Algorithmic learning of XML Schema definitions from XML data

Kore Nordmann

March 24, 2011

Abstract

XML currently is the main syntax for data exchange. It is the base syntax for nearly all data exchange happening on the internet, which, for example, includes XHTML and SOAP. Besides these well-specified formats there are also a lot protocols and interfaces available, which use XML as a syntax, but not have any specified schema. If a schema exists, this facilitates automation, optimization of search, integration, translation and processing of XML data. [3, 9, 11, 13, 17, 22, 23, 26, 30, 38] It is therefore desired to provide means to algorithmically inference schemas for existing XML data. This work outlines the existing algorithms, which can be used for schema inference, extends those algorithms based on an analysis of real-world XML Schema definitions and evaluates those algorithms based on example data extracted from real-world XML data.

Contents

Introduction

XML currently is the main syntax for data exchange. It is the base syntax for nearly all data exchange happening on the internet, which includes XHTML and SOAP. Besides these well-specified formats there are also a lot protocols and interfaces available, which use XML as a syntax, but not have any specified schema. If a schema exists, this facilitates automation, optimization of search, integration, translation and processing of XML data. [3, 9, 11, 13, 17, 22, 23, 26, 30, 38] It is therefore desired to provide means to algorithmically inference schemas for existing XML data.

Earlier approaches like "Inference of concise DTDs from XML data" [6] implement algorithms to inference Document Type Definitions from sets of XML documents. Since 2001 [34] the W3C worked on a new schema language, XML Schema, which exceeds the power of Document Type Definitions [36] and itself is XML. Extending the Document Type Definition inferencing algorithms to the contextual power of XML Schema definitions is therefore desired.

Inferencing Document Type Definitions requires to solve mainly one problem, to inference concise regular expressions from a set of words. The algorithms Rewrite and Crx described in "Inference of concise DTDs from XML data" are able to learn regular expressions for Document Type Definitions and are discussed and extended in this work.

This work can also be used for inferencing XML Schema definitions, as shown in "Inferring XML Schema Definitions from XML Data" [7], since the regular expressions available in XML Schema form a superset of the regular expressions available in Document Type Definitions. The additionally available syntactical features in XML Schema, which are not used by the algorithms mentioned before are *Counting patterns* and the `<all>` syntax for sets of children with irrelevant order.

The main difference between Document Type Definitions and XML Schema definitions is the fact, that Document Type Definitions only allow to define one type for each element label. This means, that each label has exactly one regular expression associated, which defines the children which may occur below the given element. XML Schema allows to define types of elements based on the element name or the ancestors of an element. Also the same type may be used for multiple elements with different labels.

This structural difference makes it harder to learn concise XML Schema definitions. In "Inferring XML schema definitions from XML data" [7] the authors show an approach of learning k-local XML Schema definitions. This approach results in far too many types, which requires merging of types. The aim is to reduce a set of types with

similar or same regular expressions into one single XML Schema type definition. The aforementioned paper introduces a simple distance metric on the inferenced child patterns and merges them based on that. Different extensions for this will be discussed and evaluated in this paper.

In this thesis the types not only learned k-local, but based on the full ancestor path, and experiment with different approaches for merging the types, based on the analysis of real-world XML Schema definitions. Based on real-world data training and verification data was extracted to evaluate the algorithms.

Part I.

State of the art

Overview

This part outlines the current state of the art in schema inferencing from XML data. Since no formal metric exists, which is able to judge the quality of inferenced schemas there are usually two steps involved in developing algorithms for schema inference:

- Analysis of existing schemas

  Analysing existing real-world schemas provides insight about the used schema language features and the structure of schemas. This knowledge can be reused when developing new algorithms for schema inference.

- Developing algorithms

  Developing new algorithms for schema inference.

The following two chapters start with describing the current state of schema analysis which is relevant to this work and the algorithms described in the following chapter. The algorithms described build the base for the work in the next part of this work.

Chapter 1.

Schema analysis

The problem setting of schema inference implies that schemas can only be learned from positive data only. Gold [20] showed that it is not possible to learn the class of all regular expressions from positive examples only. Thus it is necessary to find out about common characteristics of schemas, which are already used, to develop algorithms which can learn meaningful subsets of the target schema language.

For this the schema languages need to be structurally analyzed and the schemas instances already used can be statistically analyzed to find out about the used features. In chapter II a custom analysis is presented, which bases on already existing analysis presented in this chapter.

1.1. Schema usage

Barbosa et al. [2, 28] showed that only about half of the XML documents they found on the web are referring to a schema at all. Since the other XML documents do not have any defined structure no real statements on their structure and complexity can be made.

In another study Bex et al. [5, 27] showed that about two-thirds of the schemas found on the web and in schema repositories are not valid according to the W3C XML Schema specification [36]. This renders those schemas mostly useless for their application, but it might still be possible to analyze them structurally, as shown in chapter II. A similar observation was made by Sahuguet [1] regarding Document Type Definitions.

1.2. Schema structure

XML Schema and Document Type Definition both use regular expression to define the content models or child patterns for elements (see section 2.1 for details). Analysing the used regular expressions is one of the most important parts.

1.2.1. Regular expressions

The structure of regular expressions in schemas has for example been analyzed by Bex et al. [5]. In this study 109 Document Type Definitions and 93 XML Schema definitions were downloaded from the XML Cover pages [10] and analyzed.

They found out that the vast majority of the regular expressions found in the schemas are simple, i.e. 92% in Document Type Definitions and 97% in XML Schema definitions, where a simple regular expression is defined as:

Definition 1:
A base symbol is a regular expression a, $a?$, or a^* where $a \in \Sigma$; a factor is of the form e, e^*, or $e?$ where e is a disjunction of base symbols. A ***simple regular expression*** *is ϵ, $\emptyset$, or a sequence of factors.*

This definition matches the definition of CHAREs in structure, which are discussed later in this work in subsection 1.2.2. The authors claim in a later paper [6] that 99% of the analyzed expressions are indeed CHAREs, which also means, that every element in the regular expression only occurs once.

The occurrence counts of elements in regular expressions are important, since regular expressions with only one occurrence for each element are possible to learn from positive examples only. [6] Another analysis [12] based on a larger dataset supported the numbers by finding out that 97.57% of the expressions were **simple**. In both papers these statistics include `PCDATA`, `ANY` and `EMPTY` regular expressions.

Most of the existing analysis focussed on Document Type Definition schema properties. The analysis in "PG 530: Pattern Based Schema Languages" [12] included some XML Schema specific aspects, like counting the occurrence of the `<all>` regular expression syntax, which occurred in 6.52% of all analyzed schemas. No paper could be found which analyzed the usage of **counting patterns** in XML Schema regular expressions.

XML Schema makes it possible to not only use ?, $*$ and $+$ as quantifiers, but also provide custom **counting patterns** as quantifiers, so that $a2,5$ would mean, that the element a may occur from 2 to 5 times. Just like `<all>` this can also be expressed using the common syntax but it would be interesting to see how broadly it is used in real XML Schema definitions. **Counting patterns** in XML Schema can be assigned not only to elements, but also to sequences and choices by using the `minOccurs` and `maxOccurs` attributes with values other then 0, 1 and `unbounded`.

1.2.2. XML Schema type properties

XML Schema employs a different typing mechanism then Document Type Definition, which makes it possible to associate different content models with elements of the same name, based on the ancestors of the element. This is discussed in more detail in 4.1.

Bex et al. [5] analyzed this based on their small set of only 30 XML Schema definitions and found out that only 5 of those schemas (15%) used the XML Schema typing mechanism – all other schemas could as well be expressed by Document Type Definitions. The type for the element depended on the direct parent in all of these cases.

In "PG 530: Pattern Based Schema Languages" [12] the same analysis was performed on a much larger input set of XML Schema definitions which showed that 22.48% of the XML Schema definitions used the XML Schema typing mechanism.

All the mentioned papers performed additional analysis with different focal points. The results mentioned here are relevant to this work and summarizing them all would exceed the scope of this work.

Chapter 2.

Schema learning algorithms

The early work in this area focussed on learning Document Type Definitions, which reduces to learning concise regular expression from positive example strings. [6] The paper "Inference of Concise DTDs from XML Data" [6] presents three algorithms for learning CHAREs and SOREs.

Since Document Type Definitions can be abstracted by context-free grammars with regular expressions (RE) on the right side, learning Document Type Definitions reduces to learning of REs describing all strings occurring below that element name in the XML corpus. Like Gold [20] shows the class of all REs cannot be learned from positive examples only. The challenge was to identify subclasses of RE which can be learned from positive examples only, are deterministic RE, which is required for Document Type Definition, and are concise.

"Inference of Concise DTDs from XML Data" identifies two such classes of regular expressions:

- The class of **Single Occurrence Regular Expressions** SORE, which are regular expressions, where every element name occurs only once.

- The class of **Chain Regular Expressions** CHARE, which are SOREs, which contain of a sequence of factors $f_1..f_n$, where each factor is a non-empty choice over element names, optionally with a multiplier. The factors follow the form $(a_1 + .. + a_k)?$, where a_i is an element name, $k \geq 1$ and the ? optionally one of the available multipliers ?, $*$, or $+$.

Since each element name only occurs once in each SORE by definition, each SORE and therefore also each CHARE is obviously deterministic (one-unambiguous) [6, 8], as required for Document Type Definitions.

As metioned in section 1.1 92% in Document Type Definitions and 97% in XML Schema definitions are **simple regular expression** and the in [6] the authors claim that 99% of the analyzed expressions are CHAREs.

The two algorithms *REWRITE* and *CRX* described in more detail in this chapter will be used and extended for the use with XML Schema definitions in this work.

2.1. Inferring Single Occurrence Regular Expressions

This section describes the *REWRITE* algorithm as presented in "Inference of Concise DTDs from XML Data" [6].

2.1.1. Definitions

For this chapter Σ is a finite set of symbols, the elements names of the schema. Every $a \in \Sigma$ is a regular expression, as well as rr', $r + r'$, $r?$, $r*$ and $r+$ are regular expressions, if r and r' are regular expressions. The language defined by a regular expression is denoted as $L(r)$.

By $RE(\Sigma)$ the class of all regular expressions over Σ is denoted. Document Type Definitions are abstracted as a mapping from Σ symbols to regular expressions over Σ:

Definition 2:
A Document Type Definition is a pair (d, s) where d is a function that maps Σ-symbols to regular expressions over Σ ($RE(\Sigma)$), and $s \in \Sigma$ is the start symbol. A tree satisfies the Document Type Definition if its root is labeled by s for every node u with label a, the sequence $a_1..a_n$ of labels of its children matches the regular expression $d(a)$.

The regular expression $d(a)$ is also referred to as the *element definition* or the *content model* of a. The Document Type Definition specification additionally requires the regular expressions to be deterministic, which can be ignored here, since the presented algorithms are limited to SOREs, which are always deterministic as mentioned earlier.

For the presented algorithms an automaton is used, which deviate from the usual automaton definition and is defined as:

Definition 3:
*For a set S, an S-*labeled graph *G is a tuple $(V, E, \lambda, s_{in}, s_{out})$ where V is a finite set of nodes, $E \subseteq V \times V$ is the edge relation, $\lambda : V\ s_{in}, s_{out} \to S$ is the labeling function, and $s_{in}, s_{out} \in V$ are the source and sink, respectively.*

The idea behind this, that every edge carries the label of the state it points to. s_{in} and s_{out} play the role of the unique start and end state.

Definition 4:
An automaton *is a Σ-labeled graph.*

The definition of a SOA continues intuitively from that:

Definition 5:
An Single Occurrence Automaton *(SOA) is a automaton, where every* Σ *symbol is assigned to at most one state.*

The paper also says that a SOA A is equivalent to an SORE when there exists an SORE r such that $L(A) = L(r)$.

2.1.2. Inferring SOAs

A SOA can be inferenced from a set of input strings $W = \{w_1, .., w_n\}$ by splitting the input words w_i into their *2-grams* and adding nodes and edges for each element of the 2-gram $G^2_{w_i}$. In this context in "Inference of Concise DTDs from XML Data" also the following proposition is proved, which provides the background for this:

Proposition 1:
Every SORE is 2-testable. More precisely, for every SORE r*, there is an up to isomorphism unique SOA* A_r *such that* $L(r) = L(A_r)$.

Also the algorithm *2T-INF* in the paper "Inference of k-Testable Languages in the Strict Sense and Application to Syntactic Pattern Recognition" [18] is referenced, which infers the sets I, F and S from a set of input strings $W = \{w_1, .., w_n\}$ with $w_i \in \Sigma*$. I_W are simply all fist symbols, F_W all last symbols, and $S_W = \bigcup_{i \leq n} G^2_{w_i}$.
The corresponding automaton G_W can then be constructed easily from those sets:

1. Create a state for each element.
2. Create a separate initial and final state
3. Construct an edge from the initial state to each element name in I_W and an edge from each element name in F_W to the final state.
4. Create an edge for every $ab \in S_W$ from a to b.

The result of this is a SOA describing the input strings W. An example for this can be found in [6].

2.1.3. From SOA to SORE

The main work regarding learning of SOREs in the paper is the conversion of SOAS to SOREs. Two different algorithms are presented, *REWRITE* and *iDTD*. While *REWRITE* inferences regular expressions, which are equivalent to the provided automaton, the *iDTD* algorithm, "*which is an adaptation of* REWRITE*, [..] attempts*

to produce an SORE which describes a (as small as possible) superset of $L(G_W)$*.*" [6] This is implemented, because not every SOA can be converted into a SORE using the *REWRITE* algorithm.

Only the *REWRITE* algorithm will be covered in this work, since it is used later for inferencing the regular expressions.

The *REWRITE* algorithm employs a state-elimination algorithm, transforming the input automaton into a regular expression. This is done by merging several nodes into one single node, or by removing edges from the automaton. The labels on the given SOA are considered scalar regular expressions, and applying the reduction steps the node labels will be replaced by more complex regular expressions, until only one node is left, or no more reductions can be applied. The used reduction steps are:

1. *DISJUNCTION*

Precondition: $W = \{r_1, .., r_n\}$ is a set of states with $n \geq 2$ such that every two nodes r_i , r_j have the same predecessor and successor set. This implies that either (i) there are no edges in G between $r_1, .., r_n$ at all or (ii) that, for each i, j there is an edge (r_i, r_j) in $G*$.

Action: Remove $r_1, .., r_n$, add a new node $r = r_1 + .. + r_n$, redirect all incoming and outgoing edges of $r_1, .., r_n$ to r. In case of (ii) add the edge (r, r).

2. *CONCATENATION*

Precondition: $W = \{r_1, .., r_n\}$ is a maximal set of states, $n \geq 2$, such that there is an edge from every r_i to r_{i+1} , every node besides r_1 has only one incoming edge, and every node besides r_n has only one outgoing edge.

Action: Remove $r_1, .., r_n$, add a new node $r = r_1..r_n$, redirect all incoming edges of r_1 and all outgoing edges of r_n to r. (In particular: if G has an edge (r_n, r_1) then (r, r) is added.)

3. *SELF-LOOP*

Precondition: $(r, r) \in E$.

Action: Delete (r, r), relabel r by $r+$.

4. *OPTIONAL*

Precondition: Every $r' \in Pred(r)$, $Succ(r) \subseteq Succ(r')$. (Thus: every node that can be reached through r from a predecessor, can also be reached directly from that predecessor.)

Action: Relabel r by $r?$, remove all edges (r', r'') such that $r' \in Pred(r)$ and $r'' \in Succ(r)$ $\{r\}$.

An example for this algorithm can be found in [6].

The paper proves, that this algorithm transforms every SOA into the equivalent SORE if one exists. It also mentions that the complexity of this algorithm is $\mathcal{O}(n^4)$.

Also an extension of *REWRITE* is presented which is always able to produce a SORE from a SOA, but is not very efficient for the configuration, which is guaranteed to always produce a SORE. The paper also presents an algorithm which inferences CHAREs, a subclass of SOREs, called *CRX*.

2.2. Inferring Chain Regular Expressions

This section describes the *CRX* algorithm as presented in "Inference of Concise DTDs from XML Data" [6], which is always able to provide a CHARE for each input language. As mentioned earlier each CHARE also is a SORE.

The algorithm shown here will be extended later in this work to inference regular expressions using the additional syntactical features of XML Schema regular expressions.

Similar to the *REWRITE* algorithm the *CRX* algorithm operates on a set of strings, $W = \{w_1, .., w_n\}$. This set of strings defines an order $\rightarrow_W$ on the alphabet, which is defined as:

$$a, b \in \Sigma, a \rightarrow_W b \iff \exists w_i = w_{i_1}..w_{i_n} \in W | w_{i_k} = a \wedge w_{i_{k+1}} = b$$

Additionally the relation $\approx_W \subseteq \Sigma \times \Sigma$ is defined as:

$$a \approx_W b \iff a \rightarrow_W^* b \wedge b \rightarrow_W^* a$$

Where $\rightarrow_W^*$ is the reflexive, transitive closure of $\rightarrow_W$. $\approx_W$ then is obviously an equivalence relation. Let Γ_W the be the set of equivalence classes if $\approx_W$.

The equivalence classes are written in the form $[a_1, .., a_n]$, and a node of the form $[a]$ is called a singleton.

Let $\preceq_W$ be the partial order on Γ_W, which is induced by $\rightarrow_W^*$. For every $\preceq_W$, we denote by H_W its Hasse diagram, i.e., the graph of $\preceq_W$ without transitive edges.

Further we define $succ(\gamma)$ as the set of nodes, which can be reached from γ by a single edge. Accordingly we define $pred(\gamma)$ as the set of nodes, from which with a single edge γ can be reached.

The *CRX* algorithm performed to build the CHARE from W is outlined in Algorithm 1.

An example for this algorithm can be found in [6].

2.3. Inferring k-Occurrence Regular Expressions

The paper "Inference of Concise DTDs from XML Data" only considered Single Occurrence Regular Expressions, since studies in that paper showed that those are the most common regular expressions found in schemas (See section 1.1). Further studies like "Expressiveness and Complexity of XML Schema" showed that even for the cases

Algorithm 1 *CRX*

Compute the set Γ_W of equivalence classes of $\approx_W$
while A maximum set of singleton nodes $\gamma_1, .., \gamma_n$, where $pred(\gamma_1) = .. = pred(\gamma_n)$ and $succ(\gamma_1) = .. = succ(\gamma_n)$ exists **do**
  Replace $\gamma_1, .., \gamma_n$ by $\gamma = \cup_{i=1}^{n} \gamma_i$ and redirect all incoming and outgoing edges from γ_1 to γ.
end while
Compute a topological sort $\gamma_1, .., \gamma_k$ of the nodes
for all $i \in \{1, .., k\}$ $(\gamma_i = [a_1, .., a_n])$ **do**
  if every string in W contains exactly one occurrence of a symbol in $\{a_1, .., a_n\}$ **then**
    $r(\gamma_i) = (a_1 + .. + a_n)$
  else if every string in W contains at most one occurrence of a symbol in $\{a_1, .., a_n\}$ **then**
    $r(\gamma_i) = (a_1 + .. + a_n)?$
  else if every string in W contains at least one of $a_1, .., a_n$ and there is a string that contains at least two occurrences of symbols **then**
    $r(\gamma_i) = (a_1 + .. + a_n)^+$
  else
    $r(\gamma_i) = (a_1 + .. + a_n)^*$
  end if
end for
return Γ_W

where elements occur multiple times, the number of occurrences is generally small. [27]

For those cases the paper "Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data" [4] introduces k-Occurrence Regular Expressions (k-ORE), where k is the number of maximum occurrences of one element in the regular expression. SOREs are therefore *1-ORE*, and the regular expression $(a + b)^*a$, for example, would be a *2-ORE*.

The paper shows that it is possible to learn the class of deterministic k-ORE from positive examples only and introduces an algorithm *iDRegEx*, which derives deterministic k-ORE for increasing values of k and employs a *minimum description length* argument to choose the best resulting regular expression.

The presented algorithm to learn a single k-ORE, *iKORE*, uses a probabilistic approach to learn a k-Occurrence Automaton (k-OA), defined equivalently to the earlier defined SOA. For training the *Baum-Welsh algorithm* [32] is used, which trains *Hidden Markov Models*.

The resulting k-OA is then modified by two further steps, described in the paper, to receive a *deterministic k-OA*. Using the *KOATOKORE* algorithm the k-OA is then transformed into a k-ORE. The *KOATOKORE* algorithm is described in [6], but called *iDTD* in that paper.

The *iDRegEx* is not investigated further in this paper, since the results of the algorithm depend on probabilistic, which makes comparing regular expressions for equivalence much harder, which is required for inferring XML Schema definitions, as shown later in this work.

2.4. XML Schema definitions

The earlier sections described the learning of regular expressions, which is obviously an essential part for learning schema definitions – for XML Schema, Document Type Definition and RelaxNG. But XML Schema also has a typing system, which basically allows to reuse the content model for multiple elements and which allows to have elements with the same label to have different content models.

2.4.1. Structure of XML Schema

Consider the following XML example:

```
1 <?xml version="1.0" ?>
2 <store>
3   <sale>
4     <item>
5       <name>Potatoes</name>
6       <price>3.99 EUR</price>
7     </item>
```

```
8     </sale>
9     <warehouse>
10      <item>
11        <name>Potatoes</name>
12        <count>42</count>
13      </item>
14    </warehouse>
15  </store>
```

When writing a Document Type Definition for this XML document, it is only possible to define one common content model for the `<item>` element:

```
1  <!ELEMENT store (sale, warehouse)>
2  <!ELEMENT sale (item+)>
3  <!ELEMENT warehouse (item+)>
4  <!ELEMENT item (name, (count + price))>
```

With XML Schema, on the other hand, one can define different types for the item elements based on their ancestors. The following schema defines more specific types for the two different `<item>` elements.

```
1  <?xml version="1.0" ?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema">
3    <element name="store" type="store" />
4    <complexType name="store">
5      <sequence>
6        <element name="sale" type="sale" />
7        <element name="warehouse" type="warehouse" />
8      </sequence>
9    </complexType>
10   <complexType name="sale">
11     <sequence>
12       <element maxOccurs="unbounded" name="item" type="sale_item" />
13     </sequence>
14   </complexType>
15   <complexType name="warehouse">
16     <sequence>
17       <element maxOccurs="unbounded" name="item" type="warehouse_item" /
             >
18     </sequence>
19   </complexType>
20   <complexType name="sale_item">
21     <sequence>
22       <element name="name" type="string" />
23       <element name="price" type="string" />
24     </sequence>
25   </complexType>
26   <complexType name="warehouse_item">
27     <sequence>
28       <element name="name" type="string" />
29       <element name="count" type="string" />
```

```
30      </sequence>
31    </complexType>
32  </schema>
```

By introducing artificial types, an element can not only depend on direct parents, but on any ancestors in the type tree. In fact Martens et. al. [27] showed that the content model of an element is completely determined by the labeled path from the root to that element.

The W3C Specification on XML Schema [36] additionally specifies the **Element Declaration Consistency**, which requires multiple occurrences of one element in one content model to use the same type. Hence the following type definition would be illegal:

```
1  <?xml version="1.0" ?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema">
3    <!-- ... -->
4    <complexType name="warehouse">
5      <choice>
6        <element maxOccurs="unbounded" name="item" type="sale_item" />
7        <element maxOccurs="unbounded" name="item" type="warehouse_item" /
             >
8      </choice>
9    </complexType>
10   <!-- ... -->
11 </schema>
```

Formally the **type automaton** of XML Schema definitions thus can be specified as the following triple: [7]

Definition 6:
A XML Schema type automaton 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

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

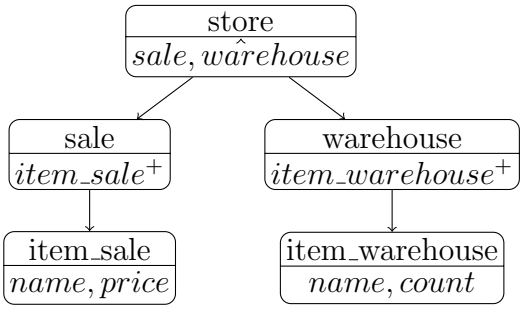
where λ denotes the empty string and a ranges over element names; and a mapping τ that assigns a type to each pair (t, a) with the element name a occurring in $\rho(t)$.

XML Schema notation

For a shorter notation of XML Schema definitions typeset element names will be typesetted in `monospace` and element types will be typesetted in *italic* in this work. The earlier XML Schema can be written more compact then, like:

$$
\begin{aligned}
\mathit{store} &\rightarrow \texttt{<sale>}[\mathit{sale}], \texttt{<warehouse>}[\mathit{warehouse}] \\
\mathit{sale} &\rightarrow \texttt{<item>}[\mathit{sale\_item}]^{+} \\
\mathit{warehouse} &\rightarrow \texttt{<item>}[\mathit{warehouse\_item}]^{+} \\
\mathit{sale\_item} &\rightarrow \texttt{<name>}[\lambda], \texttt{<price>}[\lambda] \\
\mathit{warehouse\_item} &\rightarrow \texttt{<name>}[\lambda], \texttt{<count>}[\lambda]
\end{aligned}
$$

Another notation, used in this work, is the tree notation, which shows the full type automaton and the type dependencies:



When learning XML Schema instead of just Document Type Definition this means, that one needs to learn types for each path, instead of for each element name. Some of those types may be merged and reused to reduce the complexity of the schema. The paper "Inferring XML Schema Definitions from XML Data" [7] presents an algorithm for doing that, which will be described now and extended later in this work.

The path used for this does not necessarily span to the root of the document, but may be limited to some level of ancestors. A schema where each type only depends at maximum on ancestors k level above the current element is called **k-local**. Document Type Definitions therefore are always only **1-local** schemas.

2.4.2. Inference of local XML Schema definitions

For the purpose of this algorithm "Inferring XML Schema Definitions from XML Data" [7] considers an **XML fragment** a (possibly empty) sequence $< \mathtt{a_1} > f_1 < /\mathtt{a_1} > .. < \mathtt{a_n} > f_n < /\mathtt{a_n} >$ of elements where $a_1, .., a_n$ are element names, and $f_1, .., f_n$ are themselves XML fragments. Attributes are ignored in this paper, as well as data values. The inference of atomic data values has already been studied in "Efficient Schema Extraction from Multiple and Large XML Documents". [21]

Furthermore paths(f) is written for the set of all labeled paths starting at a root element in f, if f is an XML fragment. Additionally strings(f, p) is defined as the set of all strings of element names occurring below an occurrence of path p in f.

Based on Definition 4.1 the paper provides a XML Schema validation algorithm which then leads to the inferencing algorithm and definitions, which will be further reused in this work.

The semantics of a XML Schema are given by the following algorithm to validate a XML fragment $f = <\mathtt{a_1}> f_1 </\mathtt{a_1}> .. <\mathtt{a_n}> f_n </\mathtt{a_n}>$ against a type t in a XML Schema $D = (T, \rho, \tau)$. [27, 29] First the string of element names $a_1..a_n$ is matched against the regular expression $\rho(t)$. If the check fails, the fragment is rejected, otherwise each f_i is validated against the type $\tau(t, a_i)$ of a_i in t and accept the fragment if all validations (recursively) succeed. This validation algorithm also implies, that the content model of an element is completely determined by the labeled path from the root to the element, as mentioned earlier.

For $f = <\mathtt{a_1}> f_1 </\mathtt{a_1}> .. <\mathtt{a_n}> f_n </\mathtt{a_n}>$ to be of type t, each f_i must be valid with respect to $\tau(t, a_i)$. This is true only if $f_i = <\mathtt{b_1}> g_1 </\mathtt{b_1}> .. <\mathtt{b_m}> g_m </\mathtt{b_m}>$ and every g_j is valid with respect to $\tau(\tau(t, a_i), b_j)$. This reasoning can be continued until the desired element is reached, where its child fragment h must be of type $\tau(..\tau(\tau(t, a_i), b_j).., c)$ with $aibj..c$ as the labeled path from the root to the element.

This leads to the following alternative view on validation, which forms the cornerstone of the papers inference algorithms. Let, for a path $p = ab..c, \tau(s, p) \to t$ denote that $\tau(..\tau(\tau(s, a), b).., c)$ is defined and equals t. Let $L(r)$ denote the set of all strings matched by regular expression r.

Proposition 2:
([7]) An XML fragment f has type s in an XML Schema (T, ρ, τ) iff for every path $p \in paths(f)$ there exists t such that $\tau(s, p) \to t$ and $strings(f, p) \subseteq L(\rho(t))$.

As already mentioned the **locality** of a content model in a XML Schema basically defines the length of the minimum path, which is required to differentiate a type from another with the same element name. It is defined formally in [7]:

The formal definition of such **k-local** XML Schema definitions is as follows. Let $p|_k$ stand for the path formed by the k last element names of a path p (if $length(p) \leq k$ then we take $p|_k = p$). Two paths p and q are **k-equivalent** if $p|_k = q|_k$. In particular, when $length(p) < k$, p is only k-equivalent to itself.

Definition 7:
*A pair (D, s) with D an XML Schema and s a type in D is called **k-local** if for all k-equivalent p and q such that $\tau(s, p) \to t$ and $\tau(s, q) \to t$ we have $t = t$.*

For learning the content models for the types in the XML Schema the algorithms from "Inference of Concise DTDs from XML Data" [6] are reused. Thus the algorithm learns XML Schema definitions with only Single Occurrence Regular Expressions (SORE), and in the paper the learned schemas are called SOXSD (Single Oc-

currence Xml Schema Definitions) accordingly. This also ensures the learned regular expressions are deterministic, as required by the specification. [36]

The SORE are learned from Single Occurrence Automatons (SOA) as, described earlier in chapter 2. The SOA contains of states, which, "*except for the initial and final state, are element names.*" [7]

Those SOAs are learned for paths which are limited to a length k for **k-local SOXSD**.

For the XML example in subsection 2.4.1 the resulting 2-local XML Schema would look like:

$$
\begin{aligned}
(\lambda, \texttt{<store>}) &\rightarrow \texttt{<sale>}, \texttt{<warehouse>} \\
(\texttt{<store>}, \texttt{<sale>}) &\rightarrow \texttt{<item>} \\
(\texttt{<store>}, \texttt{<warehouse>}) &\rightarrow \texttt{<item>} \\
(\texttt{<sale>}, \texttt{<item>}) &\rightarrow \texttt{<name>}, \texttt{<price>} \\
(\texttt{<warehouse>}, \texttt{<item>}) &\rightarrow \texttt{<name>}, \texttt{<count>} \\
(\texttt{<item>}, \texttt{<name>}) &\rightarrow \lambda \\
(\texttt{<item>}, \texttt{<price>}) &\rightarrow \lambda \\
(\texttt{<item>}, \texttt{<count>}) &\rightarrow \lambda
\end{aligned}
$$

The result might differ from the expectation, since for three different elements an equivalent type is learned, but they do not reference the same type. With incomplete input sets of XML data this problem gets even worse. Consider the following example:

```
1  <?xml version="1.0" ?>
2  <store>
3    <sale>
4      <category>
5        <item/>
6        <item/>
7      </category>
8    </sale>
9    <warehouse>
10     <category>
11       <item/>
12     </category>
13   </warehouse>
14 </store>
```

Note that in this example the `<item>` elements are the same. Despite that the following 2-local SOXSD would be learned:

$$
\begin{aligned}
(\lambda, \texttt{<store>}) &\rightarrow \texttt{<sale>}, \texttt{<warehouse>} \\
(\texttt{<store>}, \texttt{<sale>}) &\rightarrow \texttt{<category>} \\
(\texttt{<store>}, \texttt{<warehouse>}) &\rightarrow \texttt{<category>} \\
(\texttt{<store>}, \texttt{<category>}) &\rightarrow \texttt{<item>}^{+} \\
(\texttt{<warehouse>}, \texttt{<category>}) &\rightarrow \texttt{<item>} \\
(\texttt{<category>}, \texttt{<item>}) &\rightarrow \lambda
\end{aligned}
$$

It is obvious that the difference between the two `<category>` content models occurs due to the sparse input. The resulting SOREs are not equal, even, with a more complete XML corpus, they probably are meant to be equivalent.

2.4.3. Minimization

To tackle the problem of too many types the authors present the algorithm *MINIMIZE*, which locates equivalent types and merges the type definitions. Types are considered equivalent if the XML fragments produces by this type are equal – which is, for example, the case for the empty elements in 2.4.2. The full algorithm for learning k-local SOXSD including the *MINIMIZE* step is called *iLOCAL* and works on an XML corpus $\mathcal{C}$, where r is the root type of the schema:

$$\text{XSD}(D, r) = i\text{LOCAL}(\mathcal{C}, k)$$

This does not apply to 2.4.2, though, since the XML fragments for the two `<category>` elements are different. The paper introduce the *REDUCE* algorithm for that, which merges similar type depending on a provided similarity threshold ϵ.

The REDUCE algorithm

The goal of *REDUCE* is to not only merge equal, but also similar types. The similarity of two types is measured based on the inferenced SOAs, which are learned for two types. If the types are similar enough to be considered equal the XML Schema is subsequently adapted. The adaption can be considered as a generalization of the schema to compensate missing input data in the XML corpus.

To calculate the similarity of the SOAs the algorithm is adapted to include the **support** for each edge in the SOA, denoted by $supp_A(a, b)$ for the edge from a to b in the automaton A. The adapted SOAs obviously need to be stored during the algorithm, so that they are available for the comparison. The SOA for a type s is denoted by:

$$\text{soa}(s) := \text{iSOA}(k\text{-strings}(\mathcal{C}, p|_k))$$

Where $\mathcal{C}$ is the XML corpus and *iSOA* is the modified algorithm for learning the SOAs. k-strings$(\mathcal{C}, p|_k)$ are defined as the set of all strings in $\mathcal{C}$ that occur below paths that are k-equivalent to p:

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

Based on these extra data structure the similarity of two types s and t in an inferred XML Schema $(D, r) = i\text{LOCAL}(\mathcal{C}, k)$ can be calculated. Let $\text{dist}(A, B)$ then be the **normalized edit distance** between the support-annotated SOAs $A = (V, E)$ and $B = (W, F)$ of those types:

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

$\text{dist}(A, B)$ intuitively calculates the dissimilarity of A and B by counting the number of edges, which are available in A, but not in B, weighted by the support these edges have and normalized by the overall support of all edges in A, and the same for the edges in the opposite direction. If both SOAs are the same the dissimilarity will be 0, obviously.

The similarity of two types s and t, or **edit distance**, is then given by:

$$\text{dist}_D(s, t) := \max_{(s',t')\in \text{reach}_D(s,t)} \text{dist}(\text{soa}(s'), \text{soa}(t'))$$

This describes the maximum edit distance between any two types, which can be reached by the same path in the current type subtree.

Based on the dist_D the algorithm merges types, which dissimilarity is below some specified threshold value. Merging the types involves the adjunction of the SOAs for the types. The algorithm, as presented in [7], is shown in Algorithm 2.

$(s', t') \in \text{reach}_D(s, t)$ is defined as all elements which can be reached from the types s and t using the same path p such that $\tau(s, p) \to s'$ and $tau(t, p) \to t'$.

With higher threshold values there is a risk of false merges, e.g. with a threshold value of 2 all types would be merged, no matter how different they are. The paper evaluates different values for the threshold ϵ against a hand-crafted example XML Corpus and checks for false positives and false negatives. Their results listed in Table 2.1.

With the used input it shows that starting by a threshold of 0.2 the number of false negatives increases, which means that types which are not meant to be merged are merged by the algorithm. In this work this approach, and similar approaches, will be evaluated against more examples.

Algorithm 2 *REDUCE*

let $(T, \rho, \tau) = D$
initialize $M := \{(s,t) \in T^2 | 0 < \text{dist}_D(s,t) < \epsilon\}$
while M is non-empty **do**
  for all $(s,t) \in M$ **do**
    for all $(s',t') \in \text{reach}_D(s,t)$ **do**
      set $\text{soa}(s') := soa(s') \uplus soa(t')$
      set $\text{soa}(t') := soa(s')$
      for all $a \in \text{elems}_D(t') - \text{elems}_D(s')$ **do**
        add $(s',a) \to \tau(t',a)$ to τ
      end for
      for all $a \in \text{elems}_D(s') - \text{elems}_D(t')$ **do**
        add $(t',a) \to \tau(s',a)$ to τ
      end for
    end for
  end for
  recompute $M := \{(s,t) \in T^2 | 0 < \text{dist}_D(s,t) < \epsilon\}$
end while
for all type $t \in T$ **do**
  replace each $t \to \rho(t)$ in ρ by $t \to \text{TOSORE}(\text{soa}(t))$
end for

ϵ	false pos.	false neg.
0.01	2	0
0.05	0	0
0.10	0	0
0.15	0	0
0.20	0	3
0.50	0	3

Table 2.1.: *REDUCE* evaluation results

Part II.

Development

Overview

In this part the existing analysis and algorithms, described in the last part, are extended. This parts follows the common structure by performing additional analysis on larger and more current sets of XML data. Based on the data gathered in the analysis the existing algorithms are adapted and extended. Afterwards a methodology for evaluation of the resulting schemas will be presented and the developed algorithms will be compared on this basis.

Chapter 3.

Schema analysis

Besides the existing analysis of schemas own additional statistics were required to verify which improvements to the algorithms would prove most valuable. This approach already proved itself important during earlier works on inferencing Document Type Definitions and XML Schema definitions. With the use of statistics it was possible to find about valid simplifications for child patterns, for example. [6]

For some of the required statistics extra steps had to be performed to calculate accurate statistics. The analysis of XML Schema definitions included the following steps:

1. Finding XML Schema definitions
2. Schema downloading
3. Schema normalization
4. Schema ranking
5. Calculation of statistics

Those steps will be described in further detail in this chapter.

3.1. Finding XML Schema definitions

The goal is to analyse schemas, which are currently used to describe XML data for real life applications. There are different ways to get those schemas. The number of schemas used to generate the statistics often were relatively small in earlier studies [6, 5, 24].

For this study different search engines and known schema repositories were used to locate as many schemas as possible. The only popular search engine, which still indexed XML Schema definitions was *Google*, while *Bing* and *Yahoo!* did not offer any results when restricting the file type to XML Schema definitions.

To find as many XML Schema definitions as possible, the search queries were parametrized. Google offers multiple ways to restrict a search, like:

- Country codes to limit the originating area
- Count and offset of search results
- File type of found documents

All these parameters were used to widen the set of potential results. The resulting URLs of potential XML Schema definitions often led to false positives.

The *Stylus XML Schema Library* also provided an index of well-known XML Schema definitions, which were also downloaded.

Altogether the searches resulted in 2733 unique URLs.

3.2. Schema downloading

The unique URLs, which were located in the schema fetching step only point to *potential* XML Schema definitions. To receive a set of unique XML Schema definitions the schema downloader had to complete additional tasks:

- Detect if the downloaded item really is a XML Schema
- Fix small fixable errors
- Remove duplicate schemas

Those tasks are implemented in the schema downloader and are now described in more detail.

Since several of the URLs pointed to textual descriptions of a schema, rather then to the schema itself, a method had to be implemented, to detect if the downloaded file really is a XML Schema. Since there exist many partially invalid schemas, which still could be processed like a real schema, it has been decided to not be too strict about that. Those partially invalid schemas most likely were written by humans, which then provides important insight into the properties of common XML Schema definitions.

2% of the schemas contained XML errors, which were mostly simple syntax errors, which libxml2 were able to fix on loading. libxml2 was the library used to process and analyse the XML Schema definitions and is generally one of the most used XML libraries.

Since several XML Schema and XML authors do not handle namespaces correctly the checking for the namepsace of the document was not feasible. The check, which was performed to verify if a document is a XML Schema was to ensure that the root element has the local name `<schema>`. About 23% of the URLs found earlier pointed to documents not containing a `<schema>` element as the root node.

The `targetNamespace` attribute of a XML Schema contains the namespace a XML Schema defines. The contained name can therefor be used as an unique identifier

Anonymous XML Schema definitions	318
Namespaced XML Schema definitions	1139
Sum	1511

Table 3.1.: Found XML Schema definitions

of the schema. Later versions of the same schema would contain a different version string in the target namespace. With this namespace it is possible to remove most XML Schema duplicates, which were found under multiple URLs.

On the other hand there are a lot of XML Schema definitions, which do not define a target namespace. For those schemas the XML markup was normalized and hashed to generate a custom unique identifier for those schemas. This allowed us to remove all obvious duplicates (about 22%) and left the number of schemas, shown in Table 3.1.

3.3. Schema normalization

One part of the analysis are statistics on the type automaton, which has been defined in Definition 4.1. XML Schema itself offers syntactic sugar when specifying types, which makes them hard to analyse. For this a normalization step is performed, which transforms the XML Schema in a semantically equivalent XML Schema, which only uses very basic syntax.

The syntactical elements which are eliminated during this process are:

- Simplify types
  - Extract anonymous types
  - Inline groups
  - Inline attribute groups
  - Evaluate extensions
  - Evaluate restrictions
  - Normalize simple content
  - Remove unused types
- General markup simplifications
  - Removing comments and annotations
  - Simplify simple types

These normalizations are now described in further detail.

3.3.1. Extract anonymous types

Types are normally identified by a string and referenced by that string from any `<element>` in the schema, which should be validated against this type. But XML Schema does not enforce that each type must be referenced by a name, you can also specify a type inline.

```
<?xml version="1.0" ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="root" name="root">
  <complexType name="root">
    <sequence>
      <element name="child">
        <complexType>
          <sequence>
            <element name="gradchild" type="string"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</schema>
```

In the example above the element named `<root>` has an explicit type definition and is associated with the type also named `<root>`. In this type definition a sequence is defined which only contains one child element, named `<child>`, which has an implicit type definitions. There is no type name specified, but the complex type is defined directly in the element. The child element again contains only one element named `<grandchild>` which contains a simple type definition.

This normalization now takes implicitly defined types, like for the `<child>` element above, and converts them into a new explicitly defined type with a unique name. The new type is declared on a global level of the schema:

```
<?xml version="1.0" ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="root" name="root">
  <complexType name="root">
    <sequence>
      <element name="child" type="djf3924jsdg"/>
    </sequence>
  </complexType>
  <complexType name="djf3924jsdg">
    <sequence>
      <element name="gradchild" type="string"/>
    </sequence>
  </complexType>
</schema>
```

This makes it easier to analyse type dependencies and analyse the expressions which are defined by the types.

3.3.2. Inlining groups

XML Schema defines groups, which are basically sub-expressions, which may be reused in type definitions. These groups can define the same structures like any type definitions can do. Each group may be reused by any number of type definitions. A very similar construct exists for attribute lists, so called `<attributeGroup>`s.

To properly analyse the complexity of expressions, those groups need to inlined recursively into the type expressions. This has to be done recursively, since one group may reference another group.

During the group inlining step it showed that two schemas actually contained a direct group self-reference, which is obviously invalid, since it would lead to an expression of infinite size. Those direct self references are ignored. Beside self-references group-inlining is implemented straight-forward, ignoring groups from external schemas.

3.3.3. Evaluating inheritance

In the context of the XML Schema typing mechanism it makes sense to be able to further refine existing types. XML Schema implements those by with the `<extension>` and `<restriction>` elements. Both allow to modify the child pattern of a type defined elsewhere.

An `<extension>` allows to extend an existing type with additional allowed elements, like the following example shows by extending the `extended` type from the `base` type and adding an additional element:

```
1  <?xml version="1.0"?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema">
3    <element name="root" type="extended"/>
4
5    <complexType name="base">
6      <sequence>
7        <element name="child1" type="string"/>
8      </sequence>
9    </complexType>
10
11   <complexType name="extended">
12     <complexContent>
13       <extension base="base">
14         <sequence>
15           <element name="child2" type="string"/>
16         </sequence>
17       </extension>
18     </complexContent>
19   </complexType>
20 </schema>
```

An `<extension>` only allows appending, like mentioned in [36]:

> This specification allows only appending, and not other kinds of extensions.

A `<restriction>` only modifies the simple type definitions of the referenced base type. Since simple types are not analyzed here, the child pattern from the referenced type may just be inlined ignoring the defined modifications.

3.3.4. Other simplifications

For easier manual introspection and faster processing of the schemas, information irrelevant to this analysis was removed from the schemas. All `<annotation>` elements, which are comments in XML Schema, XML comments and unused types were removed. Additionally the simple type definitions all are replaced by the same string type.

3.4. Schema ranking

XML Schema supports `<import>` to include the definitions of another schema, so that its types may be referenced, extended, or the groups may be used in type definitions.

Those import statements are not resolved during the schema normalization, to make it possible to analyze the schemas with and without influence of the referenced schemas. Some XML Schema definitions, like the XML XML Schema are used very often, and would have a larger influence.

Beside the unweighted statistics two ranking methods were implemented to verify if the value distribution differences significantly between the popular and the unpopular XML Schema definitions.

- Schema support
- PageRank

The weighting is defined as a function over Σ, the set of all analyzed XML Schema definitions to a number in $\mathbb{R}$. The value provided by the weighting function function for a given XML Schema is multiplied with the results from the statistics. For the unweighted results the following function is used:

$$\omega(\Sigma) = 1$$

3.4.1. Schema support

Schema support can formally be specified as the following function:

Definition 8:
$\omega(\Sigma)$ is a function mapping each XML Schema in Σ to a number $n \in \mathbb{N}$.

$$\omega(\Sigma) \in \mathbb{N}$$

Where Σ is the set of analysed schemas and n is the support of a XML Schema. The support of a XML Schema is the number of references in other XML Schema definitions to the XML Schema plus 1.

The addition of 1 to the basic support of each schema is used to leave a minimal relevance to the unreferenced XML Schema definitions.

3.4.2. PageRank

The PageRank is an algorithm originally developed by Brin and Page in "The PageRank Citation Ranking: Bringing Order to the Web" [31] to implement an objective importance metric for web sites by emulating an idealized random surfer.

The PageRank algorithms can be mapped to evaluate importance of nodes in any graph, like the CodeRank metric shown in "CodeRank: A New Family of Software Metrics" [**?**]. The XML Schema definitions with its `<import>` statements can be considered a directed graph:

Definition 9:
The SchemaGraph is a directed graph $G = (\Sigma, I)$, where Σ are the analyzed schemas and I are the edges, denoted by the `<import>` statements in the analyzed schemas pointing from the analyzed schema to the referenced schema.

On this graph the PageRank can be calculated as described in "The PageRank Citation Ranking: Bringing Order to the Web" [31] - for this implementation a equally distributed source ranking had been chosen. An excerpt of the schemas with the highest rank, as determined by the PageRank algorithm, can be found in Table 3.2.

3.5. Calculation of statistics

With the normalized XML Schema definitions as a base and the three different schema weighting functions the relevant statistics can be calculated. The statistics focus on the differences between Document Type Definitions and XML Schema definitions, which are the XML Schema typing mechanism and the differences in child pattern regular expressions. The properties of simple type usage are ignored, because they are irrelevant for this work, and analysis of those has already been researched. [21]

The following statistics were calculated:

Schema	PageRank
http_www_w3_org_2001_SMIL20_	21.28
http_www_w3_org_XML_1998_namespace	18.99
http_www_w3_org_1999_xlink	7.24
http_www_it_ojp_gov_jxdm_appinfo_1	5.74
http_purl_org_dc_elements_1_1_	3.62
http_www_w3_org_2000_09_xmldsig_	3.31
urn_oasis_names_specification_ubl_schema_xsd_CoreComponent...	3.30
http_www_opengis_net_gml	2.95
http_java_sun_com_products_oss_xml_Common	2.79
...	

Table 3.2.: PageRank of XML Schema definitions

- Child patterns
  - `<all>` occurrences
  - Counting pattern usage
- Typing mechanism
  - Usage of types
  - Ancestor depth

The statistics are discussed in further depth in the following sections.

3.5.1. Child patterns

Child pattern analysis is one of the three major differences between XML Schema and Document Type Definition. The class of allowed regular expressions in Document Type Definition and XML Schema is the same, but XML Schema allows several syntactical constructs which are intended to make schema authoring easier. This analysis is intended to analyze the usage of those constructs to evaluate if the algorithms for learning the regular expressions should be adapted to learn XML Schema definitions, which are closer to real world schemas.

`<all>` occurrences

`<all>` is new to XML Schema and allows to specify a set of elements, which may occur in an expression with insignificant order. *RelaxNG* introduced the ampersand (&) in their compact syntax for this which is also used in this work for a shorter notation. [14] An example for a conversion of a `<all>` concatenation of the elements `<a>`, `<b>` and `<c>` to the simplified syntax of Document Type Definitions therefore is:

$$a\&b\&c = (abc + acb + bac + bca + cab + cba)$$

The equivalent XML Schema code would look like:

```
<!-- ... -->
  <all>
    <element name="a" type="a" />
    <element name="b" type="b" />
    <element name="c" type="c" />
  </all>
<!-- ... -->
```

The occurrences of `<all>` were analyzed per schema and per type.

	No weighting	Support	PageRank
Per type	2154 / 62439 (3.45%)	3.45%	0.52%
Per schema	90 / 1511 (5.96%)	6.02%	0.90%

Table 3.3.: Occurrence of `<all>`

The numbers in Table 3.3 show that `<all>` is used in a significant number of XML Schema definitions, even the number is significantly lower for the most popular schemas. The results weighted by the **PageRank** clearly show that `<all>` is mainly used in XML Schema definitions, which are not reused by other XML Schema definitions.

Counting pattern usage

XML Schema supports simple counting patterns applied to `<sequence>`, `<choice>`, `<all>` and `<element>` nodes. Counting patterns specify how often a subexpression may occur. For each element it allows to specify the minimum and maximum number of occurrences. An example for a conversion of a counting pattern applied to a `<element>` to the simplified syntax used in Document Type Definitions could look like:

$$a\{2,5\} = aaa?a?a?$$

The equivalent XML Schema code would look like:

```
<!-- ... -->
  <element name="a" type="a" minOccurs="2" maxOccurs="5" />
<!-- ... -->
```

Counting patterns were analyzed per subexpression for the elements mentioned before, per type and per schema. There is a certain set of counting patterns, which can be expressed using the common counting modifiers ?, $*$ or $+$ in Document Type Definition, too. The values which can be mapped to those counting modifiers are

considered **standard**. Setting the minimum and maximum occurrences is of course optional. If nothing was set by the schema author the subexpression was counted as **default**. **Non standard** are all those counting patterns, which contain values other then 0, 1 and unbounded:

Default
No counting patterns were provided by the schema author.

Standard
Only counting patterns were provided, which could be mapped to ?, $*$ or $+$.

Non-Standard
All other counting patterns

First, the aggregated usage of counting patterns per schema, while a schema is considered "default", if only default values are used, and is considered "standard" if only default and standard patterns are used. If there is only one non-standard pattern, it is considered "non-standard".

	No weighting	Support	PageRank
minOccurs			
Default	102 / 1511 (6.75%)	6.10%	5.71%
Standard	1337 / 1511 (88.48%)	90.22%	90.68%
Non-Standard	72 / 1511 (4.77%)	3.68%	3.62%
maxOccurs			
Default	99 / 1511 (6.55%)	5.94%	5.57%
Standard	1267 / 1511 (83.85%)	87.23%	87.60%
Non-Standard	145 / 1511 (9.60%)	6.84%	6.82%
Aggregated			
Default	59 / 1511 (3.90%)	3.97%	3.67%
Standard	1278 / 1511 (84.58%)	87.76%	88.14%
Non-Standard	174 / 1511 (11.52%)	8.27%	8.19%

Table 3.4.: Counting patterns per schema

As shown in Table 3.4 there is no significant difference between the unweighted and the weighted results – counting patterns seem equally distributed across schemas.

That only about 10% of the XML Schema definitions (unweighted: 11.52%, pagerank: 8.19%) ever use non-standard counting patterns of any type anywhere in their schemas is a quite low number, which is even more obvious when looking at the per-type usage of counting patterns.

Table 3.5 shows that .41% of the types are using non-standard minimum occurrence patterns and only .86% of types using non-standard maximum occurrence patterns. That leaves 98.89% of all type definitions with default or standard counting patterns.

	No weighting	Support	PageRank
minOccurs			
Default	15587 / 62439 (24.96%)	24.97%	25.10%
Standard	46596 / 62439 (74.63%)	74.59%	74.44%
Non-Standard	256 / 62439 (0.41%)	0.45%	0.45%
maxOccurs			
Default	17722 / 62439 (28.38%)	28.38%	28.51%
Standard	44179 / 62439 (70.76%)	70.74%	70.60%
Non-Standard	538 / 62439 (0.86%)	0.88%	0.89%
Aggregated			
Default	10357 / 62439 (16.59%)	16.98%	16.91%
Standard	51389 / 62439 (82.30%)	81.90%	81.96%
Non-Standard	693 / 62439 (1.11%)	1.12%	1.14%

Table 3.5.: Counting patterns per type

Additionally one may take a look at the distribution of counting patterns to see, if there are some very common multipliers.

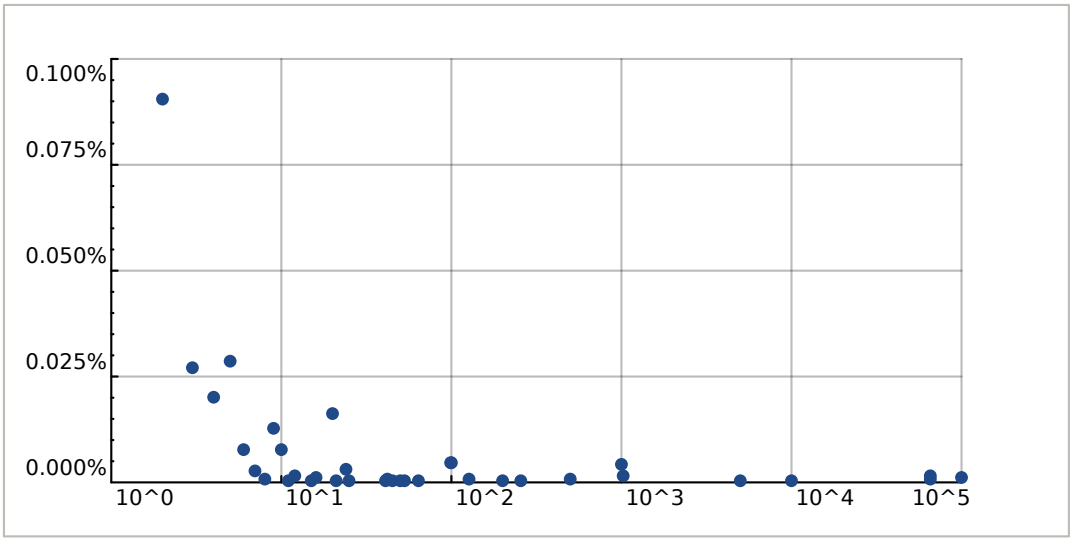


Figure 3.1.: Max occurrence distribution

Like shown in Figure 3.1 the max occurrence counts are distributed in a quite broad range. The only significant peak is around 2, which is used in .08% if `<element>` definitions, 19 (of 11669) `<choice>` and 39 (of 34829) `<sequence>` definitions.

The high numbers ($> 10^3$) for `maxOccurs` are most probably used by schema authors, which did not know about `unbounded`.

The distribution of values used for minimum occurrences shown in Figure 3.2 also

show a significant peak for 2. Still the general usage of `minOccurs` is very low and shows a broad distribution of values.

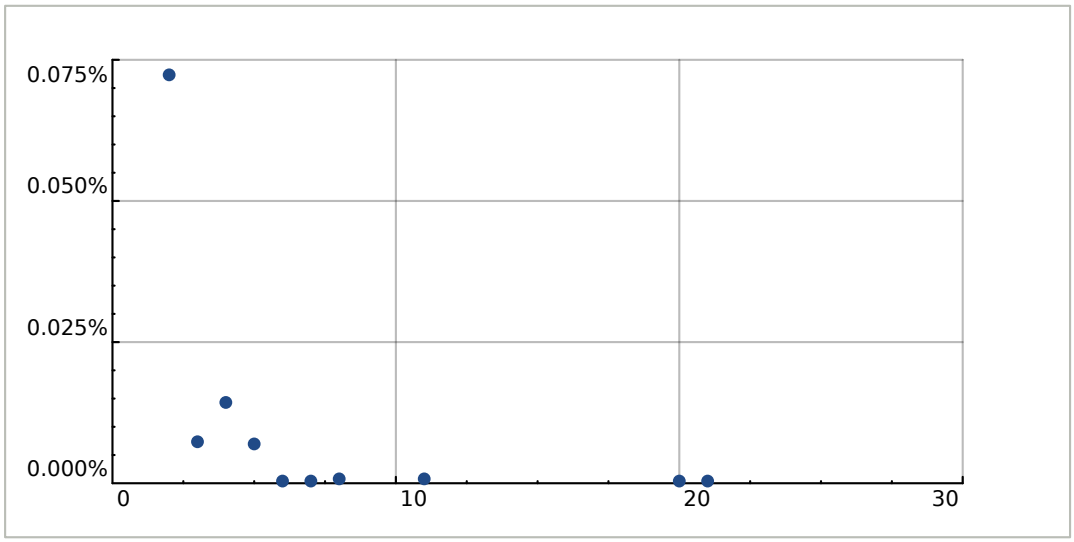


Figure 3.2.: Min occurrence distribution

Conclusion

The usage of counting patterns is insignificant, with a broad distribution of used values. The statistical analysis shows that it is most probably not relevant to learn counting patterns from input data. Especially learning maximum occurrence counts for patterns would most probably require negative examples.

On the other hand `<all>` is used in a significant amount of schemas, even not used in the most popular ones. It seems sensible to extend the regular expression learning algorithms to be able to also learn `<all>` regular expressions.

3.5.2. Typing mechanism

The main difference between XML Schema and Document Type Definition is the typing mechanism, as already discussed in section 2.3.

This is different from the XML Schema as defined in 4.1, where the regular expressions are regular expressions of (label, type) tuples. This implies, that the content model of an element is completely determined by the labeled path from the root to that element. [7] This path is called **ancestor path**.

Ancestor depth

A per schema analysis of the ancestor depth of types shows how many of the XML Schema definitions are using the XML Schema typing mechanism. The **ancestor**

depth is defined as:

Definition 10:
The ancestor depth of an element is the minimum path length required to determine the type of an element, sufficient to differentiate the element from elements with the same label but a different type.

An ancestor depth > 1 means, that at least one element requires the parent element to determine its type, and therefore the schema is not a Document Type Definition anymore.

Ancestor depth	No weighting	Support	PageRank
1	1233 (81.60%)	88.25%	87.67%
2	171 (11.32%)	7.33%	7.62%
3	60 (3.97%)	2.46%	2.63%
4	32 (2.12%)	1.35%	1.42%
5	13 (0.86%)	0.53%	0.57%
6	2 (0.13%)	0.08%	0.09%

Table 3.6.: Ancestor depth per schema

Table 3.6 shows that 82% of the analysed schemas are not using the XML Schema typing mechanism. Weighted by the PageRank the number of schemas not using the XML Schema typing mechanism increases significantly to nearly 88%. This implies, that the most popular XML Schema definitions most probably can be written as Document Type Definitions.

XML Schema types are still broadly used in other schemas, with a maximum ancestor depth of 6, found in 2 schemas. Compared with the ancestor depth per type usage statistics in Table 3.7 one can see that only few types per schema are using a high ancestor depth.

Ancestor depth	No weighting	Support	PageRank
1	74686 (97.81%)	97.79%	99.66%
2	1106 (1.45%)	1.47%	0.22%
3	355 (0.46%)	0.47%	0.07%
4	125 (0.16%)	0.16%	0.03%
5	84 (0.11%)	0.11%	0.02%
6	2 (0.00%)	0.00%	0.00%

Table 3.7.: Ancestor depth per type

Usage of types

In Document Type Definition the type of an element is only determined by its label. In XML Schema it is determined by its label and the ancestor path. If the same label occurs multiple times in a schema and is associated with different types, the schema cannot be expressed as a Document Type Definition anymore. Those types are furthermore call **XSD-types**.

Since types are identified by a type in XML Schema it is possible that the same type is used with multiple elements with different labels. This analysis researches how these two features are used in real world XML Schema definitions.

Types per label	No weighting	Support	PageRank
1	95.72%	98.43%	97.40%
2	2.30%	0.85%	1.40%
3	0.35%	0.13%	0.21%
4	0.68%	0.25%	0.41%
5	0.10%	0.03%	0.06%
6	0.06%	0.02%	0.03%
7	0.02%	0.01%	0.01%
8	0.41%	0.15%	0.25%
9	0.05%	0.02%	0.03%
10	0.01%	0.01%	0.01%
	...		
1008	0.00%	0.00%	0.00%

Table 3.8.: Usage of multiple types for the same label

With the schema normalization described in section 3.3 it is quite simple to analyze the type label ratios. Since each type is extracted into a named `<complexType>` definition in the global schema element, the "types per label" analysis in Table 3.8 results in fetching all `<element>` nodes, grouping them by the label, which is defined by the `name` attribute, and checking how many different types are referenced.

The "labels per type" statistics in Table 3.9 worked analogous. For both analysis only schema-local `<complexType>` definitions were taken into account. Statistics based on all types do not differ significantly – even if also simple types are taken into account. For types, which are not declared locally in the schema, it cannot be decided, though, if it is a simple or complex type. Therefore only local complex type definitions were used for this statistic.

Table 3.8 shows the usage of the main XML Schema feature, how often are different types used for the label. If this number is greater then 1, the type definitions cannot be expressed as a Document Type Definition. 95.7% of the labels only have one type definition assigned and this feature is is used less in the popular schemas.

Also interesting, and not mentioned in other papers [7], is the XML Schema feature shown in Table 3.9. Not only the same label may be used with different types, but also the same type may be reused with multiple elements with different labels. Like the statistics show, this is used in a similar amount of types, while it is used more in the popular schemas.

The schema definition of "US GAAP Taxonomies, Release 2009", for example, uses the same string complex type definition for 6091 different labels.

Labels per type	No weighting	Support	PageRank
1	95.52%	90.53%	93.63%
2	2.05%	3.95%	2.75%
3	0.82%	1.39%	1.01%
4	0.50%	1.07%	0.72%
5	0.24%	0.70%	0.41%
6	0.18%	0.51%	0.31%
7	0.12%	0.19%	0.16%
8	0.07%	0.20%	0.12%
9	0.07%	0.12%	0.09%
10	0.06%	0.19%	0.11%
	...		
6091	0.00%	0.00%	0.00%

Table 3.9.: Usage of multiple labels for the same type

3.5.3. Comparison

The data set fetched two years earlier for [12] was still available during this research. It contained over 8000 XML Schema definitions. Significant differences between the numbers presented here and performing the same analysis on those schemas could not be found.

There are other papers performing similar analysis on other sets of XML Schema definitions. Since the schema simplifications described above are not performed in those works the numbers might not be comparable.

In [5], dating back to 2004, 5 out 30 (16.7%) XML Schema definitions used XSD-types, as opposed to 18.4% in this study. Given the low number of schemas used in that study no tendency can be extrapolated from that.

The XML Schema feature `<all>` has been researched in [24] with data sets of XML Schema definitions dating back to 2005 and 2008, containing 199 and 233 valid XML Schema files, respectively. For the 2005 data set `<all>` occurred in 10.55% of all schemas and for the 2008 data set it occurred in 19.82% of all schemas. Those numbers differ quite a bit from the 6% found in this analysis and support the incentive

to integrate support for `<all>` in the schema inferencing algorithms. It can only be assumed that the XML Schema definitions analysed in those studies belong to an adverse subset of the schemas fetched in this work and in [12] which were both analyzed in this work.

3.6. Conclusion

The usage of XML Schema types in XML Schema definitions is significant. About 7% of the schemas are even using an ancestor depth of > 2. This leads to the conclusion, that the algorithms for schema learning should be extended to also inference different types for the same label.

The usage of the same type for multiple different labels is as popular as using different types for the same label. Therefore the algorithms should also learn schema definitions in which the same type is used for different labels. This may also lead to more compact schema definitions.

The usage of `<all>` in XML Schema definitions is also significant, as opposed to counting patterns. Therefore the algorithms should also be extended to support inferencing `<all>`.

Chapter 4.

Schema learning

The schema analysis in chapter II showed that some improvements to the existing algorithms for learning of XML Schema definitions, which were described in subsection 1.2.2, are meaningful. It has been found out, that he typing mechanism of XML Schema is used in practice, so that improvements to the type merging algorithms should be researched and evaluated. Additionally extensions to the regular expression syntax are also used in XML Schema definitions, so that the corresponding algorithms should also be adapted for that.

4.1. XML Schema definition

First the XML Schema formalization presented in 4.1, as provided by [7] is extended. This formalization of XML Schema defines XML Schema definitions as a triple $D = (T, \rho, \tau)$, consisting of a finite set of types T; a mapping ρ from T to regular expressions r.

This does not specify a set of potential root elements, e.g. elements which may occur as root nodes in the validated XML instances. The Document Type Definition definition, presented in 2, on the other hand defines a single possible root node. In Document Type Definitions one single root node has to be defined explicitly in the schema, while in XML Schema definitions the set of root nodes is only defined implicitly, thus often overlooked.

As stated in [12] all elements defined in `<element>` nodes, which are direct children of the `<schema>` node are possible root elements in the XML instances, which are validated against the XML Schema. This feature is used in practice, like the following extracts from the Docbook [15] schema show:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified">
3   <!-- ... -->
4   <!-- doc:A book. -->
5   <xs:element name="book" substitutionGroup="book.class"/>
6   <!-- ... -->
7   <!--
8     An Article is a chapter-level, stand-alone document that is often,
9     but need not be, collected into a Book.
```

```
10    -->
11    <xs:element name="article" substitutionGroup="article.class"/>
12    <!-- ... -->
13  </xs:schema>
```

The XML Schema defines both, `<book>` and `<article>`, as valid root nodes while `<article>` also is a valid child of the `<book>` element.

The following example shows a XML Schema which only consists of one valid root element by defining all other elements locally in the type definitions:

```
1  <?xml version="1.0" ?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema">
3    <element name="root" type="root" />
4    <complexType name="root">
5      <sequence>
6        <element name="child" type="child" />
7      </sequence>
8    </complexType>
9    <complexType name="child">
10     <sequence>
11       <element maxOccurs="unbounded" name="grandchild" type="string" />
12     </sequence>
13   </complexType>
14 </schema>
```

This XML Schema will only validate XML instances, which contain the `<root>` element as a root node. The nodes `<child>` and `<grandchild>` are no valid root nodes.

Therefore the following extension of the XML Schema type automaton definition is proposed:

Definition 11:
A XML Schema type automaton is a 5-tuple $D = (T, \rho, \tau, \sigma, \tau_\sigma)$ 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 $\rho(t)$; and the start expression σ, which is defined as the regular expression $a_1 + .. + a_n$ and a mapping τ_σ that assigns a type in T to each a occurring in σ.

The XML Schema validation algorithm is then extended to validate the root XML fragment against the start expression σ and then continues as described in section 2.4.1. This will be referenced later in the extensions of the XML Schema definitions learning algorithm again.

4.2. Regular expression learning enhancements

There are two major changes in the regular expression syntax available in XML Schema compared to Document Type Definition. In XML Schema it is possible to define counting patterns other then just ?, * and $^+$, but the analysis in chapter II showed that they are not used enough in practice to make the effort of modifying the regular expression algorithms to learn counting patterns.

Another syntactic feature available in XML Schema, but not in Document Type Definition, is the `<all>` syntax element. `<all>` allows to define a set of children with irrelevant order but a defined number of occurrences of those children. To keep the regular expression deterministic there are some limitations imposed by the XML Schema specification [36]. The `<all>` element may only occur at the top level of regular expressions and may not contain `<sequence>` or `<choice>` elements. We denote `<all>` by &. The syntax for regular expressions available in XML Schema therefore can be defined as:

$$
\begin{aligned}
s &:== \lambda|a|l|r \\
r &:== \lambda|a|r,r|r+r|r^*|r^+|r? \\
l &:== m|m?|m^*|m^+ \\
m &:== a|a\&m
\end{aligned}
$$

Each `<all>` expression can be translated into a common `<choice>` expression. The counting pattern applied to the `<all>` expression affects each element, so that $(a\&b\&c)^+$ indicates that each of the elements a, b and c must at least occur once in the matched string, while their order is irrelevant. Therefore the following translations of `<all>` patterns into common expression are valid:

$$
\begin{aligned}
(a\&b\&c) &:== (abc+acb+bac+bca+cab+cba) \\
(a\&b\&c)^+ &:== (((a+b+c)^*a^+(a+b+c)^*b^+(a+b+c)^*c^+(a+b+c)^*)+ \\
&\quad ((a+b+c)^*a^+(a+b+c)^*c^+(a+b+c)^*b^+(a+b+c)^*)+ \\
&\quad ((a+b+c)^*b^+(a+b+c)^*a^+(a+b+c)^*c^+(a+b+c)^*)+ \\
&\quad ((a+b+c)^*b^+(a+b+c)^*c^+(a+b+c)^*a^+(a+b+c)^*)+ \\
&\quad ((a+b+c)^*c^+(a+b+c)^*a^+(a+b+c)^*b^+(a+b+c)^*)+ \\
&\quad ((a+b+c)^*c^+(a+b+c)^*b^+(a+b+c)^*a^+(a+b+c)^*)) \\
(a\&b\&c)^* &:== (a+b+c)^*
\end{aligned}
$$

Since order is not always relevant in XML instances this obviously makes it easier to write XML Schema definitions which explicitly allow this. The translation of $(a\&b\&c)^*$ shows that inferring `<all>` is only relevant for counting patterns enforcing at least

one occurrence of every element in the expression. For the other cases a common `<choice>` pattern can be used to achieve the same effect.

4.2.1. Extending REWRITE

The REWRITE algorithm, like described in chapter 2, does not maintain any counting information for the elements found in the input strings and thus cannot be extended meaningfully to inference `<all>`.

For a set of input strings like $\{abc, acb, bac, bca, cab, cba\}$ the algorithm instead fails to inference a regular expression. Since it is known that REWRITE is not always able to return a regular expression for each set of input strings another algorithm, like CRX, should be used as a fallback algorithm anyways. Thus only the CRX algorithm has been extended to support `<all>` in this work.

4.2.2. Extending CRX

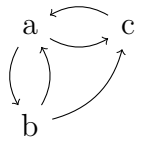
The CRX algorithm, as described in subsection 2.1.3, already maintains counting information about the elements occurring in the input strings and the extension to also learn `<all>` patterns is pretty straight forward. Now the algorithm eCRX is presented, which is an extension to the original CRX algorithm, additionally inferring `<all>` patterns.

Since `<all>` patterns may not occur inside other constructs, and no other constructs, like a `<sequence>` or `<choice>`, may occur inside an `<all>` pattern the following subset of regular expressions will be inferenced by this algorithm: The class of **extended Chain Regular Expressions** (eCHARE), which are CHAREs, or patterns following the form $(a_1\&..\&a_k)?$ where a_i is an element name, $k \geq 1$ and the ? optionally one of the available multipliers ?, $*$, or $+$.

Based on H_W, as defined in the original algorithm in subsection 2.1.3, `<all>` can obviously be inferenced if H_W only has one node, meaning all elements are equal regarding $\preceq_W$. The algorithm is then modified as shown in Algorithm 3.

Example

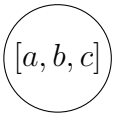
To illustrate the algorithm the set of input strings $W = \{abc, bac, cab\}$ is used. The graph induced by $\rightarrow_W$ then looks like:



The resulting calculated set of equivalency classes then looks trivial, like:

Algorithm 3 *eCRX*

Compute the set Γ_W of equivalence classes of $\approx_W$
if $|\Gamma_W| = 1 \wedge \gamma_1 = [a_1, .., a_n] | n > 1$ **then**
  if every string in W contains exactly one occurrence of a symbol in $\{a_1, .., a_n\}$ **then**
    $r(\gamma_1) = (a_1 \& .. \& a_n)$
  else if every string in W contains at most one occurrence of a symbol in $\{a_1, .., a_n\}$ **then**
    $r(\gamma_1) = (a_1 \& .. \& a_n)?$
  else if every string in W contains at least one of $a_1, .., a_n$ and there is a string that contains at least two occurrences of symbols **then**
    $r(\gamma_1) = (a_1 \& .. \& a_n)^+$
  else
    $r(\gamma_1) = (a_1 \& .. \& a_n)^*$
  end if
  return Γ_W
end if
while A maximum set of singleton nodes $\gamma_1, .., \gamma_n$, where $pred(\gamma_1) = .. = pred(\gamma_n)$ and $succ(\gamma_1) = .. = succ(\gamma_n)$ exists **do**
  Replace $\gamma_1, .., \gamma_n$ by $\gamma = \cup_{i=1}^{n} \gamma_i$ and redirect all incoming and outgoing edges from γ_1 to γ.
end while
Compute a topological sort $\gamma_1, .., \gamma_k$ of the nodes
for all $i \in \{1, .., k\}$ $(\gamma_i = [a_1, .., a_n])$ **do**
  if every string in W contains exactly one occurrence of a symbol in $\{a_1, .., a_n\}$ **then**
    $r(\gamma_i) = (a_1 + .. + a_n)$
  else if every string in W contains at most one occurrence of a symbol in $\{a_1, .., a_n\}$ **then**
    $r(\gamma_i) = (a_1 + .. + a_n)?$
  else if every string in W contains at least one of $a_1, .., a_n$ and there is a string that contains at least two occurrences of symbols **then**
    $r(\gamma_i) = (a_1 + .. + a_n)^+$
  else
    $r(\gamma_i) = (a_1 + .. + a_n)^*$
  end if
end for
return Γ_W



Since every element occurs exactly once in each of the input strings, the algorithm inferences the following regular expression:

$$a\&b\&c$$

4.3. Type merging

The typing mechanism in XML Schema is used for two different use cases. The first use case, described and handled in [7], is to use different types for elements with the same name. This is a major difference compared with Document Type Definition. The typing mechanism can also be used in a second way, to use the same type for elements with different names.

Re-using the same type for elements with different names is even more common in real-world XML Schema definitions then using different types for elements with the same name, like the analysis in Table 3.5.2 showed. One use case for this, for example, are markup languages like *Docbook* and *XHTML*, which allow different inline markup to be stacked recursively. In case of *XHTML* the inline markup like `<strong>`, `<em>` and `<a>` may again contain any inline markup, and thus all those elements have the same generic inline markup type assigned.

The algorithm presented in "Inferring XML Schema Definitions from XML Data" [7] only merges type definitions for elements with the same label. In this work a different algorithm will be presented which also merges types of elements with different labels.

During practical evaluation of the learning algorithm it showed that attributes of the elements also provide important information for type merging. The algorithm will include information about attributes and tries to enhance the type merging results based on that. As mentioned in [7] in practice it is not sufficient to check for equality of the type automatons in elements or for the same of attributes, thus different methods of comparing the type automatons and attribute lists will be presented, evaluated and compared.

4.3.1. The algorithm

The algorithm presented in [7] and described in subsection 2.4.3 uses $\text{reach}_D(s, t)$ for two types s and t in the schema D and compares the resulting set of types (s', t'). Since $reach_D$ operates on the labeled path p from the types s and t to reach the types s' and t' and checks them for equality or similarity only types for elements with the same label are merged.

The algorithm itself used in this paper is simpler by moving the responsibility for maintaining the type tree consistence to the comparators. The algorithm *Merge* is shown in Algorithm 4. It compares all types with each other using the function similar() which implements different comparison methods described later. If to types are similar enough they are merged. The merging itself (expressed by $\uplus$) again is slightly different from the algorithm described in [7], since it also needs to merge attribute definitions and simple type information.

One important constraint about similar() is, that it must not evaluate types as similar where the same label is used with different types. This would result in lost information about the type tree and the resulting schema might not validate the input XML anymore.

Algorithm 4 *Merge*

let $(T, \rho, \tau) = D$
set $changed = 1$
while $changed$ **do**
  set $changed = 0$
  for all $(s, t) \in T^2$ **do**
    if similar(s, t) **then**
      set soa$(s) := soa(s) \uplus soa(t)$
      set soa$(t) := soa(s)$
      for all $a \in \text{elems}_D(t) - \text{elems}_D(s)$ **do**
        add $(s, a) \to \tau(t, a)$ to τ
      end for
      for all $a \in \text{elems}_D(s) - \text{elems}_D(t)$ **do**
        add $(t, a) \to \tau(s, a)$ to τ
      end for
      $changed = 1$
    end if
  end for
end while
for all type $t \in T$ **do**
  replace each $t \to \rho(t)$ in ρ by $t \to$ TOSORE(soa(t))
end for

Example

Without a concrete definition of similar() and ignoring attributes the execution of the algorithm is shown as an example for the following XML document:

```
1 <?xml version="1.0" ?>
2 <store>
```

```
3     <sale>
4       <category>
5         <item/>
6         <item/>
7         <item/>
8       </category>
9     </sale>
10    <warehouse>
11      <category>
12        <item/>
13        <item/>
14      </category>
15    </warehouse>
16  </store>
```

Using the unmerged 2-local SOXSD, as described in section 2.4.1, the input schema D for *Merge* would look like:

$$
\begin{aligned}
(\lambda, \texttt{<store>}) &\rightarrow \texttt{<sale>}sale, \texttt{<warehouse>}warehouse \\
(\texttt{<store>}, \texttt{<sale>}) &\rightarrow \texttt{<category>}category_1 \\
(\texttt{<store>}, \texttt{<warehouse>}) &\rightarrow \texttt{<category>}category_2 \\
(\texttt{<sale>}, \texttt{<category>}) &\rightarrow (\texttt{<item>}item)^+ \\
(\texttt{<warehouse>}, \texttt{<category>}) &\rightarrow (\texttt{<item>}item)^+ \\
(\texttt{<category>}, \texttt{<item>}) &\rightarrow \lambda
\end{aligned}
$$

If the two types *warehouse* and *sale* are compared in an early iteration of the *Merge* algorithm they must be considered as dissimilar since the labels in the SOA are associated with the different types $category_1$ and $category_2$.

Once the two types $category_1$ and $category_2$ are compared they would be merged by the algorithm, since they contain the same SOA and their labels point to the same type *item*. The algorithm would then set the *change* flag to 1 and reevaluate the types in D. Eventually it will compare *warehouse* and *store* again, which now point to the same type $category_1$ and can be merged. This will result in the intended schema:

$$
\begin{aligned}
store &\rightarrow \texttt{<sale>}sale, \texttt{<warehouse>}sale \\
sale &\rightarrow \texttt{<category>}category_1 \\
category_1 &\rightarrow (\texttt{<item>}item)^+ \\
item &\rightarrow \lambda
\end{aligned}
$$

Importance of type checking

The importance of checking that the labels in the compared content models are associated with the same type can be shown with the following 3-local SOXSD as example

input:

$$
\begin{aligned}
(\lambda, \lambda, \texttt{<store>}) &\rightarrow \texttt{<sale>}sale, \texttt{<warehouse>}warehouse \\
(\lambda, \texttt{<store>}, \texttt{<sale>}) &\rightarrow \texttt{<category>}category_1 \\
(\lambda, \texttt{<store>}, \texttt{<warehouse>}) &\rightarrow \texttt{<category>}category_2 \\
(\texttt{<store>}, \texttt{<sale>}, \texttt{<category>}) &\rightarrow (\texttt{<item>}item_1)^+ \\
(\texttt{<store>}, \texttt{<warehouse>}, \texttt{<category>}) &\rightarrow (\texttt{<item>}item_2)^+ \\
(\texttt{<sale>}, \texttt{<category>}, \texttt{<item>}) &\rightarrow (\texttt{<name>}name, \texttt{<price>}price) \\
(\texttt{<warehouse>}, \texttt{<category>}, \texttt{<item>}) &\rightarrow (\texttt{<name>}name, \texttt{<count>}count) \\
(\texttt{<category>}, \texttt{<item>}, \texttt{<name>}) &\rightarrow \lambda \\
(\texttt{<category>}, \texttt{<item>}, \texttt{<price>}) &\rightarrow \lambda \\
(\texttt{<category>}, \texttt{<item>}, \texttt{<count>}) &\rightarrow \lambda
\end{aligned}
$$

Note that this example contains two different definitions for the <item> elements aggregated in the <category> elements, which depend on their grandparents.

Assuming the algorithm now starts by comparing the two types *sale* and *warehouse* the SOAs are the same, but the labels are associated with two different types. If those now would be merged, both would, for example, point to $category_1$ and the information about $item_2$ and its <count> child would have been lost and the XML, which lead to the 3-local SOXSD, could not be validated any more with the resulting schema, since the <count> element would be rejected during the validation.

The only elements which can be merged in this example are the empty elements <name>, <price> and <count>. The schema resulting from the *Merge* algorithm therefore would look like:

$$
\begin{aligned}
store &\rightarrow \texttt{<sale>}sale, \texttt{<warehouse>}warehouse \\
sale &\rightarrow \texttt{<category>}category_1 \\
warehouse &\rightarrow \texttt{<category>}category_2 \\
category_1 &\rightarrow (\texttt{<item>}item_1)^+ \\
category_2 &\rightarrow (\texttt{<item>}item_2)^+ \\
item_1 &\rightarrow (\texttt{<name>}name, \texttt{<price>}name) \\
item_2 &\rightarrow (\texttt{<name>}name, \texttt{<count>}name) \\
name &\rightarrow \lambda
\end{aligned}
$$

4.3.2. Implementation of similar()

similar() is intended to compare attributes and the SOA defining a type. Attributes in XML Schema are defined as unordered set [36] and there is no way that the child el-

ements depend on attributes, as opposed to *RelaxNG* [12]. In *RelaxNG* the attributes are part of the regular expression defining the content model, but this work focusses on XML Schema. Therefore the types maintain a set of attributes, which occur in the given type and maintain information about each attribute if it is optional or required.

Since attributes and the regular expression are independent two different comparators, one for the regular expression and one for the attribute sets are used by similar(). Assuming that elements with the same label are more likely to share the same type optionally different attribute or element comparators can be used for elements with the same label. The different pattern and attribute comparators are explained in the following two subsections.

4.3.3. Attribute comparison

During the evaluation of the algorithms against real world XML data and the manual introspection of the resulting XML Schema definitions it has been noticed that a lot of false positive merges occurred because of ignoring attributes in the input XML documents. This was especially obvious when learning the schema for one specific *XHTML* based website.

XHTML contains a header and a body section. The header defines metadata like the document title and author, style information and references to scripts. Most of those elements do not contain any children but very different sets of attributes. Additionally there are also various other elements not containing children, but different attributes in the *XHTML* body, like `<img>` elements. Ignoring the attributes this lead to very unspecific results like the following type, which is then used by all elements without any children:

```
1  <?xml version="1.0" ?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema">
3    <!-- ... -->
4    <complexType name="html/head/meta">
5      <attribute name="http-equiv" type="string"/>
6      <attribute name="content" type="string"/>
7      <attribute name="rel" type="string"/>
8      <attribute name="type" type="string"/>
9      <attribute name="href" type="string"/>
10     <attribute name="media" type="string"/>
11     <attribute name="src" type="string"/>
12     <attribute name="class" type="string"/>
13     <attribute name="value" type="string"/>
14     <attribute name="selected" type="string"/>
15     <attribute name="name" type="string"/>
16     <attribute name="width" type="string"/>
17     <attribute name="height" type="string"/>
18     <attribute name="alt" type="string"/>
19     <attribute name="title" type="string"/>
20     <attribute name="colspan" type="string"/>
```

```
21      <attribute name="disabled" type="string"/>
22      <attribute name="checked" type="string"/>
23      <attribute name="onclick" type="string"/>
24    </complexType>
25    <!-- ... -->
26  </schema>
```

To be able to evaluate the algorithms against real world XML data the attributes had to be taken into account while merging the types. To compare the equality of attribute definitions various comparators where defined and evaluated later.

Since the default state for attributes is optional, the example above shows all attributes as optional, while, for example, the `<img>` element requires the `src` and `alt` attributes and all input documents specify those. This is the logical result of false positive merges.

Comparator: Strict

The *Strict* attribute set comparator is the most strict one. It evaluates two sets of attributes as similar if both sets contain the same attributes and the same subsets of attributes are flagged as optional. For two types s and t, if $a(s)$ denotes the set of all attribute names associated with this type and $oa(s)$ the set of optional attributes the *Strict* comparator is evaluated as:

$$a(s) = a(t) \wedge ao(s) = ao(t)$$

Comparator: Same

The *Same* attribute set comparator is less strict then the *Strict* comparator and ignores the optional flag in the attribute sets. It just checks that both types contain the same set of attributes and therefore evaluated as:

$$a(s) = a(t)$$

Comparator: Equal

The *Equal* attribute set comparator is less strict then the *Same* comparator. It checks that at least all required attributes are available in both attribute sets:

$$a(s) - ao(s) \subseteq a(t) \wedge a(t) - ao(t) \subseteq a(s)$$

Comparator: Merge

The *Merge* attribute set comparator is the least strict comparator and just considers all attribute sets as equal. It therefore mimics the behaviour of ignoring attributes during the comparison of types.

Merging attribute sets

If two types s and t are considered similar by similar() their attribute sets need to be merged. Attributes occurring only in $a(s)$ or $a(t)$ are flagged as optional in the resulting attribute set. Attributes flagged as optional in either s or t are also flagged as optional in the resulting type s'. Only attributes flagged as required in both types s and t are also flagged as required in s'. Therefore the following applies:

$$\begin{aligned} a(s') &= a(s) \cup a(t) \\ ao(s') &= ao(s) \cup ao(t) \cup (a(s) \cap a(t)) \end{aligned}$$

4.3.4. Element comparison

For the comparison of the regular expressions (element patterns) contained in the content model definitions one algorithm has already been introduced in [7]. This work introduces other comparators for the element patterns, which will later be compared during the evaluation.

Comparator: Reduce

As introduced in [7] in the *REDUCE* algorithm and explained in subsection 2.4.3 this algorithm calculates a distance between two patterns, as described by their SOAs. In this work the direct distance between two automatons is used, and not the maximum distance in the current type tree, since the *Merge* algorithm operates locally on each type and maintains the integrity of the type tree by other means. Therefore the following distance definition is used between the support-annotated SOAs $A = (V, E)$ and $B = (W, F)$:

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

Two element patterns are considered similar if the distance is below a defined threshold. For valid threshold example values see subsection 2.4.3 or the source paper. [7]

Comparator: Equal

This *Equal* element pattern comparator considers types similar, if the two automatons of the compared types $A = (V, E)$ and $B = (W, F)$ are equal:

$$V = W \wedge E = F$$

Comparator: Subsumed

The *Subsumed* element pattern comparator considers types similar, if one of the automatons subsumes the other automaton:

$$(V \subseteq W \wedge E \subseteq F) \vee (W \subseteq V \wedge F \subseteq E)$$

Comparator: Node-Based

The *Node-Based* element pattern comparator considers types similar, if the same nodes are contained in both automatons, ignoring the vertices between those nodes:

$$V = W$$

Comparator: Node-Subsumed

The *Node-Subsumed* element pattern comparator considers types similar, if the nodes in one of the automatons subsume the nodes in the other automaton:

$$V \subseteq W \vee W \subseteq V$$

Merging element patterns

If two types s and t are considered similar by similar() their element patterns need to be merged. This is done by adding all nodes and vertices from one automaton to the other. Merging the two SOAs $soa(s) = A = (V, E)$ and $soa(t) = B = (W, F)$ thus will result in $C = (X, G)$:

$$X = V \cup W$$
$$G = E \cup F$$

In case of a support-annotated SOA as required for the *Reduce* comparator the support will be maintained as described in [7]:

$$\text{supp}_C(a, b) := \text{supp}_A(a, b) + \text{supp}_B(a, b)$$

Where $\text{supp}_A(a, b) = 0$ is assumed for simplicity, if there is no such edge in A, and similarly for B.

4.4. Evaluation

Evaluating the schema learning algorithms is crucial but very hard. There exist large sets of XML instances, partially associated with schemas. But one cannot expect to learn the exact same schema when learning from a set of XML documents, since schema definitions are often less specific then they could be [6] or the input data is to sparse to learn the real schema behind the data. Checking if one schema subsumes another schema is still topic of current research and the author of this work is not aware of any algorithm which can verify this.

Evaluation on real world data can only happen by manual introspection, which is obviously tedious given the large amount of existing XML instances and schemas.

Therefore one has to look for other ways of automatic verification of the learning algorithms and type comparators. In "Inferring XML Schema Definitions from XML Data" [7] the authors use one XML instance and manually check the false positive and false negative merges for different configurations of their algorithm. The XML instance used in that paper demonstrates one aspect of type merging which was not handled before by other work in this area.

For this work the algorithms were checked manually against different sets of XML instances and the results were continually introspected manually. Based on that different XML instances were defined which show demonstrate certain aspects of schema learning and prepared for automatic verification. In this section the source datasets, the extracted examples and the verification methodology will be described.

4.4.1. Source datasets

As source datasets for manual verification of the algorithms and extraction of common structures for automatic verification sets of valid XML instances with known structures were required. It is important to know the expected structure beforehand, so that it is possible to verify the results and detect inconsistencies in the inferred schema.

As a base for the extracted examples the following XML corpus were used:

Docbook
: A set of Docbook [15] documents was used, generated by an application using a well-known, documented subset of Docbook. Docbook is a popular XML-based text-markup format using recursive structures and a lot elements share the same type.

XHTML
: XHTML (HTML) [37] is probably the most popular XML based language and due to its really well known structure it is easy to manually detect false positive and false negative merges in the resulting schema. For the manual introspection

the valid XHTML pages from one web application were used as input. This lead to an application specific XHTML subset being learned, but also provided valid input, like mentioned in subsection 4.3.2.

dblp
The *dblp Computer Science Bibliography* [25] provides a very large set of XML documents, following a well known Document Type Definition. The data has been used for schema learning before [6], which already showed that the schema definition could be stricter then the available schema is. The schema does not require the features of XML Schema and therefore provides a good way to verify that the XML Schema inference algorithm still works fine with this kind of data. Additionally the data set is quite large, so that it is safe to assume all intended variants of the schema are already used.

Webfrap
Webfrap is an application build by SAP, intended to provide a framework for Model Driven Development. For this the models are specified in quite complex and domain specific XML documents, which do not yet have a schema assigned. Together with the developers of Webfrap this enabled verification of the algorithms against a large set of complex XML documents.

4.4.2. Extracted examples

From the mentioned manually analyzed XML corpus several examples were extracted and prepared for automatic evaluation. The motivation behind those examples is explained in this section. The XML source of the examples can be found in the appendix in Part IV.

Store
This example is the XML source code used in "Inferring XML Schema Definitions from XML Data" [7] and used for evaluation of the algorithm in that paper.

Empty Types
The *Empty Types* example shows the case of multiple different empty elements, which can only be told apart by their attributes as found commonly in HTML documents. This example models the case described in subsection 4.3.2.

Attributes
The *Attributes* example models the case of different attribute sets for the same element in different subtrees of the document. This was a common case in all test corpus, except for the dblp corpus. An example for this is an XHTML `<img>` element, which contains a `title` attribute only in one subsection of the

site, or a Docbook `<para>` element, which only contains `class` attributes in some sections.

Ancestor Depth
Since the analysis showed an ancestor depth greater then 2 is not uncommon the *Ancestor Depth* example models a XML document, which requires at least a 3-local schema for correct schema detection.

Reoccurrent
A very common case in all XML corpus, except for dblp, was the same element occurring in different subtrees with slightly different sets of children. This es especially true for languages like Docbook and XHTML, where large sets of *inline elements* exist, which may be used in paragraphs. With incomplete corpus this often leads to false type specialization.

4.4.3. Verification

To verify the results of the type merging process each element was annotated with the expected type. This was done using an attribute in a custom namespace. After the type merging one can now check automatically if elements with the same annotated type were not merged successfully, or if two elements with different annotated types were merged. The *Empty Types* example shows how these annotations look like:

```
1 <html xmlns:sl="http://example.com/SchemaLearning" sl:type="html">
2   <head sl:type="head">
3     <meta name="keywords" value="key, word" sl:type="meta" />
4     <meta name="description" value="This is a description" sl:type="meta
          " />
5     <title sl:type="title">Title</title>
6     <script type="text/ecmascript" src="scripts.js" sl:type="script" />
7   </head>
8 </html>
```

The attributes in the custom namespace *http://example.com/SchemaLearning* are used solely for this evaluation. The `type` attribute specifies the type for each element in the example.

To verify the accuracy of the result the RandIndex [33] metric was used. RandIndex is a metric originally developed to check the accuracy of clustering algorithms and returns a value between 0 and 1 describing the success of the clustering.

The assignment of elements to types can be considered as clustering the elements in the source schema into an expected set of clusters (types). For a set of elements $S = \{e_1, .., e_2\}$ and two partitions of S to compare, $X = \{x_1, .., x_r\}$ and $Y = \{y_1, .., y_s\}$ the following is defined:

$$
\begin{aligned}
a &= |S^*|, \text{where} S^* = \{(o_i, o_j) | o_i, o_j \in X_k, o_i, o_j \in Y_l\} \\
b &= |S^*|, \text{where} S^* = \{(o_i, o_j) | o_i \in X_{k_1}, o_j \in X_{k_2}, o_i \in Y_{l_1}, o_j \in Y_{l_2}\} \\
c &= |S^*|, \text{where} S^* = \{(o_i, o_j) | o_i, o_j \in X_k, o_i \in Y_{l_1}, o_j \in Y_{l_2}\} \\
d &= |S^*|, \text{where} S^* = \{(o_i, o_j) | o_i \in X_{k_1}, o_j \in X_{k_2}, o_i, o_j \in Y_l\}
\end{aligned}
$$

Intuitively a and b are the items both clusters agree on, and c and d are the items the clusters disagree on. The index R is now defined as:

$$R = \frac{a + b}{a + b + c + d}$$

R results in values between 0 and 1, where 1 means that the clusters are entirely the same, and 0 indicating that there are no similarities between the clusterings.

4.4.4. Experiment settings

The evaluation was run for all the mentioned examples, with different configurations of the *Merge* algorithm.

First the input SOXSD are calculated with a different locality. For the evaluation the following localities were evaluated:

- 1-local (like Document Type Definitions), 2-local, 3-local, 4-local
- Full path

  This always uses the full path to the element, without limiting it to any specified length, like defined for k-local SOXSD.

As described in section 4.3.1 the type comparison function similar() can be configured in different ways. Beside setting an attribute and element pattern comparator it is possible to define another attribute and pattern comparator for elements with the same label. For the evaluation the comparators were naively ordered by their permissiveness. Each comparator was used as it is for elements with the same label and elements with different labels. Each comparator was also used with each more permissive comparator for elements with the same label.

The *Reduce* comparator additionally allows custom thresholds values. It was used with thresholds of 0.05, 0.1, 0.25 and 0.5, which also were used in the original paper introducing the algorithm. [7]

A full matrix of the different comparators and their combinations used in the experiment can be found in the full result in the appendix in Section 4.6.5.

Locality	Accuracy
1-local	0.9162
2-local	0.9280
3-local	0.9247
4-local	0.9155
Full Path	0.9155

Table 4.1.: Ancestor depth

4.4.5. Evaluation results

The full evaluation results can be found in 4.6.5. The result value distribution is small. The best results (0.946) are accomplished using a 2-local SOXSD as input with the *Node-Based* element pattern comparator for elements with different labels and the *Node-Subsumed* element pattern comparator for elements with the same label with different combinations of attribute set comparators.

Even with the low general variety of values there are several conclusions one can draw from the results:

Ancestor depth

The results showed that using a high value for k in the input k-local SOXSD quickly leads to false negative merges, which reduces the quality of the resulting schema. In table Table 4.1 the results show that the best results are accomplished with a 2-local SOXSD even one of the five evaluation examples explicitly requires a 3-local SOXSD to be learned correctly.

Label based merging

The examples contain examples of elements with different labels which are intended to be merged and elements with the same label which are not intended to be merged. Still from the results it is obvious that using a more permissive element pattern comparator for elements with the same label results in better accuracy – the best results are accomplished using the very permissive *Node-Subsumed* (section 4.3.4) element pattern comparator for elements with the same label. Using the *Node-Subsumed* element pattern comparator also for nodes with different labels then again results in very poor accuracy ratings.

Attribute merging

Table 4.2 shows the results grouped by the different attribute set comparators used in the evaluation. Ignoring the attribute sets (*Merge*) lead to the worst results, and using a more permissive attribute set comparator for elements with the same label

Attribute pattern comparator	Accuracy
Strict	0.9221
Strict + Same	0.9221
Strict + Equal	0.9239
Strict + Merge	0.9239
Same	0.9221
Same + Equal	0.9239
Same + Merge	0.9239
Equal	0.9195
Equal + Merge	0.9195
Merge	0.8987

Table 4.2.: Attribute comparators

again results in slightly better results. The best results are accomplished using the *Strict* or *Same* comparator for elements with different labels and using *Equal* or *Merge* for elements with the same label.

Pattern merging

There are no clear advantages for any of the pattern merging algorithms itself. Choosing the correct pattern merging algorithm generally depends on the completeness of the XML corpus which is analyzed. For a more complete corpus a less permissive algorithm will obviously provide better results, since the number of false positive merges will be reduced.

A more detailed analysis with sets of existing real-world XML instances, which do not yet define a schema yet, could provide more detailed insight here. But this requires either domain experts for the input XML, who can verify the results or other algorithmic ways to automatically judge schema quality. Once algorithms are developed which can test if one schema subsumes another schema one could also perform this analysis on sets of XML instances with already defined schemas – but this is still topic of current research.

4.5. Software

To perform the analysis a software had to be written which implements all the algorithms mentioned in this work. The software is available at `https://github.com/kore/XML-Schema-learner` and licensed under GPL 3 [16].

The software offers a simple Command Line Tool, which can be used to inference schemas from XML data:

```
$ ./learn --help
Schema Learner
by Kore Nordmann

Usage: ./learn [-t <type>] <xml-files>

General options:

-t / --type      Type of the schema to generate. Currently implemented schema
                 languages: dtd, xsd, bonxai

-h / --help      Display this help output

XML Schema / BonXai specific options:

-l / --locality Locality of the types when inferencing XML Schema schemata.
                 Valid values are integer numbers, or "n".

-a / --attrComp Algorithm used to compare attributes in the type merger.
                 Available algorithms are "strict", "same", "equals", "merge".
                 Defaults to "equals".

-p / --ptrnComp Algorithm used to compare patterns in the type merger. Available
                 algorithms are "exact", "reduce", "node-based", "subsumed" and
                 "node-subsumed". If not specified, no types will be merged.

--snAttrComp     Attribute comparator used for elements with the same name.
                 Only need to be specified if it differs from the attrComp.

--snPtrnComp     Pattern comparator used for elements with the same name. Only
                 need to be specified if it differs from the ptrnComp.
```

The software is written in a clean object-oriented way and can be extended by additional regular expression inferencing algorithms, type merging algorithms and also produce other schema languages. For now it implements learning Document Type Definitions [35], XML Schema definitions [36] and Bonxai schemas [12].

The algorithm used for type merging can be selected using shell Command Line arguments and special comparators can be specified for elements with the same label, as discussed in section 4.3.4.

Describing the software class diagram or the implementation in detail would exceed the scope of this thesis.

Part III.

Outlook

Outlook

The software described in section 4.4.5 already provides users with a way to infer schemas, which can then be used to validate further documents, based on the algorithms described and developed in this work.

As mentioned in section 4.3.4 during the evaluation further tests of the type comparators with real world XML data could provide further insight in which situations which of the element pattern comparators are best.

This requires either working together with domain experts for the processed XML corpus to manually verify the quality of the inferred schema and extract more use cases for automatic verification or ways to algorithmically judge the quality of the inferred schemas. One way to automatically verify the inferred schemas would be checking if the inferred schema is subsumed by existing schemas for an analyzed XML corpus. Testing for schema subsumption is still topic of current research.

A useful way to extend the software is adding means for simple type inference. Inferencing simple types from XML instances has already been researched [21] and is just a matter of extending the software, which already provides the necessary extension points.

This work provides the means to infer XML Schema definitions. It has been shown that the capabilities of RelaxNG are a strict superset of XML Schema [19]. Therefore the software could be extended to also generate RelaxNG schemas, without using the additional features. Additionally the algorithms can be extended to also infer schemas using the full contextual power of RelaxNG, which especially means that attributes would be part of the regular expressions describing the content model of elements.

RelaxNG also has the capability to define schemas for multiple namespace in one schema definition file, while XML Schema and Document Type Definition require multiple schema definition files for that. Since namespaces do not structurally change anything when inferencing schemas from XML data, namespaces are currently ignored. A valid extension to the software would be to maintain namespace information of elements and attributes and use them when outputting the schemas. In case of XML Schema or Document Type Definition this could result in multiple schemas with cross-references.

Part IV.

Appendix

4.6. XML Source code examples

4.6.1. Store

```
<store xmlns:sl="http://example.com/SchemaLearning" sl:type="store">
  <order sl:type="order">
    <customer sl:type="customer">
      <name sl:type="textfield">John Mitchell</name>
      <email sl:type="textfield"> j.mitchell@example.com </email>
    </customer>
    <item sl:type="oder_item">
      <id sl:type="textfield"> I18F </id>
      <price sl:type="textfield"> 100 </price>
    </item>
  </order>
  <stock sl:type="stock">
    <item sl:type="stock_item">
      <id sl:type="textfield"> IG8 </id>
      <qty sl:type="textfield"> 10 </qty>
      <supplier sl:type="supplier">
        <name sl:type="textfield"> Al Jones </name>
        <email sl:type="textfield"> a.j@example.net </email>
        <email sl:type="textfield"> a.j@example.org </email>
      </supplier>
    </item>
    <item sl:type="stock_item">
      <id sl:type="textfield"> J38H </id>
      <qty sl:type="textfield"> 30 </qty>
      <item sl:type="stock_item">
        <id sl:type="textfield"> J38H1 </id>
        <qty sl:type="textfield"> 10 </qty>
        <supplier sl:type="supplier">
          <name sl:type="textfield"> Al Jones </name>
          <email sl:type="textfield"> a.j@example.org </email>
        </supplier>
      </item>
      <item sl:type="stock_item">
        <id sl:type="textfield"> J38H2 </id>
        <qty sl:type="textfield"> 1 </qty>
        <supplier sl:type="supplier">
          <name sl:type="textfield"> Al Jones </name>
          <email sl:type="textfield"> a.j@example.org </email>
        </supplier>
      </item>
    </item>
  </stock>
</store>
```

4.6.2. Empty Types

```
<html xmlns:sl="http://example.com/SchemaLearning" sl:type="html">
  <head sl:type="head">
    <meta name="keywords" value="key, word" sl:type="meta" />
    <meta name="description" value="This is a description" sl:type="meta
        " />
    <title sl:type="title">Title</title>
    <script type="text/ecmascript" src="scripts.js" sl:type="script" />
  </head>
</html>
```

4.6.3. Attributes

```
<html xmlns:sl="http://example.com/SchemaLearning" sl:type="html">
  <body sl:type="body">
    <h1 sl:type="header">Top level header</h1>
    <img sl:type="image" src="foo.png" alt="foo" />
    <h2 sl:type="header">Second level header</h2>
    <p sl:type="para">Some text</p>
    <ul sl:type="ul">
      <li sl:type="li" class="first">
        <p sl:type="para" class="note">Note...</p>
        <img sl:type="image" src="foo.png" alt="foo" />
      </li>
      <li sl:type="li">
        <img sl:type="image" src="foo.png" alt="foo" width="16" height="
            16" />
        <p sl:type="para">Note...</p>
      </li>
    </ul>
    <p>And another para with an inline image: <img sl:type="image" src="
        foo.png" alt="foo" title="Inline image" /></p>
  </body>
</html>
```

4.6.4. Ancestor Depth

```
<store xmlns:sl="http://example.com/SchemaLearning" sl:type="store">
  <order sl:type="order">
    <category sl:type="order_category">
      <name sl:type="string">Audi</name>
      <car sl:type="order_car"><type sl:type="string">A 4</type></car>
      <car sl:type="order_car"><type sl:type="string">A 6</type></car>
      <car sl:type="order_car"><type sl:type="string">A 8</type></car>
    </category>
    <category sl:type="order_category">
      <name sl:type="string">BMW</name>
      <car sl:type="order_car"><type sl:type="string">X3</type></car>
      <car sl:type="order_car"><type sl:type="string">X4</type></car>
      <car sl:type="order_car"><type sl:type="string">525</type></car>
    </category>
```

```
15    </order>
16    <stock sl:type="stock">
17      <category sl:type="stock_category">
18        <name sl:type="string">Audi</name>
19        <car sl:type="stock_car">
20          <type sl:type="string">A 4</type>
21          <price sl:type="string">34958 EUR</price>
22        </car>
23        <car sl:type="stock_car">
24          <type sl:type="string">A 6</type>
25          <price sl:type="string">54958 EUR</price>
26        </car>
27      </category>
28      <category sl:type="stock_category">
29        <name sl:type="string">BMW</name>
30        <car sl:type="stock_car">
31          <type sl:type="string">X4</type>
32          <price sl:type="string">34958 EUR</price>
33        </car>
34        <car sl:type="stock_car">
35          <type sl:type="string">525</type>
36          <price sl:type="string">54958 EUR</price>
37        </car>
38      </category>
39    </stock>
40  </store>
```

4.6.5. Reoccurent

```
1   <store xmlns:sl="http://example.com/SchemaLearning" sl:type="store">
2     <order sl:type="order">
3       <customer sl:type="customer">
4         <name sl:type="textfield">John Mitchell</name>
5         <email sl:type="textfield"> j.mitchell@example.com </email>
6       </customer>
7       <item sl:type="oder_item">
8         <id sl:type="textfield"> I18F </id>
9         <price sl:type="textfield"> 100 </price>
10      </item>
11    </order>
12    <stock sl:type="stock">
13      <item sl:type="stock_item">
14        <id sl:type="textfield"> IG8 </id>
15        <qty sl:type="textfield"> 10 </qty>
16        <supplier sl:type="supplier">
17          <name sl:type="textfield"> Al Jones </name>
18          <email sl:type="textfield"> a.j@example.net </email>
19          <email sl:type="textfield"> a.j@example.org </email>
20        </supplier>
21      </item>
```

```
22      <item sl:type="stock_item">
23        <id sl:type="textfield"> J38H </id>
24        <qty sl:type="textfield"> 30 </qty>
25        <item sl:type="stock_item">
26          <id sl:type="textfield"> J38H1 </id>
27          <qty sl:type="textfield"> 10 </qty>
28          <supplier sl:type="supplier">
29            <name sl:type="textfield"> Al Jones </name>
30            <email sl:type="textfield"> a.j@example.org </email>
31          </supplier>
32        </item>
33        <item sl:type="stock_item">
34          <id sl:type="textfield"> J38H2 </id>
35          <qty sl:type="textfield"> 1 </qty>
36          <supplier sl:type="supplier">
37            <name sl:type="textfield"> Al Jones </name>
38            <email sl:type="textfield"> a.j@example.org </email>
39          </supplier>
40        </item>
41      </item>
42    </stock>
43  </store>
```

Pattern comparator	Strict	Strict + Same	Strict + Equal	Strict + Merge	Same	Same + Equal	Same + Merge	Equal	Equal + Merge	Merge
Exact	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.904
Exact + Reduce(.05)	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.904
Exact + Reduce(.1)	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.904
Exact + Reduce(.25)	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.904
Exact + Reduce(.5)	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.904
Exact + Node-Based	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.904
Exact + Subsumed:	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.904
Exact + Node-Subsumed:	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.904
Reduce(.05):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.05) + Reduce(.1):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.05) + Reduce(.25):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.05) + Reduce(.5):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.05) + Node-Based:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.05) + Subsumed:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.05) + Node-Subsumed:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.1):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.1) + Reduce(.25):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.1) + Reduce(.5):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.1) + Node-Based:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.1) + Subsumed:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.1) + Node-Subsumed:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.25):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.25) + Reduce(.5):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.25) + Node-Based:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.25) + Subsumed:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.25) + Node-Subsumed:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.5):	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.5) + Node-Based:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.5) + Subsumed:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Reduce(.5) + Node-Subsumed:	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.922	0.902
Node-Based:	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.907
Node-Based + Subsumed:	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.907
Node-Based + Node-Subsumed:	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.927	0.907
Subsumed:	0.881	0.881	0.881	0.881	0.881	0.881	0.881	0.846	0.846	0.816
Subsumed + Node-Subsumed:	0.881	0.881	0.881	0.881	0.881	0.881	0.881	0.846	0.846	0.816
Node-Subsumed:	0.879	0.879	0.879	0.879	0.879	0.879	0.879	0.827	0.827	0.797

Table 4.3.: Results for 1-local

4.7. Evaluation results

Results from the evaluation of the type merging algorithms with the examples shown in Part IV.

If only one comparator is shown in a table row or column it means the same comparator has been used for elements with and without the same labels. If two algorithms are mentioned like "Node-Based + Subsumed" it means that the first algorithm ("Node-Based") has been used for elements with different labels, and the second ("Subsumed") for elements with the same label.

The best values are highlighted as **bold text** and can be found in Table 4.4.

Pattern comparator	Strict	Strict + Same	Strict + Equal	Strict + Merge	Same	Same + Equal	Same + Merge	Equal	Equal + Merge	Merge
Exact:	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Exact + Reduce(.05):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Exact + Reduce(.1):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Exact + Reduce(.25):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Exact + Reduce(.5):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Exact + Node-Based:	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Exact + Subsumed:	0.930	0.930	0.933	0.933	0.930	0.933	0.933	0.933	0.933	0.913
Exact + Node-Subsumed:	0.940	0.940	0.943	0.943	0.940	0.943	0.943	0.943	0.943	0.923
Reduce(.05):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.05) + Reduce(.1):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.05) + Reduce(.25):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.05) + Reduce(.5):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.05) + Node-Based:	0.930	0.930	0.931	0.931	0.930	0.931	0.931	0.928	0.928	0.908
Reduce(.05) + Subsumed:	0.932	0.932	0.936	0.936	0.932	0.936	0.936	0.936	0.936	0.916
Reduce(.05) + Node-Subsumed:	0.939	0.939	0.943	0.943	0.939	0.943	0.943	0.943	0.943	0.923
Reduce(.1):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.1) + Reduce(.25):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.1) + Reduce(.5):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.1) + Node-Based:	0.930	0.930	0.931	0.931	0.930	0.931	0.931	0.928	0.928	0.908
Reduce(.1) + Subsumed:	0.932	0.932	0.936	0.936	0.932	0.936	0.936	0.936	0.936	0.916
Reduce(.1) + Node-Subsumed:	0.939	0.939	0.943	0.943	0.939	0.943	0.943	0.943	0.943	0.923
Reduce(.25):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.25) + Reduce(.5):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.25) + Node-Based:	0.930	0.930	0.931	0.931	0.930	0.931	0.931	0.928	0.928	0.908
Reduce(.25) + Subsumed:	0.932	0.932	0.936	0.936	0.932	0.936	0.936	0.936	0.936	0.916
Reduce(.25) + Node-Subsumed:	0.939	0.939	0.943	0.943	0.939	0.943	0.943	0.943	0.943	0.923
Reduce(.5):	0.932	0.932	0.934	0.934	0.932	0.934	0.934	0.931	0.931	0.911
Reduce(.5) + Node-Based:	0.930	0.930	0.931	0.931	0.930	0.931	0.931	0.928	0.928	0.908
Reduce(.5) + Subsumed:	0.932	0.932	0.936	0.936	0.932	0.936	0.936	0.936	0.936	0.916
Reduce(.5) + Node-Subsumed:	0.939	0.939	0.943	0.943	0.939	0.943	0.943	0.943	0.943	0.923
Node-Based:	0.926	0.926	0.928	0.928	0.926	0.928	0.928	0.924	0.924	0.904
Node-Based + Subsumed:	0.929	0.929	0.932	0.932	0.929	0.932	0.932	0.932	0.932	0.912
Node-Based + Node-Subsumed:	0.942	0.942	**0.946**	**0.946**	0.942	**0.946**	**0.946**	**0.946**	**0.946**	0.926
Subsumed:	0.901	0.901	0.901	0.901	0.901	0.901	0.901	0.874	0.874	0.844
Subsumed + Node-Subsumed:	0.911	0.911	0.911	0.911	0.911	0.911	0.911	0.884	0.884	0.854
Node-Subsumed:	0.909	0.909	0.909	0.909	0.909	0.909	0.909	0.873	0.873	0.843

Table 4.4.: Results for 2-local

Pattern comparator	Strict	Strict + Same	Strict + Equal	Strict + Merge	Same	Same + Equal	Same + Merge	Equal	Equal + Merge	Merge
Exact:	0.926	0.926	0.928	0.928	0.926	0.928	0.928	0.925	0.925	0.905
Exact + Reduce(.05):	0.926	0.926	0.928	0.928	0.926	0.928	0.928	0.925	0.925	0.905
Exact + Reduce(.1):	0.926	0.926	0.928	0.928	0.926	0.928	0.928	0.925	0.925	0.905
Exact + Reduce(.25):	0.926	0.926	0.928	0.928	0.926	0.928	0.928	0.925	0.925	0.905
Exact + Reduce(.5):	0.926	0.926	0.928	0.928	0.926	0.928	0.928	0.925	0.925	0.905
Exact + Node-Based:	0.928	0.928	0.929	0.929	0.928	0.929	0.929	0.926	0.926	0.906
Exact + Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Exact + Node-Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Reduce(.05):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.05) + Reduce(.1):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.05) + Reduce(.25):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.05) + Reduce(.5):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.05) + Node-Based:	0.928	0.928	0.929	0.929	0.928	0.929	0.929	0.926	0.926	0.906
Reduce(.05) + Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Reduce(.05) + Node-Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Reduce(.1):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.1) + Reduce(.25):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.1) + Reduce(.5):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.1) + Node-Based:	0.928	0.928	0.929	0.929	0.928	0.929	0.929	0.926	0.926	0.906
Reduce(.1) + Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Reduce(.1) + Node-Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Reduce(.25):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.25) + Reduce(.5):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.25) + Node-Based:	0.928	0.928	0.929	0.929	0.928	0.929	0.929	0.926	0.926	0.906
Reduce(.25) + Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Reduce(.25) + Node-Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Reduce(.5):	0.927	0.927	0.929	0.929	0.927	0.929	0.929	0.925	0.925	0.905
Reduce(.5) + Node-Based:	0.928	0.928	0.929	0.929	0.928	0.929	0.929	0.926	0.926	0.906
Reduce(.5) + Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Reduce(.5) + Node-Subsumed:	0.930	0.930	0.934	0.934	0.930	0.934	0.934	0.934	0.934	0.914
Node-Based:	0.926	0.926	0.928	0.928	0.926	0.928	0.928	0.925	0.925	0.905
Node-Based + Subsumed:	0.929	0.929	0.932	0.932	0.929	0.932	0.932	0.932	0.932	0.912
Node-Based + Node-Subsumed:	0.931	0.931	0.935	0.935	0.931	0.935	0.935	0.935	0.935	0.915
Subsumed:	0.908	0.908	0.908	0.908	0.908	0.908	0.908	0.880	0.880	0.850
Subsumed + Node-Subsumed:	0.910	0.910	0.910	0.910	0.910	0.910	0.910	0.883	0.883	0.853
Node-Subsumed:	0.909	0.909	0.909	0.909	0.909	0.909	0.909	0.872	0.872	0.842

Table 4.5.: Results for 3-local

Pattern comparator	Strict	Strict + Same	Strict + Equal	Strict + Merge	Same	Same + Equal	Same + Merge	Equal	Equal + Merge	Merge
Exact:	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Reduce(.05):	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Reduce(.1):	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Reduce(.25):	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Reduce(.5):	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Exact + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Exact + Node-Subsumed:	0.922	0.922	0.925	0.925	0.922	0.925	0.925	0.925	0.925	0.905
Reduce(.05):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.05) + Reduce(.1):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.05) + Reduce(.25):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.05) + Reduce(.5):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.05) + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Reduce(.05) + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Reduce(.05) + Node-Subsumed:	0.921	0.921	0.925	0.925	0.921	0.925	0.925	0.925	0.925	0.905
Reduce(.1):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.1) + Reduce(.25):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.1) + Reduce(.5):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.1) + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Reduce(.1) + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Reduce(.1) + Node-Subsumed:	0.921	0.921	0.925	0.925	0.921	0.925	0.925	0.925	0.925	0.905
Reduce(.25):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.25) + Reduce(.5):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.25) + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Reduce(.25) + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Reduce(.25) + Node-Subsumed:	0.921	0.921	0.925	0.925	0.921	0.925	0.925	0.925	0.925	0.905
Reduce(.5):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.5) + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Reduce(.5) + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Reduce(.5) + Node-Subsumed:	0.921	0.921	0.925	0.925	0.921	0.925	0.925	0.925	0.925	0.905
Node-Based:	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Node-Based + Subsumed:	0.919	0.919	0.923	0.923	0.919	0.923	0.923	0.923	0.923	0.903
Node-Based + Node-Subsumed:	0.923	0.923	0.926	0.926	0.923	0.926	0.926	0.926	0.926	0.906
Subsumed:	0.900	0.900	0.900	0.900	0.900	0.900	0.900	0.873	0.873	0.843
Subsumed + Node-Subsumed:	0.903	0.903	0.903	0.903	0.903	0.903	0.903	0.875	0.875	0.845
Node-Subsumed:	0.902	0.902	0.902	0.902	0.902	0.902	0.902	0.865	0.865	0.835

Table 4.6.: Results for 4-local

Pattern comparator	Strict	Strict + Same	Strict + Equal	Strict + Merge	Same	Same + Equal	Same + Merge	Equal	Equal + Merge	Merge
Exact:	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Reduce(.05):	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Reduce(.1):	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Reduce(.25):	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Reduce(.5):	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Exact + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Exact + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Exact + Node-Subsumed:	0.922	0.922	0.925	0.925	0.922	0.925	0.925	0.925	0.925	0.905
Reduce(.05):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.05) + Reduce(.1):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.05) + Reduce(.25):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.05) + Reduce(.5):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.05) + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Reduce(.05) + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Reduce(.05) + Node-Subsumed:	0.921	0.921	0.925	0.925	0.921	0.925	0.925	0.925	0.925	0.905
Reduce(.1):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.1) + Reduce(.25):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.1) + Reduce(.5):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.1) + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Reduce(.1) + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Reduce(.1) + Node-Subsumed:	0.921	0.921	0.925	0.925	0.921	0.925	0.925	0.925	0.925	0.905
Reduce(.25):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.25) + Reduce(.5):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.25) + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Reduce(.25) + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Reduce(.25) + Node-Subsumed:	0.921	0.921	0.925	0.925	0.921	0.925	0.925	0.925	0.925	0.905
Reduce(.5):	0.918	0.918	0.919	0.919	0.918	0.919	0.919	0.916	0.916	0.896
Reduce(.5) + Node-Based:	0.918	0.918	0.920	0.920	0.918	0.920	0.920	0.916	0.916	0.896
Reduce(.5) + Subsumed:	0.920	0.920	0.924	0.924	0.920	0.924	0.924	0.924	0.924	0.904
Reduce(.5) + Node-Subsumed:	0.921	0.921	0.925	0.925	0.921	0.925	0.925	0.925	0.925	0.905
Node-Based:	0.917	0.917	0.919	0.919	0.917	0.919	0.919	0.915	0.915	0.895
Node-Based + Subsumed:	0.919	0.919	0.923	0.923	0.919	0.923	0.923	0.923	0.923	0.903
Node-Based + Node-Subsumed:	0.923	0.923	0.926	0.926	0.923	0.926	0.926	0.926	0.926	0.906
Subsumed:	0.900	0.900	0.900	0.900	0.900	0.900	0.900	0.873	0.873	0.843
Subsumed + Node-Subsumed:	0.903	0.903	0.903	0.903	0.903	0.903	0.903	0.875	0.875	0.845
Node-Subsumed:	0.902	0.902	0.902	0.902	0.902	0.902	0.902	0.865	0.865	0.835

Table 4.7.: Results for Full path

List of Tables

List of Figures

Bibliography

[1] Arnaud Sahuguet. Everything you ever wanted to know about DTDs, but were afraid to ask. In *WebDB-2000*, 2000.

[2] D. Barbosa, L. Mignet, and P. Veltri. Studying the XML web: Gathering statistics from an XML sample. *World Wide Web*, 9(2):187–212, 2006.

[3] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008.

[4] G. J. Bex, W. Gelade, F. Neven, and S. 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.

[5] G. J. Bex, F. Neven, and J. V. den Bussche. DTDs versus XML schema: A practical study. In S. Amer-Yahia and L. Gravano, editors, *WebDB*, pages 79–84, 2004.

[6] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 115–126. VLDB Endowment, 2006.

[7] G. J. Bex, F. Neven, and S. 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.

[8] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *ic*, 142(2):182–206, 1998.

[9] D. Che, K. Aberer, and M. T. Özsu. Query optimization in XML structured-document databases. *VLDB J*, 15(3):263–289, 2006.

[10] R. Cover. The CoverPages. `http://xml.coverpages.org/`, Nov 2010. [Online; Accessed November 2010].

[11] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 28(2):431–??, 1999.

[12] N. Douib, O. Garbe, D. Günther, D. Oliana, J. Kroniger, F. Lücke, T. Melikoglu, K. Nordmann, G. Oezen, T. Schlitt, L. Schmidt, J. Westhoff, and D. Wolff. PG 530: Pattern based schema languages. Oct. 2009.

[13] F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML documents in relational databases. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *VLDB*, pages 1297–1300. Morgan Kaufmann, 2004.

[14] T. O. for the Advancement of Structured Information Standards. RELAX NG compact syntax. `http://www.oasis-open.org/committees/relax-ng/compact-20021121.html`, November 2002. [Online; Accessed November 2010].

[15] O. for the Advancement of Structured Information Standards (OASIS). DocBook V4.5 W3C XML Schema. `http://www.docbook.org/xsd/4.5/`, Nov 2010. [Online; Accessed November 2010].

[16] I. Free Software Foundation. GNU general public license. `http://www.gnu.org/licenses/gpl-3.0.txt`, Nov 2010. [Online; Accessed November 2010].

[17] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: making XML count. In M. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA*, pages 181–191, pub-ACM:adr, 2002. ACM Press.

[18] P. Garcia and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-12(9):920–925, Sept. 1990.

[19] W. Gelade and F. Neven. Succinctness of pattern-based schema languages for XML. In M. Arenas and M. I. Schwartzbach, editors, *DBPL*, volume 4797 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2007.

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

[21] J. Hegewald, F. Naumann, and M. Weis. XStruct: Efficient schema extraction from multiple and large XML documents. In R. S. Barga and X. Zhou, editors, *ICDE Workshops*, page 81. IEEE Computer Society, 2006.

[22] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE*, page 198, 2000.

[23] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. *CoRR*, cs.DB/0406016, 2004. informal publication.

[24] A. H. F. Laender, M. M. Moro, C. Nascimento, and P. Martins. An X-ray on web-available XML schemas. *SIGMOD Record*, 38(1):37–42, 2009.

[25] M. Ley. The DBLP computer science bibliography. `http://www.informatik.uni-trier.de/~ley/db/`, Nov 2010. [Online; Accessed November 2010].

[26] I. Manolescu, D. Florescu, and D. K. Kossmann. Answering XML queries over heterogeneous data sources. In *Proceedings of the 27th International Conference on Very Large Data Bases(VLDB '01)*, pages 241–250, Orlando, Sept. 2001. Morgan Kaufmann.

[27] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, Sept. 2006.

[28] L. Mignet, D. Barbosa, and P. Veltri. The XML web: a first study. In *WWW*, pages 500–510, 2003.

[29] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology (TOIT)*, 5(4):660–704, Nov. 2005.

[30] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *CoRR*, abs/cs/0606065, 2006. informal publication.

[31] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report SIDL-WP-1999-0120, Stanford University, Nov. 1999.

[32] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[33] W. M. Rand. Objective criteria for the evaluation of clustering methods. *American Statistical Association Journal*, 66(336):846–850, 1971.

[34] W3C. XML Schema part 1: Structures. `http://www.w3.org/TR/2001/PR-xmlschema-1-20010316/`, March 2001. [Online; Accessed November 2010].

[35] W3C. Extensible markup language (XML) 1.0 (fifth edition). `http://www.w3.org/TR/REC-xml/#dt-doctype`, November 2008. [Online; Accessed November 2010].

[36] W3C. W3C XML Schema definition language. `http://www.w3.org/TR/xmlschema11-1/`, December 2009. [Online; Accessed 13-December-2009].

[37] W3C. XHTML™ 1.1 - module-based XHTML - second edition. http://www.w3.org/TR/xhtml11/, Nov 2010. [Online; Accessed November 2010].

[38] G. Wang, M. Liu, J. X. Yu, B. Sun, G. Yu, J. Lv, and H. Lu. Effective schema-based XML query optimization techniques. In *IDEAS*, pages 230–235. IEEE Computer Society, 2003.