated. The *collapseParallelTransitions* function collapses the parallel transitions as shown in Figure 5.9. The appropriate regular expressions are joined by the | operator.

Algorithm 25 SAtoRegex

Input: an automaton *A*

Output: the computed regular expression

```

superFinalState = state()
for all state IN A.finalStates do
    state.createLambdaTransitionTo(superFinalState);
end for
while A.states.size > 2 do
    state = A.getLowestWeight();
    state.collapseParallelTransitions();
    loop = state.getLoop
    for all pair of backTransition and transition do
        regex = createRegexFor(backTransition, loop, transition)
        A.addTransition(backTransition.from, regex, transition.to)
    end for
    state.removeAllTransitions();
    A.removeState(state)
end while
A.initialState.collapseParallelTransitions();
loop = A.initialState.getLoop
transition = A.initialState.getTransitionToSuperFinalState();
return createRegexFor(loop, transition);

```

Chapter 6

Implementation

Schema Builder has been implemented in the JAVA language in the version JDK 6.0. It has been implemented as a swing application, using swing GUI tools. For a XML processing, the built-in Java SAX parser has been used. The Java language has been selected because of the platform independence, the swing is used, because it is more user-friendly than the command line application.

6.1 Architecture

The application is composed of the code itself, third-party packages and the ant build script.

6.1.1 Third Party Packages

Third-party packages are located in the `/lib/` directory. The packages with a common prefix `jung` is used for the automata visualization. The `colt` library, the `concurrent` library and the Apache `commons libraries` are used by the JUNG visualization tool [8]. The `log4j` library is used for logging.

6.1.2 Code Structure

The source code is located in the `/src/` directory and is divided into packages. The package structure corresponds to the directory structure in the filesystem. Two main packages splits the source code to the two logical blocks.

The `cz.cuni.mff.schemabuilder.swing` package contains the GUI definition. It is a frontend of the application.

On the other hand, the `cz.cuni.mff.schemabuilder.core` package contains the core of the application. It contains all data structure definitions, the core

algorithms and all the code that is not bounded to the GUI. It represents the backend of the application.

The `core` package is further divided into the following subpackages:

automaton contains a schema automaton data structure definition and contains all methods that manipulate with the structure.

builder contains class `SchemaBuilder`, the class that directs the whole computation. The subpackage `algorithms` contains implemented third-party algorithms.

document contains a XML document data structure and the parser that builds it.

exception contains exception definitions

util contains some useful universal classes

XMLSchema contains a XML Schema data structure

6.2 Implemented Fragments

SchemaBuilder application is an implementation of Schema Builder as presented in Chapter 5. However, there are parts that have not been implemented yet.

The permutation criterion in ACO heuristic determines only *first-level candidates*. A user then decide whether to substitute or not. Finding *second-level candidates* is not implemented and, thus, permutation method is not available for the automatic generalisation.

Besides the automatic permutation substitution, the group substitution is also unimplemented yet. These two fragments have been left for the future work.

6.3 Building and Executing

For building the executable jar file, the *ant* tool is used. The version 1.7 or later is required. To build the jar, execute a command `$>ant build.xml`. The generated jar file will be stored in the subdirectory `/build/`, that already contains the pre-built executable jar file `SchemaBuilder.jar`. The jar can be executed by the command `$>java -jar SchemaBuilder.jar`. Note, that at least JRE 6.0 is required.

Appendix B contains a simple user guide with screenshots and examples.

Chapter 7

Experimental Results

In this chapter, experimental results of SchemaBuilder application are presented. Sets of documents have been provided as an input to SchemaBuilder and the results have been analysed and compared with the expected schema. The documents can be divided according to their origin or complexity. The documents can be split into real-world XML documents and documents synthetic, created to cover all features of the application. The documents have various structure and complexity. Four sets of documents have been tested, which sufficiently cover the abilities of SchemaBuilder. Documents have been firstly passed through the application without any help from a user, and the resulting XML Schema has been stored as a `test1.xsd`. After it, the same documents have been processed using user interaction. The result has been stored as a `test2.xsd`.

7.1 Testing Sets

The following sets of documents have been tested. The tested documents with inferred XSDs are placed on CD in the directory `/data/`.

Set 1

Documents in this set have a sort of *table structure*. In such document, the root element contains an arbitrary number of a single element with a sequential structure. SchemaBuilder without user interaction infers XML Schema from this set of documents correctly. The only difference is setting the root type to a^+ instead of a^* , when it is possible. Actually, this can be viewed not as a mistake, but as a useful feature. However, a user can simply edit the proper schema automaton to get the a^* content model.

Set 2

DTD describing this set uses all possible constructs. It uses a choice operator, all occurrence operators and it uses various combinations of the operators. Having the set of documents, SchemaBuilder without user interaction suffers from the following behaviour. It does not join some type clusters, that should be joined and, in some cases, it infers less generalised schema automata than it should. It is partially caused by the insufficient input examples. Having a complex automaton, the unoptimal schema inference phase make the content model big, not readable and in many cases nondeterministic.

User interaction helps to merge type clusters properly and also schema automata are, if not optimal, then closer to the expected content model. But even optimal automata are sometimes transformed to non-user-friendly content model.

Set 3

DTD describing this set does not contain a choice operator, but often contains * operator. The DTD also contains two types, on which inheritance can be applied. SchemaBuilder behaves similar as on the Set 2. Some type clusters are leaved unmerged, some content models are wrongly inferred. The biggest problem is that it cannot infer a content model of the type $a^*b^*c^*d^*$. The content model usually consists of many choice particles nested within each other.

As on the Set 2, the user helps to merge type clusters properly. The user can also mark inheritance and make more concise content models. The size of the generated XSD is reduced to $\frac{1}{3}$.

Set 4

It is a set of synthetic XML documents and contains constructs such as more types for the same element name, one type for different element names, permutation particle and inheritance. SchemaBuilder without any user interaction suffers from the less generalised schema and it cannot recognise the all particle and inheritance. User interaction helps to obtain the expected schema.

7.1.1 Conclusion

User interaction helps mainly in places, where input examples are poor and do not cover whole possibilities. It also helps a lot in the phase of merging type clusters and it is the only way of how to infer inheritance. However,

the results are still not optimal. The biggest problem is when elements are optional. For example when a content model should be of the form ab^*c^* , SchemaBuilder infers a content model like $((ab + |a)|ac+)|ab + c+$. This is caused by the implementation of the XML Schema inference phase and by the fact, that such content models are hardly represented by automata without lambda edges. The next problem is that expressions like $a+$ are sometimes defined as aa^* .

7.2 Types Inference

One of the interesting parts of the computation is, how type clusters are merged. The clusters are compared mutually and they are merged or not according to the tree edit distance (TED). Table 7.2 shows, how user interaction can decrease the number of inferred types. When user interaction (UI) is not used a *BUILDERS_DECISION* merge type is selected. Thus, two types are merged, if the TED is under layer 2. It has been found, that if the layer 2 parameter is set to 10, almost all clusters, that should be merged and no clusters, that should not be merged, are really merged. Thus, the parameter is set to 10 for each test. When UI is used, *TED_TWO_LAYERS* merge type is selected. Thus, a user must decide, whether to merge clusters or not, if TED is under layer 1 but over layer 2. If TED is under layer 2, clusters are merged automatically. Thus, layer 1 has been looked for, that makes the set of inferred types to be optimal. In the proposed tests, where user decisions have been made, the results are optimal. The only exception is Set 1, where the optimal set has been inferred without user interaction.

Set	UI	Layer 1	Decisions	Number of types		
				DTD	Before	After
Set 1	no	0	0	32/7	32	8/7
Set 2	no	0	0	28	48	17/16
Set 2	yes	70	14	28	48	13/12
Set 3	no	0	0	23/11	33	9/8
Set 3	yes	70	5	23/11	33	8/7
Set 4	no	0	0	8	25	10/9
Set 4	yes	40	10	8	25	9/8

Table 7.1: Number of inferred types according to the initial parameters

When the number of types is marked as 32/7, it means, that there are 32 types in common and only 7 types have defined complex content (i.e. an

element may contain subelements). Note, that in every tests SchemaBuilder infers exactly one type, that has simple content. Elements in this type have assigned type `xs:string`.

Chapter 8

Conclusion

The aim of this thesis has been finding new possibilities in the automatic construction of an XML schema for a given set of XML documents. Firstly the existing solutions have been analysed and some of them described. It has been found, that no analysed solution focuses on user interaction in a process of schema inference. Thus this thesis has focused on how user interaction can help in the process of XML schema inference.

This thesis contains a discussion of possible ways of user interaction and also an experimental implementation. As a schema description language, XML Schema has been used, because it contains constructs, that cannot be easily determined by automatic computation, but a user can recognise them.

As a default algorithm, Schema Miner [14] (see Section 4.2.2) has been used, because it is heuristic method, which best suits with user interaction. The type inference phase has been improved by allowing the same type for different element names and by allowing a user to determine whether to merge type clusters or not. After the type inference phase, the user can mark inheritance between types. Inheritance is a construction, which has not been considered yet in any of the analysed solutions. In the phase of schema generalisation, the user can mark states to merge or can request for a help. The algorithm can in addition merge states automatically. In the final phase, XML Schema is inferred from a given set of type clusters.

Thanks to user interaction, the proposed method gives more concise and user-friendly XSDs than the method without user interaction. Contrary to Schema Miner, the proposed algorithm allows to define an inheritance between types and allows to define the same type for elements with different names.

8.1 Future Work

The general aim has been fulfilled. The discussion outlines the possible ways of how a user can help with the schema inference. The experimental implementation contains almost every algorithm outlined in the discussion. However, there are several solutions, that have not been implemented or that are worth improving.

The groups extraction is one of them. The user can mark a subautomaton and replaces it with an auxiliary *group state*. The subautomaton is then represented as a globally defined group, which the auxiliary state references. An implementation of this feature has been left for the future work.

The XML Schema inference phase constructs a regular expression for a given schema automaton using the state removal method. This algorithm is based on the heuristic, which selects the next state to remove. The used weight heuristic provides better and more concise regular expression than the random selection. However, it does not always produce a deterministic regular expression, especially for more complicated automata. Thus, implementing better algorithm with preventing nondeterministic regular expressions should be an issue for the future work.

There are XML Schema constructs that are not covered by the proposed method. For example substitution groups or integrity constraints. The inheritance is also reduced to identifying inheritance between two existing types. The improvement would be to define one abstract supertype for a set of subtypes. Together with the substitution groups it is a useful XML Schema construct. Finally, the only simple type inferred by the proposed algorithm is `xs:string`. Identifying simple types is also left for the future work.

Bibliography

- [1] Helena Ahonen. *Generating Grammars for Structured Documents Using Grammatical Inference Methods*. PhD thesis, University of Helsinki, Department of Computer Science, Helsinki, Finland, November 1996.
- [2] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from xml data. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 825–834, New York, NY, USA, 2008. ACM.
- [3] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring xml schema definitions from xml data. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 998–1009. VLDB Endowment, 2007.
- [4] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Xtract: a system for extracting document type descriptors from xml documents. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 165–176, New York, NY, USA, 2000. ACM.
- [5] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.
- [6] John E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, November 2000.
- [7] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [8] Tom Nelson Joshua O'Madadhain, Danyel Fisher. Jung graph visualization tool. <http://jung.sourceforge.net/>, 2008.

- [9] MURATA Makoto. Relax ng home page. <http://www.relaxng.org/>, 2009.
- [10] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The xml web: a first study. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 500–510, New York, NY, USA, 2003. ACM.
- [11] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keong Ng. Dtd-miner: A tool for mining dtd from xml documents. In *WECWIS '00: Proceedings of the Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000)*, page 144, Washington, DC, USA, 2000. IEEE Computer Society.
- [12] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keong Ng. Re-engineering structures from web documents. In *DL '00: Proceedings of the fifth ACM conference on Digital libraries*, pages 67–76, New York, NY, USA, 2000. ACM.
- [13] Christoph Neumann. Converting deterministic finite automata to regular expressions. 2005.
- [14] Ondřej Vošta. Automatická konstrukce schématu pro množinu xml dokumentů. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Software Engineering, Prague, Czech Republic, 2005.
- [15] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an xsd. In *Proceedings of the 13th International Conference on Database Systems for Advance Applications*, pages 35–50, New Delhi, India, 2008. Springer Berlin / Heidelberg.
- [16] W3C. World wide web consortium. <http://www.w3.org/>, 1994-2008.
- [17] W3C. Xml schema. <http://www.w3.org/XML/Schema>, 2004.
- [18] W3C. Xml schema part 1: Structures second edition. <http://www.w3.org/TR/xmlschema-1/>, 2004.
- [19] W3C. Xml schema part 2: Datatypes second edition. <http://www.w3.org/TR/xmlschema-2/>, 2004.
- [20] W3C. Extensible markup language (xml) 1.0 (fifth edition). <http://www.w3.org/TR/2008/REC-xml-20081126/>, 2008.

- [21] W3C. On sgml and html. <http://www.w3.org/TR/REC-html40/intro/sgmltut.html>, 2009.
- [22] Raymond K. Wong and Jason Sankey. On structural inference for xml data. *Technical Report UNSW-CSE-TR-0313*, 2003.

Appendix A

Content of CD

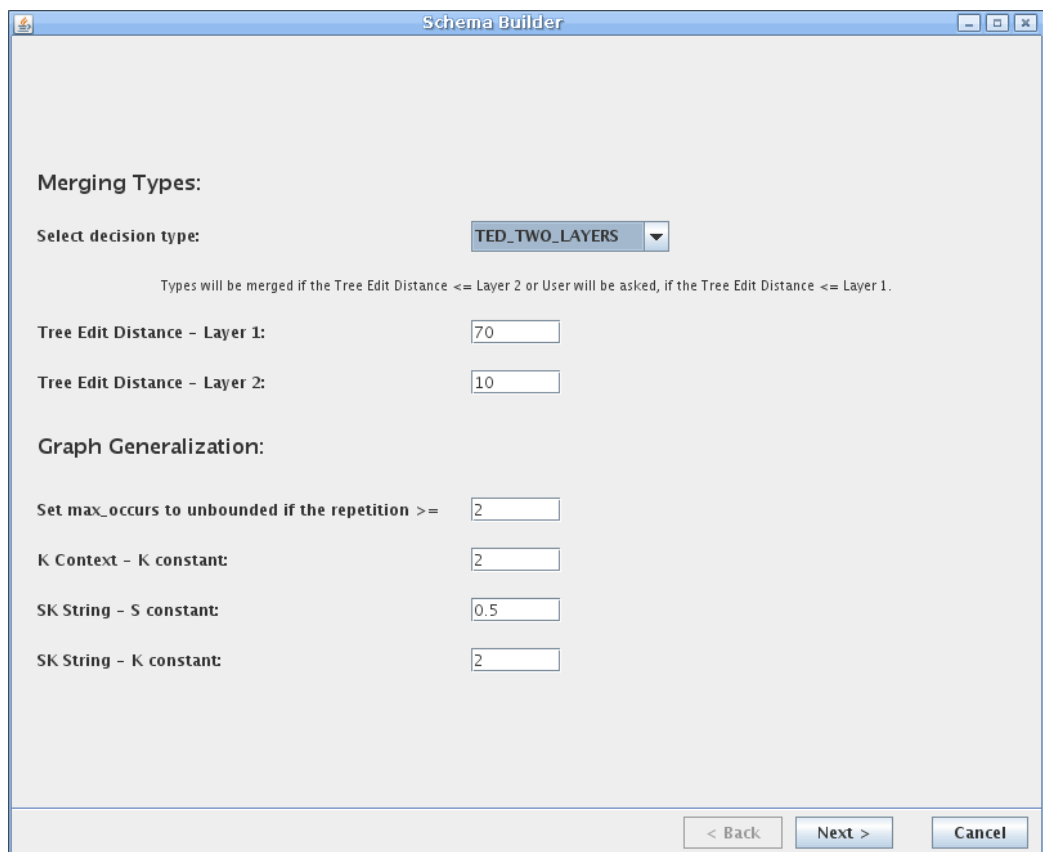
Part of this thesis is the enclosed CD. CD contains source codes of the experimental implementation together with a documentation and built application. It also contains experimental data and results. Finally, the text of this thesis is present. CD has the following structure:

- `content.txt` A file with this text
- `thesis.pdf` A PDF file with the text of this thesis
- `/impl/` Source codes and libraries of the experimental implementation
- `/javadoc/` A generated *javadoc* documentation.
- `/data/` Experimental data and results.

Appendix B

SchemaBuilder Guide

Execute the application by the command `$>java -jar SchemaBuilder.jar`. The following window will be displayed.

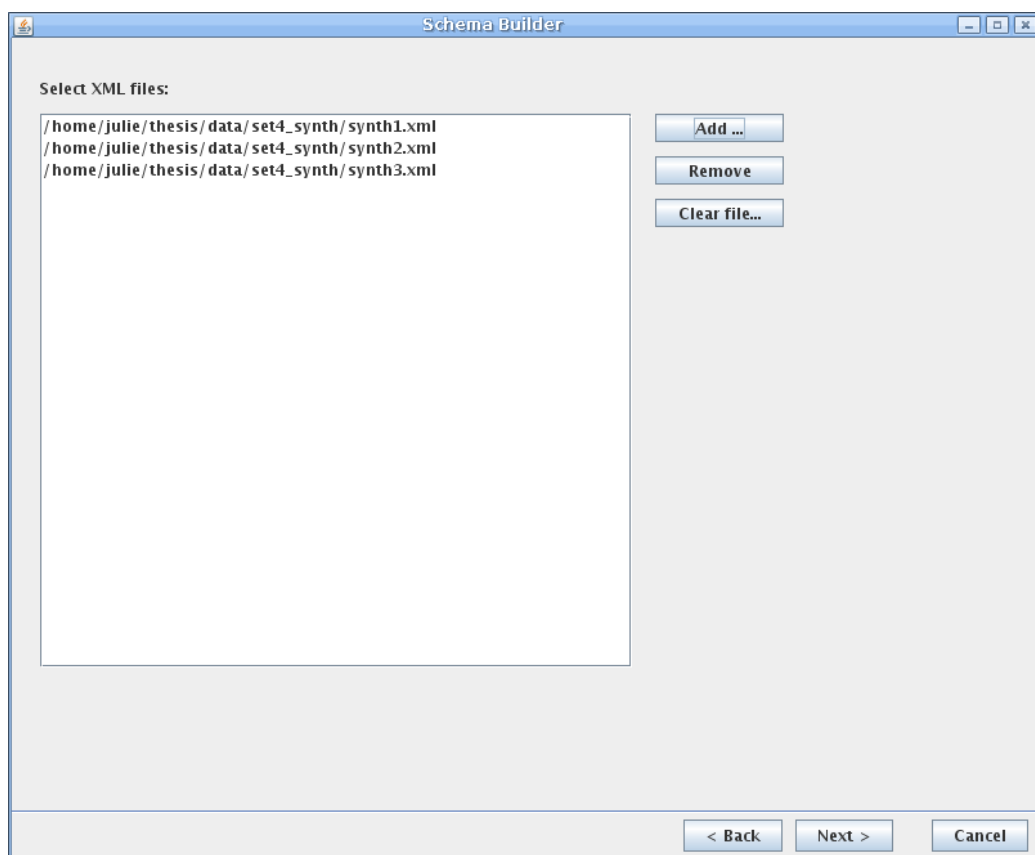


The screenshot shows a window titled "Schema Builder" with the following configuration options:

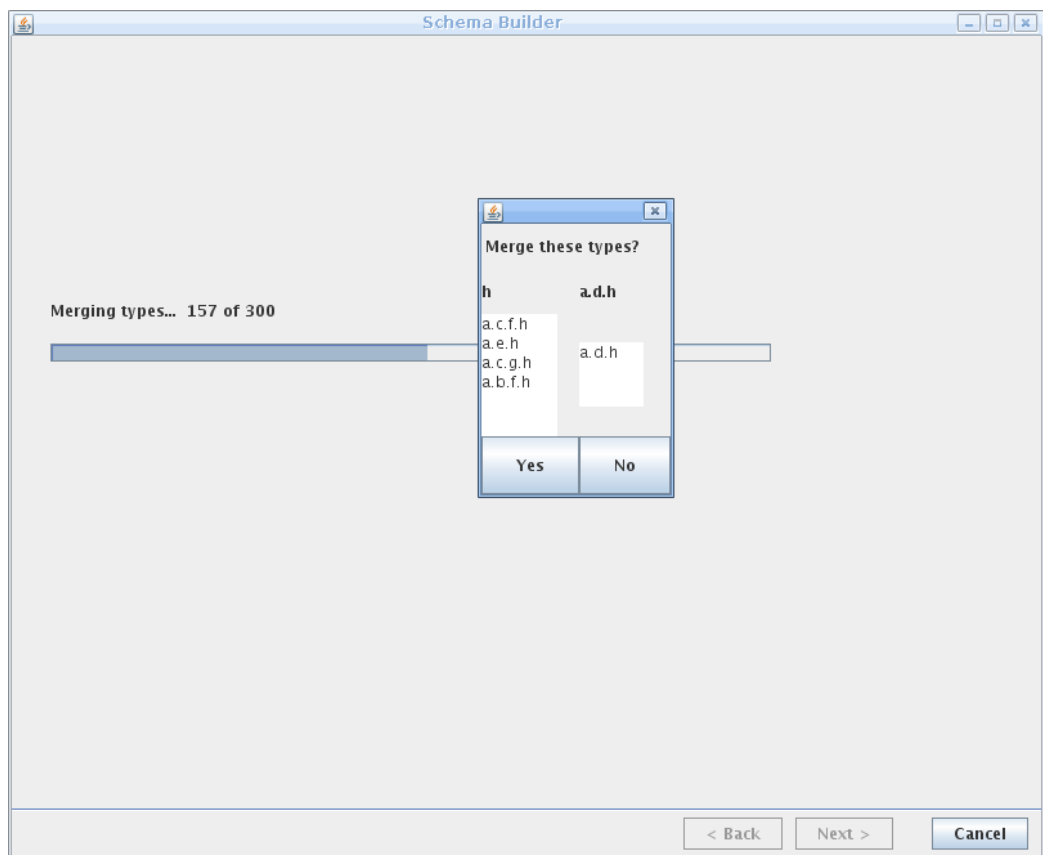
- Merging Types:**
 - Select decision type: **TED_TWO_LAYERS** (dropdown menu)
 - Types will be merged if the Tree Edit Distance \leq Layer 2 or User will be asked, if the Tree Edit Distance \leq Layer 1.
 - Tree Edit Distance - Layer 1:
 - Tree Edit Distance - Layer 2:
- Graph Generalization:**
 - Set max_occurs to unbounded if the repetition \geq :
 - K Context - K constant:
 - SK String - S constant:
 - SK String - K constant:

At the bottom right, there are three buttons: "< Back", "Next >", and "Cancel".

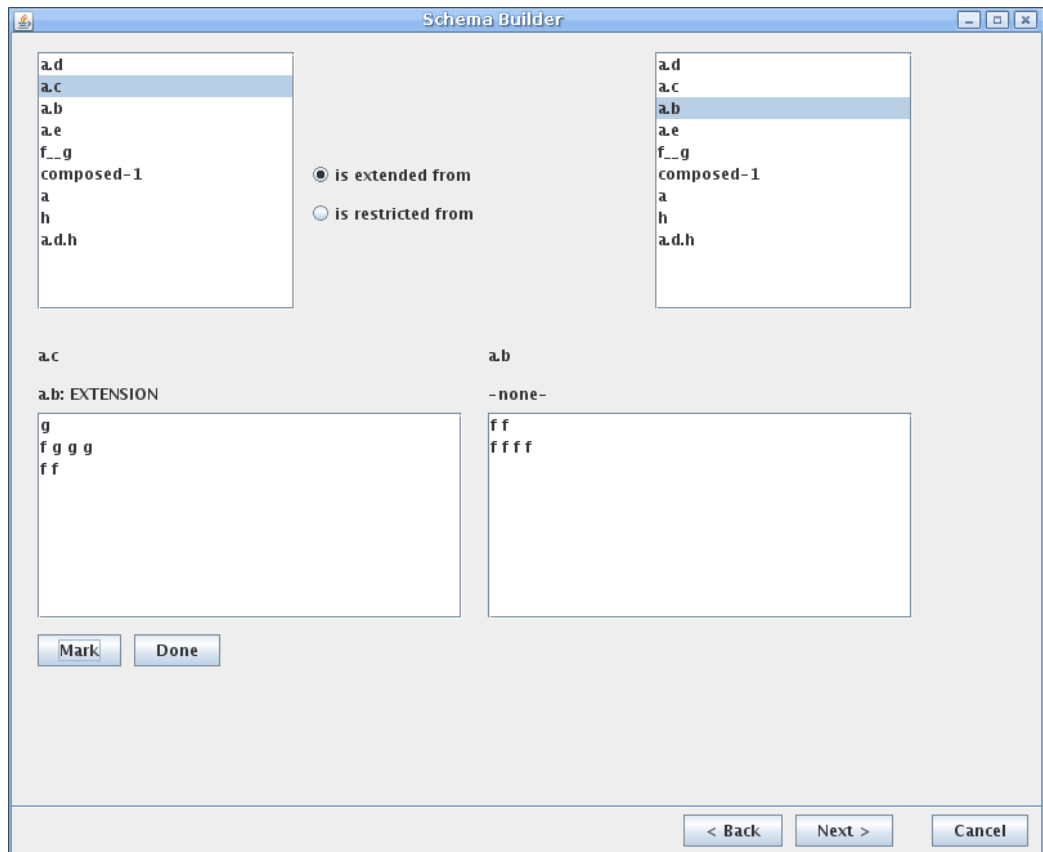
Set the application parameters here, then click on "Next". On the next window choose XML files.



The selected files will be used for the inference algorithm. Click on “Next”, progress bars will be displayed.

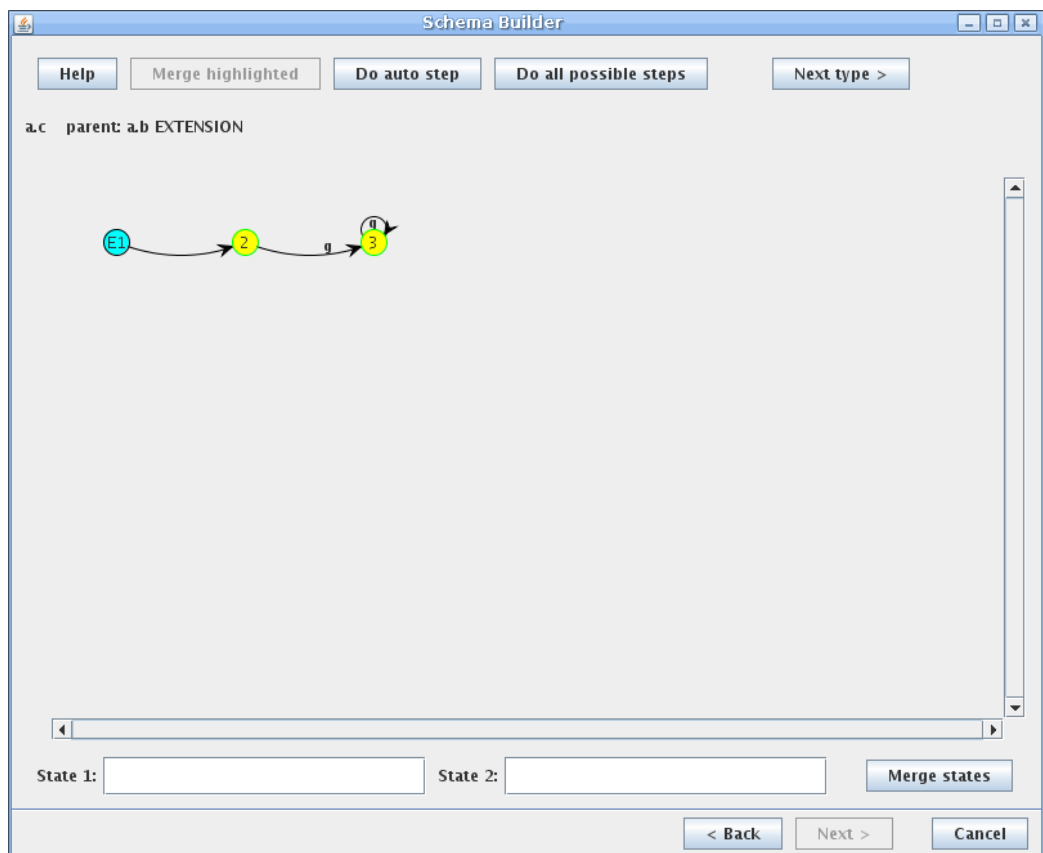


When you choose an interaction during the merge types phase, a **modal dialog** may be displayed. The type is identified by its name in bold. In the white frame a set of elements with paths, that belongs to the type, are displayed. Click on “Yes” to merge the types, click on “No” not to merge the types. After the merge types phase the following window will be displayed.



This is an **inheritance panel**. On the top of the panel two type lists are displayed. The lists contain all types inferred by the previous phase. Below a type list, a name and a parent of the selected type is displayed. Below it a set of strings, that represents the type is displayed.

You can mark inheritance between types. Select a subtype in the left type list. Select inheritance type by the option between the two type lists. Select a supertype in the right type list. Click on “Mark” to mark the inheritance. When you are done, click on “Done” or “Next”. The following window will be displayed.



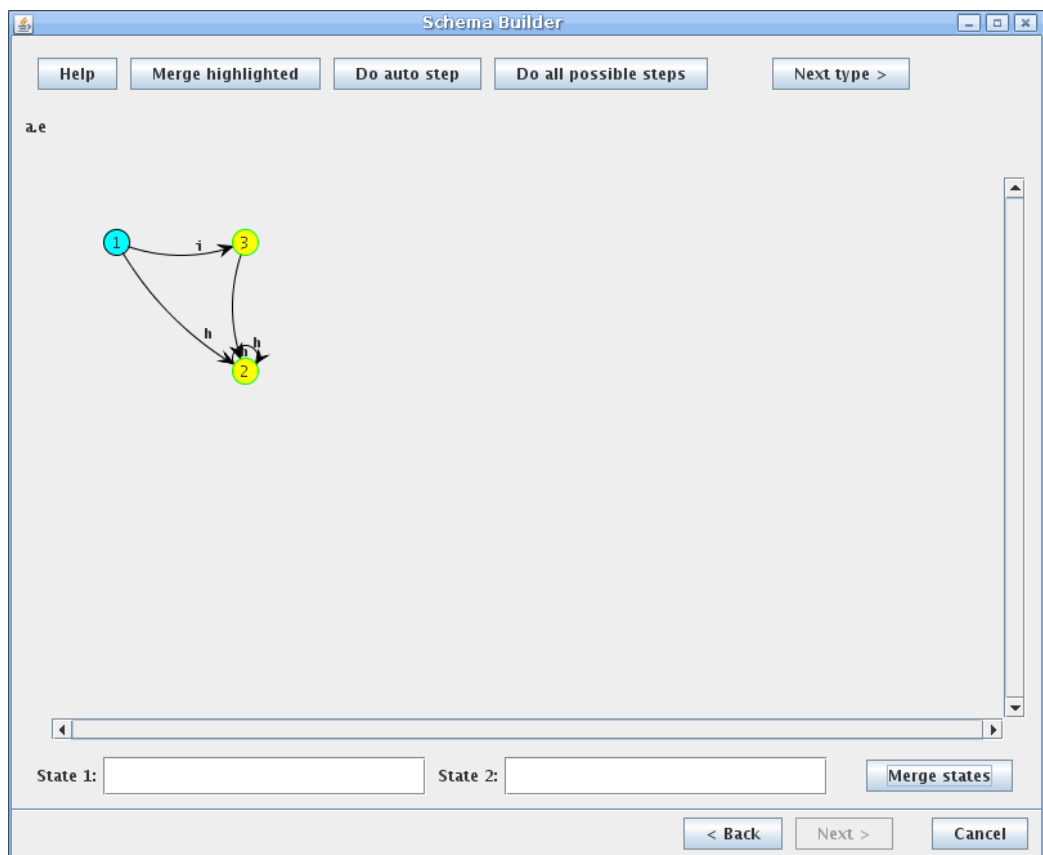
This is a **type panel**. Here a schema automaton of a type is displayed. You can merge states of the schema automaton to generalise and simplify it.

On the top of the panel buttons are located. Use “Help” button if you want to sign the best candidates to merge. If you want to merge the highlighted candidates, click on “Merge highlighted” button. Use “Do auto step” if you want the application to merge one pair of the best candidates. Use “Do all possible steps” if you want the application to merge pairs of the best candidates unless there is no improvement.

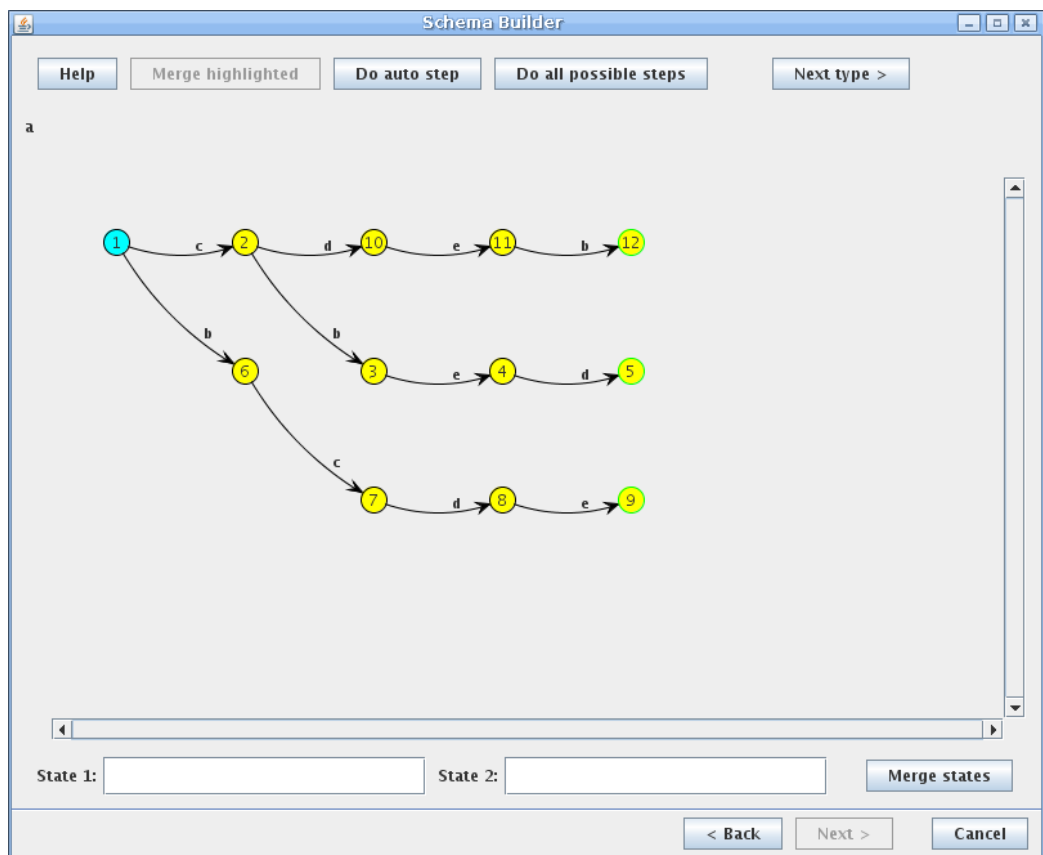
Below the buttons a name and a parent of the actual type is displayed. Below it a schema automaton visualisation is placed. States are labelled with their IDs, transitions are labelled with element names. An initial state is filled by cyan and highlighted states are filled by red. Final states are drawn by green. Extension states are signed by E , permutation states are signed by P .



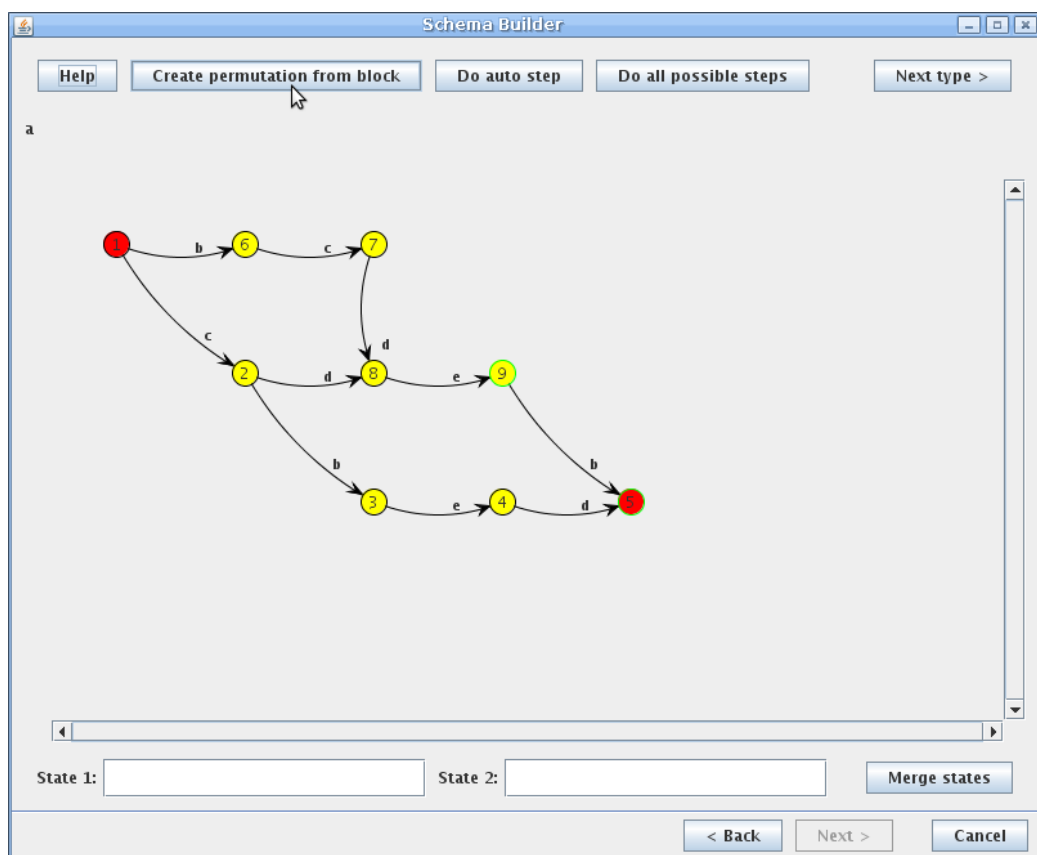
Below the schema automaton, two fields for two state IDs and one action button are located. Fill the fields with the two IDs of states, which you want to merge. Then click on “Merge states”. The two states will be merged.



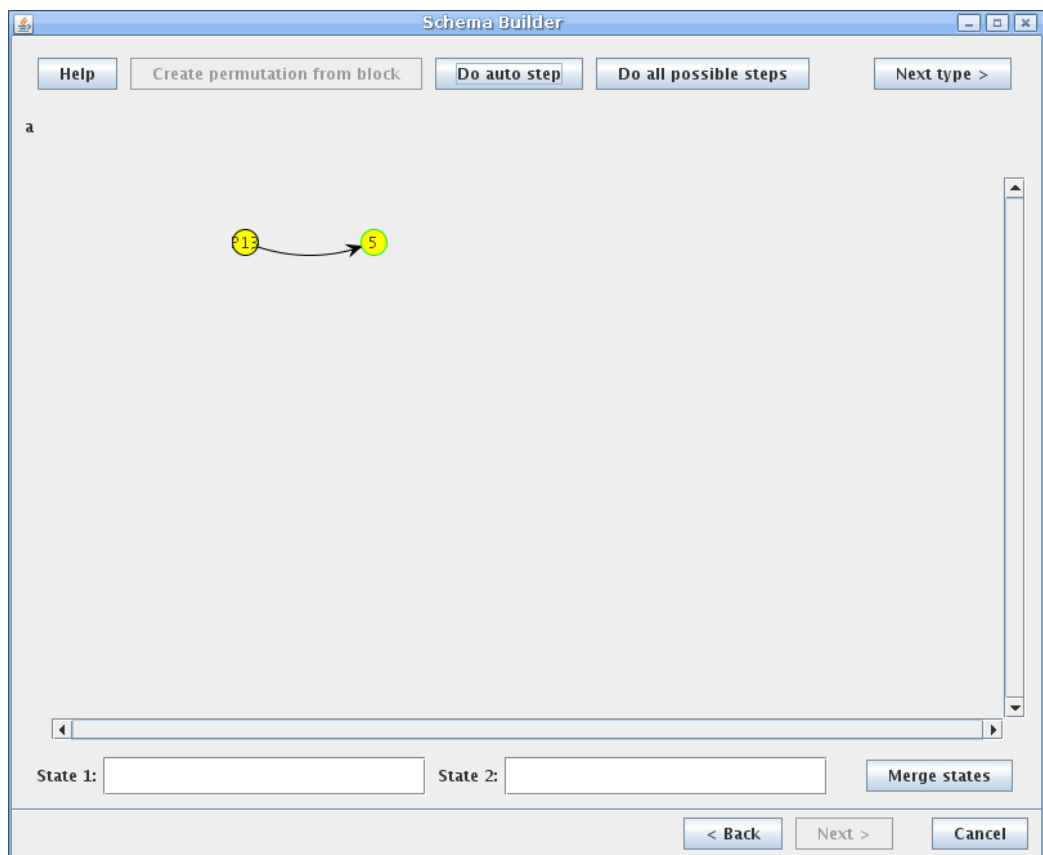
When schema automaton is generalised sufficiently, click on “Next Type”. The type panel will show the schema automaton of the next type.



This automaton is a candidate for **permutation**. Doing automatic steps you can get an automaton displayed in the following figure.



Clicking on “Help” button an application finds the permutation block and highlights the start and end state. You can click on “Create permutation from block” button. The result is shown in the following figure.



Now the automaton is generalised. You can click on “Next type” button. When are no types left, the XML Schema document will be generated.



You can save the generated schema by clicking on “Save schema” button.