transnationale UNIVERSITEIT LIMBURG

School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Discovering structure in semi-structured data

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen, richting Informatica
te verdedigen door

Geert Jan Bex

Promotor: Prof. dr. F. Neven

18 december 2008

D/2008/2451/51

Acknowledgments

First and foremost, I would like to thank my adviser, Frank Neven, for his guidance, support and encouragement throughout the research reported here. I'd also like to thank Jan Van den Bussche, Thomas Schwentick, Wim Martens, Karl Tuyls, Stijn Vansummeren and Wouter Gelade, all of whom I had the pleasure to collaborate with in the context of this thesis.

I would like to express my gratitude to Frank, Stijn, Wouter and Natalia Kwasnikowska: I am indebted to them for their assistance and support in a difficult period.

I would also like to mention my appreciation of those I collaborated with on research topics that are not reported on in this thesis. It is a pleasure to work with Goele Hollanders, Marc Gyssens, Ronald Westra and Karl Tuyls on an interesting topic in bioinformatics/machine learning, and with Klaus Schmidt and Koos Jaap van Zwieten on biomechanics.

Infolab, our research group, provides an atmosphere that is stimulating to work in, so I would like to thank all members. Dirk Leinders deserves a special commendation for managing to share an office for several years and providing feedback to many of my crazy ideas. It should also be mentioned that is was nice to work with Frank, Marc and Bart Kuijpers when teaching various subjects.

Finally, I'd like to thank my family and friends for their support.

Diepenbeek, December 2008

Contents

1 Introduction

Data can be represented in many ways, the most common one in natural language text. Below is a description of a store by way of example.

> *The store is flourishing, quite a number of orders have been placed. John Mitchell, who's email address is* `j.mitchell@yahoo.com` *has ordered a DVD box (118F for US$100) which is currently not in stock, although there happen to be 10 IG8 DVDs that are supplied by Al Jones (*`a.j@gmail.com` *or* `a.j@dot.com`*)...*

Although this is quite intelligible for humans, a computer would have a hard time processing the information presented in this form. It is just represented in free format that lacks any kind of formal structure. At the other end of the spectrum, this same information could be stored in a relational database. The data is organized in tables, each column having its specific type, all in a rigorous structure that can easily be manipulated by a computer program and is described by a database schema.

As the term implies, semi-structured data takes the middle ground between unstructured and structured data. The concept traces its roots to the 1960s when the need arose to create machine readable documents. IBM's Generalized Markup Language [Gol96] was succeeded by the Standard Generalized Markup Language [ME95] that has been in use by various large companies as a document format. Markup was used to add some structure and/or semantics to textual documents. Although extensively used by professionals, semi-structured data entered the lay person's ecosystem with the advent of the World Wide Web in the form of HyperText Markup Language, or HTML. A few years later, researchers and entrepreneurs realized that the web had potential as an application platform, but that HTML was not a suitable data format for such a purpose. Although HTML served well as a format to specify how a web page should be rendered, it could not be harnessed into conveying semantics or structure of data. A working group was formed that borrowed ideas from SGML to define a more lightweight semi-structured data format: XML.

1.1 Motivation

The eXtensible Markup Language (XML) serves as the lingua franca for data exchange on the Internet [ABS99]. Because XML documents in general can be of any form, most communities and applications impose structural constraints on the documents that are to be exchanged or processed. These constraints can be formally specified in a *schema*, which is written in a schema language such as the *Document Type Definitions* (DTDs) or the *XML Schema Definitions* (XSDs) [TBMM01].

The advantages offered by the presence of a fully specified schema are numerous. First and foremost, a schema allows automatic validation of the input document structure, which not only facilitates automatic processing but also ensures soundness of the input. Unvalidated input data from web requests is considered as the number one vulnerability for web applications [OWA04]. The presence of a schema also allows for automation and optimization of search, integration, and processing of XML data (cf., e.g., [BFG05, DFS99, KSSS04, MFK01, NS03, WLY$^+$03]). Moreover, various software development tools such as Castor[1] and SUN's JAXB[2] rely on schemas to perform object-relational mappings for persistence. Furthermore, the existence of schemas is imperative when integrating (meta) data through schema matching [RB01] and in the area of generic model management [Ber03, Mel04]. A final advantage of a schema is that it assigns meaning to the data. That is, it provides a user with a concrete semantics of the document and aids in the specification of meaningful queries over XML data. Although the examples mentioned here just scrape the surface of current applications, they already underscore the importance of schemas accompanying XML data.

Unfortunately, in spite of the above mentioned advantages, the presence of a schema is not mandatory and many XML documents are not accompanied by one. For instance, in a recent study, Barbosa et al. [BMV05, MBV03] have

[1]`http://www.castor.org/`
[2]`http://java.sun.com/webservices/jaxb/`

shown that approximately half of the XML documents available on the web do not refer to a schema. In another study, we have noted that about two-thirds of XSDs gathered from schema repositories and from the web are not valid with respect to the W3C XML Schema specification [TBMM01], rendering them essentially useless for immediate application (see Chapter 6). A similar observation was made by Sahuguet [Sah00] concerning DTDs.

Based on the lack of schemas in practice, it is essential to devise algorithms that can infer a schema for a given collection of XML documents when none, or no syntactically correct one, is present. This is also acknowledged by Florescu [Flo05] who emphasizes that in the context of data integration:

> *"We need to extract good-quality schemas automatically from existing data and perform incremental maintenance of the generated schemas."*

It should be noted that even when a schema is already available, there are situations where inference can be useful. One such situation is *schema cleaning*: sometimes a schema is too general with respect to the XML data that it is supposed to describe. In that case, it can be advantageous to infer a new schema based solely on the data at hand. This situation is nicely illustrated by the following real-world example taken from the Protein Sequence Database DTD [Mik02], which gives the following definition for the `refinfo`-element:

```
authors, citation, volume?, month?, year, pages?,
  (title | description)?, xrefs?
```

An analysis of the available XML corpus (683 megabyte of data) using our inference algorithms shows that the `refinfo`-element is better described by the following expression:

```
authors, citation, (volume | month), year, pages?,
  (title | description)?, xrefs?
```

Note that the latter is more strict than the former, as it emphasizes that `volume` and `month` do not occur together: either one specifies a month of publication for a given journal article, or the volume that it has appeared in, but not both. As this example illustrates, schema inference algorithms can be used to better understand the semantics of a given XML dataset, making it possible to adapt an existing schema when necessary. In general, schema inference can be used to restrict schemas to a relevant subset of data needed by the application at hand, thereby facilitating difficult tasks like schema matching and data integration. Indeed, as argued by Hinkelman [Hin05], industry-level standards are too loosely defined in general, which can result in XML schemas where many business structures are formally specified as being optional.

The second situation where schema inference is useful even though a schema already exists is in the presence of *noisy XML data*. In such a situation, part or all of the data that needs to be processed is rejected by the existing schema. For instance, we have harvested and investigated a corpus of XHTML documents from the web and found that an astonishing 89 % of 2092 documents was not valid with respect to the XHTML Transitional specification [PAA+02]. In this case, the inference of a new schema based on the corpus and its comparison with the XHTML Transitional specification provides a uniform view of the kind of errors made. Further, given that one often has no choice but to deal with such noisy data, one may infer a new schema from a subset of the corpus (deleting documents that make unacceptable errors) and work with that schema rather than with the official specification to retain at least a minimal validation.

The situations sketched above in which schema inference is potentially helpful motivated us to develop a tool intended to complement existing high-quality schema editors like Stylus Studio[3] and `<oXygen/>`[4] to assists in such scenarios. In Chapter 5 this tool, SchemaScope, is introduced.

1.2 Problem setting

Based on the above observations, it is hence essential to devise algorithms that can automatically infer a DTD or XSD from a given corpus of XML documents.

As illustrated in Figure 1.1, an DTD is essentially a mapping d from element names to regular expressions over element names. An XML document is valid with respect to d if for every occurrence of an element name e in the document, the word formed by its children belongs to the language of the corresponding regular expression $d(e)$. For instance, the DTD in Figure 1.1 requires each `store` element to have zero or more `order` children, which must be followed by a `stock` element. Likewise, each order must have a `customer` child, which must be followed by one or more `item` elements.

To infer a DTD from a corpus of XML documents $\mathcal{C}$ it hence suffices to look, for each element name e that occurs in a document in $\mathcal{C}$, at the set of element name words that occur below e in $\mathcal{C}$, and to infer from this set the corresponding regular expression $d(e)$. As such, the inference of DTDs reduces to the inference of regular expressions from sets of positive example words. To illustrate, from the words `id price`, `id qty supplier`, and `id qty item item` appearing under `<item>` elements in a sample XML corpus, we could derive

[3] `http://www.stylusstudio.com/`
[4] `http://www.oxygenxml.com/`

the rule

```
  <!ELEMENT item (id, price | (qty, (supplier | item*)))>
```

```
<!ELEMENT store (order*, stock)>
<!ELEMENT order (customer, item+)>
<!ELEMENT customer (first, last, email*)>
<!ELEMENT item (id, price | (qty, (supplier | item+)))>
<!ELEMENT stock (item*)>
<!ELEMENT supplier (first, last, email*)>
```

Figure 1.1: An example DTD.

While the inference of XSDs is more complicated than the inference of DTDs, recent characterizations [MNSB06] show that the structural core of XML Schema Definition (that is, the sets of trees that are definable by XSDs) correspond to DTDs extended with vertical regular expressions. Therefore, one cannot hope to successfully infer XSDs without good algorithms for inferring regular expressions. As such, Chapters 2 and 3 focus on the inference of regular expressions (and therefore, by the reduction above, on the inference of DTDs). The inference of XSDs, building on the algorithms presented here, is treated in Chapter 4.

In particular, let Σ be a fixed set of alphabet symbols (also called element names), and let Σ^* be the set of all words over Σ.

Definition 1.1 (Regular Expressions, REs)**.** In this work, we are interested in learning regular expressions r, s of the form

$$r, s ::= \emptyset \mid \varepsilon \mid a \mid r \,.\, s \mid r + s \mid r? \mid r^+,$$

where parentheses may be added to avoid ambiguity. Here, ε denotes the empty word; a ranges over symbols in Σ; $r\,.\,s$ denotes concatenation; $r + s$ denotes disjunction; r^+ denotes one-or-more repetitions; and $r?$ denotes the optional regular expression. That is, the language $\mathcal{L}(r)$ accepted by regular expression r is given by:

$$\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset \\
\mathcal{L}(\varepsilon) &= \{\varepsilon\} \\
\mathcal{L}(a) &= \{a\} \\
\mathcal{L}(r\,.\,s) &= \{vw \mid v \in \mathcal{L}(r), w \in \mathcal{L}(s)\} \\
\mathcal{L}(r + s) &= \mathcal{L}(r) \cup \mathcal{L}(s) \\
\mathcal{L}(r^+) &= \{v_1 \ldots v_n \mid n \geq 1 \text{ and } v_1, \ldots, v_n \in \mathcal{L}(r)\} \\
\mathcal{L}(r?) &= \mathcal{L}(r) \cup \{\varepsilon\}.
\end{aligned}$$

Note that the Kleene star operator (denoting zero or more repetitions as in r^*) is not allowed by the above syntax. This is not a restriction, since r^* can always be represented as $(r^+)?$ or $(r?)^+$. Conversely, the latter can always be rewritten into the former for presentation to the user.

The class of *all* regular expressions is actually too large for our purposes, as both DTDs and XSDs require the regular expressions occurring in them to be *deterministic* (also sometimes called one-unambiguous [BKW98]). Intuitively, a regular expression is deterministic if, without looking ahead in the input word, it allows to match each symbol of that word uniquely against a position in the expression when processing the input in one pass from left to right. For instance, $(a + b)^*a$ is not deterministic as already the first symbol in the word aaa could be matched by either the first or the second a in the expression. Without look-ahead, it is impossible to know which one to choose. The equivalent expression $b^*a(b^*a)^*$, on the other hand, is deterministic.

Definition 1.2. Let $\overline{r}$ stand for the regular expression obtained from r by replacing the ith occurrence of alphabet symbol a in r by $a^{(i)}$, for every i and a. For example, for $r = b^+a(ba^+)?$ we have $\overline{r} = {b^{(1)}}^+a^{(1)}(b^{(2)}{a^{(2)}}^+)?$. A regular expression r is *deterministic* if there are no words $wa^{(i)}v$ and $wa^{(j)}v'$ in $\mathcal{L}(\overline{r})$ such that $i \neq j$.

Equivalently, an expression is deterministic if it is translated using the Glushkov construction [BK93] into a deterministic finite automaton rather than a non-deterministic one [BKW98]. Not every non-deterministic regular expression is equivalent to a deterministic one [BKW98]. Thus, semantically, the class of deterministic regular expressions forms a strict subclass of the class of all regular expressions.

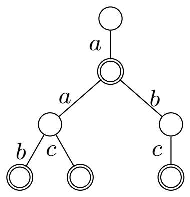
Learning in the limit. For the purpose of inferring DTDs from XML data, we are hence in search of an algorithm that, given enough sample words of a target deterministic regular expression r, returns a deterministic expression r' equivalent to r. In the framework of *learning in the limit* [Gol67], such an algorithm is said to learn the deterministic regular expressions from positive data.

Definition 1.3. Define a *sample* to be a finite subset of Σ^* and let $\mathcal{R}$ be a subclass of the regular expressions. An algorithm M mapping samples to expressions in $\mathcal{R}$ is said to *learn* $\mathcal{R}$ *in the limit from positive data* if (1) $S \subseteq \mathcal{L}(M(S))$ for every sample S and (2) to every $r \in \mathcal{R}$ we can associate a so-called *characteristic sample* $S_r \subseteq \mathcal{L}(r)$ such that, for each sample S with $S_r \subseteq S \subseteq \mathcal{L}(r)$, $M(S)$ is equivalent to r.

Intuitively, the first condition says that M must be *sound*; the second that M must be *complete*, given enough data. A class of regular expressions $\mathcal{R}$ is *learnable in the limit from positive data* if an algorithm exists that learns $\mathcal{R}$. For the class of all regular expressions, it was shown by Gold that no such algorithm exists [Gol67]. The same holds for the class of deterministic regular expressions, as shown below.

Theorem 1.4. *The class of deterministic regular expressions is not learnable in the limit from positive data.*

Proof. It was shown by Gold [Gol67, Theorem I.8], that any class of regular expressions that contains all non-empty finite languages as well as at least one infinite language is not learnable in the limit from positive data. Since deterministic regular expressions like a^* define an infinite language, it suffices to show that every non-empty finite language is definable by a deterministic expression. Hereto, let S be a finite, non-empty set of words. Now consider the prefix tree T for S. For example, if $S = \{a, aab, abc, aac\}$, we have the following prefix tree:



Nodes for which the path from the root to that node forms a string in S are marked by double circles. In particular, all leaf nodes are marked.

By viewing the internal nodes in T with two or more children as disjunctions, internal nodes in T with one child as conjunctions, and adding a question mark for every marked internal node in T, it is straightforward to transform T into a regular expression. For example, with S and T as above we get $r = a\,.(b\,.\,c + a\,.(b + c))?$. Clearly, $\mathcal{L}(r) = S$. Moreover, since no node in T has two edges with the same label, r must be deterministic. □

Theorem 1.4 immediately excludes the possibility for an algorithm to infer the full class of DTDs. In practice, however, regular expressions occurring in DTDs and XSDs are *concise* rather than arbitrarily complex. Indeed, a study of 819 DTDs and XSDs gathered from the Cover Pages [Cov03] (including many high-quality XML standards) as well as from the web at large, reveals that regular expressions occurring in practical schemas are such that every alphabet symbol occurs at most k times, with k small. Actually, in 98% of the cases $k = 1$. The reader is referred to Chapter 6 for a detailed analysis and discussion.

Definition 1.5. A regular expression is *k-occurrence* if every alphabet symbol occurs at most k times in it.

For example, the expressions `customer . order`$^+$ and (`school`$+$`institute`)$^+$ are both 1-occurrence, while `id` .(`qty`$+$`id`) is 2-occurrence (as `id` occurs twice). Observe that if r is k-occurrence, then it is also l-occurrence for every $l \geq k$. To simplify notation in what follows, we abbreviate 'k-occurrence regular expression' by k-ORE and also refer to the 1-OREs as 'single occurrence regular expressions' or SOREs.

Note that, since every alphabet symbol can occur at most once in a SORE, every SORE is necessarily deterministic. Given their importance in practical schemas, Chapter 2 focuses on the inference of SOREs. The inference of deterministic k-OREs for $k > 1$ is treated in Chapter 3.

To appreciate the difference between inferring DTDs and XSDs, consider the XML document in Figure 1.2 that contains information about store orders and stock contents. Orders hold customer information and list the items ordered, with each item stating its id and price. The stock contents consists of the list of items in stock, with each item stating its id, the quantity in stock, and—depending on whether the item is atomic or composed from other items—some supplier information or the items of which they are composed, respectively. It is important to emphasize that order items do not include supplier information, nor do they mention other items. Moreover, stock items do not mention prices.

DTDs are incapable of distinguishing between order items and stock items because the content model of an element can only depend on the element's name in a DTD, and not on the context in which it is used [MLMK05]. For example, although the DTD in Figure 1.1 describes all intended XML documents, it also allows supplier information to occur in order items and price information to occur in stock items.

XML Schema, in contrast, is based on type definitions, and therefore does allow the content model of an element to depend on the context it is used in [MLMK05, MNSB06]. For instance, it can be specified that an item is an order item when it occurs under an order element. In particular, XML schema can *exactly* describe the set of all intended documents, as we will show in Section 4.1. It is precisely this ability that makes inferring XSDs significantly more difficult than inferring DTDs. Indeed, whereas DTD inference basically reduces to generating regular expressions from sets of sample strings, inferring XSDs also entails identifying from a corpus of XML document the contexts in which elements bear different content models. Existing DTD inference engines do not identify such contexts and therefore always return schemas like the one in Figure 1.1 that are too general with respect to the target schema.

```
<store>
  <order>
    <customer>
       <name>John Mitchell</name>
       <email> j.mitchell@yahoo.com </email>
    </customer>
    <item> <id> I18F </id>
           <price> 100 </price>
    </item>
    <item> ... </item> ... <item> ... </item>
  </order>
  <order> ... </order> ... <order> ... </order>
  <stock>
    <item>
      <id> IG8 </id> <qty> 10 </qty>
      <supplier> <name> Al Jones </name>
                  <email> a.j@gmail.com </email>
                  <email> a.j@dot.com </email>
       </supplier>
     </item>
     <item>
       <id> J38H </id> <qty> 30 </qty>
       <item>
          <id> J38H1 </id> <qty> 10 </qty>
          <supplier> ... </supplier>
       </item>
       <item>
          <id> J38H2 </id> <qty> 1 </qty>
          <supplier> ... </supplier>
      </item>
      <item> ... </item> ... <item> ... </item>
   </item>
   ...
   <item> ... </item>
 </stock>
</store>
```

Figure 1.2: Example XML document.

Given that any DTD can be represented by an XSD and that it is—in general—impossible to learn the class of all DTDs, it is by extension impossible to learn the class of all XSDs from positive examples only. As the framework for XML schema inference is exactly such that only positive example documents are provided, it is unrealistic to develop inference algorithms for the class of all XSDs. One of the main challenges therefore is to identify subclasses of XSDs that are widely used in practice and that can be learned efficiently from positive data only.

An examination of 225 XSDs gathered from the Cover Pages [Cov03] (including many high-quality XML standards) as well as from the web at large, reveals that in more than 98% of the XSDs occurring in practice the content model of an element depends only on the label of the element itself, the label of its parent, and (sometimes) the label of its grandparent [MNSB06]. In Figure 1.2, for example, an item is an order item only if it is occurs in an order element. It is a stock item only if it occurs in a stock element or in an item element. We say that an XSD is *k-local* if its content models depend only on labels up to the k-th ancestor.

Although the class of k-local XSDs by itself is still too general to be learned from positive examples only, we show in Chapter 4 that the class of local XSDs with content models given by regular expressions in which each element name occurs at most once, *can* be learned efficiently from positive data only. The restriction to such *single occurrence regular expressions* (SOREs for short) has already been motivated above. Furthermore, as shown in Chapter 2, SOREs can be learned efficiently from positive data only, in contrast to the class of all regular expressions. Also, SOREs have the added benefit of succinctness: since every element name can occur only once, the size of a SORE is always linear in the number of different element names occurring in the corpus. The inferred content models are therefore naturally comprehensible.

In principle, one could consider k-ORE content models in the context of XSD inference, however, the combined complexity would require unrealistically large samples and incur large running times.

The study of real-world DTDs and XSDs much of this work is motivated by is discussed in more detail in Chapter 6. This chapter also details how the corpora we use in our experiments have been obtained.

Finally, Appendix 6.2 gives a short overview of the software that can be of general interest developed in the course of the research reported here.

Publications Chapter 2 is based on work presented at VLDB 2006 [BNST06] and submitted to VLDB Journal, Chapter 3 on a contribution to the WWW conference 2008 [BGNV08] and an article submitted to ACM Transactions on the Web, Chapter 4 at VLDB 2007 [BNV07], SchemaScope was demonstrated at SIGMOD 2008 [BNV08]. Finally, the analysis in Chapter 6 is an extended version of the one presented at WebDB 2004 [BNV04], the WWW 2005 conference [BMNS05] and an article in ACM Transactions on Database Systems [MNSB06].

1.3 Related work

Schema inference Schemas for semi-structured data have been defined in [BDFS97, FS98, QWG+96] and their inference has been addressed in [GW97, NUWC97, NAM98]. The methods in [NUWC97, GW97] focus on the derivation of a graph summary structure (called full representative object or dataguide) for a semi-structured database. This data structure contains all paths in the database. Approximations of this structure are considered by restricting to paths of a certain length. The latter then basically reduces to the derivation of an automaton from a set of bounded length strings. Naively restricting the algorithms to trees rather than graphs is inappropriate since no order is considered between the children of a node so that DTD-like schemas can not be derived. However, even the use of more sophisticated encodings of the XML documents using edges between siblings would be to no avail since no algorithms are given to translate the obtained automata to regular expressions. In [NAM98], a schema is a typing by means of a datalog program. The complexity of optimal schema inference is NP-hard. Again, no algorithms are given to transform datalog types into regular expressions. So, these approaches can therefore not be used to derive DTDs, not even when the semi-structured database is tree-shaped.

DTD inference In the context of DTD inference, [SW01] proposes several approaches to generate probabilistic string automata to represent REs. To transform these into actual REs, and hence to obtain DTDs, the authors refer to the methods of [Aho96]. The latter provides a method to translate one-unambiguous non-probabilistic string automata to REs, as given by Brüggemann-Klein and Wood [BKW98], followed by a post-processing simplification step. Apart from several case analyzes based on a dictionary example, no systematic study of the effectiveness of the approach is provided. In particular, in contrast to our results, no target class is given for which the set of transformations is complete.

There are only a few papers describing systems for direct DTD inference [GGR+03, MAC03, Chi01]. Only one of them is available for testing: XTRACT [GGR+03]. In Section 2.5, we make a detailed comparison with our proposal. In contrast to our approach, the XTRACT systems generates for every separate string a regular expression while representing repeated subparts by introducing Kleene-*. In a second step, the system factorizes common subexpressions of these candidate regular expressions using algorithms from the logic optimization literature. Finally, in the third step, XTRACT applies the Minimum Description Length (MDL) principle to find the best RE among the candidates. Although the approach has been shown to work on real-world DTDs [GGR$^+$03] the XML data complying to these DTDs was generated. We report in Section 2.5 that XTRACT has two kinds of shortcomings on real-world XML data: (1) it generates large, long-winded, and difficult to interpret regular expressions; and (2) it cannot handle large data sets (over 1000 strings). The latter is due to the NP-hard submodule in the third step of the XTRACT algorithm [Fer04]. The former problem seems to be more fundamental. The final step results in expressions consisting of disjunctions of regular expressions while in practice the large majority of regular expressions are concatenations of disjunctions [BNV04]. As a result, larger data sets result in larger regular expressions.

Min et al. [MAC03] propose an adaptation of the XTRACT approach to a restricted class of regular expressions which form a subclass of SOREs. Although the system, according to the conducted experiments [MAC03] outperforms XTRACT in accuracy and efficiency, it seems that the two fundamental shortcomings described above remain present. It would thus be surprising if the system performed much better than XTRACT on real-world data.

Similarly to Ahonen's [Aho96], Chidlovskii's approach [Chi01] relies on the translation of Glushkov automata to regular expressions which, in general, can lead to an exponential size increase.

Trang [Cla03] is state of the art software written by James Clark intended as a schema translator for the schema languages DTDs, Relax NG, and XML Schema. In addition, Trang allows to infer a schema for a given set of XML documents. We discuss Trang further in Section 2.5.1.

Language inference Most of the learning of regular languages from positive examples in the computational learning community is directed towards inference of automata as opposed to inference of REs [AS83, Pit89, Sak97]. As noted by Fernau [Fer04] and argued in the previous paragraph, first using learning algorithms for deterministic automata and then transforming these into regular expressions, in general leads to unmanageable and long-winded regular expressions. Some approaches to inference of REs for restricted cases have been considered. For instance, Brāzma [Brā93] showed that REs without union can be approximately learned in polynomial time from a set of examples satisfying some criteria. Recently, Fernau [Fer05] provided a learning algorithm for regular expressions that are finite unions of pairwise left-aligned union-free regular expressions. These expressions are different from the expressions we consider here: they are not included in the class of SOREs and do not contain all CHAREs. The development is purely theoretical, no experimental validation has been performed.

Automata to RE translation Although heuristics for automata to RE translations [DM04, HW05] have been proposed, all of them are optimizations of the classical state elimination algorithm. In particular, they investigate the best order to eliminate states when going from automata to REs. So, they focus on the class of all automata for which, as explained above, an exponential increase in size cannot be avoided in general. Further, the methods remain theoretical as no experimental analysis has been performed. Caron and Ziadi [CZ00] devise an algorithm deciding whether an automaton is Glushkov. If so, the automaton can be rewritten into a short equivalent regular expression. There method works in a top-down fashion, i.e., it derives the top nodes of the parse tree corresponding to the regular expression first, and subsequently proceeds downward in the tree. Consequently, the method first derives the largest subexpressions of the expression, making it harder to devise heuristics in the presence of missing data. In contrast, our approach is bottom-up, i.e., starting from the leaf nodes of the parse tree, composing them into the smallest subexpressions.

Learning of tree automata Fernau [Fer02] proposed a general framework of function distinguishability to learn tree automata for ranked trees. In essence, the framework gives a learning algorithm for the settings where a function is given which determines how to merge states. As XSDs constitute a subset of the regular unranked tree languages [MNSB06], our approach can be seen as an instantiation of that framework generalized to unranked trees. In contrast to the ranked setting, our algorithm has to deal with regular expressions inference as well. Unranked tree language inference received recent interest in the context of wrapper induction [CLN04]. However, this setting considers the inference of node-selecting queries in the presence of both positive and negative examples which makes the problem entirely different. The most related to our work is [RBV05], where it is shown that unranked (m, n)-contextual tree languages can be learned from positive data only. Basically, an algorithm is presented that learns a tree language from the sets of (m, n)-forks appearing in sample trees. Here, an (m, n)-fork is a subtree of depth n and width m. The (m, n) contextual languages form a strict superset of local SOXSDs. It hence follows immediately that local SOXSDs can be learned from positive data only. Our Theorem 4.5 shows, however, that for local SOXSDs we do not need to resort to general (m, n)-forks but that already path-shaped forks suffice. In addition, the techniques of [RBV05] are geared towards inferring queries, not XSDs. As such, no optimizations in the direction of schema inference have been attempted.

2

Inferring single-occurrence regular expressions

We consider the problem of inferring a concise Document Type Definition (DTD) for a given set of XML-documents, a problem that basically reduces to learning *concise* regular expressions from positive examples strings. We identify two classes of concise regular expressions—the single occurrence regular expressions (SOREs) and the chain regular expressions (CHAREs)—that capture the vast majority of expressions used in practical DTDs. For the inference of SOREs we present several algorithms that on a given set of example strings first infer an automaton using known techniques and then translate that automaton to a corresponding SORE, possibly repairing the automaton when no equivalent SORE can be found. In the process, we introduce a novel automaton to regular expression rewrite technique which is of independent interest. We show that our algorithms outperform existing systems in accuracy, conciseness and speed. When only a very small amount of XML data is available, however, (for instance when the data is generated by Web service requests or by answers to queries), these algorithms produce regular expressions that are too specific. Therefore, we introduce a novel learning algorithm CRX that directly infers CHAREs (which form a subclass of SOREs) without going through an automaton representation. We show that CRX performs very well within its target class on very small data sets.

2.1 A complete algorithm for inferring SOREs

Our goal in this section is to infer a SORE s equivalent to a target SORE r given only a finite sample $S \subseteq \mathcal{L}(r)$. To this end, we first learn from S a *single occurrence automaton* (SOA for short). A SOA is a specific kind of deterministic finite state automaton in which all states, except for the initial and final state, are element names. Figure 2.1 gives an example. Note that in contrast to the classical definition of automata, no edges are labeled: all incoming edges in a state a are assumed to be labeled by a. As such, a word $a_1, \ldots, a_n$ is accepted if there is an edge from the initial state to a_1, an edge from a_1 to a_2,..., and an edge from a_n to the final state. In other words, the SOA in Figure 2.1 accepts the same language as $a \,.\, b \,.(c + d^+)$.

Definition 2.1 (SOA)**.** Let *src* and *sink* be two special symbols, distinct from the element names, that will serve as the initial and final state, respectively. A *single occurrence automaton* is a finite directed graph $G = (V, E)$ such that

1. $\{src, sink\} \subseteq V$ and all nodes in $V - \{src, sink\}$ are element names; and
2. the edge relation E is such that *src* has only outgoing edges; *sink* has only incoming edges; and every $v \in V - \{src, sink\}$ is reachable by a walk from *src* to *sink*.

In this context, an *accepting run* for a word $a_1 \ldots a_n$ is a walk $src\, a_1 \ldots a_n$ *sink* from *src* to *sink* in G. In what follows we write $\mathcal{L}(G)$ for the set of all words accepted by G; $V(G)$ for the set of G's vertices, and $E(G)$ for G's edge relation.

2.1.1 Learning an automaton

Given a sample S, we can learn an automaton G that accepts all words in S by means of the algorithm 2T-INF shown in Algorithm 1. Its behavior is illustrated in Figure 2.1 on the sample $S = \{abc, abdd\}$ and in Figure 2.2 on the sample $S = \{bacacdacde, cbacdbacde, abccaadcde\}$.

2T-INF was introduced by Garcia and Vidal [GV90], who also proved the following proposition.

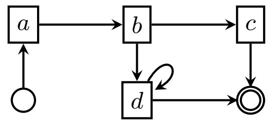


Figure 2.1: The SOA accepting the same language as the SORE $a \,.\, b \,.(c+d^+)$.

Algorithm 1 2T-INF

```
Input: a finite set of sample strings $S$
Output: a SOA $G$ such that $S \subseteq L(G)$
1: Let $V$ be the set of states consisting of all element names occurring in $S$
   plus the initial state $src$ and final state $sink$
2: Initialize $E := \emptyset$
3: for each string $a_1 \dots a_n$ in $S$ do
4:    add the edges $(src, a_1), (a_1, a_2), \dots, (a_n, sink)$ to $E$
5: return $G = (V, E)$
```

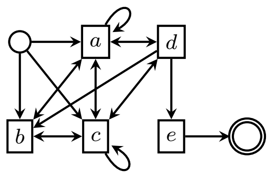


Figure 2.2: The SOA generated by 2T-INF for the sample $S = \{bacacdacde, cbacdbacde, abccaadcde\}$.

Proposition 2.2 (Garcia and Vidal [GV90]). 2T-INF *is* sound *in the sense that* $S \subseteq \mathcal{L}(\text{2T-INF}(S))$ *for each sample* S. *Moreover,* 2T-INF *is* minimal *in the sense that for each SOA* G *with* $S \subseteq \mathcal{L}(G)$, 2T-INF$(S)$ *is a subgraph of* G *and hence* $\mathcal{L}(\text{2T-INF}(S)) \subseteq \mathcal{L}(G)$.

It turns out that 2T-INF is also complete for building a SOA representation of a target SORE r, provided that its input sample is *representative* with respect to r.

Definition 2.3 (Representative sample)**.** A word v of length 2 is said to be 2-*gram* of a set of words W if it occurs as a subword in some $w \in W$. A sample S is *representative* of a SORE r if $S \subseteq \mathcal{L}(r)$ and

1. for every $a \in \Sigma$ that starts a word in $\mathcal{L}(r)$ there is a word in S that starts with a;
2. for every $a \in \Sigma$ that ends a word in $\mathcal{L}(r)$ there is a word in S that ends with a; and
3. every 2-gram of $\mathcal{L}(r)$ is a 2-gram of S.

If S is not representative of r, then we say that S does *not cover* r.

For instance, the sample $\{bacacdacde, cbacdbacde, abccaadcde\}$ is representative for $((b?.(a+c)^+).d)^+.e$ but $\{bacacdacde, cbacdbacde\}$ is not since it does not contain the 2-gram ab.

Proposition 2.4. *If S is a representative sample of* SORE r *then* $\mathcal{L}(2\textsc{t-inf}(S)) = \mathcal{L}(r)$.

Proof. It is not hard to see that every SORE r can be transformed into an equivalent SOA G_r: we take as nodes of G_r all element names occurring in r plus the initial state *src* and the final state *sink*; for each alphabet symbol that starts a word in $\mathcal{L}(r)$ we add the edge (src, a) to G_r; for each alphabet symbol that ends a word in $\mathcal{L}(r)$ we add an edge $(a, sink)$ to G_r, and for each alphabet symbol b that follows an alphabet symbol a in a word in $\mathcal{L}(r)$ we add the edge (a, b) to G_r. Now reason as follows. Clearly, $S \subseteq \mathcal{L}(r) = \mathcal{L}(G_r)$. Hence, $2\textsc{t-inf}(S)$ is a subgraph of G_r by Proposition 2.2. Since S is a representative sample of r, however, every edge of G_r must also be in $2\textsc{t-inf}(S)$. As such, $2\textsc{t-inf}(S) = G_r$ and hence $\mathcal{L}(2\textsc{t-inf}(S)) = \mathcal{L}(G_r)$. $\square$

2.1.2 From SOA to SORE

Proposition 2.4 shows that it is possible to learn a SOA representation of a target SORE r, provided that we are given enough data. To transform this SOA into a regular expression, an obvious approach would be to use known techniques such as the classical state elimination algorithm (cf., e.g., [HU79]). Unfortunately, as already hinted upon by Fernau [Fer04, Fer05] and as we illustrate below, it is very difficult to get *concise* regular expressions from an automaton representation. For instance, the classical state elimination algorithm applied to the SOA generated by 2T-INF in Figure 2.2 yields the expression:[1]

$$\begin{aligned}&(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d) + (b + aa^*b + (c +\\ &aa^*c)(c + aa^*c)^*(b + aa^*b))(aa^*b + (c + aa^*c)(c + aa^*c)^*\\ &(b + aa^*b))^*(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d)))(aa^*d +\\ &(c + aa^*c)(c + aa^*c)^*(d + aa^*d) + (b + aa^*b + (c + aa^*c)(c +\\ &aa^*c)^*(b + aa^*b))(aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))^*\end{aligned}$$

which differs quite a bit from the equivalent SORE

$$((b?(a + c))^+d)^+e \qquad (\ddagger).$$

Actually, results by Ehrenfeucht and Zeiger [EZ76], Gelade and Neven [GN08], and Gruber and Holzer [GH08] show that it is impossible in general to generate concise regular expressions from automata: there are automata, even SOAs as generated by 2T-INF, for which the number of occurrences of alphabet symbols in the smallest equivalent expressions is exponential in the size of the automaton. For such automata, a concise regular expression representation hence does not exist.

[1] Transformation computed by JFLAP: `www.jflap.org`.

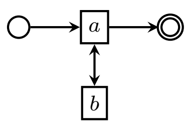


Figure 2.3: A SOA not equivalent to any SORE. It accepts the same language as $a(ba)^+$.

These results imply that there are SOAs G for which an equivalent SORE does not exist (Figure 2.3 gives a simple example). Note, however, that when such a SORE r does exist, its size is always linearly bounded by the number of states of G. Indeed, since every alphabet symbol can occur at most once in r, the size of r is linearly bounded by the alphabet symbols that it mentions. Since G and r are equivalent, these symbols are exactly the states of G (minus *src* and *sink*). Hence, the SOREs constitute a well-behaved and concisely representable subset of the regular languages. It is therefore natural to investigate how transform a given SOA into an equivalent SORE when such a SORE exists. Clearly, the example above illustrates that the classical state elimination algorithm does not suffice for this purpose.

For that reason, we introduce in this section a novel graph-rewriting approach for transforming SOAs into SOREs. While our approach is related to the classical state-elimination algorithm for transforming an arbitrary automaton into a regular expression, we do not eliminate states by introducing additional edges (thereby duplicating subexpressions) but instead replace sets of states by single states (taking care to avoid duplication). In addition, there a two rewriting steps that only removes edges.

Just as the classical algorithm, it is necessary for the definition of the graph rewrite rules to define a generalization of SOAs in which internal states are allowed to be labeled by SOREs (as opposed to element names from Σ). This generalization is defined as follows. Call two regular expressions r and s *alphabet-disjoint* if r and s have no alphabet symbol in common. For example, $(a+b)?$ and c^+ are alphabet-disjoint, whereas $(a+b)$ and $b?c^+$ are not. Call an expression r *proper* if it accepts at least one non-empty word (i.e., it is not equivalent to $\emptyset$, nor to ε).

Definition 2.5. A *generalized single occurrence automaton* (generalized SOA for short) is a finite graph $G = (V, E)$ such that

1. $\{src, sink\} \subseteq V$ and all vertices in $V - \{src, sink\}$ are pairwise alphabet-disjoint proper SOREs; and

2. the edge relation E is such that *src* has only outgoing edges; *sink* only incoming edges; and every $v \in V$ is reachable by a walk from *src* to *sink*.

A word $w \in \Sigma^*$ is accepted by G if there is a walk $\mathit{src}\, r_1 \dots r_n\, \mathit{sink}$ in G and a division of w into subwords $w = w_1 \dots w_n$ such that $w_i \in \mathcal{L}(r_i)$, for $1 \leq i \leq n$. Again, we write $\mathcal{L}(G)$ for the set of all words accepted by G.

Figure 2.6 shows some examples. Clearly, every SOA is also a generalized SOA. In what follows, we write $\mathrm{Pred}_G(r)$ for the set of all direct predecessors of s in G, and $\mathrm{Succ}_G(s)$ for the set of all direct successors of s in G:

$$\begin{aligned}\mathrm{Pred}_G(s) &:= \{r \mid (r, s) \in E(G)\},\\ \mathrm{Succ}_G(s) &:= \{t \mid (s, t) \in E(G)\}.\end{aligned}$$

Furthermore, we write $\mathrm{Pred}^-_G(r)$ for $\mathrm{Pred}_G(r) - \{r\}$ and similarly $\mathrm{Succ}^-_G(s)$ for $\mathrm{Succ}_G(s) - \{s\}$. Finally, we write

$$\mathrm{Pred}^+_G(s) := \begin{cases}\mathrm{Pred}_G(s) \cup \{s\} & \text{if } s = s'^+\\ \mathrm{Pred}_G(s) & \text{otherwise}\end{cases}$$

$$\mathrm{Succ}^+_G(s) := \begin{cases}\mathrm{Succ}_G(s) \cup \{s\} & \text{if } s = s'^+\\ \mathrm{Succ}_G(s) & \text{otherwise.}\end{cases}$$

Rewrite rules. Our system of rewrite rules consists of the seven rules shown in Figure 2.4: one rule to introduce disjunction $(r + s)$, four rules to introduce concatenation $(r \,.\, s$, $r? \,.\, s$, $r \,.\, s?$, and $r? \,.\, s?)$, one rule to introduce iteration (r^+), and one rule to introduce optionals $(r?)$. At the basis of the first five rules lies the *contraction* of two states r and s into a single new state t, which is defined as follows.

Definition 2.6 (State contraction)**.** Let G be a generalized SOA; let r and s be states in G; and let t be a state not in G. The *contraction* of r and s into t is the generalized SOA $G[r, s/t]$ obtained from G as follows:

1. Add t as a new state to G;
2. make every $v \in \mathrm{Pred}(r) - \{r, s\}$ a predecessor of t;
3. make every $w \in \mathrm{Succ}(r) - \{r, s\}$ a successor of t;
4. add a loop $t \to t$ if $r \in \mathrm{Succ}(s)$; and
5. remove r, s and all of their incoming and outgoing edges.

Note that state contraction is not symmetric due to step (4).

For example, the contraction $G[a, c/a + c]$ of the generalized SOA G in Figure 2.6(a), is shown in Figure 2.6(b). Similarly, the contraction $G[b, a +$

DISJUNCTION $r + s$

Precondition	*Action*
(1) $r, s \notin \{\mathit{src}, \mathit{sink}\}$ and	replace G by $G[r, s/r + s]$
(2.a) $\mathrm{Pred}_G(r) = \mathrm{Pred}_G(s)$ and $\mathrm{Succ}_G(r) = \mathrm{Succ}_G(s)$ or	
(2.b) $\mathrm{Pred^+}_G(r) = \mathrm{Pred^+}_G(s)$ and $\mathrm{Succ^+}_G(r) = \mathrm{Succ^+}_G(s)$	

CONCATENATION $r \,.\, s$

Precondition	*Action*
(1) $r, s \notin \{\mathit{src}, \mathit{sink}\}$ and	Replace G by $G[r, s/r \,.\, s]$
(2) $\mathrm{Succ}_G(r) = \{s\}$ and $\mathrm{Pred}_G(s) = \{r\}$	

CONCATENATION $r \,.\, s?$

Precondition	*Action*
(1) $r, s \notin \{\mathit{src}, \mathit{sink}\}$	Replace G by $G[r, s/r \,.\, s?]$
(2) $\mathrm{Pred}_G(s) = \{r\}$	
(3) $\mathrm{Succ}_G(r) - \{r, s\} = \mathrm{Succ}_G(s) - \{r, s\}$ and	
(4) if $r \in \mathrm{Succ}_G(s)$ then $r \in \mathrm{Succ}^+(r)$	
(5) if $r \in \mathrm{Succ}_G(r)$ then $r \in \mathrm{Succ}_G(s)$	

CONCATENATION $r? \,.\, s$

Precondition	*Action*
(1) $r, s \notin \{\mathit{src}, \mathit{sink}\}$	Replace G by $G[r, s/r? \,.\, s]$
(2) $\mathrm{Succ}_G(r) = \{s\}$	
(3) $\mathrm{Pred}_G(r) - \{r, s\} = \mathrm{Pred}_G(s) - \{r, s\}$ and	
(4) if $s \in \mathrm{Pred}_G(r)$ then $s \in \mathrm{Pred}^+(s)$	
(5) if $s \in \mathrm{Pred}_G(s)$ then $s \in \mathrm{Pred}_G(r)$	

CONCATENATION $r? \,.\, s?$

Precondition	*Action*
(1) $r, s \notin \{\mathit{src}, \mathit{sink}\}$	Replace G by $G[r, s/r? \,.\, s?]$
(2) $s \in \mathrm{Succ}_G(r)$	
(3) $\mathrm{Succ}_G(r) - \{r, s\} = \mathrm{Succ}_G(s) - \{r, s\}$	
(4) $\mathrm{Pred}_G(r) - \{r, s\} = \mathrm{Pred}_G(s) - \{r, s\}$	
(5) $\mathrm{Pred}_G(r) \times \mathrm{Succ}_G(s) \subseteq E(G)$ and either	
(6a) $r \in \mathrm{Succ}_G(s)$, $r \in \mathrm{Succ}^+(r)$, and $s \in \mathrm{Succ}^+(s)$; or	
(6b) $r \notin \mathrm{Succ}_G(s)$, $r \notin \mathrm{Succ}_G(r)$, and $s \notin \mathrm{Succ}_G(s)$	

ITERATION r^+

Precondition	*Action*
(1) $r \notin \{\mathit{src}, \mathit{sink}\}$ and	Replace G by $G[r/r^+] \ominus (r^+, r^+)$
(2) $r \in \mathrm{Succ}_G(r)$	

OPTIONAL $r?$

Precondition	*Action*
(1) $r \notin \{\mathit{src}, \mathit{sink}\}$ and	Replace G by $G[r/r?] \ominus (\mathit{src}, \mathit{sink})$
(2) $E(G) = \{(\mathit{src}, r), (r, \mathit{sink}), (\mathit{src}, \mathit{sink})\}$	

Figure 2.4: Rewrite rules

$c/b? .(a + c)]$ of the generalized SOA G in Figure 2.6(b) is shown in Figure 2.6(c). Note that if $r = s$, then $G[r, s/t]$ is simply a substitution of r by the new state t. To simplify notation, we simply write $G[r/t]$ for such contractions in what follows.

In addition to contraction, the rewrite rules also use the following operation.

Definition 2.7. If G is a generalized SOA and r, s are states in G, then we write $G \ominus (r, s)$ to denote the generalized SOA obtained from G by removing the edge from r to s, if present.

In what follows, we write $G \rightsquigarrow H$ to indicate that G rewrites to H in a single step according to the rewrite rules in Figure 2.4, and $G \rightsquigarrow^* H$ to indicate that G rewrites to H in zero or more steps.

The following proposition shows that the rewrite rules are sound:

Proposition 2.8. *If G is a generalized SOA and $G \rightsquigarrow H$ then H is also a generalized SOA and $\mathcal{L}(G) = \mathcal{L}(H)$.*

Proof. First observe that, since all states in a generalized SOA are pairwise alphabet-disjoint proper SOREs, the new states $r + s$; $r \,.\, s$; $r? \,.\, s$; $r \,.\, s?$; $r? \,.\, s?$; r^+; and $r?$ introduced by the rewrite rules in Figure 2.4 must themselves be proper SOREs alphabet-disjoint with the remaining states. As such, all states in H are pairwise alphabet-disjoint proper SOREs. To show that H is a generalized SOA, it hence remains to show that every state in H participates in a walk from *src* to *sink*. Hereto, observe that H is either

- $G[r, s/t]$ for some t. Then, since G is a generalized SOA, r and s participate in a walk from *src* to *sink*. In particular, there is a walk form *src* to r in G, and a walk form s to *sink*. Then, by definition of state contraction, there is a walk from *src* to t and from t to *sink* in H, i.e., t participates in a walk form *src* to *sink* in H.

- $G[r/r^+] \ominus (r^+, r^+)$. Then, by definition of state contraction and since r participates in a walk from *src* to *sink* in G, r^+ must participate in a walk form *src* to *sink* in $G[r/r^+]$. This walk can always be transformed into a walk from *src* to *sink* in H by removing the edge (r^+, r^+) should it occur.

- $G[r/r?] \ominus (src, sink)$. Then, by definition of state contraction and since r participates in a walk from *src* to *sink* in G, $r?$ must participate in a walk form *src* to *sink* in $G[r/r?]$. Since the edge $(src, sink)$ cannot occur in this walk, $r?$ also participates in a walk from *src* to *sink* in H.

To see that $\mathcal{L}(G) = \mathcal{L}(H)$ we reason by a case analysis on the rewrite rule used to transform G into H. For economy of space, we only illustrate this reasoning for DISJUNCTION $r + s$; the other cases are similar.

So, suppose that G was rewritten into H by DISJUNCTION $r + s$, i.e $H = G[r, s/r+s]$. Then r and s have the same (extended) predecessor and successor set. As such,

$$\begin{aligned} s \in \mathrm{Succ}_G(r) &\Leftrightarrow s \in \mathrm{Succ}^+_G(s) \\ &\Leftrightarrow s \in \mathrm{Pred}^+_G(s) \\ &\Leftrightarrow s \in \mathrm{Pred}_G(r) \\ &\Leftrightarrow r \in \mathrm{Succ}_G(s) \\ &\Leftrightarrow r \in \mathrm{Succ}^+(r) \end{aligned}$$

Therefore, G is either as illustrated in Figure 2.5(a) or as illustrated in Figure 2.5(b). In both cases, the corresponding H is also shown.

Now suppose that $w = w_1 \dots w_n \in \Sigma^*$ is recognized by the walk $\mathit{src}, t_1, \dots, t_n, \mathit{sink}$ in G with $w_i \in \mathcal{L}(t_i)$ for $1 \leq i \leq n$. Let the sequence $\mathit{src}, t'_1, \dots, t'_n, \mathit{sink}$ be obtained from $\mathit{src}, t_1, \dots, t_n, \mathit{sink}$ by replacing every occurrence of r and s by $r + s$. By inspection of Figures 2.5(a) and 2.5(b), it is not difficult to see that $\mathit{src}, t'_1, \dots, t'_n, \mathit{sink}$ is walk in H. Moreover, $w_i \in \mathcal{L}(t'_i)$ by construction for $1 \leq i \leq n$. Therefore, $w \in \mathcal{L}(H)$ and hence $\mathcal{L}(G) \subseteq \mathcal{L}(H)$. Conversely, suppose that $w = w_1 \dots w_n \in \Sigma^*$ is recognized by $\mathit{src}, t'_1, \dots, t'_n, \mathit{sink}$ in H with $w_i \in \mathcal{L}(t'_i)$ for $1 \leq i \leq n$. Determine v_i as follows:

$$t_i = \begin{cases} t'_i & \text{if } t'_i \neq r + s \\ r & \text{if } t'_i = r + s \text{ and } w_i \in \mathcal{L}(r) \\ s & \text{if } t'_i = r + s \text{ and } w_i \in \mathcal{L}(s) \end{cases}$$

By inspection of Figures 2.5(a) and 2.5(b) it is not difficult to see that $\mathit{src}, t_1, \dots, t_n, \mathit{sink}$ is a walk in G. Moreover, $w_i \in \mathcal{L}(t_i)$ for $1 \leq i \leq n$. Therefore $w \in \mathcal{L}(G)$ and hence $\mathcal{L}(H) \subseteq \mathcal{L}(G)$. As such, $\mathcal{L}(G) = \mathcal{L}(H)$.

□

Since each rewrite rule either contracts two states into a single state or removes an edge from G, the size of H is always smaller than G. Therefore:

Proposition 2.9. *The system of rewrite rules in Figure 2.4 is* terminating*: there is no infinite sequence of rewrite steps $G \rightsquigarrow H \rightsquigarrow I \rightsquigarrow \dots$*

Our algorithm REWRITE, shown in Algorithm 2, then operates as follows. First, it checks whether the input SOA G corresponds to the empty language ($\emptyset$) or the empty word (ε) in lines $1 - 5$. If so, it returns the corresponding

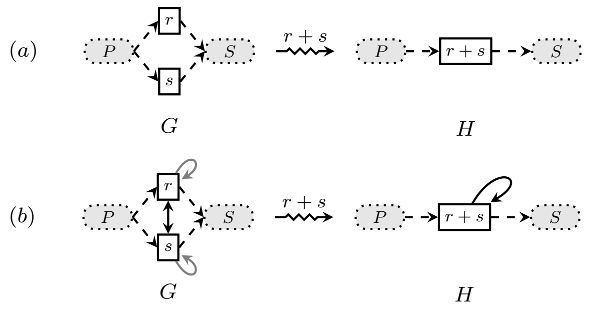


Figure 2.5: Illustration of the proof of Proposition 2.8. P is the set $\text{Pred}_G(r) - \{r, s\} = \text{Pred}_G(s) - \{r, s\}$. S is the set $\text{Succ}_G(r) - \{r, s\} = \text{Succ}_G(s) - \{r, s\}$. The gray loops on r and s in case (b) indicate that $r \in \text{Succ}^+_G(r)$ and $s \in \text{Succ}^+_G(s)$.

regular expression. Otherwise, it rewrites G until no further rules apply. If the resulting SOA is *final*, i.e., if $E(G) = \{(\mathit{src}, r), (r, \mathit{sink})\}$ with r distinct from *src* and *sink*, then clearly $\mathcal{L}(G) = \mathcal{L}(r)$, and r is returned as result. If the resulting SOA is not final, then G is not equivalent to a SORE (as we formally show further on), and REWRITE fails. To illustrate, Figure 2.6 shows an example run of REWRITE on the example SOA from Figure 2.2.

Theorem 2.10. *On input SOA G,* REWRITE *fails if and only if G is not equivalent to a* SORE*. Otherwise,* REWRITE *returns a* SORE *equivalent to G. Moreover,* REWRITE *operates in time $\mathcal{O}(n^5)$ where n is the number of states in G.*

Note that the complexity $\mathcal{O}(n^5)$ is reasonable since when we apply REWRITE to the result of 2T-INF on a sample S, n corresponds to the (typically small) number of distinct element names occurring in S, not the total number or total length of words in S.

The remainder of this section is devoted to the proof of Theorem 2.10, which is divided into three steps. First, we show that REWRITE is sound:

Proposition 2.11. *If* REWRITE(G) *does not fail then it returns a* SORE *equivalent to G, for any SOA G.*

Proof. We distinguish three cases.

1. If *sink* is not reachable from *src* then REWRITE$(G) = \emptyset$ (clearly a SORE) and $\mathcal{L}(G) = \emptyset = \mathcal{L}(\emptyset)$, as desired.

2. If $E(G) = \{(\mathit{src}, \mathit{sink})\}$ then REWRITE$(G) = \varepsilon$ (again clearly a SORE), and $\mathcal{L}(G) = \{\varepsilon\} = \mathcal{L}(\varepsilon)$, as desired.

Algorithm 2 REWRITE

Input: a SOA G

Output: a SORE r such that $\mathcal{L}(r) = \mathcal{L}(G)$

```
 1: if $sink$ is not reachable from $src$ in $G$ then
 2:    return $\emptyset$
 3: else if $E(G) = \{(src, sink)\}$ then
 4:    return $\varepsilon$
 5: else
 6:    while a rewrite rule from Figure 2.4 can be applied do
 7:       perform the rewrite rule on $G$
 8:    if $G$ is final then
 9:       return the corresponding regular expression
10:    else
11:       fail
```

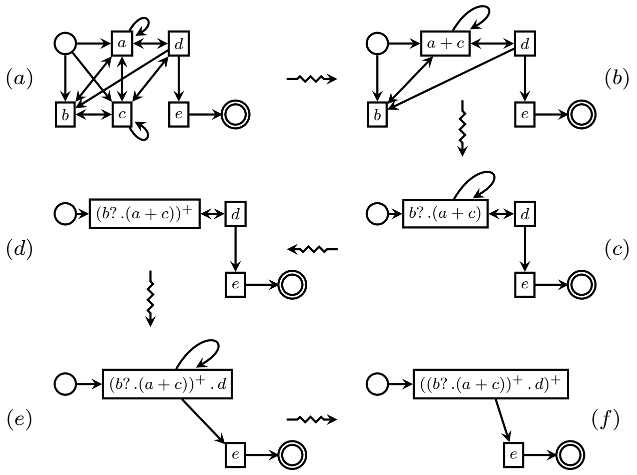


Figure 2.6: An execution of REWRITE on the example automaton in Fig. 2.2. Step (1) applies DISJUNCTION $r + s$ with $r = a$ and $s = b$. Step (2) applies CONCATENATION $r? \,.\, s$ with $r = b$ and $s = a + c$. Step (3) applies ITERATION r^+ with $r = b?.(a + c)$. Step (4) applies CONCATENATION $r \,.\, s$ with $r = (b?.(a + c))^+$ and $s = d$. Step (5) applies ITERATION r^+ with $r = (b?.(a + c))^+ \,.\, d$. One more application of CONCATENATION $r \,.\, s$ with $r = ((b?.(a + c))^+ \,.\, d)^+$ and $s = e$ (not shown) leads to the resulting expression $((b?.(a + c))^+ \,.\, d)^+ \,.\, e$.

3. Otherwise, G is rewritten into a final generalized SOA H with $E(H) = \{(src, t), (t, sink)\}$ (t distinct from *src* and *sink*) and $\textsc{rewrite}(G) = t$. In particular, t is a SORE. By Proposition 2.8, $\mathcal{L}(G) = \mathcal{L}(H)$ and thus, since $E(H) = \{(src, t), (t, sink)\}$, $\mathcal{L}(G) = \mathcal{L}(H) = \mathcal{L}(t) = \mathcal{L}(\textsc{rewrite}(G))$, as desired.

□

Next, we show that REWRITE has the claimed complexity.

Proposition 2.12. REWRITE *operates in time $\mathcal{O}(n^5)$, where n is the number of states of its input G.*

Proof. We assume that checking whether there is an edge from state r to state s can be done in constant time (for instance, using an adjacency matrix representation). To see that REWRITE runs in time $\mathcal{O}(n^5)$ under this assumption, let us check that lines 1-4, lines 6-7, and lines 8-10 all run in $\mathcal{O}(n^5)$.

(Lines 1-4*).* Since G has at most n^2 edges, checking whether *sink* is reachable from *src* can be done in time $\mathcal{O}(n^2)$ using depth first search. Moreover, checking whether $E(G) = \{(src, sink)\}$ can also be done in time $\mathcal{O}(n^2)$.

(Lines 6-7*).* Suppose that $\vec{G} = G_1, G_2, \dots, G_k$ is the sequence of generalized SOAs produced by lines 6-7 when rewriting $G = G_1$ until no further rewrite rule applies. Since rewrite rules never introduce new states without also removing a state, every G_i has at most n states. Now reason as follows:

- The rule for optionals can be applied at most once in $\vec{G}$ since the automaton that it returns is always final, and since no rewrite rule applies to a final generalized SOA. Checking the preconditions of the rule for optionals can be done in time $\mathcal{O}(n^2)$, and its action can be performed in time $\mathcal{O}(n)$. As such, the total time spent in $\vec{G}$ on applying the rewrite rule for optionals is bounded by $\mathcal{O}(n^2)$.

- Since the rewrite rules for disjunction and concatenation contract two states into a single one, these rewrite rules can be applied at most n times in $\vec{G}$. Since of all their preconditions can be checked in time $O(n^4)$ (by iterating over all pairs of states r and s in the current automaton G_i and comparing Pred(r), Pred(s), Succ(r) and Succ(s) as desired) and since state contraction can be done in time $\mathcal{O}(n)$, the total time spent in $\vec{G}$ on the rewrite rules for disjunction and concatenation is bounded by $\mathcal{O}(n \times n^4) = \mathcal{O}(n^5)$.

- Since the rule for iteration removes the loop of the state to which it is applied, and since each generalized SOA contains at most n loops, there can be at most n consecutive applications of this rule before another rewrite rule is applied. By the remarks above, there are at most n applications of the other rewrite rules, so the rewrite rule for iteration can be applied at most n^2 times in $\vec{G}$. Since its precondition can be checked in constant time, and since its action can be done in time $\mathcal{O}(n)$, the total time spent in $\vec{G}$ on the rewrite rule for iteration is bounded by $\mathcal{O}(n^2 \times n) = \mathcal{O}(n^3)$.

(Lines 6-7*).* Finally, checking whether a generalized SOA is final and extracting the corresponding regular expression can be done in time $O(n^2)$.

In summary, lines 1-4 run in time $\mathcal{O}(n^2)$, lines 6-7 run in time $\mathcal{O}(n^4)$, and lines 8-11 run in time $\mathcal{O}(n^2)$, yielding a total running time of $\mathcal{O}(n^4)$. □

Finally, we show that REWRITE(G) fails if, and only if G is not equivalent to a SORE, or equivalently, that REWRITE(G) does not fail if, and only if, G is equivalent to a SORE. This is actually the most involved part of the proof of Theorem 2.10. Proposition 2.11 already shows that if REWRITE(G) does not fail, then G is equivalent to a SORE. Hence, we remain to show:

Proposition 2.13. *If SOA G is equivalent to a* SORE*, then* REWRITE(G) *does not fail.*

Essentially, we prove this proposition in two steps. Call a generalized SOA *proper* if $\mathcal{L}(G) \neq \emptyset$ and $\mathcal{L}(G) \neq \{\varepsilon\}$.

1. We first show that for any proper SOA G equivalent to a SORE there exists a sequence of rewrite steps that ends in a final automaton (Corollary 2.20).

2. In addition, we show that if proper G can be rewritten into a final automaton by a particular sequence of rewrite steps, then *any* sequence of rewrite steps on G ends in a final automaton (Corollary 2.28).

As such, REWRITE(G) cannot fail when G is equivalent to a SORE: either G is not proper, in which case lines 1-4 of Algorithm 2 return a valid expression, or G is proper and will hence be rewritten into a final automaton, in which case line 9 returns a valid expression.

Step (1) above is obtained by showing that for every proper SOA G equivalent to a SORE, there exists a SORE r of a special form from which we can deduce a sequence of automata $G_1, G_2, \dots, G_n$ such that $G_1 = G$; every G_i can be rewritten by one of the rewrite rules of Figure 2.4 into G_{i+1}; and G_n is final. Here, the special form that r takes is that of a *saturated factor*, a notion that is introduced in the following definitions.

Definition 2.14 (Factors, terms, basic terms)**.** Let the sets of *factors*, *terms*, and *basic terms* be given by the syntax

$$\begin{array}{lccl} \text{factors} & f & ::= & t \mid t? \\ \text{terms} & t & ::= & e \mid e^{+} \\ \text{basic terms} & e & ::= & a \mid f\,.\,f \mid t + t, \end{array}$$

where a ranges over symbols in Σ.

For instance, $(\texttt{name}^{+} + \texttt{id})$ is a basic term (and hence also a term), $(\texttt{name}^{+} + \texttt{id})^{+}$ is a term (and hence also a factor), and $(\texttt{name}?\,.\,\texttt{email}?)\,.\,\texttt{address}?$ is a factor. Note that repetitions of ? and $^{+}$ (as in $\texttt{name}??$ and $\texttt{name}^{++}$) and optionals immediately below a disjunction (as in $(\texttt{name}? + \texttt{id})$) are disallowed by the syntax of factors, terms and basic terms. This is not a restriction, as the following proposition shows.

Proposition 2.15. *For every proper* SORE r *there exists an equivalent* SORE *factor* f*.*

Proof. Consider the following system of rewrite rules on regular expressions.

$$\begin{array}{rcl@{\qquad}rcl} s?^{+} & \to & s^{+}? & s?? & \to & s? \\ s^{++} & \to & s^{+} & (s_1? + s_2) & \to & (s_1 + s_2)? \\ (s_1 + s_2?) & \to & (s_1 + s_2)? & s + \varepsilon & \to & s? \\ \varepsilon + s & \to & s? & s\,.\,\varepsilon & \to & s \\ \varepsilon\,.\,s & \to & s & \varepsilon? & \to & \varepsilon \\ \varepsilon^{+} & \to & \varepsilon & s + \emptyset & \to & s \\ \emptyset + s & \to & s & s\,.\,\emptyset & \to & \emptyset \\ \emptyset\,.\,s & \to & \emptyset & \emptyset? & \to & \emptyset \\ \emptyset^{+} & \to & \emptyset & & & \end{array}$$

Since each rewriting rule either reduces the size of the original expression, or moves an optional ? outward in the original expression, any sequence of rewrite steps starting from r must end in an expression f on which no rule is applicable. Since each rewrite rule clearly produces an expression equivalent to the original expression, it readily follows that r is equivalent to f. Moreover, since none of the rewrite rules duplicates a subexpression and r is a SORE, so is f. It remains to show that f is a factor. Hereto, we first verify by structural induction on f that, since none of the above rewrite rules are applicable to f or any of its subexpressions, f is either $\emptyset$, ε, or a factor. Then, since r is proper and f is equivalent to r, f must be proper and thus cannot be $\emptyset$ or ε. As such, f is a factor, as desired. $\square$

Definition 2.16 (Saturation)**.** A regular expression r is *nullable* if $\varepsilon \in \mathcal{L}(r)$. A regular expression is *saturated* if for every subexpression $r\,.\,s$, if r is nullable then it is an optional $r'?$ and similarly, if s is nullable then it is an optional $s'?$.

For instance, the expression (`name`? . `email`?) is saturated, while the following expression (`name`? . `email`?) . `address`? is not.

We can easily make any expression saturated by replacing all nullable but non-optional subexpressions r and s occurring in a concatenation $r\,.\,s$ by $r?$ and $s?$, respectively. For instance, in the expression (`name`? . `email`?) . `address`? we would replace (`name`? . `email`?) by (`name`? . `email`?)?, so that we obtain (`name`? . `email`?)? . `address`?. It is readily verified that saturating a factor in this way yields again a factor. Hence, by Proposition 2.15:

Corollary 2.17. *For every proper* SORE r *there exists an equivalent saturated factor* SORE f.

Definition 2.18. A generalized SOA is *ε-transitive* if for all distinct states r, s, and t with $(r,s) \in E$, $(s,t) \in E$, and $\varepsilon \in \mathcal{L}(s)$ we also have $(s,t) \in E$. A generalized SOA is *well-looped* if there is no loop of the form (s^+, s^+) in E. A generalized SOA is *well-behaved* if it is ε-transitive, well-looped and all of its states, except *src* and *sink*, are saturated terms.

Note that SOAs are trivially well-behaved.

Proposition 2.19 (Backwards Rewriting)**.** *If G is a well-behaved generalized SOA but not a SOA (i.e., it contains at least one state that is not src, sink or an alphabet symbol) then there exists a well-behaved generalized SOA F such that $F \rightsquigarrow G$.*

Proof. Pick $x \in V(G) - (\Sigma \cup \{src, sink\})$ arbitrarily. Note that since G is well-behaved, x is a saturated term. We then construct F by expanding x into one or more new states and edges depending on both the shape of x and the presence of a loop on x, as shown in Fig. 2.7 and 2.1.2. There, we range over saturated basic terms by e, e_1, and e_2; and over saturated terms by t, t_1, and t_2.

For instance, if $x = e^+$ with e a basic term then there can be no loop on x since G is well-looped. As shown in Fig. 2.7, we then construct F by replacing e^+ by e and adding the edge (e,e). It is readily verified that since G is well-behaved, so is F. Moreover, since $e \in \mathrm{Succ}_F(e)$, we can apply ITERATION e^+ on F, which yields G, as desired.

As another example, when $x = e_1?\,.\,e_2?$ has a loop, then we construct F by (see Fig. 2.1.2):

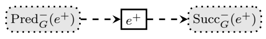
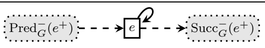
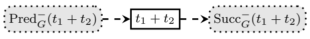
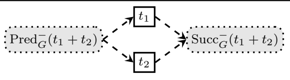
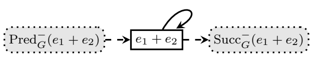
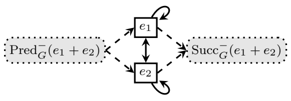
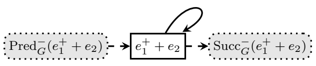
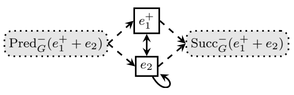
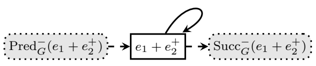
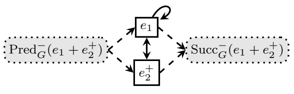
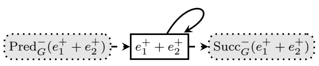
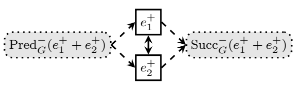
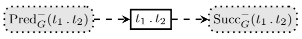
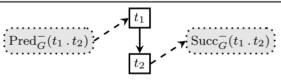
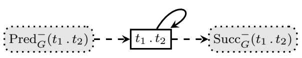
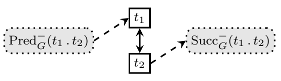
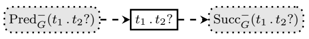
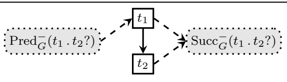
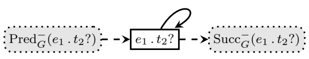
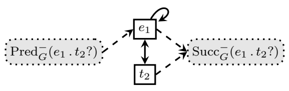
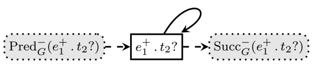
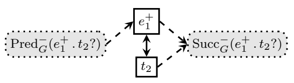
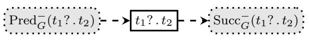
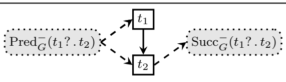
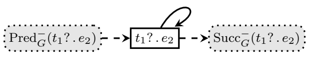
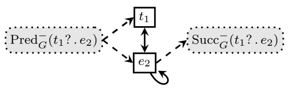
x	loop on x?	then G looks like	and we take F to be
e^+	no		
$t_1 + t_2$	no		
$e_1 + e_2$	yes		
$e_1^+ + e_2$	yes		
$e_1 + e_2^+$	yes		
$e_1^+ + e_2^+$	yes		
$t_1 \,.\, t_2$	no		
$t_1 \,.\, t_2$	yes		
$t_1 \,.\, t_2?$	no		
$e_1 \,.\, t_2?$	yes		
$t_1^+ \,.\, t_2?$	yes		
$t_1? \,.\, t_2$	no		
$t_1? \,.\, e_1$	yes		

Figure 2.7: Illustration of the proof of Proposition 2.19.

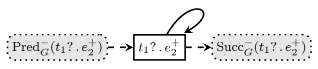
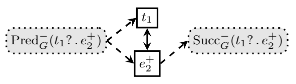
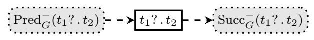
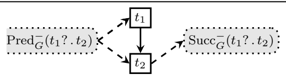
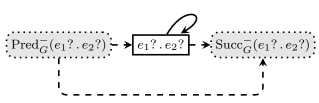
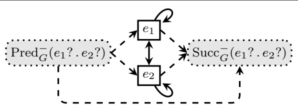
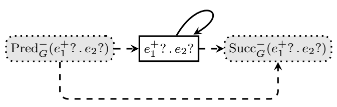
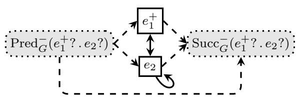
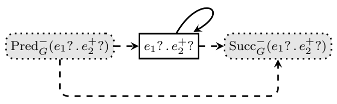
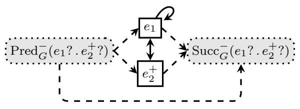
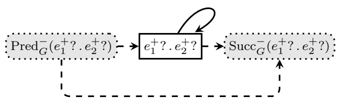
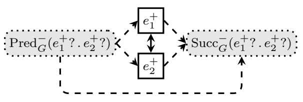
x	loop on x?	then G looks like	and we take F to be
$t_1?\,.\,e_2^+$	yes		
$t_1?\,.\,t_2?$	no		
$e_1?\,.\,e_2?$	yes		
$e_1^+?\,.\,e_2?$	yes		
$e_1?\,.\,e_2^+?$	yes		
$e_1^+?\,.\,e_2^+?$	yes		

Figure 2.8: Illustration of the proof of Proposition 2.19 (Continued from Fig. 2.7).

- expanding x into two new states e_1 and e_2;
- making every $v \in \mathrm{Pred}^{-}_{G}(x)$ a predecessor of both e_1 and e_2
- making every $w \in \mathrm{Succ}^{-}_{G}(x)$ a successor of both e_1 and e_2; and
- adding the edges $(e_1, e_1), (e_1, e_2), (e_2, e_1), (e_2, e_2)$.

Since x was saturated, so are e_1 and e_2. Hence, all states in F are saturated terms. Moreover, since G was well-looped, and since we have only added loops on the basic terms e_1 and e_2 (which themselves cannot be of the form e^+), F is also well-looped. Finally, since G is ε-transitive, so is F. Hence F is well-behaved. Furthermore, we can apply CONCATENATION $e_1? \,.\, e_2?$ on F:

1. $e_1, e_2 \notin \{\mathit{src}, \mathit{sink}\}$;
2. $e_2 \in \mathrm{Succ}_F(e_1)$;
3. $\mathrm{Succ}_F(e_1) - \{e_1, e_2\} = \mathrm{Succ}_F(e_2) - \{e_1, e_2\}$;
4. $\mathrm{Pred}_F(e_1) - \{e_1, e_2\} = \mathrm{Pred}_F(e_2) - \{e_1, e_2\}$;
5. Since $\varepsilon \in \mathcal{L}(e_1? \,.\, e_2?)$ and G is ε-transitive, there is an edge in G from every $v \in \mathrm{Pred}^{-}_{G}(e_1? \,.\, e_2?)$ to every $w \in \mathrm{Succ}^{-}_{G}(e_1? \,.\, e_2?)$. Since these edges were left untouched when constructing F, it is readily verified that $\mathrm{Pred}_F(e_1) \times \mathrm{Succ}_F(e_2) \subseteq E(F)$;
6. Finally, $e_1 \in \mathrm{Succ}_F(e_2)$, $e_1 \in \mathrm{Succ^+}_F(e_1)$, and $e_2 \in \mathrm{Succ}^+(e_2)$.

Then it is easy to see that applying CONCATENATION $e_1? \,.\, e_2?$ on F yields G, as desired.

The other cases are similar. □

Corollary 2.20. *For every SOA G that is equivalent to a proper* SORE *r there exists a final generalized SOA H such that $G \rightsquigarrow^* H$.*

Proof. Since r is a proper SORE there exists a saturated factor f equivalent to r by Corollary 2.17. Let the saturated factor h be defined as follows:

- if $f = t?$ for some t, then $h := f$;
- if $f = t$ for some t and $\varepsilon \in \mathcal{L}(f)$, then $h := t?$;
- otherwise $f = t$ for some t with $\varepsilon \notin \mathcal{L}(t)$, and $h := f$.

Note that h is equivalent to f (and hence to r) and that $\varepsilon \in \mathcal{L}(h)$ if, and only if, h is an optional saturated term $t?$. Now let H be the final generalized SOA with $V(H) = \{\mathit{src}, h, \mathit{sink}\}$, and $E(H) = \{(\mathit{src}, h), (h, \mathit{sink})\}$. We claim that $G \leadsto^* H$, which can be seen as follows. Let G_n be defined by case analysis on h:

- if $h = t?$ for some saturated term t, then take

  $$V(G_n) := \{\mathit{src}, t, \mathit{sink}\} \text{ and}$$
  $$E(G_n) := \{(\mathit{src}, t), (t, \mathit{sink}), (\mathit{src}, \mathit{sink})\}.$$

  Clearly, G_n is well-behaved by construction. Moreover, $G_n \leadsto H$.

- otherwise $h = t$ for some saturated term t with $\varepsilon \notin \mathcal{L}(t)$ and we take $G_n := H$. Again, G_n is well-behaved by construction. Moreover, $G_n \leadsto^* H$.

By repeated application of Proposition 2.19 we can start from G_n and keep rewriting backwards along a sequence of well-behaved generalized SOAs until we end up with a SOA G_0 such that $G_0 \leadsto^* G_n$. By soundness of the rewrite rules (Proposition 2.8) we then have $\mathcal{L}(G_0) = \mathcal{L}(G_n) = \mathcal{L}(H) = \mathcal{L}(h) = \mathcal{L}(r)$. Since also $\mathcal{L}(G) = r$ and since G and G_0 are SOAs, it must be the case that $G = G_0$. Indeed, if G and G_0 were to differ in a state or an edge between states, then—since each state in a SOA participates in a walk from from *src* to *sink*—we can always construct a walk $\mathit{src}, a, \ldots, b, \mathit{sink}$ in one automaton (say G) that is not a walk in the other. Hence, $a \ldots b \in \mathcal{L}(G)$ but $a \ldots b \notin \mathcal{L}(G_0)$, contradicting $\mathcal{L}(G) = \mathcal{L}(G_0)$. Therefore, $G = G_0 \leadsto^* G_n \leadsto^* H$ with H final, as desired. □

Having established that any proper SOA equivalent to a SORE can always be rewritten into a final generalized SOA using a specific sequence of rewrite steps, it remains to show that *any* sequence of rewrite steps starting from that SOA ends in a final generalized SOA. Thereto, the following definitions and observations are in order.

Definition 2.21 (Isomorphism)**.** Let $r \sim s$ denote that regular expressions r and s are both iterated (i.e., $r = r'^{+}$ and $s = s'^{+}$ for some r' and s'), or are both not iterated (i.e., $r \neq r'^{+}$ and $s \neq s'^{+}$ for any r' and s'). Two generalized SOAs G and H are *isomorphic*, denoted $G \simeq H$, if there exists a one-to-one onto mapping $\rho \colon V(G) \to V(H)$ such that

1. $\rho(\mathit{src}) = \mathit{src}$; $\rho(\mathit{sink}) = \mathit{sink}$; and $v \sim \rho(v)$ for all $v \in V(G) - \{\mathit{src}, \mathit{sink}\}$; and

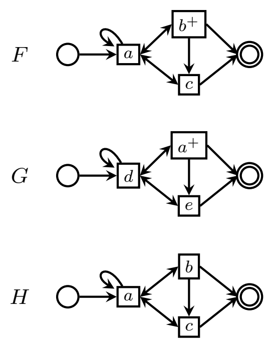


Figure 2.9: Illustration of isomorphism: $F \simeq G$, $F \not\simeq H$, and $G \not\simeq H$.

2. $(v, w) \in E(G) \Leftrightarrow (\rho(v), \rho(w)) \in E(H)$ for all $v, w \in V(G)$.

To illustrate, the generalized SOAs F and G in Figure 2.9 are isomorphic, but F and H are not.

Observe that if $H \simeq F$, then H is final if and only if F is also final. Hence, in order to show that $G \rightsquigarrow^* H$ with H final implies that any sequence of rewrite steps starting from G ends in a final automaton, it suffices to show that for any $G \rightsquigarrow^* F$ with no rule applicable to F we have $H \simeq F$. Thereto, we show that our rewrite rules are *confluent modulo* $\simeq$, a concept from the theory of abstract reduction systems that is defined as follows.

Definition 2.22 (Joinable, confluence)**.** Let $\equiv$ be an equivalence relation. Two generalized SOAs H_1 and H_2 are *joinable modulo* $\equiv$, denoted $H_1 \downarrow_\equiv H_2$, if they can be made equivalent through rewriting, i.e., if there exist I_1 and I_2 such that $H_1 \rightsquigarrow^* I_1$; $H_2 \rightsquigarrow^* I_2$; and $I_1 \equiv I_2$. Rewriting is said to be *confluent modulo* $\equiv$ if $G \rightsquigarrow^* H_1$ and $G \rightsquigarrow^* H_2$ implies $H_1 \downarrow_\equiv H_2$, for all G, H_1, and H_2.

Proposition 2.23. *Suppose that there is a sequence of rewrite steps starting from G that ends in a final automaton. If rewriting is confluent modulo $\simeq$, then any sequence of rewrite steps starting from G ends in a final automaton.*

Proof. Suppose that we rewrite $G \rightsquigarrow^* H_1$ with no rewrite rule applicable to H_1. By hypothesis, there is some final H_2 such that $G \rightsquigarrow^* H_2$. Hence $H_1 \downarrow_\simeq H_2$ by confluence of rewriting modulo $\simeq$. As such, there exist I_1 and I_2 such that $H_1 \rightsquigarrow^* I_1$; $H_2 \rightsquigarrow^* I_2$; and $I_1 \simeq I_2$. However, since no rewrite rule applies to H_1 (by hypothesis), nor to H_2 (as H_2 is final), H_1 must equal I_1 and H_2 must equal I_2. As such, $H_1 \simeq H_2$. Then, since H_2 is final, so is H_1. □

It is a classical result in the theory of abstract rewrite systems [Hue80, Ohl98] that in order to show that rewriting is confluent modulo an equivalence relation $\equiv$, it suffices to show that rewriting is *terminating*; *locally consistent with respect to* $\equiv$; and *locally confluent modulo* $\equiv$.

Definition 2.24. Rewriting is *terminating* if there is no infinite sequence $G_1 \rightsquigarrow G_2 \rightsquigarrow \dots$. Let $\equiv$ be an equivalence relation. Rewriting is *locally consistent with respect to* $\equiv$ if $G \rightsquigarrow G'$ and $G \equiv H$ implies that there exists $H' \equiv G'$ such that $H \rightsquigarrow H'$, for all G, G', and H. Rewriting is *locally confluent modulo* $\equiv$ if $G \rightsquigarrow H_1$ and $G \rightsquigarrow H_2$ implies $H_1 \downarrow_{\equiv} H_2$, for all G, H_1, and H_2.

We have already established that rewriting is terminating in Proposition 2.9. It remains to show the other two properties.

Proposition 2.25 (Local consistency w.r.t. $\simeq$)**.** *Let $G \simeq H$ be two isomorphic generalized SOAs. If $G \rightsquigarrow G'$ then there exists $H' \simeq G'$ such that $H \rightsquigarrow H'$.*

Proof. Let $\rho\colon V(G) \to V(H)$ be the one-to-one onto mapping that testifies $G \simeq H$. To ease notation, we extend ρ pointwise to sets. For example, $\rho(\{r, s, t\}) = \{\rho(r), \rho(s), \rho(t)\}$. It is then readily verified that

1. $\rho(\mathrm{Pred}_G(v)) = \mathrm{Pred}_H(\rho(v))$;
2. $\rho(\mathrm{Pred}^+_G(v)) = \mathrm{Pred}^+_H(\rho(v))$;
3. $\rho(\mathrm{Succ}_G(v)) = \mathrm{Succ}_H(\rho(v))$; and
4. $\rho(\mathrm{Succ}^+_G(v)) = \mathrm{Succ}^+_H(\rho(v))$,

for all $v \in V(G)$. As such, if r, s are states in G that satisfy the preconditions of one of the rewrite rules for disjunction or concatenation, then $\rho(r)$ and $\rho(s)$ also satisfy these preconditions in H. Hence, the same rewrite rule is applicable to H. Similarly, if r is a state in G that satisfies the precondition of Iteration or Optional, then $\rho(r)$ also satisfies this precondition, and hence the same rewrite rule is applicable to H. It is then readily verified that the results of applying the same rewrite rule on two isomorphic automata are again isomorphic. □

Proposition 2.26 (Local Confluence modulo $\simeq$)**.** *If $G \rightsquigarrow H_1$ and $G \rightsquigarrow H_2$ then $H_1 \downarrow_{\simeq} H_2$.*

Proof. To ease the discussion that follows, let $\overset{\alpha}{\rightsquigarrow}$ with α a regular expression be the binary relation on generalized SOAs such that $F_1 \overset{\alpha}{\rightsquigarrow} F_2$ if, and only if, F_1 rewrites into F_2 in a single step, and α is the new state introduced by this rewriting. For instance, $F_1 \overset{r+s}{\rightsquigarrow} F_2$ indicates that we obtain F_2 by applying Disjunction $r + s$ on F_1. Similarly, $F_1 \overset{t^+}{\rightsquigarrow} F_2$ indicates that we obtain F_2 by applying Iteration t^+ on F_1. Note that α uniquely determines the rewrite rule used, as well as the states that are contracted during the application of that rule.

Now clearly, since $G \rightsquigarrow H_1$ and $G \rightsquigarrow H_2$ there exist α_1 and α_2 such that $G \overset{\alpha_1}{\rightsquigarrow} H_1$ and $G \overset{\alpha_2}{\rightsquigarrow} H_2$. The proof then proceeds by a case analysis on α_1 and α_2. We first observe that if α_1 and α_2 are alphabet-disjoint, then $\overset{\alpha_1}{\rightsquigarrow}$ and $\overset{\alpha_2}{\rightsquigarrow}$ *commute*: there is some I such that $G \overset{\alpha_1}{\rightsquigarrow} H_1 \overset{\alpha_2}{\rightsquigarrow} I$ and $G \overset{\alpha_2}{\rightsquigarrow} H_2 \overset{\alpha_1}{\rightsquigarrow} I$. For instance, suppose that $\alpha_1 = r + s$ and $\alpha_2 = t^+$, with r, s, and t all alphabet-disjoint. Then, $H_1 = G[r, s/r + s]$ since $G \overset{r+s}{\rightsquigarrow} H_1$ and hence, by definition of state contraction,

$$\mathrm{Succ}_{H_1}(t) = \mathrm{Succ}_G(t) - \{r, s\} \cup \{r + s \mid r \in \mathrm{Succ}_G(t)\}. \tag{2.1}$$

Then, since $G \overset{t^+}{\rightsquigarrow} H_2$, G satisfies all preconditions of ITERATION t^+. In particular, $t \in \mathrm{Succ}_G(t)$ and hence, since r, s, and t are all distinct, also $t \in \mathrm{Succ}_{H_1}(t)$ by (2.1). Therefore, ITERATION t^+ is indeed applicable to H_1. Let I_1 be its result. Using a similar reasoning it can be verified that DISJUNCTION $r + s$ is applicable to H_2, yielding an automaton I_2. Then

$$\begin{aligned} I_1 &= H_1[t/t^+] \ominus (t^+, t^+) \\ &= G[r, s/r + s][t/t^+] \ominus (t^+, t^+) \\ &= G[t/t^+] \ominus (t^+, t^+)[r, s/r + s] \\ &= H_2[r, s/r + s] \\ &= I_2 \end{aligned}$$

and hence $G \overset{\alpha_1}{\rightsquigarrow} H_1 \overset{\alpha_2}{\rightsquigarrow} I$ and $G \overset{\alpha_2}{\rightsquigarrow} H_2 \overset{\alpha_1}{\rightsquigarrow} I$, as claimed. The reasoning for the other alphabet-disjoint combinations of α_1 and α_2 is similar.

Since commutativity clearly implies $H_1 \downarrow_{\simeq} H_2$, it remains to verify the proposition in the cases where α_1 and α_2 are not alphabet-disjoint. By definition of the rewrite rules, α_1 and α_2 are of the form

$$r + s \mid r \,.\, s \mid r? \,.\, s \mid r \,.\, s? \mid r? \,.\, s? \mid r^+ \mid r?,$$

where r and s range over states of G. Since all states of G are alphabet-disjoint, α_1 and α_2 can share an alphabet symbol only if they share a whole state of G. Using these observations, Table 2.1 lists all possible combinations of α_1 and α_2 of the above form that share a state in G. In that table, the regular expressions r,s, and t range over distinct (and hence alphabet-disjoint) states of G.

Now observe that for most combinations of α_1 and α_2 in Table 2.1 we cannot have $G \overset{\alpha_1}{\rightsquigarrow} H_1$ and simultaneously $G \overset{\alpha_2}{\rightsquigarrow} H_2$.

α_1	Possible values for α_2								
$r + s$	$\underline{r + s}$	$r . s$	$r . s?$	$\underline{r? . s}$	$r? . s?$	$\underline{r^+}$	$\underline{s^+}$	$r?$	$s?$
	$\underline{s + r}$	$s . r$	$s . r?$	$s? . r$	$\underline{s? . r?}$				
	$\underline{r + t}$	$r . t$	$r . t?$	$r? . t$	$\underline{r? . t?}$				
	$\underline{t + r}$	$t . r$	$t . r?$	$t? . r$	$\underline{t? . r?}$				
	$\underline{s + t}$	$s . t$	$s . t?$	$s? . t$	$\underline{s? . t?}$				
	$\underline{t + s}$	$t . s$	$t . s?$	$t? . s$	$\underline{t? . s?}$				
$r . s$	$r + s$	$\underline{r . s}$	$r . s?$	$r? . s$	$r? . s?$	r^+	s^+	$r?$	$s?$
	$s + r$	$s . r$	$s . r?$	$s? . r$	$s? . r?$				
	$r + t$	$r . t$	$r . t?$	$r? . t$	$r? . t?$				
	$t + r$	$\underline{t . r}$	$t . r?$	$\underline{t? . r}$	$t? . r?$				
	$s + t$	$\underline{s . t}$	$\underline{s . t?}$	$s? . t$	$s? . t?$				
	$t + s$	$t . s$	$t . s?$	$t? . s$	$t? . s?$				
$r . s?$	$r + s$	$r . s$	$\underline{r . s?}$	$r? . s$	$r? . s?$	$\underline{r^+}$	s^+	$r?$	$s?$
	$s + r$	$s . r$	$s . r?$	$s? . r$	$s? . r?$				
	$r + t$	$r . t$	$r . t?$	$r? . t$	$r? . t?$				
	$t + r$	$\underline{t . r}$	$t . r?$	$\underline{t? . r}$	$t? . r?$				
	$s + t$	$s . t$	$s . t?$	$\underline{s? . t}$	$\underline{s? . t?}$				
	$t + s$	$t . s$	$t . s?$	$t? . s$	$t? . s?$				
$r? . s$	$r + s$	$r . s$	$r . s?$	$\underline{r? . s}$	$r? . s?$	r^+	$\underline{s^+}$	$r?$	$s?$
	$s + r$	$s . r$	$s . r?$	$s? . r$	$s? . r?$				
	$r + t$	$r . t$	$r . t?$	$r? . t$	$r? . t?$				
	$t + r$	$t . r$	$\underline{t . r?}$	$t? . r$	$\underline{t? . r?}$				
	$s + t$	$\underline{s . t}$	$\underline{s . t?}$	$s? . t$	$s? . t?$				
	$t + s$	$t . s$	$t . s?$	$t? . s$	$t? . s?$				
$r? . s?$	$\underline{r + s}$	$r . s$	$r . s?$	$r? . s$	$r? . s?$	$\underline{r^+}$	$\underline{s^+}$	$r?$	$s?$
	$\underline{s + r}$	$s . r$	$s . r?$	$s? . r$	$\underline{s? . r?}$				
	$\underline{r + t}$	$r . t$	$r . t?$	$r? . t$	$\underline{r? . t?}$				
	$\underline{t + r}$	$t . r$	$\underline{t . r?}$	$t? . r$	$\underline{t? . r?}$				
	$\underline{s + t}$	$s . t$	$s . t?$	$\underline{s? . t}$	$\underline{s? . t?}$				
	$\underline{t + s}$	$t . s$	$t . s?$	$t? . s$	$\underline{t? . s?}$				
r^+	$\underline{r + s}$	$r . s$	$\underline{r . s?}$	$r? . s$	$\underline{r? . s?}$	$\underline{r^+}$	$r?$		
	$\underline{s + r}$	$s . r$	$s . r?$	$\underline{s? . r}$	$\underline{s? . r?}$				
$r?$	$r + s$	$r . s$	$r . s?$	$r? . s$	$r? . s?$	r^+	$r?$		
	$s + r$	$s . r$	$s . r?$	$s? . r$	$s? . r?$				

Table 2.1: All possible combinations of α_1 and α_2 that share a state in the proof of Proposition 2.26. For each possible value for α_1, only the corresponding values for α_2 that are underlined can be simultaneously applied to G.

- For instance, suppose for the purpose of contradiction that $G \overset{r+s}{\rightsquigarrow} H_1$ and $G \overset{r\,.\,s}{\rightsquigarrow} H_2$. Since $G \overset{r\,.\,s}{\rightsquigarrow} H_2$, G satisfies the precondition of CONCATENATION $r\,.\,s$, and hence $\mathrm{Succ}_G(r) = \{s\}$. Also, since G is a generalized SOA, there is a walk from r to *sink*. Since s is the only successor of r, such a walk can only exist if s has a successor $u \notin \{r, s\}$. Since $G \overset{r+s}{\rightsquigarrow} H_1$, G satisfies the precondition of DISJUNCTION $r+s$, and hence u must also be a successor of r. But then $\mathrm{Succ}_G(r) \neq \{s\}$, which gives the desired contradiction.

- As another example, suppose for the purpose of contradiction that $G \overset{r+s}{\rightsquigarrow} H_1$ and $G \overset{r\,.\,t?}{\rightsquigarrow} H_2$ with r, s, and t distinct states in G. Since $G \overset{r\,.\,t?}{\rightsquigarrow} H_2$, G satisfies the precondition of CONCATENATION $r\,.\,t?$, and hence $\mathrm{Pred}_G(t) = \{r\}$. As such, $t \in \mathrm{Succ}_G(r)$. Then, since $G \overset{r+s}{\rightsquigarrow} H_1$, G satisfies the precondition of DISJUNCTION $r+s$. In particular, t must also be a successor of s. But then $s \in \mathrm{Pred}_G(t)$, and hence $\mathrm{Pred}_G(t) \neq \{r\}$, which gives the desired contradiction.

By similar examination of the other combinations of α_1 and α_2 in Table 2.1 we obtain that for each possible value of α_1, only the corresponding values for α_2 that are underlined in Table 2.1 can be applied simultaneously to G. For each such simultaneously applicable combination, we illustrate in Figure 2.10 how H_1 and H_2 can be joined. For economy of space, we have omitted the trivial cases where $\alpha_1 = \alpha_2$. For the same reason, cases that are equivalent to one of the already shown cases are also omitted. For instance, the case where $\alpha_1 = r\,.\,s$ and $\alpha_2 = t?\,.\,r$ is not shown. Note however, that since the roles of α_1 and α_2 can always be swapped, and since r, s, t can always be permuted, this case is treated in the equivalent case $\alpha_1 = r?\,.\,s$ and $\alpha_2 = s\,.\,t$.

- For instance, consider the case where $\alpha_1 = r + s$ and $\alpha_2 = r?\,.\,s?$. Since $G \overset{r?\,.\,s?}{\rightsquigarrow} H_2$, G satisfies the precondition of CONCATENATION $r?\,.\,s?$, and hence $s \in \mathrm{Succ}_G(r)$. Moreover, since $G \overset{r+s}{\rightsquigarrow} H_1$, G also satisfies the precondition of DISJUNCTION $r + s$ and r, s therefore have the same (extended) predecessor and successor set. As such,

$$\begin{aligned} s \in \mathrm{Succ}_G(r) &\Rightarrow s \in \mathrm{Succ}_G^+(s) \\ &\Rightarrow s \in \mathrm{Pred}_G^+(s) \\ &\Rightarrow s \in \mathrm{Pred}_G(r) \\ &\Rightarrow r \in \mathrm{Succ}_G(s) \end{aligned}$$

Also, by the precondition of CONCATENATION $r?\,.\,s?$, $\mathrm{Pred}_G(r) \times \mathrm{Succ}_G(s) \subseteq E(G)$. Hence, G, H_1, and H_2 are as illustrated in Figure 2.11. Clearly,

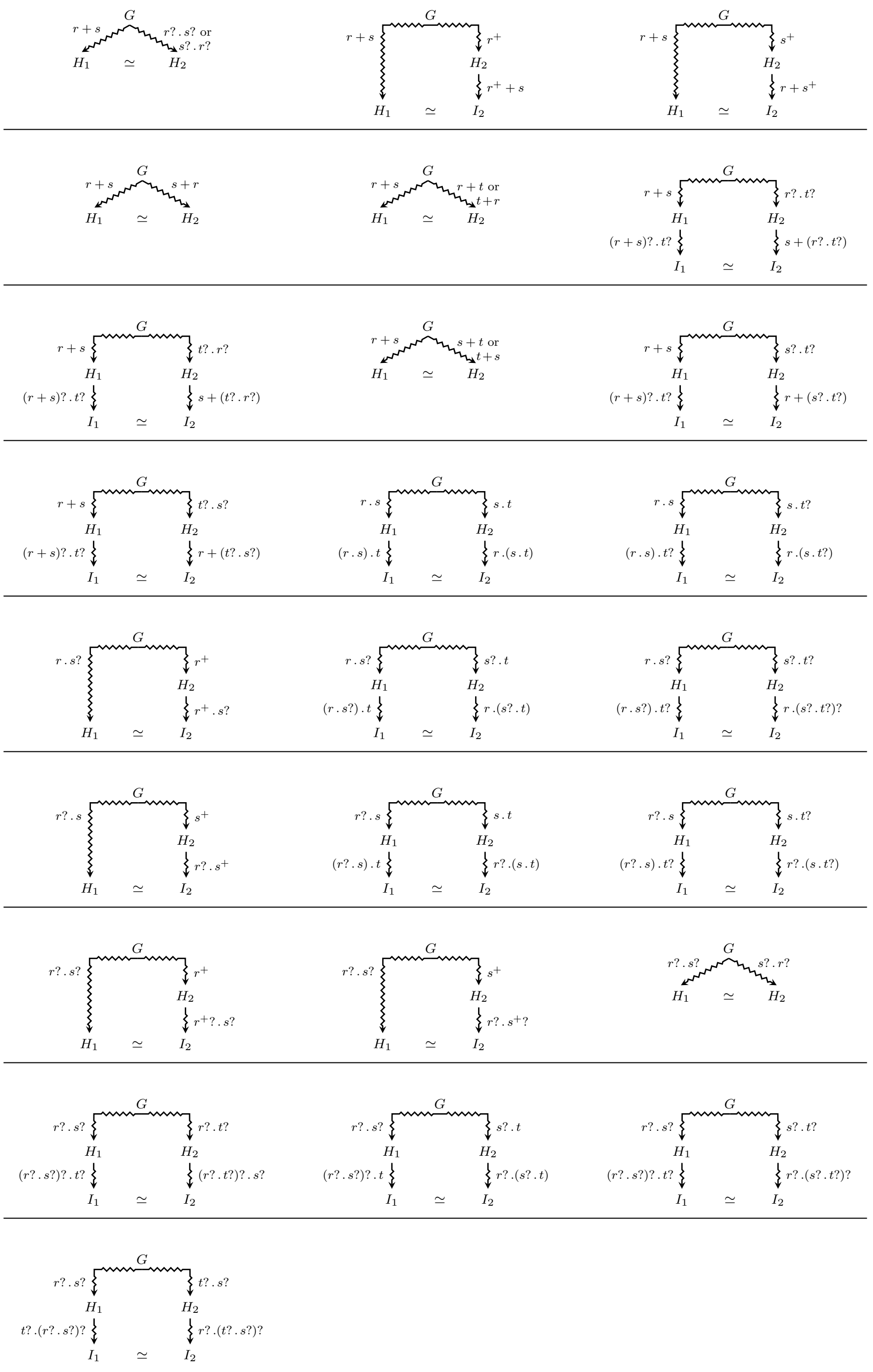


Figure 2.10: Illustration of the proof of Proposition 2.26.

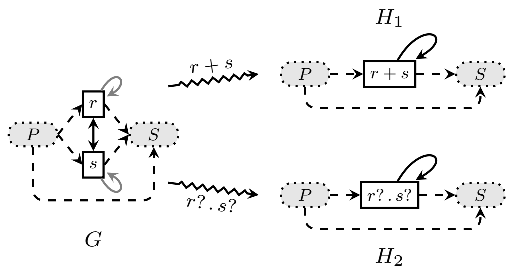


Figure 2.11: Illustration of the interaction of $r + s$ and $r?\,.\,s?$ in the proof of Proposition 2.26. P is the set $\mathrm{Pred}_G(r) - \{r, s\} = \mathrm{Pred}_G(s) - \{r, s\}$. S is the set $\mathrm{Succ}_G(r) - \{r, s\} = \mathrm{Succ}_G(s) - \{r, s\}$. The gray loops on r and s indicate that $r \in \mathrm{Succ}^+_G(r)$ and $s \in \mathrm{Succ}^+_G(s)$.

the one-to-one onto mapping $\rho\colon V(H_1) \to V(H_2)$ that is the identity on every state in $V(H_1)$ except $r + s$, for which $\rho(r + s) = r?\,.\,s?$, shows that H_1 and H_2 are isomorphic. As such, $H_1 \downarrow_{\simeq} H_2$.

- As another example, consider the case where $\alpha_1 = r\,.\,s?$ and $\alpha_2 = s?\,.\,t?$. Since $G \overset{r\,.\,s?}{\rightsquigarrow} H_1$, G satisfies the precondition of CONCATENATION $r\,.\,s?$, and hence $\mathrm{Pred}_G(s) = \{r\}$. In particular, $t \notin \mathrm{Pred}_G(s)$. Therefore, $s \notin \mathrm{Succ}_G(t)$. Since $G \overset{s?\,.\,t?}{\rightsquigarrow} H_1$, G also satisfies the precondition of CONCATENATION $s?\,.\,t?$. By this precondition, the fact that $s \notin \mathrm{Succ}_G(t)$ implies $s \notin \mathrm{Succ}_G(s)$ and $t \notin \mathrm{Succ}_G(t)$. We then distinguish two possibilities.

  (1) $r \in \mathrm{Succ}_G(s)$. Since s and t have the same predecessors (modulo s and t) by the precondition of CONCATENATION $s?\,.\,t?$, also $r \in \mathrm{Succ}_G(t)$. Moreover, by the precondition of CONCATENATION $r\,.\,s?$, $r \in \mathrm{Succ}^+_G(r)$. As such, G, H_1, and H_2 are as illustrated in Figure 2.12. Clearly, H_1 satisfies the precondition of CONCATENATION $(r\,.\,s?)\,.\,t?$, while H_2 satisfies the precondition of CONCATENATION $r\,.(s?\,.\,t?)?$. Let I_1 and I_2 be the results of this rewriting, as shown in Figure 2.12. Clearly, the one-to-one onto mapping $\rho\colon V(H_1) \to V(H_2)$ that is the identity on every state in $V(H_1)$ except $(r\,.\,s?)\,.\,t?$, for which $\rho((r\,.\,s?)\,.\,t?) = r\,.(s?\,.\,t?)?$, shows that I_1 and I_2 are isomorphic. As such, $H_1 \downarrow_{\simeq} H_2$.

  (2) $r \notin \mathrm{Succ}_G(s)$. Since s and t have the same predecessors (modulo s and t) by the precondition of CONCATENATION $s?\,.\,t?$, also $r \notin \mathrm{Succ}_G(t)$. Then, by the precondition of CONCATENATION $r\,.\,s?$, also $r \notin \mathrm{Succ}_G(r)$. A reasoning similar to case (1) then shows that again $H_1 \downarrow_{\simeq} H_2$.

The reasoning for the other simultaneously applicable combinations is similar.

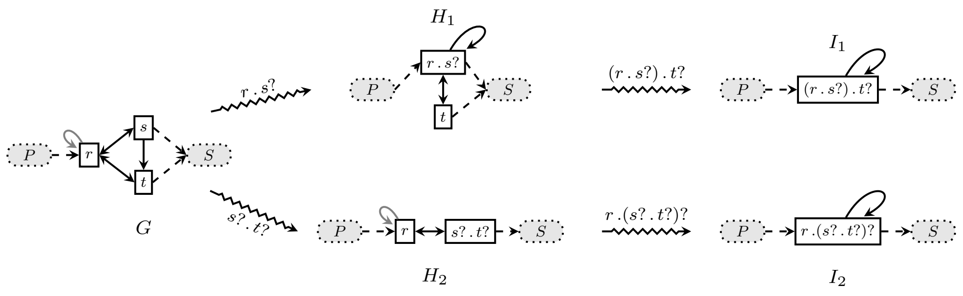


Figure 2.12: Illustration of the interaction of $r\,.\,s?$ and $s?\,.\,t?$ in the proof of Proposition 2.26. P is the set $\mathrm{Pred}_G(r) - \{r, s, t\}$. S is the set $\mathrm{Succ}_G(s) - \{r, s, t\} = \mathrm{Succ}_G(t) - \{r, s, t\}$. The gray loop on r indicates that $r \in \mathrm{Succ}^+_G(r)$.

$\square$

Corollary 2.27 (Confluence modulo $\simeq$). *If $F \rightsquigarrow^* G_1$ and $F \rightsquigarrow^* G_2$ then $G_1 \downarrow_\simeq G_2$.*

Hence, by Proposition 2.23:

Corollary 2.28. *Suppose that there is a sequence of rewrite steps starting from G that ends in a final automaton. Then any sequence of rewrite steps starting from G ends in a final automaton.*

This concludes the proof of Proposition 2.13, and hence also the proof of Theorem 2.10.

2.1.3 Discussion

It should be noted that while the result of REWRITE is always a SORE, this SORE need not be easy to read (depending on the order of rewriting). For instance, as illustrated in Figure 2.12, it is possible for REWRITE to generate an expression $r\,.(s?\,.\,t?)?$. Clearly, the optional around $(s?\,.\,t?)$ is redundant. Removing it leads to the simpler $r\,.(s?\,.\,t?)$. For presentation to the user, it is therefore advisable to post-process the result of REWRITE (and its variations in Section 2.2) using a regular expression simplification algorithm.

2.2 Dealing with missing data

The results of Section 2.1 suggest the following method to infer a SORE from a given sample S:

1. First, use 2T-INF to learn from S an automaton representation G of the target SORE r.

2. Next, convert G into a SORE using REWRITE.

If S is a representative sample of r then G is equivalent to r by Proposition 2.4. Therefore, REWRITE(G) does not fail by Theorem 2.10, and hence REWRITE(G) is equivalent to r.

Unfortunately, real-world samples are rarely representative. For instance, for target $r = (a_1 + \cdots + a_n)^+$ and increasing values of n, it is increasingly unlikely that a sample bears witness to each of the n^2 2-grams needed to represent r. On such non-representative samples, 2T-INF will construct an automaton for which $\mathcal{L}(G)$ is a strict subset of $\mathcal{L}(r)$. In particular, this automaton need not be equivalent to a SORE, and REWRITE(G) can fail. Figure 2.13 shows an example.

For that reason, we present in this section two modifications of REWRITE that "repair" G when rewriting gets stuck in a non-final automaton. The first modification, RWR, picks a single repair when rewriting gets stuck, independent of how the repair affects G. The second modification, RWR^2, in contrast, considers multiple repair strategies and selects the one that extends G in a minimal way. The repair rules used by both algorithms are shown in Figure 2.14. After a repair rule is applied, the automaton necessarily satisfies the precondition of the corresponding rewrite rule. Now note:

Proposition 2.29. *Let G be a proper generalized SOA. If G is not final and no rewrite rule applies to G, then at least one of the repair rules in Figure 2.14 applies to G.*

Proof. Since G is proper, it recognizes at least one non-empty word. Clearly, this can only happen when *src* has a successor r distinct from *sink*. We distinguish two cases.

- Either r has a successor s distinct from *src*, *sink*, and r. Clearly, REPAIR $r?\,.\,s?$ is then applicable to G.

- If r does not have such a successor s, then we claim that *src* has another successor t, distinct from *src*, *sink*, and r. Indeed, suppose for the purpose of contradiction that no such successor exists. Then, since every state in G participates in a walk from *src* to *sink*, either $E(G) =$

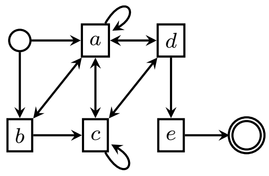


Figure 2.13: The SOA generated by 2T-INF for the non-representative sample $S = \{bacacdacde, abccaadcde\}$. The only rewrite rules that can be applied are ITERATION a^+ and ITERATION c^+, after which REWRITE gets stuck in a non-final automaton and fails.

$\{(src, r), (r, sink)\}$, or $E(G) = \{(src, r), (r, r), (r, sink)\}$. In the first case G is final, in the second we can rewrite G using ITERATION r^+— a contradiction in both cases. As such, the claimed t exists. Then, since $src \in \mathrm{Pred}_G(r) \cap \mathrm{Pred}_G(t)$, REPAIR $r + t$ is applicable to G.

□

As such, we can always apply a repair rule if rewriting gets stuck in a non-final automaton, after which rewriting can continue.

2.2.1 A greedy approach: RWR

An outline of RWR (short for REWRITE with REPAIRS), is shown in Algorithm 3. Like REWRITE, it first checks whether its input G is equivalent to $\emptyset$ or ε. Otherwise, G is rewritten using the rewrite rules in Figure 2.4 until a final automaton is reached, arbitrarily selecting a repair rule when rewriting gets stuck. (In our implementation we first check whether there are r and s for which REPAIR $r \,.\, s?$ can be applied. Then we check whether there are r and s for which REPAIR $r? \,.\, s$ can be applied. Next, we check for REPAIR $r + s$ and finally for REPAIR $r? \,.\, s?$.)

Since the repair rules add edges to G, thereby increasing $\mathcal{L}(G)$, we may conclude:

Theorem 2.30. *For a SOA G, RWR always produces a SORE r with $L(G) \subseteq L(r)$. Moreover, if G is equivalent to a* SORE*, then $\mathcal{L}(G) = \mathcal{L}(r)$.*

(The second statement follows by Theorem 2.10.) Combined with Proposition 2.4, we hence obtain:

Corollary 2.31. *Let M be the composition of* 2T-INF *with* RWR*, i.e., $M(S) :=$ RWR(2T-INF(S)). Then M learns the class of SOREs from positive data.*

Repair $r + s$

Precondition	*Add edges*
$\mathrm{Pred}_G(r) \cap \mathrm{Pred}_G(s) \neq \emptyset$ or	(1) for all $u \in (\mathrm{Pred}_G(r) \cup \mathrm{Pred}_G(s)) - \{r, s\}$,
$\mathrm{Succ}_G(r) \cap \mathrm{Succ}_G(s) \neq \emptyset$	add (u, r), (u, s)
	(2) for all $u \in (\mathrm{Succ}_G(r) \cup \mathrm{Succ}_G(s)) - \{r, s\}$,
	add (r, u), (s, u)
	(3) if $r \in \mathrm{Succ}_G(s)$ or $s \in \mathrm{Succ}_G(r)$ then
	(a) add (r, s), (s, r)
	(b) if $r \notin \mathrm{Succ}^+_G(r)$, add (r, r)
	(c) if $s \notin \mathrm{Succ}^+_G(s)$, add (s, s)

Repair $r \,.\, s?$

Precondition	*Add edges*
$\mathrm{Pred}_G(s) = \{r\}$	(1) for all $u \in \mathrm{Succ}_G(s) - \{r, s\}$, add (r, u)
	(2) for all $u \in \mathrm{Succ}_G(r) - \{r, s\}$, add (s, u)
	(3) if $r \in \mathrm{Succ}_G(s)$ and $r \notin \mathrm{Succ}^+_G(r)$, add (r, r)

Repair $r? \,.\, s$

Precondition	*Add edges*
$\mathrm{Succ}_G(r) = \{s\}$	(1) for all $u \in \mathrm{Pred}_G(s) - \{r, s\}$, add (u, r)
	(2) for all $u \in \mathrm{Pred}_G(r) - \{r, s\}$, add (u, s)
	(3) if $r \in \mathrm{Succ}_G(s)$ and $s \notin \mathrm{Succ}^+_G(s)$, add (s, s)

Repair $r? \,.\, s?$

Precondition	*Add edges*
$s \in \mathrm{Succ}_G(r)$	(1) for all $u \in \mathrm{Pred}_G(s) - \{r, s\}$, add (u, r)
	(2) for all $u \in \mathrm{Pred}_G(r) - \{r, s\}$, add (u, s)
	(3) for all $u \in \mathrm{Succ}_G(s) - \{r, s\}$, add (r, u)
	(4) for all $u \in \mathrm{Succ}_G(r) - \{r, s\}$, add (s, u)
	(5) if $r \in \mathrm{Succ}_G(s)$ and $r \notin \mathrm{Succ}^+_G(r)$, add (r, r)
	(6) if $r \in \mathrm{Succ}_G(s)$ and $s \notin \mathrm{Succ}^+_G(s)$, add (s, s)
	(7) for all $u \in \mathit{Pred}_G(r)$, $u' \in \mathrm{Succ}_G(s)$,
	add (u, u')

Figure 2.14: Repair rules

Algorithm 3 RWR

```
Input: a SOA $G$
Output: a SORE $r$ such that $\mathcal{L}(G) \subseteq \mathcal{L}(r)$ if $G$ is not equivalent to a SORE,
    and $\mathcal{L}(G) = \mathcal{L}(r)$ otherwise.
 1: if $sink$ is not reachable from $src$ in $G$ then
 2:   return $\emptyset$
 3: else if $E(G) = \{(src, sink)\}$ then
 4:   return $\varepsilon$
 5: else
 6:   while $G$ is not final do
 7:     if a rewrite rule from Figure 2.4 can be applied then
 8:       apply the rewrite rule on $G$
 9:     else
10:       apply a repair rule from Figure 2.14
11:   return the corresponding regular expression $r$
```

2.2.2 Exploring the search space: RWR$^2_\ell$

When rewriting gets stuck, RWR arbitrarily selects a repair rule (perhaps based on some ordering of the rules as in our implementation), and discards the others. It should be clear, however, that when different repair rules are applicable, one rule may have a smaller impact on the language of the automaton than another. For that reason we present in this section a different modification of REWRITE that, in contrast to RWR, tries the "best" ℓ repair rules when there are several candidates. Here, the "best" repair rules are those that add the least number of words to the language. Since an automaton defines an infinite language in general, it is of course impossible to take all added words into account. We therefore only consider the words up to a length n, where n is twice the number of alphabet symbols in the automaton. Formally, for a language L, let $|L^{\leq n}|$ denote the number of words in L of length at most n. Moreover, say that generalized SOA H is a *repair of* generalized SOA G if H is obtained by applying a repair rule on G. Then the repairs of the current automaton G are ordered according to increasing values of $|\mathcal{L}(H)^{\leq n}|$, and the best (i.e., first) ℓ among them are further investigated.

The resulting algorithm, called RWR$^2_\ell$ (an abbreviation of ReWrite with ℓ best Ranked Repairs) is shown in Algorithm 4. Like REWRITE, it first checks whether its input G is equivalent to $\emptyset$ or ε. Otherwise, RWR$^2_\ell$ uses RWR$^2_\ell$-AUX to recursively rewrite and repair G until a final automaton is reached. During this recursion, H_{opt} is the best final generalized SOA found so far. Initially, on Line 4 of RWR$^2_\ell$, H_{opt} is set to the final generalized SOA that accepts all words

Algorithm 4 $\textsc{rwr}^2_\ell$

```
Input: SOA $G$
Output: a SORE $r$ such that $\mathcal{L}(G) \subseteq \mathcal{L}(r)$ if $G$ is not equivalent to a SORE,
    and $\mathcal{L}(G) = \mathcal{L}(r)$ otherwise.
1: if $sink$ is not reachable from $src$ in $G$ then
2:     return $\emptyset$
3: else if $E(G) = \{(src, sink)\}$ then
4:     return $\varepsilon$
5: else
6:     initialize the final automaton $H_{\mathrm{opt}}$ to recognize $\Sigma(G)^*$
7:     return the SORE corresponding to the final automaton computed by
       $\textsc{rwr}^2_\ell\text{-}\textsc{aux}(G, H_{\mathrm{opt}})$
```

over alphabet symbols mentioned in G. $\textsc{rwr}^2_\ell$-$\textsc{aux}$ then rewrites G in Lines 1-2 until no more rewrite rule is applicable. If the resulting G is final then it is returned. Otherwise, $\textsc{rwr}^2_\ell$-$\textsc{aux}$ computes in Line 6 all possible repairs H of G and orders them according to increasing values of $|\,\mathcal{L}(H)^{\leq n}|$. The algorithm then recursively calls itself on the ℓ best ranked repairs in lines 8-10. The test in Line 9 is an optimization: if the current repair is already worse than the best final generalized SOA H_{opt} computed so far in terms of language size, then further rewriting and repairing cannot yield a final generalized SOA that is better than H_{opt}. Lines 11 updates H_{opt} when appropriate. Finally, H_{opt} is returned.

Given its definition, it is clear that $\textsc{rwr}^2_\ell$ results in regular expressions with a smaller language size for increasing values of ℓ, of course at the cost of increased computation time. In the experiments (Section 2.5.2) the trade-off between precision and computation time of $\textsc{rwr}$ and $\textsc{rwr}^2_\ell$, for increasing values of ℓ, is investigated in more detail.

2.2.3 Efficiently computing the language size

During its executing, $\textsc{rwr}^2$ repeatedly needs to compute the language size of the possible repairs. This computation can actually be done quite efficiently for SOAs, as we show next. Of course, in general $\textsc{rwr}^2$ needs to compute the language size also for generalized SOAs, not just ordinary SOAs. Our implementation first expands such generalized SOAs into an equivalent SOA using the Glushkov construction (similar to the ideas of the proof of Proposition 2.19), and then invokes the language size computation procedure explained next.

Let $|L^{=m}|$ denote the number of words in L of length exactly m. Let G

Algorithm 5 $\textsc{rwr}^2_\ell\textsc{-aux}$

```
Input: generalized SOAs $G$ and $H_{\text{opt}}$
Output: final generalized SOA $I$ such that $\mathcal{L}(G) \subseteq \mathcal{L}(I)$ if $G$ is not equivalent
    to a SORE, and $\mathcal{L}(G) = \mathcal{L}(I)$ otherwise.
 1: while a rewrite rule from Figure 2.4 can be applied to $G$ do
 2:   perform the rewrite rule on $G$
 3: if $G$ is final then
 4:   return $G$
 5: else
 6:   compute the set $\mathcal{R}$ of all possible repairs $H$ of $G$
 7:   sort $\mathcal{R}$ in increasing order by $|\mathcal{L}(H)^{\leq n}|$
 8:   for each of the $\min(\ell, |\mathcal{R}|)$ best repairs $H$ do
 9:     if $|\mathcal{L}(H)^{\leq n}| < |\mathcal{L}(H_{\text{opt}})^{\leq n}|$ then
10:       recursively compute $H' := \textsc{rwr}^2_\ell\textsc{-aux}(H, H_{\text{opt}})$
11:       set $H_{\text{opt}} := H'$ if $|\mathcal{L}(H')^{\leq n}| < |\mathcal{L}(H_{\text{opt}})^{\leq n}|$
12:   return $H_{\text{opt}}$
```

be a SOA; and assume that $V(G) - \{src, sink\} = \{a_1, \dots, a_n\}$. Then consider the $n \times n$ matrix $\mathcal{D}$ where for $i, j \in \{1, \dots, n\}$

$$\mathcal{D}[i,j] = \begin{cases} 1 & \text{if } (a_i, a_j) \in E\text{; and,} \\ 0 & \text{otherwise.} \end{cases}$$

In addition, define the $1 \times n$ and $n \times 1$ matrices $\mathcal{I}$ and $\mathcal{F}$, respectively, as follows: for $i, j \in \{1, \dots, n\}$

$$\mathcal{I}[1,j] = \begin{cases} 1 & \text{if } (src, j) \in E\text{; and,} \\ 0 & \text{otherwise;} \end{cases}$$

and

$$\mathcal{F}[i,1] = \begin{cases} 1 & \text{if } (i, sink) \in E\text{; and,} \\ 0 & \text{otherwise.} \end{cases}$$

The following lemma is straightforward to prove by induction on n using the fact that each walk from *src* to *sink* in G uniquely determines an accepted word. Let $\mathcal{D}^m$ denote the m-times multiplication of $\mathcal{D}$, with $\mathcal{D}^0$ the unit matrix.

Lemma 2.32. *Let $m > 0$ and let G be a SOA. Then $|\mathcal{L}(G)^{=m}| = \mathcal{I} \cdot \mathcal{D}^{m-1} \cdot \mathcal{F}$.*

Since for $m = 0$, we simply have $|\mathcal{L}(G)^{=m}| = 1$ if $(src, sink) \in E$, and $|\mathcal{L}(G)^{=m}| = 0$, otherwise and since $|\mathcal{L}(G)^{\leq n}| = \Sigma_{m=0}^{n} |\mathcal{L}(G)^{=m}|$, we can determine $|\mathcal{L}(G)^{\leq n}|$ by iteratively computing the matrices $\mathcal{D}^1$ to $\mathcal{D}^m$, and applying Lemma 2.32. This immediately gives the following corollary.

Corollary 2.33. *For each $n > 0$ and SOA G, $|\mathcal{L}(G)^{\leq n}|$ can be computed in time $\mathcal{O}(n|G|^3)$.*

As already noted in the Introduction, real-worlds samples need not be valid with respect to its known schema. Errors crop up due to all sorts of circumstances. This underscores the need for a robust inference algorithm that can handle some noise in the input sample.

Noise can come in several forms. To generate a noisy subsample, we modify the target expression either by replacing a symbol by a different one from the target's expression, or by replacing it by a symbol that is not in the alphabet of the target expression. We than use the modified target expression to generate a complete sample. We define the noise level as follows:

Definition 2.34. Given a target expression r, the noise level of a sample S is the ratio $|S - \mathcal{L}(r)|/|S|$.

Here we propose an approach to filter the sample S based on the probability of its words being generated by a probabilistic automaton as we also use in Section 3.2. This probabilistic automaton has one state for each alphabet symbol, and the transition probabilities are computed using the Baum-Welsh algorithm. Given the probabilistic automaton, it is straightforward to compute the probability for each $w \in S$, so that one can rank the sample's words. One expects words that contain noise, i.e., that would be rejected by the target regular expression, to have low probability if their number is not excessively large compared to the sample's size.

To filter the sample, hoping to exclude those words that contain noise, we compute the mean μ and standard deviation σ of the sample's probabilities. A string $w \in S$ with probability $P(w)$ is excluded if $P(w) < \mu - \alpha\sigma$. The factor α is a parameter of the algorithm. The filtered sample S' is now used to derive a regular expression. It is of course possible that in the generation of S' some words needed to derive the target expression were removed. Hence there is no guarantee that the derived regular expression will be an overapproximation of the target expression. Experimental results will be discussed in Section 2.5.4.

2.3 RWR^0

It should be noted that in the conference version of this article [BNST06] we proposed a different set of rewrite and repair rules for transforming SOAs into SOREs. While those rewrite rules were claimed in [BNST06] to possess the analogue of Proposition 2.13 (namely that they always produce a SORE equivalent to the input SOA, provided that such a SORE exists), this claim is false. Worse, there exist generalized SOAs for which rewrite rules of [BNST06] are not sound. To illustrate this claim, the rewrite rules of [BNST06] are given in Figure 2.15, where G^* refers to the *ε-closure* of G, defined as follows.

Definition 2.35. Let $G = (V, E)$ be a generalized SOA. The *ε-closure* G^* of G is the graph (V, E^*) where E^* contains

- all edges of E;
- all edges (r, r) with $r = s^+$ or $r = s^+?$;
- all edges (r, s) for which there is a path from r to s in G that passes only through intermediate nodes t with $\varepsilon \in \mathcal{L}(t)$.

Figure 2.16 shows a sequence of rewrite steps using these rules starting from the SOA recognizing $(a + b)^+?$ or, equivalently, $(a? \,.\, b?)^+$. Note that the second rewrite step, which introduces $b?$, causes the automaton to become disconnected: because $a? \in \mathrm{Pred}_{G^*}(b)$ and *sink* $\in \mathrm{Succ}_{G^*}(b) - \{b\}$ it deletes $(a?, \mathit{sink})$—the only edge linking *src* to *sink*. As such, the accepted language changes from $\mathcal{L}((a+b)^+?)$ to $\emptyset$. This clearly illustrates that the Optional $r?$ rule in Figure 2.15 is unsound. For that reason, we have moved in this article to the new rewrite rules in Figure 2.4.

It is peculiar, however, that we have extensively used the rewrite of Figure 2.15 together with the repair rules in Figure 2.18 in a prototype implementation but have *never* encountered a situation where:

- we obtained a SORE r that failed to accept at least all words in the input SOA G; or
- we obtained a SORE r that accepted a strict superset of $\mathcal{L}(G)$ when G was equivalent to a SORE.

We suspect that this behavior is due to the strict order in which we apply the rewrite rules in our implementation: first Concatenation, then Disjunction, then Self-Loop, and finally Optional. To illustrate, Figure 2.17 shows a successful rewriting of the SOA accepting $(a + b)^+?$ under this order.

The algorithm RWR^0 is shown in Algorithm 6 and is based on the rewrite rules in Figure 2.15 and the repair rules in Figure 2.18. The experiments in Section 2.5 indicate that RWR^0 has no benefits over RWR and RWR^2. Moreover, as we do not have a formal proof that rewriting always behaves in the above sound and complete manner under this order, it does not make much sense to consider RWR^0 for the class of SOREs. In strong contrast, on the class of k-occurrence regular expressions ($k > 1$), RWR^0 can make a difference over RWR and RWR^2, as is shown in Chapter 3. So even without formal guarantees, RWR^0 still proofs to have its merits.

Disjunction

Precondition	*Action*
$\{r_1, \ldots, r_n\}$ with $n \geq 2$ is a subset $V(G) - \{src, sink\}$ such that every two nodes r_i, r_j have the same predecessor and successor set in G^*. Note that this implies that either 1. there are no edges in G between $r_1, \ldots, r_n$ at all; or 2. for each i, j there is an edge r_i, r_j in G^*.	• Remove $r_1, \ldots, r_n$; • add a new node $r = r_1 + \cdots + r_n$; • redirect all incoming and outgoing edges of $r_1, \ldots, r_n$ to r. • In the case of situation (2) add the edge (r, r).

Concatenation

Precondition	*Action*
$\{r_1, \ldots, r_n\}$ with $n \geq 2$ is a maximal subset of $V(G) - \{src, sink\}$ such that • there is an edge in G from r_i to r_{i+1}; • every node besides r_1 has only one incoming edge in G; and • every node besides r_n has only one outgoing edge in G.	• Remove $r_1, \ldots, r_n$; • add a new node $r = r_1 \cdots r_n$; • 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.)

Self-loop r^+

Precondition	*Action*
$(r, r) \in E(G)$	Replace G by $G[r/r^+] \ominus (r^+, r^+)$

Optional $r?$

Precondition	*Action*
$\text{Succ}_{G^*}(r) \subseteq \text{Succ}_{G^*}(s)$ for every $s \in \text{Pred}_{G^*}(r)$	• Relabel r by $r?$; • remove all edges (s, t) with $s \in \text{Pred}_{G^*}(r)$ and $t \in \text{Succ}_{G^*}(r) - \{r\}$

Figure 2.15: Set of rewrite rules introduced in the conference version of this article[BNST06].

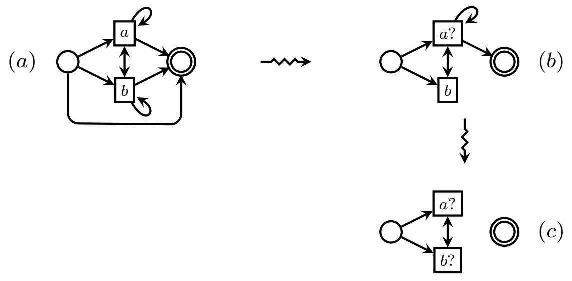


Figure 2.16: A problematic sequence of rewrite steps using the rules in Figure 2.15. The input SOA accepts the same language as $(a+b)^+?$, or, equivalently $(a?\,.\,b?)^+$. Note that the automaton resulting from by the second rewrite step is disconnected and hence accepts the empty language. Rewriting is therefore not sound.

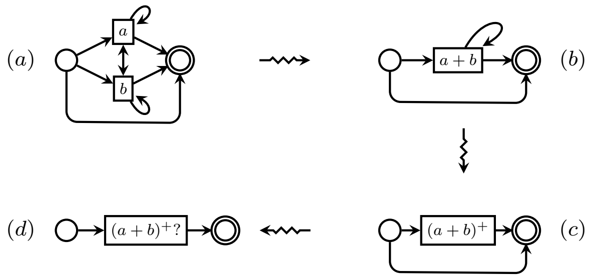


Figure 2.17: A successful sequence of rewrite steps using the rules in Figure 2.15. The input SOA accepts the same language as $(a+b)^+?$, or, equivalently $(a?\,.\,b?)^+$.

Enable-Disjunction

Precondition	*Add edges*
(1) $r, s \in V(G) - \{src, sink\}$ and either (2a) $\text{Pred}_{G*}(r) \cap \text{Pred}_{G*}(s) \neq \emptyset$; or (2b) $\text{Succ}_{G*}(r) \cap \text{Succ}_{G*}(s) \neq \emptyset$	(1) for all $u \in (\text{Pred}_{G^*}(r) \cup \text{Pred}_{G^*}(s)) - \{r, s\}$, add (u, r), (u, s) (2) for all $u \in (\text{Succ}_{G^*}(r) \cup \text{Succ}_{G^*}(s)) - \{r, s\}$, add (r, u), (s, u) (3) if $r \in \text{Succ}_G(s)$ or $s \in \text{Succ}_G(r)$ then (a) add (r, s), (s, r) (b) if $r \notin \text{Succ}_{G^*}(r)$, add (r, r) (c) if $s \notin \text{Succ}_{G^*}(s)$, add (s, s)

Enable-Optional-1

Precondition	*Add edges*
(1) $r \in V(G) - \{src, sink\}$ (2) $\varepsilon \notin \mathcal{L}(r)$ (3) $\text{Succ}_{G^*}(r) \cap \text{Succ}_{G^*}(s) \neq \emptyset$ for every $s \in \text{Pred}_{G^*}(r)$	for all $s \in \text{Pred}_{G^*}(r)$ and all $u \in \text{Succ}_{G^*}(r)$, add (s, u)

Enable-Optional-2

Precondition	*Add edges*
(1) $r \in V(G) - \{src, sink\}$ (2) $\varepsilon \notin \mathcal{L}(r)$ (3) $\text{Pred}_{G^*}(r) \neq \emptyset$	for all $s \in \text{Pred}_{G^*}(r)$ and all $u \in \text{Succ}_{G^*}(r)$, add (s, u)

Figure 2.18: Repair rules accompanying the rewrite rules in Figure 2.15. These rules are a correction of the rules presented in [BNST06]. Repairs are tried in the order shown. In particular, Enable-Optional-2 is only applied if none of the other rules are applicable.

Algorithm 6 RWR^0

Input: a SOA G

Output: a SORE r

```
 1: if $sink$ is not reachable from $src$ in $G$ then
 2:   return $\emptyset$
 3: else if $E(G) = \{(src, sink)\}$ then
 4:   return $\varepsilon$
 5: else
 6:   initialize $done$ to $false$
 7:   while not $done$ do
 8:     if there a rewrite rule in Figure 2.15 is applicable then
 9:       rewrite $G$, giving precedence to CONCATENATION, then DISJUNCTION, then SELF-LOOP, then OPTIONAL
10:     else if a repair rule in Figure 2.18 is applicable then
11:       repair $G$, giving precedence to ENABLE-DISJUNCTION, then ENABLE-OPTIONAL-1, then ENABLE-OPTIONAL-2
12:     else
13:       set $done$ to $true$
14:   if $G$ is final then
15:     return the corresponding regular expression $r$
16:   else
17:     return $\emptyset$
```

2.4 Inferring CHAREs: CRX

In this section, we present the algorithm CRX for the inference of chain regular expressions (CHAREs).

Definition 2.36 (CHAREs)**.** The class of *chain regular expressions* consists of those SOREs of the form $f_1 \cdots f_n$ where every f_i is a *chain factor*—an expression of the form $(a_1 + \cdots + a_k)$, $(a_1 + \cdots + a_k)?$, $(a_1 + \cdots + a_k)^+$, or, $(a_1 + \cdots + a_k)^+?$ with $k \geq 1$ and every a_i is an alphabet symbol.

For instance, the expression $a(b+c)^*d^+(e+f)?$ is a CHARE, while $(ab+c)^*$ and $(a^* + b?)^*$ are not.

Since each CHARE is a concatenation of alphabet-disjoint chain factors, every occurrence of an alphabet symbol in a word must be generated by the same chain factor in the target CHARE. The positional relationships between occurrences of alphabet symbols in a given sample then allow us to deduce which chain factors are present in the target CHARE, and how they are ordered.

Example 2.37. Consider the sample $S = \{u, v, w\}$ with $u = abd$, $v = bcdee$, and $w = cade$. Clearly a occurs before b in u, b occurs before c in v, and c occurs before a in w. In the target CHARE, therefore, a, b, and c must belong to the same chain factor which can only be $(a+b+c)^+$ or $(a+b+c)^+?$. Since one of $\{a, b, c\}$ is present in every word of S, we choose $(a+b+c)^+$. Similarly, d and e form chain factors by themselves. Whereas d occurs once in every word in S, e can occur zero, one or more times. Therefore, d is represented by the chain factor d, while e is represented by the chain factor $e^+?$. Since a, b, c always occur before d, which in turn always occurs before the e's, the derived CHARE is then $(a + b + c)^+de^+?$.

So, in brief, CRX computes chain factors, orders them and uses that order to generate a CHARE. Of course, the order of the chain factors is not necessarily linear. In that case, a linear order can be constructed by making the factors optional. Some care has to be taken, however, to generate factors that are disjunctions without repetitions.

Definition 2.38. Let S be a sample. We denote by $\rightarrow_S$ the partial pre-order on Σ such that $a \rightarrow_S b$ if, and only if, a immediately precedes b in some $w \in S$. (I.e., ab is a 2-gram of S.) We say that *a occurs before b* in S if $a \rightarrow_S^* b$, where $\rightarrow_S^*$ is the reflexive and transitive closure of $\rightarrow_S$.

For instance, Figure 2.19 illustrates $\rightarrow_S$ when $S = \{abccde, cccad, bfegg, bfehi\}$.

$$\begin{tikzcd}
 & b \arrow[rr] \arrow[dd] & & f \arrow[dd] & h \arrow[r] & i \\
a \arrow[ur] \arrow[drr, bend right] & & & & & \\
 & c \arrow[ul] \arrow[r] & d \arrow[r] & e \arrow[r] \arrow[uur] & g &
\end{tikzcd}$$

Figure 2.19: The partial pre-order $\rightarrow_S$ for $S = \{abccde, cccad, bfegg, bfehi\}$.

$$\begin{tikzcd}
 & \boxed{[d]} \arrow[dr] & & \boxed{[g]} & \\
\boxed{[a,b,c]} \arrow[ur] \arrow[dr] & & \boxed{[e]} \arrow[ur] \arrow[dr] & & \\
 & \boxed{[f]} \arrow[ur] & & \boxed{[h]} \arrow[r] & \boxed{[i]}
\end{tikzcd}$$

Figure 2.20: The Hasse diagram H_S of the sample $S = \{abccde, cccad, bfegg, bfehi\}$. The corresponding partial pre-order from which H_S is derived is shown in Figure 2.19.

Definition 2.39. Define $a \approx_S b$ if a occurs before b in S and b occurs before a. That is, $a \approx_S b$ if $a \rightarrow_S^* b$ and $b \rightarrow_S^* a$.

Clearly, $\approx_S$ is an equivalence relation. Let Γ_S denote the set of equivalence classes of $\approx_S$. In what follows, we denote such equivalence classes by e.g., $[a_1, \ldots, a_n]$. As usual, an equivalence class of cardinality 1 is called a *singleton.*

Definition 2.40. The *Hasse diagram* of S, denoted H_S, is the graph over Γ_S in which there is an edge from equivalence class $[a_1, \ldots, a_n]$ to class $[b_1, \ldots, b_m]$ if (1) $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_m]$ are distinct and (2) there exists $1 \leq i \leq n$ and $1 \leq j \leq m$ such that $a_i \rightarrow_S b_j$.

For instance, the Hasse diagram of the sample $S = \{abccde, cccad, bfegg, bfehi\}$ is shown in Figure 2.20. The operation of CRX is then shown in Algorithm 7 and illustrated in the following example.

Example 2.41. Consider again the sample $S = \{abccde, cccad, bfegg, bfehi\}$ and its corresponding Hasse diagram in Figure 2.20. Since $\text{Pred}_{H_S}([d]) = \text{Pred}_{H_S}([f])$ and $\text{Succ}_{H_S}([d]) = \text{Succ}_{H_S}([f])$, line 3 applies to $[d]$ and $[f]$. Although $\text{Pred}_{H_S}([g]) = \text{Pred}_{H_S}([h])$, step 7 cannot be applied as $\text{Succ}_{H_S}([g]) \neq \text{Succ}_{H_S}([h])$. Similarly $[g]$ and $[i]$ share successors, i.e. $\emptyset$, but have different predecessors. Hence, after the while loop in line 2 we obtain:

$$\begin{tikzcd}
 & & & \boxed{[g]} & \\
\boxed{[a,b,c]} \arrow[r] & \boxed{[d,f]} \arrow[r] & \boxed{[e]} \arrow[ur] \arrow[dr] & & \\
 & & & \boxed{[h]} \arrow[r] & \boxed{[i]}
\end{tikzcd}$$

A possible topological sort is $[a, b, c], [d, f], [e], [g], [h], [i]$. Since at least one of a, b and c occurs once or more in every string of W, $r([a, b, c]) = (a \mid b \mid c)^+$ is

Algorithm 7 CRX

Input: a sample S
Output: a CHARE r such that $S \subseteq L(r)$

```
 1: Compute the set $\Gamma_S$ of equivalence classes of $\approx_S$
 2: while a maximal set of singleton nodes $\gamma_1, \ldots, \gamma_\ell$ such that $\mathrm{Pred}_{H_S}(\gamma_1) = \cdots = \mathrm{Pred}_{H_S}(\gamma_\ell)$ and $\mathrm{Succ}_{H_S}(\gamma_1) = \cdots = \mathrm{Succ}_{H_S}(\gamma_\ell)$ exists do
 3:   Replace $\gamma_1, \ldots, \gamma_\ell$ by $\gamma := \cup_{j=1}^{\ell} \gamma_j$, and redirect all incoming and outgoing edges of the $\gamma_i$ to $\gamma$ in $H_S$
 4: Compute a topological sort $\gamma_1, \ldots, \gamma_k$ of the nodes
 5: for all $i \in \{1, \ldots, k\}$ $(\gamma_i = [a_1, \ldots, a_n])$ do
 6:   if every $w \in S$ contains exactly one occurrence of a symbol in $\{a_1, \ldots, a_n\}$ then
 7:     $r(\gamma_i) := (a_1 + \cdots + a_n)$
 8:   else if every $w \in S$ contains at most one occurrence of a symbol in $\{a_1, \ldots, a_n\}$ then
 9:     $r(\gamma_i) := (a_1 + \cdots + a_n)?$
10:   else if every $w \in S$ contains at least one of $a_1, \ldots, a_n$ and there is a word that contains at least two occurrences of symbols then
11:     $r(\gamma_i) := (a_1 + \cdots + a_n)^+$
12:   else
13:     $r(\gamma_i) := (a_1 + \cdots + a_n)^*$
14:   return $r(\gamma_1) \,.\, r(\gamma_2) \,.\, \cdots \,.\, r(\gamma_k)$
```

the first factor; the second factor is $(d \mid f)$ since either d or f occurs exactly once; the factor derived from $[e]$ is $e?$ since W contains a string without e and similarly for those from $[h]$ and $[i]$. Finally, g occurs multiple times in a single string. Hence the simple regular expression derived by the algorithm is $(a \mid b \mid c)^+ \cdot (d \mid f) \cdot e? \cdot g^* \cdot h? \cdot i?$ which completes step 7.

Note that the order of the chain factors in the CHARE depends on the topological sort.

Theorem 2.42. *Given a sample S,* CRX *computes a CHARE r such that $S \subseteq L(S)$.*

Proof. The theorem follows almost immediately from the construction. Clearly, CRX always outputs a CHARE. Moreover, observe that after Step 7 the computed topological sort is consistent with the order of the symbols in the words in S. More precisely, there can not exist symbols a and b, such that $a \in \gamma_i$, $b \in \gamma_j$, $i < j$, and $b \to_S^* a$. Subsequently, for each γ_i a chain factor is chosen in such a manner that it is consistent with all words $w \in S$. As these factors are ordered consistently with the order of the symbols in S, this implies that $S \subseteq L(r)$. ☐

Furthermore, on the class of CHAREs, CRX is complete:

Theorem 2.43. *For each CHARE r there is a sample S such that $\mathcal{L}(r) = \mathcal{L}(\text{CRX}(S))$.*

Proof. Denote by $\text{Sym}(r)$ the set of alphabet symbols occurring in r. We also abuse notation and, for a sample S, write $\text{Sym}(S)$ to denote the set of alphabet symbols occurring in S. Let $r = f_1 \cdots f_k$ be a CHARE. We construct the sample S such that the $\text{CRX}(S)$ is syntactically equal to r, up to commutativity of $+$. The theorem then follows.

Thereto, for every $1 \leq i \leq k$, let w_i be a word in $\mathcal{L}(f_i)$. We construct S by subsequently adding words to it. First, for all $1 \leq i \leq k-1$, $a \in \text{Sym}(f_i)$, $b \in \text{Sym}(f_{i+1})$, we add $w_1 \cdots w_{i-1} a b w_{i+2} \cdots w_k$ to S. Further, for all $1 \leq i \leq k$, we add words to S, depending on the form of f_i. Specifically, if f_i is of the form

- $(a_1 + \cdots + a_n)$, we add $w_1 \cdots w_{i-1} a_1 w_{i+1} \cdots w_k$;
- $(a_1 + \cdots + a_n)?$, we add $w_1 \cdots w_{i-1} w_{i+1} \cdots w_k$, $w_1 \cdots w_{i-1} a_1 w_{i+1} \cdots w_k$;
- $(a_1 + \cdots + a_n)^+$, we add $w_1 \cdots w_{i-1} a_1 a_1 w_{i+1} \cdots w_k$;
- $(a_1 + \cdots + a_n)^+?$, we add $w_1 \cdots w_{i-1} w_{i+1} \cdots w_k$, $w_1 \cdots w_{i-1} a_1 a_1 w_{i+1} \cdots w_k$.

We now argue that given S, CRX indeed derives an expression syntactically equal to r. First observe that already before step 7, CRX computes k nodes γ_1 to γ_k, which are linearly ordered, such that for each $1 \leq i \leq k$, γ_i contains exactly the alphabet symbols contained in f_i. Then, due to the number of occurrences of each symbol of the different chain factors, the algorithm will associate to each γ_i exactly the factor f_i, and hence CRX(S) is syntactically equivalent to r, up to commutativity of $+$. □

From Theorems 2.42 and 2.43 it readily follows:

Corollary 2.44. CRX *learns the class of CHAREs from positive data.*

The experiments in Section 2.5.3 show that the number of words in S needed in practice is very small. Actually, the prime feature that makes CRX much more robust than RWR for very small data sets is its strong generalization ability. Indeed, consider an expression of the form $(a_1 \mid \cdots \mid a_n)^*$. While REWRITE requires all n^2 2-grams of the form a_ia_j for $i, j \in \{1, \ldots, n\}$ to be present, RWR requires around $(n^2 - n)$ 2-grams. For CRX, however, the set $\{\varepsilon, a_1a_2, a_2a_3, \ldots, a_{n-1}a_n, a_na_1\}$ of size $\mathcal{O}(n)$ will suffice. This point is illustrated in practice by `example3` and `example4` in Table 2.3 where n has a value of 41 and 56 respectively. Experiments illustrate that only $400 \ll 1682$ and $500 \ll 3136$ 2-grams are needed by CRX to learn `example3` and `example4`, respectively.

The following theorem shows that CRX is optimal within the class of CHAREs when the partial order Γ_S is in fact a linear order:

Theorem 2.45. *For every sample S, if Γ_S is a linear order then for every CHARE r such that $S \subseteq \mathcal{L}(r)$ and $\mathcal{L}(r) \subseteq \mathcal{L}(\text{CRX}(S))$, we have $r = \text{CRX}(S)$, i.e. r is syntactically equal to CRX(S) up to commutativity of $\mid$.*

Proof. Assume that $\text{CRX}(S) = f_1 \cdots f_k$ and $r = g_1 \cdots g_l$. Clearly, $\text{Sym}(\text{CRX}(S)) = \text{Sym}(r) = \text{Sym}(S)$. We first argue that $k = l$. Thereto, assume for the purpose of contradiction that $k < l$. Then, there is a chain factor f in CRX(S) with $a, b \in \text{Sym}(f)$ and two chain factors g and g' in r with $a \in \text{Sym}(g)$ and $b \in \text{Sym}(g')$. We distinguish two cases:

1. If f is of the form $(a_1 + \cdots + a_n)$ or $(a_1 + \cdots + a_n)?$, then $\mathcal{L}(r) \not\subseteq \mathcal{L}(\text{CRX}(S))$.

2. If f is of the form $(a_1 + \cdots + a_n)^+?$ or $(a_1 + \cdots + a_n)^+$, by construction and since Γ_S is linearly ordered, there are words $u_1, u_2 \in S$ such that $a \rightarrow^*_{u_1} b$ and $b \rightarrow^*_{u_2} a$. However, since a and b are in different chain factors of r, either $u_1 \notin \mathcal{L}(r)$ or $u_2 \notin \mathcal{L}(r)$, and hence $S \not\subseteq \mathcal{L}(r)$.

Conversely, assume $k > l$. Then, there are chain factors f, f' in CRX(S) with $a \in \mathrm{Sym}(f)$ and $b \in \mathrm{Sym}(f')$, and a chain factor g in r with $a, b \in \mathrm{Sym}(g)$. We again distinguish two cases:

1. If g is of the form $(a_1 + \cdots + a_n)^{+}?$ or $(a_1 + \cdots + a_n)^{+}$, then $\mathcal{L}(r) \not\subseteq \mathcal{L}(\textsc{crx}(S))$.

2. If g is of the form $(a_1 + \cdots + a_n)$ or $(a_1 + \cdots + a_n)?$, by construction and since Γ_S is linearly ordered, there are words $u_1, \ldots, u_m \in S$, and symbols $c_1, \ldots, c_{m-1}$ such that $a \rightarrow^*_{u_1} c_1$, $c_m \rightarrow^*_{u_m} b$, and $c_i \rightarrow_{u_{i+1}} c_{i+1}$, for all $1 \leq i \leq m-1$. However, due to the form of g, for at least one of these u_i, $u_i \notin \mathcal{L}(r)$ must hold and hence $S \not\subseteq \mathcal{L}(r)$.

Using the same kind of argument it can be shown that $\mathrm{Sym}(f_i) = \mathrm{Sym}(g_i)$, for all $1 \leq i \leq k$. Further, since $\mathcal{L}(r) \subseteq \mathcal{L}(\textsc{crx}(S))$, for every $1 \leq i \leq k$, we have $\mathcal{L}(g_i) \subseteq \mathcal{L}(f_i)$. Since the different chain factors can only take a restricted numbers of forms, it now suffices to show that $\mathcal{L}(g_i) = \mathcal{L}(f_i)$, for all i, to show that they are also syntactically equivalent. Hence, towards a contradiction, assume $\mathcal{L}(g_i) \subsetneq \mathcal{L}(f_i)$ for some $1 \leq i \leq k$. This can only be the case if (1) $g_i = (a_1 + \cdots + a_n)$ and $f_i = (a_1 + \cdots + a_n)$; (2) $g_i = (a_1 + \cdots + a_n)^*$ and $f_i = (a_1 + \cdots + a_n)^+$; or (3) $g_i = (a_1 + \cdots a_n)?$ and f_i is one of the three other forms. However, in each of these cases, given the construction of the algorithm, one can find a word $w \in S$ such that $w \notin \mathcal{L}(r)$. Hence, for all i, $\mathcal{L}(f_i) = \mathcal{L}(g_i)$, and thus $r = \textsc{crx}(S)$. $\square$

Note that this property does not hold when Γ_S is not linear. For instance, on $S = \{abc, ade, abe\}$ CRX yields $a \cdot b? \cdot d? \cdot c? \cdot e?$ whereas the CHARE $a \cdot (b \mid d) \cdot (c \mid e)$ is a better approximation of the target language.

CRX can be efficiently executed on very large datasets by only maintaining H_S and the multiplicities of occurrences of Σ-symbols in words in S (needed for lines 6–13). From this representation, lines 2–5 can be executed. Hence, it is not necessary that the entire sample resides in main memory. The complexity of the algorithm is $\mathcal{O}(m + n^3)$, where m is the size of the sample and n the number of alphabet symbols.

2.5 Experimental evaluation

In this section we validate our approach by means of experimental analysis. Specifically, we assess the quality of the expressions returned by our algorithms on real-world corpora and DTDs, and compare it with the quality of expressions returned by XTRACT [GGR+03] and Trang [Cla03]. Next, we compare the quality of RWR^0, RWR and RWR$^2_\ell$. Subsequently, we investigate the performance of the algorithms on incomplete and noisy data. Finally, we discuss their running time performance. We abuse notation and simply write RWR for the application of 2T-INF followed by RWR, similarly for RWR^0 and RWR$^2_\ell$.

2.5.1 Real-world examples

The number of publicly available XML corpora is rather limited. We employed the XML Data repository maintained by Miklau [Mik02] as a testbed. Unfortunately, most of the corpora listed there are either very small; lack a DTD; or contain a DTD with only trivial regular expressions. Nevertheless, two of the listed corpora are interesting. Specifically, we compared XTRACT, RWR, and CRX on the Protein Sequence Database and the Mondial corpus [Mik02], a database of information on various countries. Since no real-world data could be obtained for SOREs that are not CHAREs, we generated our own XML data for a number of real-world DTDs considered in [BNV04] containing a number of sophisticated regular expressions outside the class of CHAREs.

Real-world data In this section, we only discuss RWR as RWR^0 and RWR$^2_\ell$ give precisely the same results. Table 2.2 lists all non-trivial element definitions[2] in the above mentioned DTDs together with the results derived by the inference algorithms RWR, CRX and XTRACT. It is interesting to note that only the regular expression for `authors` is not a CHARE. Moreover, no elements are repeated in any of the definitions. This should not come as a surprise given the observations discussed in the Introduction on the content models occurring in practice. The regular expression derived by the XTRACT algorithm is shown whenever it fitted the table, otherwise the number of tokens it consists of is listed. For better readability the actual output of XTRACT has been simplified by replacing expressions such as $(a_i + \varepsilon)$ by $a_i?$.

It can be verified that all regular expressions in Table 2.2 are learned quite satisfactory by RWR and CRX w.r.t. the examples extracted from the XML corpus. The numbers in the first column refer to the size of the sample. RWR and CRX always produce the same result except for `authors` where CRX cannot derive the target expression as it is not a CHARE. We note that no sample was representative of its target expression. As such, RWR always had to apply repair rules. The expressions in the table indicate that the result of these repairs are satisfactory. For a few expressions, e.g., `ProteinE(ntry)`, `refinfo` and `genetics`, the expressions produced by CRX and RWR are more strict than the corresponding one in the DTD. This is due to the data present

[2]It should be noted that the examples from the Mondial corpus are not valid according to their DTD, so for the `city` element only valid elements were used as training examples.

in the sample. For instance, for `genetics`, no a_{11} element occurs in the sample so it obviously cannot be part of the derived expression. The element `refinfo` illustrates that a_3 and a_4 are mutually exclusive in the sample and that a_8 is never followed by a_9. Inspecting the original DTD illustrates the underlying semantics:

```
authors, citation, volume?, month?, year,
   pages?, (title | description)?, xrefs?
```

Indeed, `volume` is used in the context of a journal, while `month` is used for a conference publication. Apart from the `authors` element XTRACT either produces a suboptimal expression or no expression at all. For instance, XTRACT crashes on the `ProteinE(ntry)` sample due to excessive memory consumption (more than 1 GB of RAM). Reducing the size of the sample to approximately 800 unique words yields a complex expression of 185 tokens.

Real-world regular expressions Table 2.3 lists the results of the algorithms on a number of more sophisticated regular expressions extracted from real-world DTDs discussed in [BNV04]. Since no real-world data was available for those DTDs, we have randomly generated samples using ToXgene [BM06], taking care that all relevant examples where present to ensure the target expression could be learned. Again, we list the sample size in the first column. As some of these numbers might seem artificially large, we note that, for instance, the SOA corresponding to `example3` already contains 1897 edges. Hence, a random data set of 5741 words is not unreasonably large. Note that only the first three expressions in Table 2.3 are SOREs, none of them are CHAREs. The table shows clearly that CRX yields fairly good and concise super-approximations to the original expressions. In some cases, the results produced by RWR are more precise. For XTRACT, the size of the sample had to be limited to 300–500 in order to avoid a crash. As can be seen from the table, XTRACT performed excellently on the first example, but failed to generate an expression that fitted the table in all other cases on all the sample sets we tried.

Trang We ran Trang [Cla03] on the XML data discussed in this section. In all but one case, Trang produced exactly the same output as CRX, with a notable exception: for `example1` Trang's output depends on the order in which the examples are presented, yielding either $a_1{}^*a_2?a_3{}^*$ or $a_1{}^+ \mid (a_2?a_3{}^+)$. The former is the same output as CRX, the latter is the intended RE that cannot be derived by CRX as it is outside the class of CHAREs. This inconsistency in Trang's output casts some doubt on its correctness and underscores the need

Element	**Original DTD**
Sample	**Result of CRX/RWR**
size	**Result of XTRACT**
ProteinE.	$a_1a_2a_3a_4{}^*a_5{}^*a_6{}^*a_7{}^*a_8{}^*a_9?a_{10}?a_{11}{}^*a_{12}a_{13}$
2458	$a_1a_2a_3a_4{}^+a_5{}^*a_6{}^*a_7{}^*a_8{}^*a_9?a_{10}?a_{11}{}^*a_{12}a_{13}$
843	an expression of 185 tokens
organism	$a_1a_2?a_3a_4?a_5{}^*$
9	$a_1a_2?a_3a_4?a_5{}^*$
9	$a_1((a_2a_3a_4?+a_3a_4)a_5?+a_3a_5{}^*)$
reference	$a_1a_2{}^*a_3{}^*a_4{}^*$
45	$a_1a_2{}^*a_3{}^*a_4{}^*$
45	$a_1(a_2{}^*(a_4{}^*+a_3{}^*)+a_2a_3{}^*a_4a_4+a_3{}^*a_4{}^*)$
refinfo	$a_1a_2a_3?a_4?a_5a_6?(a_7+a_8)?a_9?$
10	$a_1a_2(a_3+a_4)?a_5a_6?a_7?a_9?a_8?$
10	$a_1a_2((a_3a_5a_6a_7?+a_4a_5)a_9?+a_5(a_7+a_8)?+a_4a_5a_8)$
authors	$a_1{}^++(a_2a_3?)$
54	$a_1{}^*a_2?a_3?$ / $a_1{}^++(a_2a_3)$
54	$a_1{}^*+a_2a_3$
accinfo	$a_1a_2{}^*a_3{}^*a_4?a_5?a_6?a_7{}^*$
124	$a_1a_2{}^*a_3{}^+a_4?a_5?a_6?a_7{}^*$
124	an expression of 97 tokens
genetics	$a_1{}^*a_2?a_3?a_4?a_5?a_6?a_7?a_8?a_9?a_{10}?a_{11}{}^*a_{12}{}^*$
219	$a_1{}^*a_2?a_3?a_4?a_5?a_6?a_7?a_8?a_9?a_{10}?a_{12}{}^*$
219	an expression of 329 tokens
function	$a_1?a_2{}^*a_3{}^*$
26	$a_1?a_2{}^*a_3{}^*$
26	$(a_1(a_2?a_2?a_3{}^*+a_2{}^*(a_3a_3)^*+a_2a_2a_2a_3)+a_2(a_2a_3{}^*+a_3{}^*))$
city	$a_1a_2{}^*a_3{}^*$
9	$a_1a_2{}^*a_3{}^*$
9	$a_1(a_2{}^*a_3a_3?+a_2(a_3{}^*+a_2))?$

Table 2.2: Results of RWR, CRX and XTRACT on DTDs and sample data from the Protein Description Database and the Mondial corpora. The left column gives element names, sample size for CRX/RWR and sample size for XTRACT, respectively. The right column lists original DTD, inferred DTD by CRX/RWR and the result of XTRACT, in that order.

Element **Sample size**	**Original DTD** **Result of CRX** **Result of RWR** **Result of XTRACT**
`example1`	$a_1{}^+ \mid (a_2?a_3{}^+)$
48	$a_1{}^* a_2?a_3{}^*$
48	$a_1{}^+ \mid (a_2?a_3{}^+)$
48	$a_1{}^* \mid (a_2?a_3{}^*)$
`example2`	$(a_1a_2?a_3?)?a_4?(a_5 \mid \cdots \mid a_{18})^*$
2210	$a_1?a_2?a_3?a_4?(a_5 \mid \cdots \mid a_{18})^*$
2210	$(a_1a_2?a_3?)?a_4?(a_5 \mid \cdots \mid a_{18})^*$
300	an expression of 252 tokens
`example3`	$a_1?(a_2a_3?)?(a_4 \mid \cdots \mid a_{44})^* a_{45}{}^+$
5741	$a_1?a_2?a_3?(a_4 \mid \cdots \mid a_{44})^* a_{45}{}^+$
5741	$a_1?(a_2a_3?)?(a_4 \mid \cdots \mid a_{44})^* a_{45}{}^+$
400	an expression of 142 tokens
`example4`	$a_1?a_2a_3?a_4?(a_5{}^+ \mid ((a_6 \mid \cdots \mid a_{61})^+ a_5{}^*))$
10000	$a_1?a_2a_3?a_4?(a_6 \mid \cdots \mid a_{61})^* a_5{}^*$
10000	$a_1?a_2a_3?a_4?(a_6 \mid \cdots \mid a_{61})^* a_5{}^*$
500	an expression of 185 tokens
`example5`	$a_1(a_2 \mid a_3)^*(a_4(a_2 \mid a_3 \mid a_5)^*)^*$
1281	$a_1(a_2 \mid a_3 \mid a_4 \mid a_5)^*$
1281	$a_1((a_2 \mid a_3 \mid a_4)^+ a_5{}^*)^*$
500	an expression of 85 tokens

Table 2.3: Results of RWR, CRX and XTRACT on non-simple real-world DTDs and generated data. The left column gives element names, sample size for CRX, RWR and XTRACT, respectively. The right column lists original DTD, inferred DTD by CRX, by RWR and the result of XTRACT, in that order.

for a formal model as the cornerstone of an implementation. Indeed, there is no article or manual available describing the machinery underlying Trang. A look at the Java-code indicates that Trang is related to but different from CRX: it uses 2T-INF to construct an automaton, eliminates cycles by merging all nodes in the same strongly connected component, and then transforms the obtained DAG into a regular expression. However, no target class of REs for which Trang is complete, as is the case for CRX, is specified. As Trang is similar to CRX, it is outperformed by RWR and RWR^2_ℓ.

2.5.2 RWR versus RWR^2_ℓ

We tested the results and performance of RWR versus RWR^2_ℓ for various values of the rank cut-off parameter ℓ. The SOAs used in this test were randomly generated with 5 and 10 alphabet symbols. The results are summarized in Table 2.4. We computed the average language size of the SOAs, which is the target size. It should be noted that since no SORE corresponds to these SOAs, the target size can never be attained since the regular expression resulting from RWR or RWR^2_ℓ will necessarily be a generalization of the SOA's language. It is immediately clear from Table 2.4 that results of RWR^2_ℓ are on average better than those for RWR, and that they improve with increasing values of ℓ. For expressions of alphabet size 5, we were able to consider all possible repairs, resulting in the entry for $\ell = \infty$ in Table 2.4. This represents the smallest language that includes the SOA's language and that can be expressed by a SORE.

Of course, the results in Table 2.4 are averaged over 1000 randomly chosen SOAs. A more detailed analysis reveals that for a considerable number of SOAs, RWR actually outperforms RWR^2_ℓ for $\ell = 1$. Table 2.5 shows the number of times RWR outperforms RWR^2_ℓ for various values of ℓ. The probability that RWR outperforms RWR^2_ℓ drops rapidly for increasing values of ℓ, especially for larger alphabet sizes. The last line in Table 2.5 lists the probability that RWR derives the optimal result, i.e., that the smallest language representable by a SORE is obtained for expressions of alphabet size 5.

Although the RWR^2_ℓ algorithm clearly outperforms RWR in terms of the language size of the derived expression, there is a compelling argument in the latter's favor. In terms of running time, RWR outperforms RWR^2_ℓ with a few orders of magnitude as is discussed in Section 2.5.5.

2.5.3 Incomplete data

Unfortunately, in a real-world setting an available sample may simply contain too little information to learn the target regular expression. To formalize this,

	$\|\Sigma\| = 5$	$\|\Sigma\| = 10$
target size	0.52	0.67
RWR^0	0.88	0.98
RWR	0.80	0.96
RWR^2_ℓ		
1	0.76	0.95
2	0.73	0.92
3	0.725	0.916
4	0.722	0.911
5	0.721	0.908
∞	0.720	N/A

Table 2.4: Average language size for RWR and RWR^2_ℓ for various values of ℓ. $\ell = \infty$ denotes an exhaustive exploration of all possible repairs.

RWR^2_ℓ	$\|\Sigma\| = 5$	$\|\Sigma\| = 10$
1	28.8 %	46.3 %
2	7.6 %	7.3 %
3	3.2 %	1.2 %
4	1.3 %	0.0 %
5	0.7 %	0.0 %
∞	24.6 %	N/A

Table 2.5: Percentage of target expressions for which RWR outperforms RWR^2_ℓ.

we introduce the notion of coverage.

Definition 2.46. A sample S *covers* a deterministic automaton A if for every edge (s, t) in A there is a word $w \in S$ whose unique accepting run in A traverses (s, t). Such a word w is called a *witness* for (s, t). A sample S covers a deterministic regular expression r if it covers the automaton obtained from S using the Glushkov construction for translating regular expressions into automata [BK93].

If a sample S does not contain a witness for an edge, it may seem as if the target expression can not be learned, even if it is a SORE since the SOA derived from the data has an edge missing. However, the repair rules introduce extra edges, so this part of the algorithm may actually alleviate the problem of incomplete data. This is indeed confirmed experimentally. It turns out that even with a substantial fraction of missing witnesses, the target regular expression can be learned with an astonishing degree of success. To quantify the missing information, we introduce the following definition:

Definition 2.47. The *coverage* of a sample with respect to a target expression r is the ratio of the number of edges of the SOA derived from the sample and the SOA representing the target expression r.

The tests were done on 100 real-world regular expressions of alphabet sizes up to 10, for 10 independently selected samples of varying coverage. The results are presented in Table 2.6. The straightforward CRX is clearly outperforming all other algorithms, although this results should be approached with some caution: to give CRX a fair chance, the target expressions for this algorithm were limited to CHAREs, while the other algorithms were tested on general SOREs as well. Note that approximately 90 % of real-world expressions are in fact CHAREs, hence its superior performance is not only due to simpler target expressions. The robustness of RWR^2_1 is quite remarkable since it tends to derive more specific regular expressions than RWR^0 and RWR. One would expect the generalization ability to decrease for algorithms that yield more specific results. This expectation is borne out when one compares RWR^0 and RWR, however, RWR^2_1's greedy application of the repair rules seems to pay off in the context of incomplete data as well.

coverage	CRX	RWR^0	RWR	RWR^2_1
25.0	85 %	56 %	12 %	73 %
35.0	87 %	48 %	32 %	73 %
45.0	96 %	60 %	57 %	74 %
55.0	87 %	58 %	63 %	57 %
65.0	82 %	48 %	58 %	59 %
75.0	80 %	51 %	51 %	63 %
85.0	63 %	48 %	47 %	53 %
92.5	57 %	48 %	47 %	61 %
97.5	85 %	74 %	64 %	73 %
100.0	100 %	100 %	100 %	100 %

Table 2.6: Percentage of successfully derived expressions at various values of sample coverage for CRX, RWR^0, RWR and RWR^2_1.

2.5.4 Noise

As already noted in the Introduction, real-worlds samples need not be valid with respect to its known schema. Errors crop up due to all sorts of circumstances. This underscores the need for a robust inference algorithm that can handle some noise in the input sample.

Noise can come in several forms. To generate a noisy subsample, we modify the target expression either by replacing a symbol by a different one from the target's expression, or by replacing it by a symbol that is not in the alphabet of the target expression. We than use the modified target expression to generate a complete sample. We define the noise level as follows:

Definition 2.48. Given a target expression r, the noise level of a sample S is the ratio $|S - \mathcal{L}(r)|/|S|$.

Here we propose an approach to filter the sample S based on the probability of its words being generated by a probabilistic automaton as we also use in Section 3.2. This probabilistic automaton has one state for each alphabet symbol, and the transition probabilities are computed using the Baum-Welsh algorithm. Given the probabilistic automaton, it is straightforward to compute the probability for each $w \in S$, so that one can rank the sample's words. One expects words that contain noise, i.e., that would be rejected by the target regular expression, to have low probability if their number is not excessively large compared to the sample's size.

To filter the sample, hoping to exclude those words that contain noise, we compute the mean μ and standard deviation σ of the sample's probabilities. A string $w \in S$ with probability $P(w)$ is excluded if $P(w) < \mu - \alpha\sigma$. The factor α is a parameter of the algorithm. The filtered sample S' is now used to derive a regular expression. It is of course possible that in the generation of S' some words needed to derive the target expression were removed. Hence there is no guarantee that the derived regular expression will be an overapproximation of the target expression. Experimental results will be discussed in Section 2.5.4.

Since it was shown above that $\textsc{rwr}^2_1$ has the best overall performance, we focus solely on this algorithm in this section. In order to investigate how robust $\textsc{rwr}^2_1$ is with respect to noise we apply the algorithm to samples S with increasing noise levels and consider a range of values for the cut-off α. We compute the precision and the recall for each individual expression and use the average values of these quantities over many expressions to compute the F-value for a given noise level and cut-off so that the optimal cut-off point can be determined.

To define precision and recall, consider the sample $S = S_{\text{valid}} \cup S_{\text{invalid}}$, where $S_{\text{valid}} \subseteq S$ contains the words in S accepted by the target expression and S_{invalid} contains the words in S not accepted by the target expression. A true positive is a word in S_{valid} that is accepted by the derived expression, while a false negative is a word in S_{valid} that is rejected by the derived expression. Similarly, a false positive is a word in S_{invalid} that is accepted by the derived expression, while a true negative is a word in S_{invalid} that is rejected by the derived expression. We denote by $S_{\text{t.p.}}$ the set of true positives, by $S_{\text{t.n.}}$ the set of true negatives, by $S_{\text{f.p.}}$ the set of false positives and by $S_{\text{f.n.}}$ the set of false negatives.

Definition 2.49. The precision p, recall r, and F-value of a derived regular expression on a sample S are given by

$$\begin{aligned} p &= \frac{|S_{\text{t.p.}}|}{(|S_{\text{t.p.}}| + |S_{\text{f.p.}}|)} \\ r &= \frac{|S_{\text{t.p.}}|}{(|S_{\text{t.p.}}| + |S_{\text{f.n.}}|)} \\ F &= \frac{2pr}{p+r} \end{aligned}$$

Furthermore, we are interested in the fraction of derived regular expressions that is equivalent to the target expression.

We average over 580 SOREs obtained from a corpus of real-world DTDs. The results are shown in Figure 2.21. From the F-value we can conclude that a cut-off value $\alpha_F \approx 0.7$ yields the best balance between precision and recall. Figure 2.22 shows the fraction of derived regular expressions that is equivalent to the target expression. For noise levels increasing from 0.01 to 0.05, the F-value as well as the percentage of derived expressions equivalent to the target expression gradually decreases, as is to be expected. It should be noted that a recall $r < 1$ implies that the language represented by the derived regular expression is not a superset of the target's language. For the cut-off α_F, and a noise level of 0.01, approximately 16 % of the derived regular expressions allow false negatives, while the value for a noise level of 0.05 is 15 %. The fact that the derived expression is not a superapproximation may or may not be acceptable, depending on the application.

Another interesting observation is that the number of derived expressions that is equivalent to the target expression increases beyond the cut-off value α_F, see Figure 2.22. For a noise level of 0.01, this trend continues up to cut-off values of $\alpha_{\text{equiv.}} \approx 0.3$ where it reaches a maximum of approximately 53 %. However, at this value 20 % of the derived regular expressions are not superapproximations to their target expressions. For $\alpha < \alpha_{\text{equiv.}}$, the F-value decreases rapidly. For higher noise levels, the optimal cut-off value $\alpha_{\text{equiv.}}$ is smaller, but since it is very unlikely that one knows the noise level, it is hard to take advantage of this fact by tuning $\alpha_{\text{equiv.}}$ to a specific noise level. The overall best result will be obtained for $\alpha_{\text{equiv.}} \approx 0$ for noise levels not exceeding 0.05.

It should be noted that for a noise level of 0.01 at $\alpha_{\text{equiv.}}$, out the 53 % of derived regular expression that are equivalent to the target expression, about 7 % is not covered by the sample. The latter illustrates once more the generalization ability of the algorithms RWR^2_1 as was discussed in Section 2.5.3.

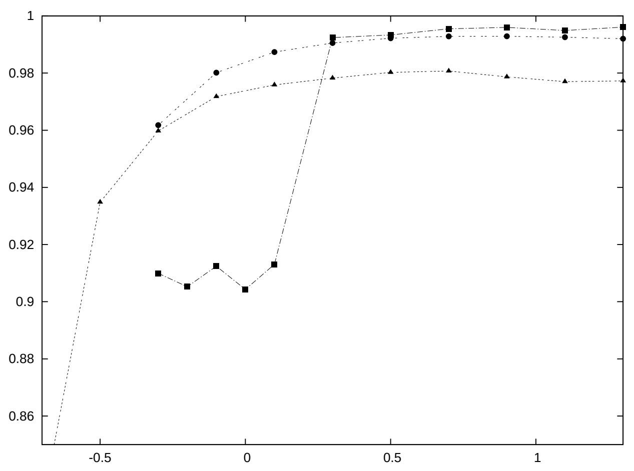


Figure 2.21: F-value as a function of the cut-off value α for noise levels of 0.01 (squares), 0.02 (circles), 0.05 (triangles)

2.5.5 Performance

As mentioned previously, the one advantage RWR has over RWR^2_ℓ is that the former's running time is much lower than the latter's. This is illustrated in Table 2.7 for 1000 target expressions of alphabet size 10. It also shows the relative running time for RWR^0, illustrating that RWR outperforms both RWR^0 and RWR^2_ℓ for any value of ℓ. However, it is interesting to note that RWR^2_1 outperforms RWR^0 by a factor of 3, and derives more specific regular expressions, again illustrating the superiority of the new algorithms over RWR^0.

	relative running time
RWR^0	$6 \cdot 10^2$
RWR^2_ℓ	
1	$2 \cdot 10^2$
2	$2 \cdot 10^3$
3	$1 \cdot 10^4$
4	$4 \cdot 10^4$
5	$1 \cdot 10^5$

Table 2.7: Relative running times of RWR^2_ℓ versus RWR for various values of ℓ.

The performance of RWR is excellent: on average it takes only ms to derive an expression of alphabet size 10. Table 2.8 shows actual running times as a function of the target expressions' alphabet size, averaged over 1000 random expressions of that alphabet size.

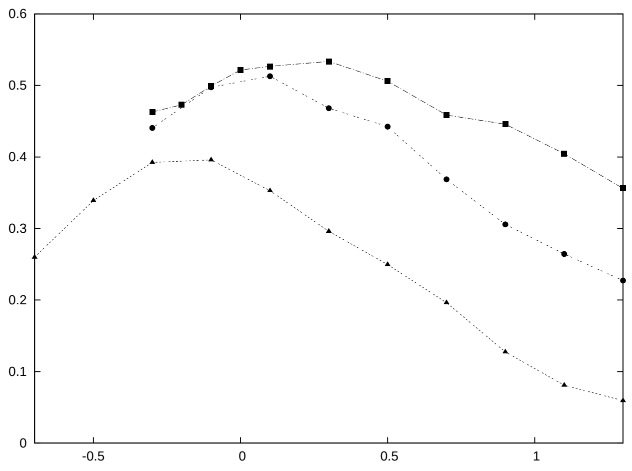


Figure 2.22: Fraction of derived expressions equivalent to the target expression as a function of the cut-off value α for noise levels of 0.01 (squares), 0.02 (circles), 0.05 (triangles)

$\|\Sigma\|$	time (ms)
5	2
10	5
15	15
20	33
50	616
100	7562

Table 2.8: Average running times in milliseconds for RWR as a function of alphabet size

With respect to the performance in terms of the number of examples, we showed in previous work RWR^0's was adequate to deal with large data sets. `Example4` with 61 symbols in Table 2.3 is derived from 10000 example words in 7 seconds while CRX only needs 3.2 seconds. More typical expressions of about 10 symbols derived from a few hundred examples take approximately a second. These figures include the time to initialize a Java Virtual Machine while the tests are done on a 2.5 GHz P4 with 512 MB of RAM. Given that RWR and RWR^2_1 outperform RWR^0 and the time required to start the virtual machine and parse the data is independent of the algorithm, our new algorithms are adequate as well. Trang slightly outperforms CRX thanks to very efficient XML parsing. We did not make a detailed comparison with XTRACT for the reason that XTRACT can not handle samples with more than 1000 words.

2.6 Conclusion

We introduced novel algorithms for the inference of concise regular expressions from positive data. For the inference of SOREs, RWR^2_ℓ was shown to yield the best experimental results. It is also quite robust when presented with incomplete and noisy data. We show that the quality of inferred expressions on real-world and synthetic data sets outperforms those returned by XTRACT where CRX is similar to Trang. CRX' generalization ability makes it highly qualified in dealing with very small data sets. Further, RWR, RWR^2_ℓ and CRX always infer succinct expressions by definition which can easily be interpreted by humans. Of independent interest, we introduced a new algorithm to transform automata into short, readable regular expressions.

3
Inferring k-occurrence regular expressions

While Chapter 2 focused on single occurrence regular expressions, here we broaden the scope to a larger class. The regular expressions occurring in practical DTDs and XSDs are such that every alphabet symbol occurs only a small number of times. As such, to infer an appropriate DTD or XML Schema Definition, in practice it suffices to learn the subclass of deterministic regular expressions in which each alphabet symbol occurs at most k times, for some small k. We refer to such expressions as k-occurrence regular expressions (k-OREs for short). Motivated by this observation, we provide a probabilistic algorithm that learns k-OREs for increasing values of k, and selects the deterministic one that best describes the sample based on a Minimum Description Length and a language size argument. The effectiveness of the method is empirically validated both on real world and synthetic data. Furthermore, the method is shown to be conservative over the simpler classes of expressions considered in Chapter 2.

3.1 Background

In this section we establish that, in contrast to the class of all deterministic expressions, the subclass of deterministic k-OREs *can* theoretically be learned in the limit from positive data, for each fixed k. We also argue, however, that this theoretical algorithm is unlikely to work well in practice.

Let $\Sigma(r)$ denote the set of alphabet symbols that occur in a regular expression r, and let $\Sigma(S)$ be similarly defined for a sample S. Define the *length* of a regular expression r as the length of it string representation, including operators and parenthesis. For example, the length of $(a \,.\, b)^{+}? + c$ is 9.

Theorem 3.1. *For every k there exists an algorithm M that learns the class of deterministic k-OREs from positive data. Furthermore, on input S, M runs in time polynomial in the size of S, yet exponential in k and $|\Sigma(S)|$.*

Proof. The algorithm M is based on the following observations. First observe that every deterministic k-ORE r over a finite alphabet $A \subseteq \Sigma$ can be simplified into an equivalent deterministic k-ORE r' of length at most $10k|A|$ by rewriting r according to the following system of rewrite rules until no more rule is applicable:

$$\begin{array}{rcl@{\qquad}rcl}
((s)) & \to & (s) & s?^{+} & \to & s^{+}? \\
s?? & \to & s? & s^{++} & \to & s^{+} \\
s+\varepsilon & \to & s? & \varepsilon+s & \to & s? \\
s\,.\,\varepsilon & \to & s & \varepsilon\,.\,s & \to & s \\
\varepsilon? & \to & \varepsilon & \varepsilon^{+} & \to & \varepsilon \\
s+\emptyset & \to & s & \emptyset+s & \to & s \\
s\,.\,\emptyset & \to & \emptyset & \emptyset\,.\,s & \to & \emptyset \\
\emptyset? & \to & \emptyset & \emptyset^{+} & \to & \emptyset
\end{array}$$

(The first rewrite rule removes redundant parenthesis in r.) Indeed, since each rewrite rule clearly preserves determinism and language equivalence, r' must be a deterministic expression equivalent to r. Moreover, since none of the rewrite rules duplicates a subexpression and since r is a k-ORE, so is r'. Now note that, since no rewrite rule applies to it, r' is either $\emptyset$, ε, or generated by the following grammar

$$\begin{array}{rcl}
t & ::= & a \mid a? \mid a^{+} \mid a^{+}? \mid (a) \mid (a)? \mid (a)^{+} \mid (a)^{+}? \\
& \mid & t_1\,.\,t_2 \mid (t_1\,.\,t_2) \mid (t_1\,.\,t_2)? \mid (t_1\,.\,t_2)^{+} \mid (t_1\,.\,t_2)^{+}? \\
& \mid & t_1+t_2 \mid (t_1+t_2) \mid (t_1+t_2)? \mid (t_1+t_2)^{+} \mid (t_1+t_2)^{+}?
\end{array}$$

It is not difficult to verify by structural induction that any expression t produced by this grammar has length

$$|t| \leq -4 + 10 \sum_{a\in\Sigma(t)} \mathit{rep}(t,a),$$

where $rep(t, a)$ denotes the number of times alphabet symbol a occurs in t. For instance, $rep(b\,.(b+c), a) = 0$ and $rep(b\,.(b+c), b) = 2$. Since $rep(r', a) \leq k$ for every $a \in \Sigma(r')$, it readily follows that $|r'| \leq 10k|A| - 4 \leq 10k|A|$.

Then observe that all possible regular expressions over A of length at most $10k|A|$ can be enumerated in time exponential in $k|A|$. Since checking whether a regular expression is deterministic is decidable in polynomial time [BKW98]; and since equivalence of deterministic expressions is decidable in polynomial time [BKW98], it follows by the above observations that for each k and each finite alphabet $A \subseteq \Sigma$ it is possible to compute in time exponential in $k|A|$ a finite set $\mathcal{R}_A$ of pairwise non-equivalent deterministic k-OREs over A such that

- every $r \in \mathcal{R}_A$ is of size at most $10k|A|$; and
- for every deterministic k-ORE r over A there exists an equivalent expression $r' \in \mathcal{R}_A$.

(Note that since $\mathcal{R}_A$ is computable in time exponential in $k|A|$, it has at most an exponential number of elements in $k|A|$.) Now fix, for each finite $A \subseteq \Sigma$ an arbitrary order $\prec$ on $\mathcal{R}_A$, subject to the provision that $r \prec s$ only if $\mathcal{L}(r) - \mathcal{L}(s) \neq \emptyset$. Such an order always exists since $\mathcal{R}_A$ does not contain equivalent expressions.

Then let M be the algorithm that, upon sample S, computes $\mathcal{R}_{\Sigma(S)}$ and outputs the first (according to $\prec$) expression $r \in \mathcal{R}_{\Sigma(S)}$ for which $S \subseteq L(r)$. Since $\mathcal{R}_{\Sigma(S)}$ can be computed in time exponential in $k|\Sigma(S)|$; since there are at most an exponential number of expressions in $\mathcal{R}_{\Sigma(S)}$; since each expression $r \in \mathcal{R}_{\Sigma(S)}$ has size at most $10k|\Sigma(S)|$; and since checking membership in $\mathcal{L}(r)$ of a single word $w \in S$ can be done in time polynomial in the size of w and r, it follows that M runs in time polynomial in S and exponential in $k|\Sigma(S)|$.

Furthermore, we claim that M learns the class of deterministic k-OREs. Clearly, $S \subseteq \mathcal{L}(M(S))$ by definition. Hence, it remains to show completeness, i.e., that we can associate to each deterministic k-ORE r a sample $S_r \subseteq L(r)$ such that, for each sample S with $S_r \subseteq S \subseteq L(r)$, $M(S)$ is equivalent to r. Note that, by definition of $\mathcal{R}_{\Sigma(r)}$, there exists a deterministic k-ORE $r' \in \mathcal{R}_{\Sigma(r)}$ equivalent to r. Initialize S_r to an arbitrary finite subset of $\mathcal{L}(r) = \mathcal{L}(r')$ such that each alphabet symbol of r occurs at least once in S, i.e., $\Sigma(S_r) = \Sigma(r)$. Let $r_1 \prec \cdots \prec r_n$ be all predecessors of r' in $\mathcal{R}_{\Sigma(r)}$ according to $\prec$. By definition of $\prec$, there exists a word $w_i \in \mathcal{L}(r) - \mathcal{L}(r_i)$ for every $1 \leq i \leq n$. Add all of these words to S_r. Then clearly, for every sample S with $S_r \subseteq S \subseteq \mathcal{L}(r)$ we have $\Sigma(S) = \Sigma(r)$ and $S \not\subseteq L(r_i)$ for every $1 \leq i \leq n$. Since $M(S)$ is the first expression in $\mathcal{R}_{\Sigma(r)}$ with $S \subseteq L(r)$, we hence have $M(S) = r' \equiv r$, as desired. $\square$

While Theorem 3.1 shows that the class of deterministic k-OREs is better suited for learning from positive data than the complete class of deterministic expressions, it does not provide a useful practical algorithm, for the following reasons.

1. First and foremost, M runs in time exponential in the size of the alphabet $\Sigma(S)$, which may be problematic for the inference of schema's with many element names.

2. Second, while Theorem 3.1 shows that the class of deterministic k-OREs is learnable in the limit for each fixed k, the schema inference setting is such that we do not know k a priori. If we overestimate k then $M(S)$ risks being an under-approximation of the target expression r, especially when S is incomplete. To illustrate, consider the 1-ORE target expression $r = a^+b^+$ and sample $S = \{ab, abbb, aabb\}$. If we overestimate k to, say, 2 instead of 1, then M is free to output $aa?b^+$ as a sound answer. On the other hand, if we underestimate k then $M(S)$ risks being an over-approximation of r. Consider, for instance, the 2-ORE target expression $r = aa?b^+$ and the same sample $S = \{ab, abbb, aabb\}$. If we underestimate k to be 1 instead of 2, then M can only output 1-OREs, and needs to output at least a^+b^+ in order to be sound. In summary: we need a method to determine the most suitable value of k.

3. Third, the notion of learning in the limit is a very liberal one: correct expressions need only be derived when sufficient data is provided, i.e., when the input sample is a superset of the characteristic sample for the target expression r. The following theorem shows that there are reasonably simple expressions r such that characteristic sample S_r of any sound and complete learning algorithm is at least exponential in the size of r. As such, it is unlikely for any sound and complete learning algorithm to behave well on real-world samples, which are typically incomplete and hence unlikely to contain all words of the characteristic sample.

Theorem 3.2. *Let $A = \{a_1, \dots, a_n\} \subseteq \Sigma$ consist of n distinct element names. Let $r_1 = (a_1a_2 + a_3 + \cdots + a_n)^+$, and let $r_2 = (a_2 + \cdots + a_n)^+ a_1 (a_2 + \cdots + a_n)^+$. For any algorithm that learns the class of deterministic $(2n + 3)$-OREs and any sample S that is characteristic for r_1 or r_2 we have $|S| \geq \sum_{i=1}^{n} (n - 2)^i$.*

Proof. First consider $r_1 = (a_1a_2 + a_3 + \cdots + a_n)^+$. Observe that there exist an exponential number of deterministic $(2n+3)$-OREs that differ from r_1 in only a single word. Indeed, let $B = A - \{a_1, a_2\}$ and let W consist of all non-empty words w over B of length at most n. Define, for every word $w = b_1 \dots b_m \in W$

the deterministic $(2n+3)$-ORE r_w such that $\mathcal{L}(r_w) = \mathcal{L}(r_1) - \{w\}$ as follows. First, define, for every $1 \leq i \leq m$ the deterministic 2-ORE r_w^i that accepts all words in $\mathcal{L}(r_1)$ that do not start with b_i:

$$r_w^i := (a_1 a_2 + (B - \{b_i\})) \, .(a_1 a_2 + a_3 + \cdots + a_n)^*$$

Clearly, $v \in \mathcal{L}(r_1) - \{w\}$ if, and only if, $v \in \mathcal{L}(r_1)$ and there is some $0 \leq i \leq m$ such that v agrees with w on the first i letters, but differs in the $(i+1)$-th letter. Hence, it suffices to take

$$r_w := r_w^1 + b_1(\varepsilon + r_w^2 + b_2(\varepsilon + r_w^3 + b_3(\cdots + b_{m-1}(\varepsilon + r_w^m + b_m \, . \, r_1) \dots)))$$

Now assume that algorithm M learns the class of deterministic $(2n+3)$-OREs and suppose that S_{r_1} is characteristic for r_1. In particular, $S_{r_1} \subseteq \mathcal{L}(r_1)$. By definition, $M(S)$ is equivalent to r for every sample S with $S_{r_1} \subseteq S \subseteq \mathcal{L}(r_1)$. We claim that in order for M to have this property, W must be a subset of S_r. Then, since W contains all words over B of length at most n, $|S_{r_1}| \geq \sum_{i=1}^{n}(n-2)^i$, as desired. The intuitive argument why W must be a subset of S_r is that if there exists w in $W - S_r$, then M cannot distinguish between r_1 and r_w. Indeed, suppose for the purpose of contradiction that there is some $w \in W$ with $w \notin S_{r_1}$. Then S_{r_1} is a subset of $\mathcal{L}(r_w)$. Indeed, $S_{r_1} = S_{r_1} - \{w\} \subseteq \mathcal{L}(r_1) - \{w\} = \mathcal{L}(r_w)$. Furthermore, since M learns the class of deterministic $(2n+3)$-OREs, there must be some characteristic sample S_{r_w} for r_w. Now, consider the sample $S_{r_1} \cup S_{r_w}$. It is included in both $\mathcal{L}(r_1)$ and $\mathcal{L}(r_w)$ and is a superset of both S_{r_1} and S_{r_w}. But then, by definition of characteristic samples, $M(S_{r_1} \cup S_{r_w})$ must be equivalent to both r_1 and r_w. This is absurd, however, since $\mathcal{L}(r_1) \neq \mathcal{L}(r_w)$ by construction.

A similar argument shows that the characteristic sample S_{r_2} of $r_2 = (a_2 + \cdots + a_n)^+ a_1 (a_2 + \cdots + a_n)^+$ also requires $\sum_{i=1}^{n}(n-2)^i$ elements. In this case, we take $B = A - \{a_1\}$ and we take W to be the set of all non-empty words over B of length at most n. For each $w = b_1 \dots b_m \in W$, we construct the deterministic $(2n+3)$-ORE r_w such that $\mathcal{L}(r_w)$ accepts all words in $\mathcal{L}(r)$ that do not end with $a_1 w$, as follows. Let, for $1 \leq i \leq m$, r_w^i be the 2-ORE that accepts all words in B^+ that do not start with b_i:

$$r_w^i := (B - \{b_i\}) \, . \, B^*$$

Then it suffices to take

$$r_w := B^+ a_1 (r_w^i + b_1(\varepsilon + r_w^2 + b_3(\cdots + b_{m-1}(\varepsilon + r_w^m + b_m B^+) \dots))).$$

A similar argument as for r_1 then shows that the characteristic sample S_{r_2} of r_2 needs to contain, for each $w \in W$, at least one word of the form $v a_1 w$ with $v \in B^+$. Therefore, $|S_{r_2}| \geq \sum_{i=1}^{n}(n-2)^i$, as desired. $\square$

3.2 The learning algorithm

In view of the observations made in Section 3.1, we present in this section a practical learning algorithm that (1) works well on incomplete data and (2) automatically determines the best value of k. Specifically, given a sample S, the algorithm derives deterministic k-OREs for increasing values of k and selects from these candidate expressions the k-ORE that describes S best. To determine the "best" expression we propose two measures: (1) a Language Size measure and (2) a Minimum Description Length measure based on the work of Adriaans and Vitányi [AV06].

Our algorithm does not derive deterministic k-OREs for S directly, but uses, for each fixed k, a probabilistic method to first learn an automaton for S, which is subsequently translated into a k-ORE. Section 3.2.1 explains how the probabilistic method that learns an automaton from S works. Section 3.2.2 explains how the learned automaton is translated into a k-ORE. Finally, Section 3.2.3, introduces the whole algorithm, together with the two measures to determine the best candidate expression.

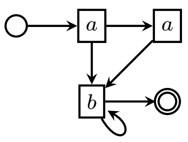
3.2.1 Probabilistically learning an deterministic automaton

The algorithm first learns a *deterministic k-occurrence automaton* (deterministic k-OA) for S. This is a specific kind of finite state automaton in which each alphabet symbol can occur at most k times. Figure 3.1(a) gives an example. Note that in contrast to the classical definition of an automaton, no edges are labeled: all incoming edges in a state s are assumed to be labeled by the label of s. In other words, the 2-OA of Figure 3.1(a) accepts the same language as $aa?b^+$.

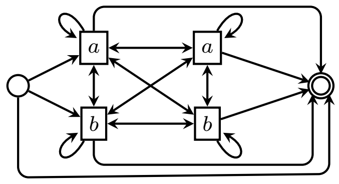
Definition 3.3 (k-OA)**.** An *automaton* is a node-labeled graph $G = (V, E, \mathit{lab})$ where

- V is a finite set of nodes (also called *states*) with a distinguished source $\mathit{src} \in V$ and sink $\mathit{sink} \in V$;
- the edge relation E is such that src has only outgoing edges; sink has only incoming edges; and every state $v \in V - \{\mathit{src}, \mathit{sink}\}$ is reachable by a walk from src to sink;
- $\mathit{lab}\colon V - \{\mathit{src}, \mathit{sink}\} \to \Sigma$ is the labeling function.

In this context, an *accepting run* for a word $a_1 \dots a_n$ is a walk $\mathit{src}\, s_1 \dots s_n\, \mathit{sink}$ from src to sink in G such that $a_i = \mathit{lab}(s_i)$ for $1 \leq i \leq n$. As usual, we denote by $\mathcal{L}(G)$ the set of all words for which an accepting run exists. An



(a) An example 2-OA. It accepts the same language as $aa?b^+$



(b) The complete 2-OA over $\{a, b\}$.

automaton is *k-occurrence* (a k-OA) if there are at most k states labeled by the same alphabet symbol. If G uses only labels in $A \subseteq \Sigma$ then G is said to be *an automaton over A*.

In what follows, we write $\mathrm{Succ}(s)$ for the set $\{t \mid (s,t) \in E\}$ of all direct successors of state s in G, and $\mathrm{Pred}(s)$ for the set $\{t \mid (t,s) \in E\}$ of all direct successors of s in G. Furthermore, we write $\mathrm{Succ}(s,a)$ and $\mathrm{Pred}(s,a)$ for the set of states in $\mathrm{Succ}(s)$ and $\mathrm{Pred}(s)$, respectively, that are labeled by a. As usual, an automaton G is *deterministic* if $\mathrm{Succ}(s,a)$ contains at most one state, for every $s \in V$ and $a \in \Sigma$.

Note that 1-OAs and SOAs are clearly equivalent definitions for the same concept. For that reason we will also refer to the 1-OAs as SOAs in this chapter.

We learn a deterministic k-OA for a sample S as follows. First, recall from Section 3.1 that $\Sigma(S)$ is the set of alphabet symbols occurring in words in S. We view S as the result of a stochastic process that generates words from Σ^* by performing random walks on the *complete* k-OA C_k *over* $\Sigma(S)$.

Definition 3.4. Define the *complete* k-OA C_k over $\Sigma(S)$ to be the k-OA $G = (V, E, lab)$ over $\Sigma(S)$ in which each $a \in \Sigma(S)$ labels exactly k states such that

- there is an edge from *src* to *sink*;
- *src* is connected to exactly one state labeled by a, for every $a \in \Sigma(S)$; and
- every state $s \in V - \{src, sink\}$ has an outgoing edge to every other state except *src*.

To illustrate, the complete 2-OA over $\{a, b\}$ is shown in Figure 3.1(b). Clearly, $\mathcal{L}(C_k) = \Sigma(S)^*$.

The stochastic process that generates words from Σ^* by performing random walks on C_k operates as follows. First, the process picks, among all states in

Succ(src), a state s_1 with probability $\alpha(src, s_1)$ and emits $lab(s_1)$. Then it picks, among all states in Succ(s_1) a state s_2 with probability $\alpha(s_1, s_2)$ and emits $lab(s_2)$. The process continues moving to new states and emitting their labels until the final state is reached (which does not emit a symbol). Of course, α must be a true probability distribution, i.e.,

$$\alpha(s,t) \geq 0; \quad \text{and} \quad \sum_{t \in \text{Succ}(s)} \alpha(s,t) = 1 \qquad (3.1)$$

for all states $s \neq sink$ and all states t. The probability of generating a particular accepting run $\vec{s} = src\, s_1 s_2 \dots s_n\, sink$ given the process $\mathcal{P} = (C_k, \alpha)$ in this setting is

$$P[\vec{s} \mid \mathcal{P}] = \alpha(src, s_1) \cdot \alpha(s_2, s_3) \cdot \alpha(s_2, s_3) \cdots \alpha(s_n, sink),$$

and the probability of generating the word $w = a_1 \dots a_n$ is

$$P[w \mid \mathcal{P}] = \sum_{\text{all accepting runs } \vec{s} \text{ of } w \text{ in } C_k} P[\vec{s} \mid \mathcal{P}].$$

Assuming independence, the probability of obtaining all words in the sample S is then

$$P[S \mid \mathcal{P}] = \prod_{w \in S} P[w \mid \mathcal{P}].$$

Clearly, the process that best explains the observation of S is the one in which the probabilities α are such that they maximize $P[S \mid \mathcal{P}]$.

To learn a deterministic k-OA for S we therefore first try to infer from S the probability distribution α that maximizes $P[S \mid \mathcal{P}]$, and use this distribution to determine the topology of the desired deterministic k-OA. In particular, we remove from C_k the non-deterministic edges with the lowest probability as these are the least likely to contribute to the generation of S, and are therefore the least likely to be necessary for the acceptance of S.

The problem of inferring α from S is well-studied in Machine Learning, where our stochastic process $\mathcal{P}$ corresponds to a particular kind of Hidden Markov Model sometimes referred to as a Partially Observable Markov Model (POMM for short). (For the readers familiar with Hidden Markov Models we note that the initial state distribution π usually considered in Hidden Markov Models is absorbed in the state transition distribution $\alpha(src, \cdot)$ in our context.) Inference of α is generally accomplished by the well-known Baum-Welsh algorithm [Rab89] that adjusts initial values for α until a (possibly local) maximum is reached.

We use Baum-Welsh in our learning algorithm iKoa shown in Algorithm 8, which operates as follows. In line 1, iKoa initializes the stochastic process $\mathcal{P}$ to the tuple (C_k, α) where

Algorithm 8 iKoa

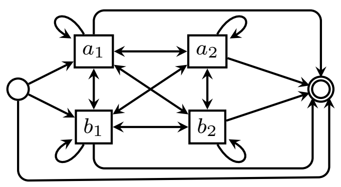
```
Input: a sample $S$, a value for $k$
Output: a deterministic $k$-OA $G$ with $S \subseteq \mathcal{L}(G)$
1: $\mathcal{P} \leftarrow \text{init}(k, S)$
2: $\mathcal{P} \leftarrow$ BaumWelsh($\mathcal{P}, S$)
3: $G \leftarrow$ Disambiguate($\mathcal{P}, S$)
4: $G \leftarrow$ Prune($G, S$)
5: return $G$
```

- C_k is the complete k-OA over $\Sigma(S)$;
- $\alpha(\mathit{src}, \mathit{sink})$ is the fraction of empty words in S;
- $\alpha(\mathit{src}, s)$ is the fraction of words in S that start with $\mathit{lab}(s)$, for every $s \in \text{Succ}(\mathit{src})$; and
- $\alpha(s, t)$ is chosen randomly for $s \neq \mathit{src}$, subject to the constraints in equation (3.1).

It is important to emphasize that, since we are trying to model a stochastic process, multiple occurrences of the same word in S *are* important. A sample should therefore not be considered as a set in Algorithm 8, but as a *bag*. Line 2 then optimizes the initial values of α using the Baum-Welsh algorithm.

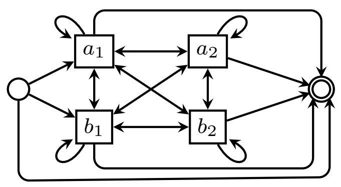
With these probabilities in hand Disambiguate, shown in Algorithm 9, determines the topology of the desired deterministic k-OA for S. In a breadth-first manner, it picks for each state s and each symbol a the state $t \in \text{Succ}(s, a)$ with the highest probability and deletes all other edges to states labeled by a. Line 7 merely ensures that α continues to be a probability distribution after this removal and line 11 adjusts α to the new topology. Line 12 is a sanity check that ensures that we have not removed edges necessary to accept all words in S; Disambiguate reports failure otherwise. The result of a successful run of Disambiguate is a deterministic k-OA which nevertheless may have edges (s, t) for which there is no *witness* in S (i.e., a word in S whose unique accepting run traverses (s, t)). The function Prune in line 4 of iKoa removes all such edges. It also removes all states $s \in \text{Succ}(\mathit{src})$ without a witness in S. Figure 3.1 illustrates a hypothetical run of iKoa.

It should be noted that BaumWelsh, which iteratively refines α until a (possibly local) maximum is reached, is computationally quite expensive. For that reason, our implementation only executes a fixed number of refinement iterations of BaumWelsh in Line 11. Rather surprisingly, this cut-off actually improves the precision of iDRegEx, as our experiments in Section 3.3 show, where it is discussed in more detail.



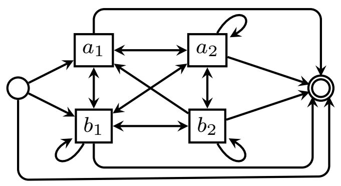
α	a_1	a_2	b_1	b_2	$sink$
src	1	\	0	\	0
a_1	0.2	0.3	0.3	0.1	0.1
a_2	0.4	0.1	0.2	0.1	0.2
b_1	0.1	0.3	0.3	0.2	0.1
b_2	0.1	0.1	0.2	0.5	0.1

(c) Process $\mathcal{P}$ returned by init with random values for α.



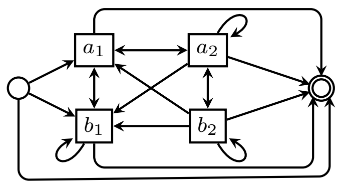
α	a_1	a_2	b_1	b_2	$sink$
src	1	\	0	\	0
a_1	0.2	0.3	0.3	0.19	0.01
a_2	0.01	0.01	0.6	0.37	0.01
b_1	0.01	0.01	0.5	0.28	0.2
b_2	0.01	0.01	0.33	0.5	0.15

(d) Process $\mathcal{P}$ after first training by BAUMWELSH.



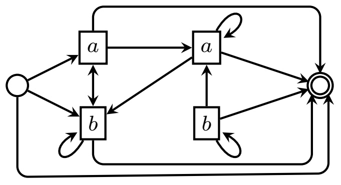
α	a_1	a_2	b_1	b_2	$sink$
src	1	\	0	\	0
a_1	0	0.5	0.49	0	0.01
a_2	0.01	0.01	0.6	0.37	0.01
b_1	0.01	0.01	0.5	0.28	0.2
b_2	0.01	0.01	0.33	0.5	0.15

(e) Process $\mathcal{P}$ after first disambiguation step (for a_1). Edges to a_1 and b_2 are removed.

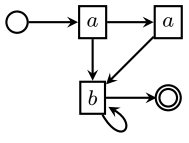


α	a_1	a_2	b_1	b_2	$sink$
src	1	\	0	\	0
a_1	0	0.5	0.49	0	0.01
a_2	0.01	0.01	0.6	0.37	0.01
b_1	0.02	0	0.78	0	0.2
b_2	0.01	0.01	0.38	0.4	0.2

(f) Process $\mathcal{P}$ after second disambiguation step (for b_1). Edges to a_2 and b_2 are removed.



(g) Automaton A returned by DISAMBIGUATE.



(h) Automaton A returned by PRUNE. It accepts the same language as $aa?b^+$.

Figure 3.1: Example run of iKOA for $k = 2$ with target language $aa?b^+$. For the process $\mathcal{P}$ in (c)-(f), the α values are listed in table-form. To distinguish different states with the same label, we have indexed the labels.

Algorithm 9 DISAMBIGUATE

```
Input: a POMM $\mathcal{P} = (G, \alpha)$ and sample $S$
Output: a deterministic $k$-OA
 1: Initialize queue $Q$ to $\{s \in \text{Succ}(\mathit{src}) \mid \alpha(\mathit{src}, s) > 0\}$
 2: Initialize set of marked states $D \leftarrow \emptyset$
 3: while $Q$ is non-empty do
 4:   $s \leftarrow \text{first}(Q)$
 5:   while some $a \in \Sigma$ has $|\text{Succ}(s, a)| > 1$ do
 6:     pick $t \in \text{Succ}(s, a)$ with $\alpha(s, t) = \max\{\alpha(s, t') \mid t' \in \text{Succ}(s, a)\}$
 7:     set $\alpha(s, t) \leftarrow \sum\{\alpha(s, t') \mid t' \in \text{Succ}(s, a)\}$
 8:     for all $t'$ in $\text{Succ}(s, a) \setminus \{t\}$ do
 9:       delete edge $(s, t')$ from $G$
10:       set $\alpha(s, t') \leftarrow 0$
11:     $\mathcal{P} \leftarrow \text{BAUMWELSH}(\mathcal{P}, S)$
12:     if $S \not\subseteq \mathcal{L}(G)$ then Fail
13:   add $s$ to marked states $D$ and pop $s$ from $Q$
14:   enqueue all states in $\text{Succ}(s) \setminus D$ to $Q$
15: return $G$
```

3.2.2 Translating k-OAs into k-OREs

Using the results in Sections 2.1 and 2.2 in the previous chapter, we can state the following theorem.

Theorem 3.5. *Let G be a SOA and let T be any of the algorithms in the family $\{\text{RWR}, \text{RWR}^2_1, \text{RWR}^2_2, \text{RWR}^2_3, \dots\}$. If G is equivalent to a* SORE *r, then $T(G)$ returns a* SORE *equivalent to r. Otherwise, $T(G)$ returns a* SORE *that is a super approximation of G, $\mathcal{L}(G) \subseteq \mathcal{L}(T(G))$.*

(Note that SOAs and SOREs are always deterministic by definition.)

In this section, we show how the SOA to SORE translation algorithms from Chapter 2 can be used to translate k-OAs into k-OREs. For simplicity of exposition, we will focus our discussion on RWR^2_1 as it is the concrete translation algorithm used in our experiments in Section 3.3, but the same arguments apply to the other algorithms in the family.

Definition 3.6. First, let $\Sigma^{(k)}$ denote the alphabet that consists of k copies of the symbols in Σ, where the first copy of $a \in \Sigma$ is denoted by $a^{(1)}$, the second by $a^{(2)}$, and so on:

$$\Sigma^{(k)} := \{a^{(i)} \mid a \in \Sigma, 1 \leq i \leq k\}.$$

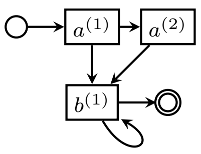


Figure 3.2: An example marking

Let *strip* be the function that maps copies to their original symbol, $\mathit{strip}(a^{(i)}) = a$. We extend *strip* pointwise to words, languages, and regular expressions over $\Sigma^{(k)}$.

For example

$$\mathit{strip}(\{a^{(1)}a^{(2)}b^{(1)}, a^{(2)}a^{(2)}c^{(2)}\}) = \{aab, aac\}$$

and

$$\mathit{strip}(a^{(1)} \,.\, a^{(2)}? \,.\, b^{(1)+}) = a \,.\, a? \,.\, b^{+}$$

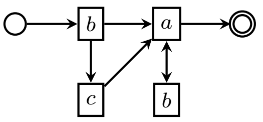
To see how we can use $\textsc{rwr}^2_1$, which translates SOAs into SOREs, to translate a k-OA into a k-ORE, observe that we can always transform a k-OA G over Σ into a SOA H over $\Sigma^{(k)}$ by processing the nodes of G in an arbitrary order and replacing the ith occurrence of label $a \in \Sigma$ by $a^{(i)}$. To illustrate, the SOA over $\Sigma^{(2)}$ obtained in this way from the 2-OA in Figure 3.1(a) is shown in Figure 3.2. Clearly, $\mathcal{L}(G) = \mathit{strip}(\mathcal{L}(H))$.

Definition 3.7. We call a SOA H over $\Sigma^{(k)}$ obtained from a k-OA G in the above manner a *marking* of G.

Note that, by Proposition 3.5, running $\textsc{rwr}^2_1$ on H yields a SORE r over $\Sigma^{(k)}$ with $\mathcal{L}(H) \subseteq \mathcal{L}(r)$. For instance, with H as in Figure 3.2, $\textsc{rwr}^2_1(H)$ returns $r = a^{(1)} \,.\, a^{(2)}? \,.\, b^{(1)+}$. By subsequently stripping r, we always obtain a k-ORE over Σ. Moreover, $\mathcal{L}(G) = \mathit{strip}(\mathcal{L}(H)) \subseteq \mathit{strip}(\mathcal{L}(r)) = \mathcal{L}(\mathit{strip}(r))$, so the k-ORE $\mathit{strip}(r)$ is always a super approximation of G. Algorithm 10, called $\textsc{rwr}^2$ with abuse of notation, summarizes the translation. By our discussion, $\textsc{rwr}^2$ is clearly sound:

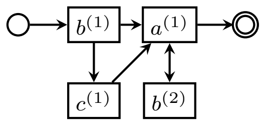
Proposition 3.8. $\textsc{rwr}^2(G)$ *is a (possibly non-deterministic)* k-ORE *with* $\mathcal{L}(G) \subseteq \mathcal{L}(\textsc{rwr}^2(G))$, *for every* k-OA G.

Note, however, that even when G is deterministic and equivalent to a deterministic k-ORE r, $\textsc{rwr}^2(G)$ need not be deterministic, nor equivalent to r. For instance, consider the 2-OA G:



```
Algorithm 10 RWR$^2$
Input: a $k$-OA $G$
Output: a $k$-ORE $r$ with $\mathcal{L}(G) \subseteq \mathcal{L}(r)$
1: compute a marking $H$ of $G$.
2: return $strip(\text{RWR}^2_1(H))$
```

Clearly, G is equivalent to the deterministic 2-ORE $bc?a(ba)^{+}?$. Now suppose for the purpose of illustration that RWR^2 constructs the following marking H of G. (It does not matter which marking RWR^2 constructs, they all result in the same final expression.)



Since H is not equivalent to a SORE over $\Sigma^{(k)}$, RWR$^2_1(H)$ need not be equivalent to $\mathcal{L}(H)$. In fact, RWR$^2_1(H)$ returns $((b^{(1)}c^{(1)}?a^{(1)})?b^{(2)}?)^{+}$, which yields the non-deterministic $((bc?a)?b?)^{+}$ after stripping. Nevertheless, G is equivalent to the deterministic 2-ORE $bc?a(ba)^{+}?$. So although RWR^2 is always guaranteed to return a k-ORE, it does not provide the same strong guarantees that RWR$^2_1$ provides (Proposition 3.5). The following theorem shows, however, that if we can obtain G by applying the Glushkov construction on r [BK93], RWR$^2(G)$ is always equivalent to r. Moreover, if r is deterministic, then so is RWR$^2(G)$. So in this sense, RWR^2 applies an inverse Glushkov construction to r. Formally, the Glushkov construction is defined as follows.

Definition 3.9. Let r be a k-ORE. Recall from Definition 1.2 that $\overline{r}$ is the regular expression obtained from r by replacing the ith occurrence of alphabet symbol a by $a^{(i)}$, for every $a \in \Sigma$ and every $1 \leq i \leq n$. Let $pos(\overline{r})$ denote the symbols in $\Sigma^{(k)}$ that actually appear in $\overline{r}$. Moreover, let the sets first$(\overline{r})$, last$(\overline{r})$, and follow$(\overline{r}, a^{(i)})$ be defined as shown in Figure 3.3. A k-OA G is a *Glushkov translation* of r if there exists a one-to-one onto mapping $\rho\colon (V(G) - \{src, sink\}) \to pos(\overline{r})$ such that

1. $v \in \text{Succ}(src) \Leftrightarrow \rho(v) \in \text{first}(\overline{r})$;
2. $v \in \text{Pred}(sink) \Leftrightarrow \rho(v) \in \text{last}(\overline{r})$;
3. $v \in \text{Succ}(w) \Leftrightarrow \rho(v) \in \text{follow}(\overline{r}, \rho(w))$; and
4. $strip(\rho(v)) = lab(v)$,

for all $v, w \in V(G) - \{src, sink\}$.

$$
\begin{aligned}
\mathrm{first}(\emptyset) &= \emptyset & \mathrm{first}(\varepsilon) &= \emptyset\\
\mathrm{first}(a^{(i)}) &= \{a^{(i)}\} & \mathrm{first}(\overline{r}?) &= \mathrm{first}(\overline{r})\\
\mathrm{first}(\overline{r}^{+}) &= \mathrm{first}(\overline{r}) & \mathrm{first}(\overline{r}+\overline{s}) &= \mathrm{first}(\overline{r}) \cup \mathrm{first}(\overline{s})
\end{aligned}
$$

$$
\mathrm{first}(\overline{r}\,.\,\overline{s}) = \begin{cases} \mathrm{first}(\overline{r}) & \text{if } \varepsilon \notin \mathcal{L}(\overline{r}),\\ \mathrm{first}(\overline{r}) \cup \mathrm{first}(\overline{s}) & \text{otherwise.} \end{cases}
$$

$$
\begin{aligned}
\mathrm{last}(\emptyset) &= \emptyset & \mathrm{last}(\varepsilon) &= \emptyset\\
\mathrm{last}(a^{(i)}) &= \{a^{(i)}\} & \mathrm{last}(\overline{r}?) &= \mathrm{last}(\overline{r})\\
\mathrm{last}(\overline{r}^{+}) &= \mathrm{last}(\overline{r}) & \mathrm{last}(\overline{r}+\overline{s}) &= \mathrm{last}(\overline{r}) \cup \mathrm{last}(\overline{s})
\end{aligned}
$$

$$
\mathrm{last}(\overline{r}\,.\,\overline{s}) = \begin{cases} \mathrm{last}(\overline{s}) & \text{if } \varepsilon \notin \mathcal{L}(\overline{s}),\\ \mathrm{last}(\overline{r}) \cup \mathrm{last}(\overline{s}) & \text{otherwise.} \end{cases}
$$

$$
\begin{aligned}
\mathrm{follow}(a^{(i)}, a^{(i)}) &= \emptyset\\
\mathrm{follow}(\overline{r}?, a^{(i)}) &= \mathrm{follow}(\overline{r}, a^{(i)})\\
\mathrm{follow}(\overline{r}^{+}, a^{(i)}) &= \begin{cases} \mathrm{follow}(\overline{r}, a^{(i)}) & \text{if } a^{(i)} \notin \mathrm{last}(\overline{r}),\\ \mathrm{follow}(\overline{r}, a^{(i)}) \cup \mathrm{first}(\overline{r}) & \text{otherwise.} \end{cases}\\
\mathrm{follow}(\overline{r}+\overline{s}, a^{(i)}) &= \begin{cases} \mathrm{follow}(\overline{r}, a^{(i)}) & \text{if } a^{(i)} \in \mathit{pos}(\overline{r}),\\ \mathrm{follow}(\overline{s}, a^{(i)}) & \text{otherwise.} \end{cases}\\
\mathrm{follow}(\overline{r}\,.\,\overline{s}, a^{(i)}) &= \begin{cases} \mathrm{follow}(\overline{r}, a^{(i)}) & \text{if } a^{(i)} \in \mathit{pos}(\overline{r}), a^{(i)} \notin \mathrm{last}(\overline{r}),\\ \mathrm{follow}(\overline{r}, a^{(i)}) \cup \mathrm{first}(\overline{s}) & \text{if } a^{(i)} \in \mathit{pos}(\overline{r}), a^{(i)} \in \mathrm{last}(\overline{r}),\\ \mathrm{follow}(\overline{s}, a^{(i)}) & \text{otherwise.} \end{cases}
\end{aligned}
$$

Figure 3.3: Definition of first($\overline{r}$), last($\overline{r}$), and follow($\overline{r}, a^{(i)}$), for $a^{(i)} \in \mathit{pos}(\overline{r})$.

Theorem 3.10. *If k-OA G is a Glushkov representation of a target k-ORE r, then $\text{RWR}^2(G)$ is equivalent to r. Moreover, if r is deterministic, then so is $\text{RWR}^2(G)$.*

Proof. Since $\text{RWR}^2(G) = \mathit{strip}(\text{RWR}^2_1(H))$ for an arbitrarily chosen marking H of G, it suffices to prove that $\mathit{strip}(\text{RWR}^2_1(H))$ is equivalent to r and that $\mathit{strip}(\text{RWR}^2_1(H))$ is deterministic whenever r is deterministic, for every marking H of G. Hereto, let H be an arbitrary but fixed marking of G. In particular, G and H have the same set of nodes V and edges E, but differ in their labeling function. Let lab_G be the labeling function of G and let lab_H the labeling function of H. Clearly, $\mathit{lab}_G(v) = \mathit{strip}(\mathit{lab}_H(v))$ for every $v \in V - \{\mathit{src}, \mathit{sink}\}$. Since G is a Glushkov translation of r, there is a one-to-one, onto mapping $\rho\colon (V - \{\mathit{src}, \mathit{sink}\}) \to \mathit{pos}(\overline{r})$ satisfying properties (1)-(4) in Definition 3.9. Now let $\sigma\colon \mathit{pos}(\overline{r}) \to \Sigma^{(k)}$ be the function that maps $a^{(i)} \in \mathit{pos}(\overline{r})$ to $\mathit{lab}_H(\rho^{-1}(a^{(i)}))$. Since lab_H assigns a distinct label to each state, σ is one-to-one and onto the subset of $\Sigma^{(k)}$ symbols used as labels in H. Moreover, by property (4) and the fact that $\mathit{lab}_G(v) = \mathit{strip}(\mathit{lab}_H(v))$ we have,

$$\mathit{strip}(a^{(i)}) = \mathit{lab}_G(\rho^{-1}(a^{(i)})) = \mathit{strip}(\mathit{lab}_H(\rho^{-1}(a^{(i)}))) = \mathit{strip}(\sigma(a^{(i)})) \qquad (\star)$$

for each $a^{(i)} \in \mathit{pos}(\overline{r})$. In other words, σ preserves (stripped) labels. Now let $\sigma(\overline{r})$ be the SORE obtained from $\overline{r}$ by replacing each $a^{(i)} \in \mathit{pos}(\overline{r})$ by $\sigma(a^{(i)})$. Since σ is one-to-one and $\overline{r}$ is a SORE, so is $\sigma(\overline{r})$. Moreover, we claim that $\mathcal{L}(H) = \mathcal{L}(\sigma(\overline{r}))$.

Indeed, it is readily verified by induction on $\overline{r}$ that a word ${a_1}^{(i_1)} \dots {a_n}^{(i_n)} \in \mathcal{L}(\overline{r})$ if, and only if, (i) ${a_1}^{(i_1)} \in \text{first}(\overline{r})$; (ii) ${a_{p+1}}^{(i_{p+1})} \in \text{follow}(\overline{r}, {a_{p+1}}^{(i_{p+1})})$ for every $1 \leq p < n$; and (iii) ${a_n}^{(i_n)} \in \text{last}(\overline{r})$. By properties (1)-(4) of Definition 3.9 we hence obtain:

$$\begin{aligned}
& \sigma({a_1}^{(i_1)}) \dots \sigma({a_n}^{(i_n)}) \in \mathcal{L}(\sigma(\overline{r})) \\
\Leftrightarrow\ & {a_1}^{(i_1)} \dots {a_n}^{(i_n)} \in \mathcal{L}(\overline{r}) \\
\Leftrightarrow\ & \mathit{src}, \rho^{-1}({a_1}^{(i_1)}), \dots, \rho^{-1}({a_n}^{(i_n)}), \mathit{sink} \text{ is a walk in } G \\
\Leftrightarrow\ & \mathit{src}, \rho^{-1}({a_1}^{(i_1)}), \dots, \rho^{-1}({a_n}^{(i_n)}), \mathit{sink} \text{ is a walk in } H \\
\Leftrightarrow\ & \mathit{lab}_H(\rho^{-1}({a_1}^{(i_1)})) \dots, \mathit{lab}_H(\rho^{-1}({a_n}^{(i_n)})) \in \mathcal{L}(H) \\
\Leftrightarrow\ & \sigma({a_1}^{(i_1)}) \dots \sigma({a_n}^{(i_n)}) \in \mathcal{L}(H)
\end{aligned}$$

Therefore, $\mathcal{L}(H) = \mathcal{L}(\sigma(\overline{r}))$.

Hence, we have established that H is a SOA over $\Sigma^{(k)}$ equivalent to the SORE $\sigma(\overline{r})$ over $\Sigma^{(k)}$. By Proposition 3.5, $\text{RWR}^2_1(H)$ is hence equivalent to $\sigma(\overline{r})$. Therefore, $\mathit{strip}(\text{RWR}^2_1(H))$ is equivalent to $\mathit{strip}(\sigma(\overline{r}))$, which by $(\star)$ above, is equivalent to $\mathit{strip}(\overline{r}) = r$, as desired.

Finally, to see that $\mathit{strip}(\text{RWR}^2_1(H))$ is deterministic if r is deterministic, let $s := \mathit{strip}(\text{RWR}^2_1(H))$ and suppose for the purpose of contradiction that

Algorithm 11 iDRegEx

```
Input: a sample $S$
Output: a $k$-ORE $r$
1: initialize candidate set $C \leftarrow \emptyset$
2: for $k = 1$ to $k_{\max}$ do
3:   for $n = 1$ to $N$ do
4:     $G \leftarrow i\text{Koa}(S, k)$
5:     if $\text{rwr}^2(G)$ is deterministic then
6:       add $\text{rwr}^2(G)$ to $C$
7: return best$(C)$
```

s is not deterministic. Then there exists $wa^{(i)}v_1$ and $wa^{(j)}v_2$ in $\mathcal{L}(\overline{s})$ with $i \neq j$. It is not hard to see that this can happen only if there exist $w'a^{(i')}v'_1$ and $w'a^{(j')}v'_2$ in $\mathcal{L}(\text{rwr}^2_1(H))$ with $i' \neq j'$. Since $\mathcal{L}(\text{rwr}^2_1(H)) = \mathcal{L}(\sigma(\overline{r}))$ we know that hence $\sigma^{-1}(w'a^{(i')}v'_1) \in \mathcal{L}(\overline{r})$ and $\sigma^{-1}(w'a^{(j')}v'_2) \in \mathcal{L}(\overline{r})$. Let $w''a^{(i'')}v''_1 = \sigma^{-1}(w'a^{(i')}v'_1)$ and $w''a^{(j'')}v''_2 = \sigma^{-1}(w'a^{(i')}v'_2)$. Since σ is one-to-one and $i' \neq j'$, also $i'' \neq j''$. Therefore, r is not deterministic, which yields the desired contradiction. □

3.2.3 The whole algorithm

Our deterministic regular expression inference algorithm iDRegEx combines iKoa and rwr^2 as shown in Algorithm 11. For increasing values of k until a maximum $k_{\max}$ is reached, it first learns a deterministic k-OA G from the given sample S, and subsequently translates that k-OA into a k-ORE using rwr^2. If the resulting k-ORE is deterministic then it is added to the set C of deterministic candidate expressions for S, otherwise it is discarded. From this set of candidate expressions, iDRegEx returns the "best" regular expression best(C), which is determined according to one of the measures introduced below. Since it is well-known that, depending on the initial value of α, BaumWelsh (and therefore iKoa) may converge to a local maximum that is not necessarily global, we apply iKoa a number of times N with independently chosen random seed values for α to increase the probability of correctly learning the target regular expression from S.

The observant reader may wonder whether we are always guaranteed to derive at least one deterministic expression such that best(C) is defined. Indeed, Theorem 3.10 tells us that if we manage to learn from sample S a k-OA which is the Glushkov representation of the target expression r, then rwr^2 will always return a deterministic k-ORE equivalent to r. When $k > 1$, there can be several k-OAs representing the same language and we could therefore learn a non-Glushkov one. In that case, RWR^2 always returns a k-ORE which is a super approximation of the target expression. Although that approximation can be non-deterministic, since we derive k-OREs for increasing values of k and since for $k = 1$ the result of RWR^2 is always deterministic (as every SORE is deterministic), we always infer at least one deterministic regular expression. In fact, in our experiments on 100 synthetic regular expressions, we derived for 96 of them a deterministic expression with $k > 1$, and only for 4 expressions had to resort to a 1-ORE approximation.

A language size measure for determining the best candidate

Intuitively, we want to select from C the simplest deterministic expression that "best" describes S. Since each candidate expression in C accepts all words in S by construction, one way to interpret "the best" is to select the expression that accepts the least number of words (thereby adding the least number of words to S). To capture this notion, we can use the concept language size that was introduced in Section 2.2.3, where we showed that $|\mathcal{L}(r)^{\leq n}|$ can be computed quite efficiently.

A minimum description length measure for determining the best candidate

An alternative measure to determine the best candidate is given by Adriaans and Vitányi [AV06], who compare the size of S with the size of the language of a candidate r. Specifically, Adriaans and Vitányi define the data encoding cost of r to be:

$$\text{datacost}(r, \mathcal{S}) := \sum_{i=0}^{n} \left(2 \cdot \log_2 i + \log_2 \binom{|\mathcal{L}^{=i}(r)|}{|\mathcal{S}^{=i}|} \right),$$

where $n = 2m + 1$ as before; $|S^{=i}|$ is the number of words in S that have length i; and $|\mathcal{L}^{=i}(r)|$ is the number of words in $\mathcal{L}(r)$ that have exactly length i. Although the above formula is numerically difficult to compute, there is an easier estimation procedure; see [AV06] for details.

In this case, the model encoding cost is simply taken to be its length, thereby preferring shorter expressions over longer ones. The best regular expression in the candidate set C is then the one that minimizes both model and data encoding cost (breaking ties arbitrarily).

We already mentioned that XTRACT [GGR$^+$03] also utilizes the Minimum Description Length principle. However, their measure for data encoding cost depends on the concrete structure of the regular expressions while ours only depends on the language defined by them and is independent of the representation. Therefore, in our setting, when two equivalent expressions are derived, the one with the smallest model cost, that is, the simplest one, will always be taken.

3.3 Experimental evaluation

In this section we validate our approach by means of an experimental analysis.

Our previous work [BGNV08] on this topic was based on a version of the RWR^0 algorithm [BNST06], we refer to this algorithm as iDRegEx(RWR^0). Unfortunately, as detailed in Section 2.3, it is not known whether RWR^0 is complete on the class of all single occurrence regular expressions. Nevertheless, the experiments in [BGNV08] which are revisited below show a good and reliable performance. However, to obtain a theoretically complete algorithm, cf. Theorem 3.10, we use the algorithm RWR^2 which is sound and complete on single occurrence regular expressions. In the remainder we focus on iDRegEx, but compare with the results for iDRegEx(RWR^0).

As mentioned in Section 3.2.3, another new aspect of the results presented here is the use of language size as an alternative measure over MDL to compare candidates. The iDRegEx(RWR^0) algorithm is only considered with the MDL criterion. We note that for alphabet size 5, the success rate of iDRegEx with the MDL criterion was only 21 %, while that of the language size criterion is 98 %. The corpus used in this experiment is described in Section 3.3.3. Therefore in the remainder of this section we only consider iDRegEx with the language size criterion. However, iDRegEx performs poorly with the MDL criterion (for the samples described in Section 3.3.3 with $|\Sigma(S)| = 5$ the success rate was only 21% in contrast to the success rate of 91% with the language size criterion), so iDRegEx, in contrast to iDRegEx(RWR^0), is only considered with the Language Size criterion of selecting the best candidate.

For all the experiments described below we take $k_{\max} = 4$ and $N = 10$ in Algorithm 11.

3.3.1 Running times

All experiments were performed with a prototype implementation of iDRegEx and iDRegEx(RWR^0) written in Java and executed on Pentium M 2.0 GHz class machines equipped with 1GB RAM. For the BaumWelsh subroutine we have gratefully used Jean-Marc François' *Jahmm* library [Fra06], which is a faithful implementation of the algorithms described in Rabiner's Hidden Markov Model tutorial [Rab89]. Since Jahmm strives for clarity rather than performance and since only limited precautions are taken against underflows, our prototype should be seen as a proof of concept rather than a polished product. In particular, underflows currently limit us to target regular expressions whose total number of symbol occurrences is at most 40. Here, the total number of symbol occurrences $occ(r)$ of a regular expression r is its length excluding the regular expression operators and parenthesis. To illustrate, the total number of symbol occurrences in $aa?b^+$ is 3. Furthermore, the lack of optimization in Jahmm leads to average running times ranging from 4 minutes for target expressions r with $|\Sigma(r)| = 5$ and $occ(r) = 6$ to 9 hours for targets expression with $|\Sigma(r)| = 15$ and $occ(r) = 30$. Running times for *i*DRegEx and *i*DRegEx(RWR^0) are similar.

As already mentioned in Section 3.2.3, one of the bottlenecks of *i*DRegEx is the application of BaumWelsh in Line 11 of Disambiguate (Algorithm 9). BaumWelsh is an iterative procedure that is typically run until convergence, i.e., until the computed probability distribution no longer change significantly. To improve the running time, we only apply a fixed number ℓ of iteration steps when calling BaumWelsh in Line 11 of Disambiguate. Experiments show that the running time performance scales linear with ℓ as one expects, but, perhaps surprisingly, the success rate improves as well for an optimal value of ℓ. This optimal value for ℓ depends on the alphabet size. These improved results can be explained as follows: applying BaumWelsh in each disambiguation step until it converges guarantees that the probability distribution for that step will have reached a local optimum. However, we know that the search space for the algorithm contains many local optima, and that BaumWelsh is a local optimization algorithm, i.e., it will converge to one of the local optima it can reach from its starting point by hill climbing. The disambiguation procedure proceeds state by state, so fine tuning the probability distribution for a disambiguation step may transform the search space so that certain local optima for the next iteration can no longer be reached by a local search algorithm such as BaumWelsh. Table 3.1 shows the performance of the algorithm for various number of BaumWelsh iterations ℓ for expressions of alphabet size 5, 10 and 15. These expressions are those described in Section 3.3.3. In this Table, $\ell = \infty$ denotes the case where BaumWelsh is ran until convergence after each disambiguation step. The Table illustrates that the success rate is actually higher for small values of ℓ. The running time performance gains increase rapidly with the expressions' alphabet size: for $|\Sigma| = 5$, we gain a factor of 3.5 ($\ell = 2$), for $|\Sigma| = 10$, it is already a factor of 10 ($\ell = 3$) and for $|\Sigma| = 15$, we gain a factor of 25 ($\ell = 3$). This brings the running time for the largest expressions we tested down to 22 minutes, in contrast with 9 hours mentioned for *i*DRegEx(RWR^0) and *i*DRegEx. The algorithm with the optimal number of BaumWelsh steps in the disambiguation process will be referred to as *i*DRegEx$^{\text{fixed}}$. In particular for small alphabet sizes ($|\Sigma| \leq 7$) we use $\ell = 2$, for large alphabet size $\ell = 3$ ($|\Sigma| < 7$). We note that the alphabet size can easily be determined from the sample.

We should also note that Experience with Hidden Markov Model learning in bio-informatics [FMSB+06] suggests that both the running time and the maximum number of symbol occurrences that can be handled can be significantly improved by moving to an industrial-strength BaumWelsh implementation. Our focus for the rest of the section will therefore be on the precision of iDRegEx.

ℓ	rate $\|\Sigma\| = 5$	rate $\|\Sigma\| = 10$	rate $\|\Sigma\| = 15$
1	95 %	80 %	40 %
2	**100 %**	75 %	50 %
3	95 %	**84 %**	**60 %**
4	95 %	77 %	50 %
∞	98 %	75 %	50 %

Table 3.1: Success rate for a limited number of BaumWelsh iterations in the disambiguation procedure, $\ell = \infty$ corresponds to iDRegEx, for $\ell = 1, \ldots, 4$ correspond to iDRegEx$^{\text{fixed}}$.

3.3.2 Real-world target expressions and real-world samples

We want to test how iDRegEx performs on real-world data. Since the number of publicly available XML corpora with valid schemas is rather limited, we have used as target expressions the 49 content models occurring in the XSD for XML Schema Definitions [TBMM01] and have drawn multiset samples for these expressions from a large corpus of real-world XSDs harvested from the Cover Pages [Cov03]. In other words, the goal of our first experiment is to derive, from a corpus of XSD definitions, the regular expression content models in the schema for XML Schema Definitions[1]. As it turns out, the XSD regular expressions are all single occurrence regular expressions.

The iDRegEx(rwr^0) algorithm infers all these expressions correctly, showing that it is conservative with respect to k since, as mentioned above, the algorithm considers k values ranging from 1 to 4. In this setting, iDRegEx performs not as well, deriving only 73 % of the regular expressions correctly. We note that for each expression that was not derived exactly, always an expression was obtained describing the input sample and which in addition is more specific than the target expression. iDRegEx therefore seems to favor more specific regular expressions, based on the available examples.

[1]This corpus was also used in [BNV07] for XSD inference.

3.3.3 Synthetic target expressions

Although the successful inference of the real-world expressions in Section 3.3.2 suggests that *i*DRegEx is applicable in real-world scenarios, we further test its behavior on a sizable and diverse set of regular expressions. Due to the lack of real-world data, we have developed a synthetic regular expression generator that is parametrized for flexibility.

Synthetic expression generation In particular, the occurrence of the regular expression operators concatenation, disjunction ($+$), zero-or-one ($?$), zero-or-more (*), and one-or-more ($^+$) in the generated expressions is determined by a user-defined probability distribution. We found that typical values yielding realistic expressions are 1/10 for the unary operators and 7/20 for others. The alphabet can be specified, as well as the number of times that each individual symbol should occur. The maximum of these numbers determines the value k of the generated k-ORE.

To ensure the validity of our experiments, we want to generate a wide range of different expressions. To this end, we measure how much the language of a generated expression overlaps with Σ^*. The larger the overlap, the greater its language size as defined in Section 3.2.3.

To ensure that the generated expressions do not impede readability by containing redundant subexpressions (as in e.g., $(a^+)^+$), the final step of our generator is to syntactically simplify the generated expressions using the following straightforward equivalences:

$$
\begin{array}{rcl}
r^* & \rightarrow & r^+? \\
r?? & \rightarrow & r? \\
(r^+)^+ & \rightarrow & r^+ \\
(r?)^+ & \rightarrow & r^+? \\
(r_1 \cdot r_2) \cdot r_3 & \rightarrow & r_1 \cdot (r_2 \cdot r_3) \\
r_1 \cdot (r_2 \cdot r_3) & \rightarrow & r_1 \cdot r_2 \cdot r_3 \\
(r_1? \cdot r_2?)? & \rightarrow & r_1? \cdot r_2? \\
(r_1 + r_2) + r_3 & \rightarrow & r_1 + (r_2 + r_3) \\
r_1 + (r_2 + r_3) & \rightarrow & r_1 + r_2 + r_3 \\
(r_1 + r_2^+)^+ & \rightarrow & (r_1 + r_2)^+ \\
(r_1^+ + r_2^+) & \rightarrow & (r_1 + r_2)^+ \\
r_1 + r_2? & \rightarrow & (r_1 + r_2)?
\end{array}
$$

Of course, the resulting expression is rejected if it is non-deterministic.

$((debab) + c)^*a$
$((((c + b)b) + a)ca) + e + d$
$(((ea)^*db) + b + a + c)^+$
$((b^+ + c + e + d)aab)^+$
$((((eabh) + d + j + c + b)^+f) + a + g + i)?$
$((((aa) + e)^+ + c)b) + b + d$
$((((d + a)^*eabcb) + c)a)?$
$((((ac) + b + d)eab) + c)^*$
$(((((bab) + c)^+ + e)?a) + d)^+$
$((((ecb)^+a) + b)^+ + d + a)?$
$((bagbfeid) + c + a + j + h)^*$
$((gdab) + a + i + c + j + e + f)^+hb$
$((h^*cdfa) + j + e + g + b + i)^*ab$
$((g + b + e + f + i + d)^*aba) + h + j + c$
$((((h + b + c + j + f)^+ + e)?aaidb) + g)?$

$(((((dbe)^*cf) + j)hac) + b + i)^*gad$
$(((((ihaaj) + d)^+ + g)b) + e + b + f + c)^+$
$(((ecgecd) + b + d + a + j + f)^*ihaba)^*$
$(l + c + d + m + n)^*aojahbegcbfidke$
$(((c + b)ab) + d + i + a)^+ + j + g + f + e + h$
$(((a?clfhabgd) + b + n + o)iedjcem)^*k$
$((a + k + f + c + m + e)^+bdieclbonjgda)^*h$
$(((k?jghadfcelifcjbhom)+$
$b + g + a + e + i + n)^+ + d)?$
$(((aedoadenhdbci) + h + k + m + j + g + b)^*$
$fccgelbifja)$
$((a^+ + f + d + o + g + n + h + c + b + j + i + e)$
$keacdlbm)$
$(((k + f + o + a + j)?edhldfhngicjmab)?cie)^*bg$
$((((a?d)^+ba) + h + g + e + c)^+ + j + i + b)?f$

Figure 3.4: A snapshot of the 100 generated expressions.

To obtain a diverse target set, we synthesized expressions with alphabet size 5 (45 expressions), 10 (45 expressions), and 15 (10 expressions) with a variety of symbol occurrences ($k = 1, 2, 3$). For each of the alphabet sizes, the expressions were selected to cover language size ranging from 0 to 1. All in all, this yielded a set of 100 deterministic target expressions. A snapshot is given in Figure 3.4.

Synthetic sample generation For each of those 100 target expressions, we generated synthetic samples by transforming the target expressions into stochastic processes that perform random walks on the automata representing the expressions (cf. Section 3.2). The probability distributions of these processes are derived from the structure of the originating expression. In particular, each operand in a disjunction is equally likely and the probability to have zero or one occurrences for the zero-or-one operator ? is 1/2 for each option. The probability to have n repetitions in a one-or-more or zero-or-more operator (* and $^+$) is determined by the probability that we choose to continue looping (2/3) or choose to leave the loop (1/3). The latter values are based on observations of real-world corpora. Figure 3.5 illustrates how we construct the desired stochastic process from a regular expression r: starting from the following initial graph,

$$\begin{tikzcd}\bigcirc \arrow[r, "1"] & \boxed{r} \arrow[r, "1"] & \circledcirc\end{tikzcd}$$

we continue applying the rewrite rules shown until each internal node is an individual alphabet symbol.

Experiments on covering samples Our first experiment is designed to test how *i*DRegEx performs on samples that are at least large enough to

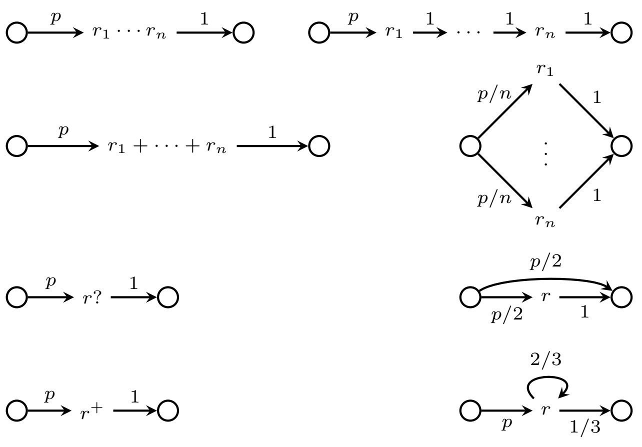


Figure 3.5: From a regular expression to a probabilistic automaton.

cover the target regular expression, as defined in Section 2.5. Intuitively, if a sample does not cover a target regular expression r then there will be parts of r that cannot be learned from S. In this sense, covering samples are the minimal samples necessary to learn r. Note that such samples are far from "complete" or "characteristic" in the sense of the theoretical framework of learning in the limit, as some characteristic samples are bound to be of size exponential in the size of r by Theorem 3.2, while samples of size at most quadratic in r suffice to cover r. Indeed, the Glushkov construction always yields an automaton whose number of states is bounded by the size of r. Therefore, this automaton can have at most $|r|^2$ edges, and hence $|r|^2$ witness words suffice to cover r.

Table 3.2 shows how iDRegEx performs on covering samples, broken up by alphabet size of the target expressions. The size of the sample used is depicted as well. The table demonstrates a remarkable precision. Out of a total of 100 expressions, 82 are derived exactly for iDRegEx. Although iDRegEx(rwr^0) outperforms iDRegEx with a success rate of 87 %, overall iDRegEx$^{\text{fixed}}$ performs best with 89 %. The performance decreases with the alphabet size of the target expressions: this is to be expected since the inference task's complexity increases. It should be emphasized that even if iDRegEx$^{\text{fixed}}$ does not derive the target expression exactly, it always yields an over-approximation, i.e., its language is a superset of the target language.

Table 3.3 shows an alternative view on the results. It shows the success rate as a function of the target expression's language size, grouped in intervals. In particular, it demonstrates that the method works well for all language sizes.

A final perspective is offered in Table 3.4 which shows the success rate in function of the average states per symbol κ for an expression. The latter quantity is defined as the length of the regular expression excluding operators, divided by the alphabet size. For instance, for the expression $a(a+b)^+cab$, $\kappa = 6/3$ since its length excluding operators is 6 and $|\Sigma| = 3$. It is clear that the learning task is harder for increasing values of κ. To verify the latter, a few extra expressions with large κ values were added to the target expressions. For the algorithm iDRegEx$^{\text{fixed}}$ the success rate is quite high for target expressions with a large value of κ. Conversely, iDRegEx(rwr^0) yields better results for $\kappa < 1.6$, while its success rate drops to around 50 % for larger values of κ. This illustrates that neither iDRegEx(rwr^0) nor iDRegEx$^{\text{fixed}}$ outperforms the other in all situations.

$\|\Sigma\|$	#regex	iDRegEx(rwr^0)	iDRegEx	iDRegEx$^{\text{fixed}}$	$\|\mathcal{S}\|$
5	45	86 %	97 %	100 %	300
10	45	93 %	75 %	84 %	1000
15	10	70 %	50 %	60 %	1500
total	100	87 %	82 %	**89 %**	

Table 3.2: Success rate on the target regular expressions and the sample size used per alphabet size for the various algorithms.

Density(r)	#regex	iDRegEx(rwr^0)	iDRegEx	iDRegEx$^{\text{fixed}}$
$[0.0, 0.2[$	24	100 %	87 %	96 %
$[0.2, 0.4[$	22	82 %	91 %	91 %
$[0.4, 0.6[$	20	90 %	75 %	85 %
$[0.6, 0.8[$	22	95 %	72 %	83 %
$[0.8, 1.0]$	12	83 %	78 %	78 %

Table 3.3: Success rate on the target regular expressions, grouped by language size.

κ	#regex	iDRegEx(rwr^0)	iDRegEx	iDRegEx$^{\text{fixed}}$
$[1.2, 1.4[$	29	96 %	72 %	83 %
$[1.4, 1.6[$	37	100 %	89 %	89 %
$[1.6, 1.8[$	24	91 %	92 %	100 %
$[1.8, 2.0[$	11	54 %	91 %	100 %
$[2.0, 2.5[$	12	41 %	50 %	50 %
$[2.5, 3.0]$	18	66 %	71 %	78 %

Table 3.4: Success rate on the target regular expressions, grouped by κ, the average number of states per symbol.

It is also interesting to note that iDRegEx successfully derived the regular expression $r_1 = (a_1a_2 + a_3 + \cdots + a_n)^+$ of Theorem 3.2 for $n = 8$, $n = 10$, and $n = 12$ from covering samples of size 500, 800, and 1100, respectively. This is quite surprising considering that the characteristic samples for these expressions was proved to be of size at least $(n-2)!$, i.e., 720, 40320, and 3628800 respectively. The regular expression $r_2 = (\Sigma \setminus a_1)^+ a_1 (\Sigma \setminus a_1)^+$, in contrast, was not derivable by *i*DREGEX from small samples.

Experiments on partially covering samples Unfortunately, samples to learn regular expressions from are often smaller than one would prefer. In an extreme, but not uncommon case, the sample does not even entirely cover the target expression. In this section we therefore test how *i*DREGEX performs on such samples.

Definition 3.11. The *coverage* of a target regular expression r by a sample S is defined as the fraction of transitions in the corresponding Glushkov automaton for r that have at least one witness in S.

Note that to successfully learn r from a partially covering sample, *i*DREGEX needs to "guess" the edges for which there is no witness in S. This guessing capability is built into *i*DREGEX(RWR^0) and *i*DREGEX in the form of repair rules as discussed in Chapter 2.5.3. Our experiments show that for target expressions with alphabet size $|\Sigma| = 10$, this is highly effective for *i*DREGEX(RWR^0): even at a coverage of 70%, half the target expressions can still be learned correctly as Table 3.5 shows. The algorithm *i*DREGEX is performing very poorly in this setting, being only successful occasionally for coverages close to 100 %. *i*DREGEX$^{\text{fixed}}$ performs better, although not as well as *i*DREGEX(RWR^0). This again illustrates that both algorithms have their merits.

coverage	*i*DREGEX(RWR^0)	*i*DREGEX	*i*DREGEX$^{\text{fixed}}$
1.0	100 %	80 %	80 %
0.9	64 %	20 %	60 %
0.8	60 %	0 %	40 %
0.7	52 %	0 %	0 %
0.6	0 %	0 %	0 %

Table 3.5: Success rate for 25 target expressions for $|\Sigma| = 10$ for samples that provide partial coverage of the target expressions.

We also experimented with target expressions with alphabet size $|\Sigma| = 5$. In this case, the results were not very promising for *i*DREGEX(RWR^0), but as Table 3.6 illustrates, *i*DREGEX and *i*DREGEX$^{\text{fixed}}$ performs better, on par with the target expressions for $|\Sigma| = 10$ in the case of *i*DREGEX$^{\text{fixed}}$. This is interesting since the absolute amount of information missing for smaller regular expressions is larger than in the case of larger expressions.

coverage	*i*DRegEx(rwr^0)	*i*DRegEx	*i*DRegEx$^{\text{fixed}}$
1.0	100 %	100 %	100 %
0.9	25 %	75 %	66 %
0.8	16 %	75 %	41 %
0.7	8 %	25 %	33 %
0.6	8 %	25 %	17 %
0.5	0 %	8 %	17 %

Table 3.6: Success rate for 12 target expressions for $|\Sigma| = 5$ with partially covering samples.

3.4 Conclusions

We presented the algorithm *i*DRegEx for inferring a deterministic regular expression from a sample of words. Motivated by regular expressions occurring in practice, we use a novel measure based on the number k of occurrences of the same alphabet symbol and derive expressions for increasing values of k. We demonstrated the remarkable effectiveness of *i*DRegEx on a large corpus of real-world and synthetic regular expressions of different densities.

Our experiments show that *i*DRegEx(rwr^0) outperforms *i*DRegEx for target expressions with a $\kappa < 1.6$ and vice versa for larger values of κ. For partially covering samples, *i*DRegEx(rwr^0) is more robust than *i*DRegEx. As κ values and sample coverage are not known in advance, it makes sense to run both algorithms and select the smallest expression or the one with the smallest language size, depending on the application at hand.

4

From DTDs to XSDs

While previous work discussed in Section 1.3 and presented in Chapters 2 and 3 has mostly focused on the inference of Document Type Definitions (DTDs for short), here we will consider the inference of XML Schema Definitions (XSDs for short), the increasingly popular schema formalism that is turning DTDs obsolete. In contrast to DTDs where the content model of an element depends only on the element's name, the content model in an XSD can also depend on the context in which the element is used. Hence, while the inference of DTDs basically reduces to the inference of regular expressions from sets of sample strings, the inference of XSDs also entails identifying from a corpus of sample documents the contexts in which elements bear different content models. Since a seminal result by Gold implies that no inference algorithm can learn the complete class of XSDs from positive examples only, we focus on a class of XSDs that captures most XSDs occurring in practice. For this class, we provide a theoretically complete algorithm that always infers the correct XSD when a sufficiently large corpus of XML documents is available. In addition, we present a variant of this algorithm that works well on real-world (and therefore incomplete) data sets.

4.1 Background

XML fragments For our purposes, an *XML fragment* is a (possibly empty) sequence `<`a_1`>` f_1 `</`a_1`>` $\ldots$ `<`a_n`>`f_n`</`a_n`>` of elements where $a_1, \ldots, a_n$ are *element names*, and $f_1, \ldots, f_n$ are themselves XML fragments. In particular, we ignore attributes (as these can straightforwardly be added) and data values (as the inference of atomic data types has already been studied [HNW06]).

As usual we abbreviate `<`a`>` `</`a`>` by `<`a`/>`. Furthermore, if f is an XML fragment, then we write paths(f) for the set of all labeled paths starting at a root element in f. For example, for the XML fragment in Figure 4.1, paths(f) includes the empty path λ, the path `store`, the path `store order`, the path `store stock`, the path `store order customer`, and so on. We write strings(f, p) for the set of all strings of element names occurring below an occurrence of path p in f. For example, for the XML fragment in Figure 4.1, strings$(f, \lambda) = \{$`store`$\}$, strings(f, `store`) $= \{$`order order stock`$\}$, and

$$\begin{aligned}\text{strings}(f, \texttt{store order}) \quad &= \quad \{\texttt{customer item item}, \\ &\qquad\ \texttt{customer item}\}\,.\end{aligned}$$

The first `order` element in Figure 4.1 ensures the presence of `customer item item`, while the second `order` element ensures the presence of `customer item`. For paths like `store order customer name` that end in a leaf of f, strings(f, p) always includes the empty string λ.

XML Schema Definitions The W3C specification [TBMM01] essentially defines an XSD D to be a collection of *type definitions*, which, if we abstract away from the concrete XML representation of XSDs, are rules like

$$\textit{store} \to \texttt{order}[\textit{order}]^*, \texttt{stock}[\textit{stock}] \qquad (\star)$$

that map type names to regular expressions over pairs $a[t]$ of element names a and type names t. Throughout this chapter we use the convention that element names are typeset in typewriter font, and type names are typeset in italic. Intuitively, this particular type definition specifies an XML fragment to be of type *store* if it is of the form

`<order>` f_1 `</order>` $\ldots$ `<order>` f_n`</order>` `<stock>` g `</stock>`

where $n \geq 0$; $f_1, \ldots, f_n$ are XML fragments of type *order*; and g is an XML fragment of type *stock*. Each type name that occurs on the right hand side of a type definition in an XSD must also be defined in the XSD, and each type name may be defined only once.

```
<store>
  <order>
    <customer> <name/> <email/> </customer>
    <item> <id/> <qty/> <price/>  </item>
    <item> <id/> <qty/> <price/>  </item>
  </order>
  <order>
    <customer> <name/> <email/> <email/> </customer>
    <item> <id/> <qty/> <price/>  </item>
  </order>
  <stock>
    <item>
       <id/> <qty/>
       <supplier/> <name/> <email/> </supplier>
     </item>
  </stock>
</store>
```

Figure 4.1: A sample XML fragment for the XSD in Figure 4.2.

It is important to remark that the 'Element Declaration Consistent' constraint of the W3C specification [TBMM01] requires multiple occurrences of the same element name in a single type definition to occur with the same type. Hence, type definition ($\star$) is legal, but

$$\mathit{persons} \to (\texttt{person}[\mathit{male}] + \texttt{person}[\mathit{female}])^{+}$$

is not, as `person` occurs both with type *male* and type *female*. Of course, element names in *different* type definitions can occur with different types (which is exactly what yields the ability to let the content model of an element depend on its context). For example, Figure 4.2 shows a legal XSD describing the intended set of store document from the Introduction. Notice in particular the use of the types item_1 and item_2 to distinguish between order items and stock items.

Due to the 'Element Declaration Consistent' constraint, each element name a occurring in the type definition of a type t is associated with a unique type $\tau(t, a)$ in this type definition. For example, for the XSD in Figure 4.2 we have

$$\begin{aligned}
\tau(\mathit{root}, \texttt{store}) &= \mathit{store}, & \tau(\mathit{store}, \texttt{order}) &= \mathit{order},\\
\tau(\mathit{store}, \texttt{stock}) &= \mathit{stock}, & \tau(\mathit{order}, \texttt{item}) &= \mathit{item}_1,\\
\tau(\mathit{item}_1, \texttt{id}) &= \mathit{emp}, & \tau(\mathit{stock}, \texttt{item}) &= \mathit{item}_2,
\end{aligned}$$

and so on. Then let $\rho(t)$ stand for the ordinary regular expression over element names only that we obtain by removing all types names in the definition of t.

$$\begin{array}{lll}
\mathit{root} & \to & \texttt{store}[\mathit{store}] \\
\mathit{store} & \to & \texttt{order}[\mathit{order}]^*, \texttt{stock}[\mathit{stock}] \\
\mathit{order} & \to & \texttt{customer}[\mathit{person}], \texttt{item}[\mathit{item}_1]^+ \\
\mathit{person} & \to & \texttt{name}[\mathit{emp}], \texttt{email}[\mathit{emp}]^+ \\
\mathit{item}_1 & \to & \texttt{id}[\mathit{emp}], \texttt{qty}[\mathit{emp}], \texttt{price}[\mathit{emp}] \\
\mathit{stock} & \to & \texttt{item}[\mathit{item}_2]^+ \\
\mathit{item}_2 & \to & \texttt{id}[\mathit{emp}], \texttt{qty}[\mathit{emp}], \\
 & & \quad (\texttt{supplier}[\mathit{person}] + \texttt{item}[\mathit{item}_2]^+) \\
\mathit{emp} & \to & \varepsilon
\end{array}$$

Figure 4.2: An XSD describing the XML document in Figure 1.2. The symbol ε denotes the empty string.

For example, for the XSD in Figure 4.2 we have

$$\begin{array}{rlrl}
\rho(\mathit{root}) = \texttt{store} & & \rho(\mathit{store}) = \texttt{order}^*, \texttt{stock} \\
\rho(\mathit{order}) = \texttt{customer}, \texttt{item}^+ & & \rho(\mathit{person}) = \texttt{name}, \texttt{email}^+
\end{array}$$

and so on. Then we can view an XSD D simply as a triple consisting only of (1) the set of types T being defined, (2) the mapping τ, and (3) the mapping ρ. Indeed, observe that the type definition of for example *order*,

$$\mathit{order} \to \texttt{customer}[\mathit{person}], \texttt{item}[\mathit{item}_1]^+$$

is easily obtained by replacing every element name a in the regular expression $\rho(\mathit{order}) = \texttt{customer}, \texttt{item}^+$ by the pair $a[\tau(\mathit{order}, a)]$. Since this view is more amenable to algorithmic manipulation, we will take it as the *definition* of an XSD, although for presentation purposes we will continue to represent XSDs as in Figure 4.2.

Definition 4.1. An XSD is a triple $D = (T, \rho, \tau)$ consisting of a finite set of *types* T; a mapping ρ from T to regular expressions r and a mapping τ that assigns a type to each pair (t, a) with the element name a occurring in $\rho(t)$.

Following the notation in DTDs, we often denote the concatenation of r and r' explicitly by r, r' in the chapter.

As was already noted in Section 1.2, the W3C specification also requires regular expressions to be *deterministic* [TBMM01]. We do not go into details here, as the regular expressions for the classes of XSDs we will be inferring are deterministic by definition.

The semantics of an XSD is given by the following simple algorithm to validate an XML fragment f = `<`a_1`>`f_1`</`a_1`>`...`<`a_n`>`f_n`</`a_n`>` against a type t in an XSD $D = (T, \rho, \tau)$ [MNSB06, MLMK05]. First, we check that the string of element names $a_1 \ldots a_n$ is matched by the regular expression $\rho(t)$. For example, when $t = \mathit{order}$ as defined in Figure 4.2, $a_1 \ldots a_n$ would be matched against $\texttt{customer}, \texttt{item}^+$. If this check fails, then the fragment is rejected. Otherwise, we validate each f_i against the type $\tau(t, a_i)$ of a_i in t, and accept the fragment if all these validations succeed. For example, when $t = \mathit{order}$ as defined in Figure 4.2 and $a_i = \texttt{item}$, f_i would be validated against $\tau(\mathit{order}, \texttt{item}) = \mathit{item}_1$. We write $\mathcal{F}(D, t)$ for the set of all XML fragments of type t in D.

Contextual power The validation algorithm above actually implies that the content model of an element occurring in $f \in \mathcal{F}(D; t)$ is completely determined by the labeled path from the root to that element – a property of XSDs first noted by Martens et al. [MNSB06]. Indeed, for $f = \texttt{<}a_1\texttt{>}f_1\texttt{</}a_1\texttt{>} \ldots \texttt{<}a_n\texttt{>}f_n\texttt{</}a_n\texttt{>}$ to be of type t, each f_i must be valid w.r.t. $\tau(t, a_i)$. This is true only if $f_i = \texttt{<}b_1\texttt{>}g_1\texttt{</}b_1\texttt{>} \ldots \texttt{<}b_m\texttt{>}g_m\texttt{</}b_m\texttt{>}$ and every g_j is valid w.r.t. $\tau(\tau(t, a_i), b_j)$. We can continue this reasoning until we reach the desired element, where we see that its child fragment h must be of type $\tau(\ldots \tau(\tau(t, a_i), b_j) \ldots, c)$ with $a_i b_j \ldots c$ the labeled path from the root to the element. This leads us to the following alternative view on validation, which forms the cornerstone of our inference algorithms. Let, for a path $p = ab \ldots c$, $\tau(s, p) \to t$ denote that $\tau(\ldots \tau(\tau(s, a), b) \ldots, c)$ is defined and equals t. Let $\mathcal{L}(r)$ denote the set of all strings matched by regular expression r.

Proposition 4.2. ([MNSB06]) *An XML fragment f has type s in an XSD (T, ρ, τ) iff for every path $p \in \mathrm{paths}(f)$ there exists t such that $\tau(s, p) \to t$ and* $\mathrm{strings}(f, p) \subseteq \mathcal{L}(\rho(t))$.

Locality The content model of an element in more than 98% of XSDs in practice turns out not to depend on the whole labeled path from the root to the element, but only on the k last element names in that path, with typically $k \leq 3$ [MNSB06]. The formal definition of such *k-local XSDs* is as follows. Let $p|_k$ stand for the path formed by the k last element names of a path p (if $\mathrm{length}(p) \leq k$ then we take $p|_k = p$). Two paths p and q are *k-equivalent* if $p|_k = q|_k$. In particular, when $\mathrm{length}(p) < k$, p is only k-equivalent to itself.

Definition 4.3. A pair (D, s) with D an XSD 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 example, (D, root) with D as in Figure 4.2 is 2-local but not 1-local since $p = \texttt{store order item}$ and $q = \texttt{store stock item}$ are 1-equivalent, yet

$$\tau(\mathit{root}, p) \to \mathit{item}_1 \text{ and } \tau(\mathit{root}, q) \to \mathit{item}_2.$$

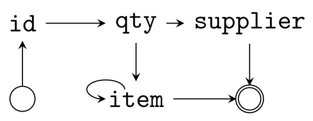


Figure 4.3: The SOA accepting the same language as the SORE $\texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^{+})$.

Observe that the 1-local XSDs are in fact just DTDs.

Single occurrence As already explained in the Introduction, a seminal result by Gold [Gol67] implies that the class of k-local XSDs is still too large to be learned from positive examples only. Fortunately, the regular expressions in more than 99 % of XSDs in practice are of a very specific form: each element name occurs at most once in them [MNSB06]. SOREs were discussed in detail in Chapter 2.

4.2 Inference of local SOXSDs

Our goal in this section is to infer a k-local SOXSD (D', t') equivalent to a target k-local SOXSD (D, t) given only a finite corpus of XML documents $\mathcal{C} \subseteq \mathcal{F}(D, t)$. This entails identifying from $\mathcal{C}$ the contexts (i.e., types) in which elements may bear different content models, as well as these content models (i.e., single occurrence regular expressions) themselves. Intuitively, we will use the paths occurring in $\mathcal{C}$ to identify types and the strings in $\mathcal{C}$ occurring below these paths to identify the SOREs. The latter essentially boils down to inferring a SORE from a set of sample strings, which can be done as was described in Chapter 2. For the results in the remainder of this section, we used the $\textsc{rwr}^2_\ell$ algorithm of Section 2.3.

The algorithm Inference of k-local SOXSDs can now be done as follows. Let paths($\mathcal{C}$) stand for the set of all paths occurring in fragments in the corpus $\mathcal{C}$. Let k-strings($\mathcal{C}, p|_k$) stand for the set of all strings in $\mathcal{C}$ that occur below paths that are k-equivalent to p:

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

Algorithm 12, $i\textsc{Local}$, then infers a k-local SOXSD from a finite corpus of XML fragments $\mathcal{C}$.

Let us illustrate $i\textsc{Local}$'s operation by running it on the corpus $\mathcal{C}$ consisting of the XML fragments of Figure 4.1 and 4.4, which both adhere to

Algorithm 12 iLocal

```
Input: a natural number $k$ and corpus $\mathcal{C}$
Output: a $k$-local SOXSD $(D, t)$ such that $\mathcal{C} \subseteq \mathcal{F}(D, t)$
1: Let the set of types $T$ consist of all $p|_k$ with $p \in \text{paths}(\mathcal{C})$
2: Initialize the mappings $\rho$ and $\tau$ to empty
3: for each type $p|_k$ in $T$ do
4:     add $p|_k \mapsto \text{RWR}^2_\ell(\text{2T-INF}(k\text{-strings}(\mathcal{C}, p|_k)))$ to $\rho$
5: for each path $pa$ in paths($C$) do
6:     add $(p|_k, a) \mapsto (pa)|_k$ to $\tau$
7: Return $(D, t)$ with $D = (T, \rho, \tau)$ and $t = \lambda$
```

the target XSD in Figure 4.2. In line 1, iLocal constructs a type $p|_k$ for each path p in paths($\mathcal{C}$). For $k = 2$, this yields the set of types shown in Figure 4.5. Next, iLocal constructs the content models for these types in lines 3 and 4. It does so by first learning a SOA (see Definition 2.1) for the set k-strings$(\mathcal{C}, p|_k)$ of all strings occurring in $\mathcal{C}$ below a path q that is k-equivalent to the type $p|_k$ under inspection, and subsequently transforming this SOA into a SORE. For $k = 2$ and $p|_k =$ `stock item`, this set of strings is $\{$`id qty supplier`$,$ `id qty item item`$\}$ as the only path 2-equivalent to `stock item` in $\mathcal{C}$ is `store stock item`. Hence, for `stock item`, iLocal will first learn the SOA from Figure 4.3, which is subsequently transformed into the SORE $\texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^+)$. Note that iLocal hence correctly infers that stock items do not contain `price` elements. After termination of the for loop in line 3 we have hence inferred the content models for all types as shown in Figure 4.6.

Finally, in lines 5 and 6 iLocal determines the types associated with the element names in these content models. It does so by adding $(p|_k, a) \mapsto (pa)|_k$ to τ, for every element name a occurring in the content model of type $p|_a$. For $k = 2$ this yields, among others,

$$\begin{aligned}
(\lambda, \texttt{store}) &\mapsto \texttt{store}, \\
(\texttt{store}, \texttt{stock}) &\mapsto \texttt{store stock}, \\
(\texttt{store stock}, \texttt{item}) &\mapsto \texttt{stock item}, \\
(\texttt{stock item}, \texttt{item}) &\mapsto \texttt{item item}, \\
(\texttt{item item}, \texttt{item}) &\mapsto \texttt{item item}.
\end{aligned}$$

Note in particular the recursion introduced in the last rule.

A careful analysis shows that for this specific example corpus, iLocal has successfully inferred the target XSD D from Figure 4.2: $\mathcal{F}(D, \mathit{root}) = \mathcal{F}(i\text{Local}(2, \mathcal{C}))$. This is actually not a coincidence, as iLocal is *complete*

```
<store>
  <stock>
    <item>
       <id/> <qty/>
       <supplier/>
          <name/> <email/> <email/>
       </supplier>
     </item>
     <item>
       <id/> <qty/>
       <item>
          <id/> <qty/>
          <supplier> <name/> <email/> </supplier>
        </item>
        <item>
          <id/> <qty/>
          <item>
            <id/> <qty/>
            <supplier> <name/> <email/> </supplier>
          </item>
          <item>
            <id/> <qty/>
            <supplier> <name/> <email/> </supplier>
          </item>
       </item>
     </item>
  </stock>
</store>
```

Figure 4.4: Another sample XML fragment the XSD in Figure 4.2.

on corpora that are "sufficiently large" in the following sense.

Definition 4.4. Let D be an XSD, let t be a type in D, and let m be the number of types in D. A corpus $\mathcal{C}$ is called *k-complete* for (D, t) if (1) $\mathcal{C} \subseteq \mathcal{F}(D, t)$; (2) paths($C$) contains all paths p of length at most $m+k+1$ such that $\tau(t, p) \to t'$ for some t'; and (3) for each such path, k-strings($\mathcal{C}, p$) contains all strings in $\mathcal{L}(\rho(t'))$ of length at most $2n$, where n is the number of different element names occurring in $\rho(t')$.

Theorem 4.5. *If (D, t) is a k-local SOXSD and corpus $\mathcal{C}$ is k-complete for (D, t), then $\mathcal{F}(i\textsc{Local}(k, \mathcal{C})) = \mathcal{F}(D, t)$.*

Proof. Let n stand for the number of types in (D, t); let $(T, \rho, \tau) = D$; and let $((T'; \rho'; \tau'), \lambda) = i\textsc{Local}(k, \mathcal{C})$. We show that for any path p we have

λ,	`store`,	`store order`,
`order customer`,	`customer name`,	`customer email`,
`order item`,	`item id`,	`item qty`,
`item price`,	`store stock`,	`stock item`,
`item supplier`,	`supplier name`,	`supplier email`,
`item item`		

Figure 4.5: The types inferred when running iLOCAL on the corpus consisting of the XML fragments in Figure 4.1 and 4.4, for $k = 2$.

$$\begin{aligned}
\lambda &\rightarrow \texttt{store}\\
\texttt{store} &\rightarrow \texttt{order}^*, \texttt{stock}\\
\texttt{store order} &\rightarrow \texttt{customer}, \texttt{item}^+\\
\texttt{order customer} &\rightarrow \texttt{name}, \texttt{email}^+\\
\texttt{order item} &\rightarrow \texttt{id}, \texttt{qty}, \texttt{price}\\
\texttt{store stock} &\rightarrow \texttt{item}^+\\
\texttt{stock item} &\rightarrow \texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^+)\\
\texttt{item supplier} &\rightarrow \texttt{name}, \texttt{email}^+\\
\texttt{item item} &\rightarrow \texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^+)
\end{aligned}$$

Figure 4.6: The content models inferred when running iLOCAL on the corpus consisting of the XML fragments in Figure 4.1 and 4.4, for $k = 2$. The types with empty content model λ have been omitted for space efficiency.

$\tau(t,p) \to s$ for some $s \in T$ if, and only if, $\tau'(\lambda,p) \to s'$ for some $s' \in T'$ and $\mathcal{L}(\rho(s)) = \mathcal{L}(\rho'(p|_k)$. The theorem then readily follows by Proposition 4.2.

($\Rightarrow$). So, let p be an arbitrary path and suppose that $\tau(t,p) \to s$. We show that $\tau'(\lambda,p) \to p|_k$ and $\mathcal{L}(\rho(s)) = \mathcal{L}(\rho'(p|_k))$ by induction on length(p).

1. If length(p) $\leq n+k+1$ then $p \in$ paths($\mathcal{C}$) since $\mathcal{C}$ is k-complete for (D,t). It is readily verified by induction on length(p) that $\tau'(\lambda,p) \to p|_k$. Moreover, since $\mathcal{C}$ is k-complete for (D,t) we know that k-strings($\mathcal{C}, p|_k$) contains all strings in $\rho(s)$ of length at most m, where m is the number of element names occurring in $\rho(s)$. By construction,

$$\rho'(p|_k) = \textsc{rwr}^2_\ell(2\textsc{t-inf}(k\text{-strings}(\mathcal{C}, p|_k))),$$

   and hence $\mathcal{L}(\rho'(p|_k)) = \mathcal{L}(\rho(s))$ by Propositions 2.2 and 2.8.

2. if $|(|p) > n+k+1$, then $p = qa_1 \dots a_k b$ for some path q and some element names $a_1, \dots a_k, b$. Since $\tau(t,p) \to s$, we clearly also have $\tau(t, qa_1 \dots a_k) \to s_1$ for some s_1 with $\tau(s_1, b) = s$. By the induction hypothesis, we hence have $\tau'(\lambda, qa_1 \dots a_k) \to a_1 \dots a_k$. Now reason as follows.

   Since $\tau(t,p) \to s$ there must exists s_2 such that $\tau(t,q) \to s_2$ and $\tau(s_2, a_1 \dots a_k) \to s$. Since there are only n types in T, it is readily verified that there exists some path q' of length at most n such that also $\tau(t,q') \to s_2$ (q' can be obtained from q by inspecting the types that are visited multiple times by $\tau(t,q)$, and by removing the corresponding substrings). Then clearly also $\tau(t, q'a_1 \dots a_k) \to s$. Since $\mathcal{C}$ is k-complete for (D,t), since $\tau(t, q'a_1 \dots a_k) \to s_2$, and since $|(|q'a_1 \dots a_k) < n+k+1$ it follows that $q'a_1 \dots a_k \in$ paths($\mathcal{C}$). But then $\tau'((q'a_1 \dots a_k b)|_k, b) = a_2 \dots a_k b$ by construction, and hence $\tau'(\lambda, q'a_1 \dots a_k b) \to a_2 \dots a_k b$, as desired.

   Moreover, since $\mathcal{C}$ is k-complete for (D,t), since $\tau(t, q'a_1 \dots a_k b) \to s$, and since $|(|q'a_1 \dots a_k b) \leq n+k+1$ we know that k-strings($\mathcal{C}, a_2 \dots a_k b$) contains all strings in $\rho(s)$ of length at most m, where m is the number of element names occurring in $\rho(s)$. By construction,

$$\rho'(a_2 \dots a_k b) = \textsc{rwr}^2_\ell(2\textsc{t-inf}(k\text{-strings}(\mathcal{C}, a_2 \dots a_k b)),$$

   and hence $\mathcal{L}(\rho'(a_2 \dots a_k b) = \mathcal{L}(\rho(s))$ by Propositions 2.2 and 2.8.

($\Leftarrow$) Conversely, suppose that $\tau'(\lambda,p) \to s'$ for some $s' \in T$. It is readily verified that then $s' = p|_k$. We show by induction on $|(|p)$ that there exists $s \in T$ such that $\tau(t,p) \to s$ and $\mathcal{L}(\rho'(p|_k)) = \mathcal{L}(\rho(s))$.

1. If $|(|p) = 0$, then p is the empty path λ. In this case, it suffices to take $s = t$. Indeed, clearly $\tau(t, \lambda) \rightarrow s$. By the reasoning in $(\Rightarrow)$ above, we know that $\mathcal{L}(\rho'(\lambda)) = \mathcal{L}(\rho(t))$.

2. If $|(|p) > 0$, then $p = qa$ for some path q and some element name a. Clearly, $\tau'(\lambda, q) \rightarrow q|_k$ and $\tau'(q|_k, a) = (qa)|_k = p|_k$. By the induction hypothesis, there exists $s_1 \in T$ such that $\tau(t, q) \rightarrow s_1$. Now observe that $(q|_k, a) \mapsto (qa)|_k$ is only added to τ' if there exists a path $q'a \in \text{paths}(\mathcal{C})$ such that $q'|_k = q|_k$ (and hence $(q'a)|_k = qa|_k$). But by Proposition 4.2 such a path $q'a$ can only occur in $\mathcal{C}$ if $\tau(t, q'a) \rightarrow s$ for some $s \in T$. The clearly, $\tau(t, q') \rightarrow s_2$ for some $s_2 \in T$ with $\tau(s_2, a) = s$. Since (D, t) is k-local and since $q'|_k = q|_k$ it follows that $s_2 = s_1$. Hence, $\tau(t, qa) \rightarrow s$. By the reasoning in $(\Rightarrow)$ above, we know that $\mathcal{L}(\rho'(p|_k)) = \mathcal{L}(\rho(s))$, as desired. □

Proposition 4.6. *$\mathcal{C} \subseteq i$*Local*$(k, \mathcal{C})$, for every $\mathcal{C}$ and k.*

Proof. By Proposition 4.2 it suffices to show that for each XML fragment $f \in \mathcal{C}$ and each path $p \in \text{paths}(f)$ there exists a type t in $((T, \rho, \tau), \lambda) = i\textsc{Local}(k, \mathcal{C})$ such that $\tau(\lambda, p) \rightarrow t$ and $\text{strings}(f, p) \subseteq \mathcal{L}(\rho(t))$. It is not difficult to see that it suffices to take $p|_k$ for t. Indeed, $\tau(\lambda, p) \rightarrow p|_k$ by construction, and

$$\begin{aligned} \text{strings}(f, p) &\subseteq k\text{-strings}(\mathcal{C}, p|_k) \\ &\subseteq \mathcal{L}(\textsc{rwr}^2_\ell(2\textsc{t-inf}(k\text{-strings}(\mathcal{C}, p|_k))) \\ &= \mathcal{L}(\rho(p|_k)), \end{aligned}$$

by Propositions 2.2 and 2.8. Hence, $f \in \mathcal{F}(i\textsc{Local}(k, \mathcal{C}))$. □

Minimization Although iLocal is complete on sufficiently large corpora, it has the disadvantage that the inferred XSDs may have more types than necessary. For instance, the inferred XSD of Figure 4.5 consists of 16 types, 8 types more than target XSD of Figure 4.2. In the worst case, $i\textsc{Local}(k, \mathcal{C})$ may return an XSD with $O(n^k)$ types where n is the number of different element names appearing in $\mathcal{C}$. For subsequent processing and presentation to the user, it is hence desirable to minimize the results of iLocal.

The algorithm Minimize due to Martens and Niehren [MN07] shown in Algorithm 13 minimizes an XSD D by unifying equivalent types in D. Here s is said to be *equivalent* to t if $\mathcal{F}(D, s) = \mathcal{F}(D, t)$.

Note that Minimize updates D during its execution. Lines 2 and 3 perform the actual unification of s and t by replacing t by s. The condition $t \neq r$ in line 1 ensures that the start type r is never removed. The equivalence condition $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ in that line can be checked as follows.

Algorithm 13 Minimize

```
Input: an XSD $D = (T, \rho, \tau)$ and type $r \in T$
Output: $(D, r)$ with redundant types in $D$ removed
1: while there are distinct types $s$ and $t$ in $T$ with $t \neq r$ and $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ do
2:     replace each $(s', a) \mapsto t$ in $\tau$ by $(s', a) \mapsto s$
3:     remove $t \mapsto \rho(t)$ from $\rho$ and $t$ from $T$
```

Definition 4.7. For an XSD $D = (T, \rho, \tau)$, let $\text{elems}_D(t)$ denote the set of all element names a for which $\tau(t, a)$ is defined. The set $\text{reach}_D(s, t)$ of *pairs of types jointly reachable from* (s, t) is the least set containing (s, t) such that $(s', t') \in \text{reach}_D(s, t)$ and $a \in \text{elems}_D(s') \cap \text{elems}_D(t')$ implies that $(\tau(s', a), \tau(t', a)) \in \text{reach}_D(s, t)$.

Clearly, $\text{reach}_D(s, t)$ can be computed by a standard fixpoint algorithm. Intuitively, $\text{reach}_D(s, t)$ is the set of all pairs (s', t') for which there exists a path p such that $\tau(s, p) \to s'$ and $\tau(t, p) \to t'$.

By the following proposition we can check that $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ by computing $\text{reach}_D(s, t)$ and verifying that $\rho(s')$ and $\rho(t')$ match the same strings for every pair $(s', t') \in \text{reach}_D(s, t)$. The latter can be done in linear time for SOREs by converting $\rho(s')$ and $\rho(t')$ to deterministic finite automata (e.g. by the Glushkov construction [BK93]) and subsequently checking equivalence.

Proposition 4.8. *Let $D = (T, \rho, \tau)$ be an XSD and let s and t be two types in D. Then $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ if, and only if, $\mathcal{L}(\rho(s')) = \mathcal{L}(\rho(t'))$ for all $(s', t') \in \text{reach}_D(s, t)$.*

4.3 Practical heuristics

Unfortunately, when *i*Local is run on an incomplete corpus, it will rarely happen that for distinct inferred types $p|_k$ and $q|_k$ that actually represent the same type in the target XSD we have $\mathcal{F}(D, p|_k) = \mathcal{F}(D, q|_k)$. For instance, if $k = 2$ and $\mathcal{C}$ consists solely of the XML fragment in Figure 4.1, then

$$2\text{-strings}(\mathcal{C}, \texttt{order customer}) = \{\texttt{name email}, \texttt{name email email}\},$$

while $2\text{-strings}(\mathcal{C}, \texttt{item supplier}) = \{\texttt{name email}\}$. Hence, although `order customer` and `item supplier` both represent the type *person* in the target XSD of Figure 4.2, we will infer

$$\rho(\texttt{order customer}) = \texttt{name}, \texttt{email},$$
$$\rho(\texttt{item supplier}) = \texttt{name}, \texttt{email}^{+}.$$

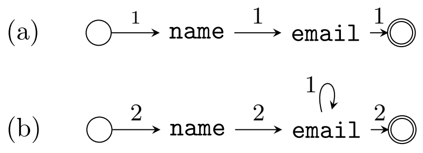


Figure 4.7: Inference of SOAs with support. SOA (a) is the result of 2T-INF({`name email`}). SOA (b) is the result of 2T-INF({`name email`, `name email email`}).

Since `order customer` and `item supplier` are hence not equivalent, MINIMIZE will fail to unify them.

This illustrates that on incomplete corpora, iLOCAL risks identifying more types than that are present in the target XSD. In practice, therefore, we need a minimization algorithm that not only unifies equivalent types, but also unifies 'similar' types. Our goal in this section is to present such an algorithm, called REDUCE. Intuitively, REDUCE measures the similarity of two types s and t in an inferred XSD $(D, \lambda) = i\text{LOCAL}(k, \mathcal{C})$ based on the SOAs learned for s and t. For all s and t that are similar enough, REDUCE subsequently adapts D such that $\mathcal{F}(D, s)$ and $\mathcal{F}(D, t)$ become equal. This adaption can be seen as generalizing the content models of s and t to compensate for missing data. Finally, the hence modified XSD D is minimized. Clearly, since all similar s and t have already been made equivalent, this causes all similar s and t to be unified.

Similarity To define the notion of 'similarity' for types in an inferred XSD $(D, \lambda) = i\text{LOCAL}(k, \mathcal{C})$ we first adapt 2T-INF such that for each edge (a, b) of the automaton A learned for a sample S we also keep the *support* $\text{supp}_A(a, b)$ of (a, b). This is the number of strings in S for which (a, b) needed to be added to the edges of A. Figure 4.7 gives an example.

Next, we adapt iLOCAL such that for each inferred type s we also keep the SOA $\text{soa}(s)$ learned for s. That is, for $s = p|_k$,

$$\text{soa}(s) := 2\text{T-INF}(k\text{-strings}(\mathcal{C}, p|_k)).$$

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

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

$$\begin{tikzcd}
\bigcirc \arrow[r, "3"] & \texttt{name} \arrow[r, "3"] & \texttt{email} \arrow[r, "3"] \arrow[loop above, "1"] & \circledcirc
\end{tikzcd}$$

Figure 4.8: Adjunction of the SOA in Figure 4.7(a) with the SOA in Figure 4.7(b).

Intuitively, $\text{dist}(A, B)$ measures the *dissimilarity* of A and B by counting the number of edges present in A but not in B and the number of edges in B but not in A, weighted by the support these edges have in the original sample. For instance, for A the SOA in Figure 4.7(a) and B the SOA in Figure 4.7(b) we have $\text{dist}(A, B) = 0 + \frac{1}{7} = \frac{1}{7}$. The smaller the value of $\text{dist}(A, B)$, the more similar A and B are. In particular, $\mathcal{L}(A) = \mathcal{L}(B)$ if $\text{dist}(A, B) = 0$.

The *edit distance* $\text{dist}_D(s, t)$ between the inferred types s and t is then defined as

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

Again, the smaller $\text{dist}_D(s, t)$ is, the more similar s and t are. In particular, $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ when $\text{dist}_D(s, t) = 0$, as $\mathcal{L}(\rho(s')) = \mathcal{L}(\rho(t'))$ for all $(s', t') \in \text{reach}_D(s, t)$ in that case.

The algorithm Reduce then operates as shown in Algorithm 14: it merges types whose edit distance is less than some threshold parameter γ. Lines 4–10 are responsible for the actual merging of the selected types s and t, and ensure that $\mathcal{F}(D, s)$ becomes equal to $\mathcal{F}(D, t)$. In particular, the operation $\text{soa}(s') \uplus \text{soa}(t')$ in line 6 stands for the *adjunction* of $\text{soa}(s')$ with $\text{soa}(t')$. This the SOA we obtain by adding to $\text{soa}(s')$ all states and edges in $\text{soa}(t')$ that are not in $\text{soa}(s')$ and setting

$$\text{supp}_{\text{soa}(s') \uplus \text{soa}(t')}(a, b) := \text{supp}_{\text{soa}(s')}(a, b) + \text{supp}_{\text{soa}(t')}(a, b),$$

(where for simplicity we assume that $\text{supp}_{\text{soa}(s')}(a, b) = 0$ if (a, b) is not an edge in $\text{soa}(s')$, and similarly for $\text{supp}_{\text{soa}(t')}$). For instance, Figure 4.8 shows the adjunction of the SOAs in Figure 4.7(a) and Figure 4.7(b). Lines 13–14 converts the updated SOAs into SOREs. Finally, line 15 minimizes the resulting XSD.

Algorithm 14 Reduce

Input: an inferred XSD $(D, r) = i\textsc{Local}(k, \mathcal{C})$ for some k and $\mathcal{C}$, and a similarity threshold γ

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

```
 1: let $(T, \rho, \tau) = D$
 2: initialize $M := \{(s,t) \in T^2 \mid 0 < \mathrm{dist}_D(s,t) < \varepsilon\}$
 3: **while** $M$ is non-empty **do**
 4:   **for** each $(s,t) \in M$ **do**
 5:     **for** each $(s',t') \in \mathrm{reach}_D(s,t)$ **do**
 6:       set $\mathrm{soa}(s') := \mathrm{soa}(s') \uplus \mathrm{soa}(t')$
 7:       set $\mathrm{soa}(t') := \mathrm{soa}(s')$
 8:       **for** each $a$ in $\mathrm{elems}_D(t') - \mathrm{elems}_D(s')$ **do**
 9:         add $(s',a) \mapsto \tau(t',a)$ to $\tau$
10:       **for** each $a$ in $\mathrm{elems}_D(s') - \mathrm{elems}_D(t')$ **do**
11:         add $(t',a) \mapsto \tau(s',a)$ to $\tau$
12:   recompute $M := \{(s,t) \in T^2 \mid 0 < \mathrm{dist}_D(s,t) < \varepsilon\}$
13: **for** each type $t$ in $T$ **do**
14:   replace each $t \mapsto \rho(t)$ in $\rho$ by $t \mapsto \textsc{rwr}^2_\ell(\mathrm{soa}(t))$
15: $\textsc{Minimize}(D, r)$
```

4.4 Experimental evaluation

In this section, we validate our approach by means of an experimental analysis. Since we used RWR^0 rather than RWR^2_ℓ, we denote the algorithm $i\textsc{Local}$ where the latter is substituted for the former with $i\textsc{Local}^0$. Let iXSD be the composition of $i\textsc{Local}^0$ and $\textsc{Reduce}$, i.e., let

$$i\text{XSD}(k, \mathcal{C}) := \textsc{Reduce}(i\textsc{Local}^0(k, \mathcal{C})).$$

We first asses iXSD's precision in Section 4.4.2 by comparing inferred XSDs with their corresponding target XSDs. We next asses iXSD's sensitivity to its parameters (the context size k and the similarity threshold γ) in Section 4.4.3. We subsequently assess iXSD's capacity to generalize on corpora with only a limited amount of data in Section 4.4.4. We conclude in Section 4.4.5 with a short discussion of the runtime performance. Let us begin with discussing the corpora used in our experiments and their corresponding target XSDs.

4.4.1 The test corpora and their target XSDs

The first corpus we consider is $\mathcal{C}_{\text{XSD}}$, which itself consists purely of XSDs in XML syntax. Hence when run on this corpus, iXSD will attempt to infer the XSD for XML Schema Definition documents as it is defined in the W3C specification [TBMM01]. This corpus is discussed in Section 6.2.

An analysis reveals that XSD for XML Schema Definition is 2-local, and is therefore a suitable target schema. The elements `attributeGroup`, `group`, and `extension` occur with different content models in two distinct contexts, `restriction` in three. The XSD for XML Schema Definitions contains a few more context dependent type definitions, but those differ only with respect to attributes, which we do not take into account here. All in all, this leaves us with 48 type definitions to infer. The total number of vertices in the corresponding SOAs is 202, while the number of edges totals 1024.

To gauge the precision of iXSD on XML documents that are described by a DTD, we reuse the real-world corpora mentioned in our previous work on DTD inference in Chapter 2.

In the context of our work on DTD inference in Chapter 2, we have already mentioned that very few corpora of XML documents exist with an interesting schema. In the present setting, this problem is aggravated by the fact that one requires an schema with at least some type definitions that depend on the context. Apart from $\mathcal{C}_{\text{XSD}}$, no suitable real-world corpus could be obtained. Hence we resorted to synthetic XSDs and XML corpora for additional experiments. We hand crafted 8 target XSDs to exhibit a specific set of features to test. All XSDs are recursive and hence define tree languages of unbounded

$$\begin{array}{lll}
\mathit{root} & \rightarrow & \mathtt{a}[a_1] \\
a_1 & \rightarrow & ((\mathtt{b}[b_1], \mathtt{c}[c]) + (\mathtt{d}[d], \mathtt{e}[e])), \mathtt{f}[f]^* \\
a_2 & \rightarrow & ((\mathtt{b}[b_2], \mathtt{d}[d]) + (\mathtt{e}[e], \mathtt{c}[c])), \mathtt{f}[f]? \\
b_1 & \rightarrow & (\mathtt{c}[c] + \mathtt{d}[d])^+ \\
b_2 & \rightarrow & \mathtt{c}[c], \mathtt{d}[d]? \\
c & \rightarrow & \varepsilon \\
d & \rightarrow & \mathtt{b}[b_2] \\
e & \rightarrow & \mathtt{b}[b_1] \\
f & \rightarrow & \mathtt{a}[a_2]
\end{array}$$

Figure 4.9: A hand crafted 2-local XSD.

depth while most have at least one content model that contains strings of unbounded length so that those tree languages have unbounded width.

In this set of hand crafted XSDs, many of the features that can make inference hard are present. In particular, all of these XSDs were constructed such that the content models of types in different contexts have the same alphabet as for the types b_1 and b_2 in Figure 4.9. Note that this particular XSD is 2-local. Our hand crafted set of XSDs, however, also contains XSDs that are 3-local but not 2-local. Moreover, one of the XSDs is 1-local, i.e., a DTD. Given that these XSDs had to be crafted by hand, they contain between 12 and 23 types. Multiple types are associated with at least two elements, while one grammar associates multiple types with six elements. For each of the XSDs, a corpus of 200 XML documents was generated using the software described in Section A.5. These corpora will be denoted by $\mathcal{C}_i$ for $1 \leq i \leq 8$.

4.4.2 Precision

In order to assess iXSD's precision we will compare, for each of the corpora $\mathcal{C}_{\text{XSD}}, \mathcal{C}_1, \ldots, \mathcal{C}_8$ described in Section 4.4.1, the inferred types with their corresponding types in the respective target XSDs. Here, "corresponding types" is defined as follows. Let (D_1, r_1) and (D_2, r_2) be the inferred and target XSD, respectively, and let $D_1 = (T_1, \rho_1, \tau_1)$ and $D_2 = (T_2, \rho_2, \tau_2)$. Then clearly, type r_1 in D_1 corresponds to type r_2 in D_2, as all fragments f valid w.r.t. r_1 in D_1 should also be valid w.r.t. r_2 in D_2 and vice versa. Now, we know that for each such particular $f = \texttt{<}a_1\texttt{>}f_1\texttt{</}a_1\texttt{>} \ldots \texttt{<}a_n\texttt{>}f_n\texttt{</}a_n\texttt{>}$, each f_i should be valid with respect to $\tau_1(r_1, a_i)$ in D_1, and similarly should be valid with regard to $\tau_2(r_2, a_i)$ in D_2. Hence, in this sense, $\tau_1(r_1, a)$ corresponds to $\tau_2(r_2, a)$. Extending this reasoning further, we say that a type t_1 in D_1 corresponds to a type t_2 in D_2 if there exists a path p such that $\tau_1(r_1, p) \rightarrow t_1$ and $\tau_2(r_2, p) \rightarrow t_2$. We remark that it is straightforward to compute all pairs of corresponding types by inspecting τ_1 and τ_2.

Now observe that there are four ways in which iXSD may be imprecise.

1. There can be a type t in the target XSD that corresponds to no type in the inferred XSD. This happens only if the corpus does not contain fragments in which type t is used.

2. There can be a type t in the target XSD that corresponds to multiple types $s_1, \ldots, s_n$ in the inferred XSD. This happens when there are multiple distinct paths $p_1, \ldots, p_n$ such that $\tau_1(r_1, p_i) \rightarrow s_i$, whereas $\tau_2(r_2, p_i) \rightarrow t$ for all i. In this case, iXSD has failed to recognize that $s_1, \ldots, s_n$ actually represent the same type t and we call $s_1, \ldots, s_n$ *false positives* of t.

3. Conversely, there can be a type s in the inferred XSD that corresponds to multiple types $t_1, \ldots, t_m$ in the target XSD. This happens when there are distinct paths $p_1, \ldots, p_n$ such that $\tau_1(r_1, p_i) \rightarrow s$ for all i, whereas $\tau_2(r_2, p_i) \rightarrow t_i$. In this case, iXSD has falsely merged the types $t_1, \ldots, t_m$ into s, and we call s a *false negative* of $t_1, \ldots, t_n$.

4. Finally, the content models inferred for corresponding types s and t may differ.

No algorithm can hope to avoid imprecision (1) based solely on positive examples. We will therefore not consider this imprecision further.

False positives/negatives As far as imprecisions (2) and (3) are concerned, iXSD produced neither false positives nor false negatives when run on $\mathcal{C}_{\text{XSD}}$ with $k = 2$. Hence the algorithm was successful in identify all and only the types in the target XSD for XML Schema Definitions. This is quite remarkable as we will detail in the next paragraph the high level of incompleteness of $\mathcal{C}_{\text{XSD}}$. The case $k = 3$ will be discussed in Section 4.4.3. The XSD inferred by iLOCAL alone contains 29 false positives, clearly illustrating the necessity and indeed the power of REDUCE.

On the synthetic corpora $\mathcal{C}_1, \ldots, \mathcal{C}_8$, iXSD performs excellently, reproducing the target XSD in each case.

Comparison of content models In order to asses imprecision (4) we first note that, although the XSD corpus $\mathcal{C}_{\text{XSD}}$ is fairly large, it nevertheless is not exhaustive. In particular, across the content models r of all target types in the XSD for XML Schema Definitions actually used in $\mathcal{C}_{\text{XSD}}$, there are 511 edges (a, b) in the SOA corresponding to r for which no string in $\mathcal{C}_{\text{XSD}}$ would actually add the edge (a, b) when learning r by 2T-INF. This is a large amount compared to 1024, the total number of edges.

To obtain a fair assessment, we therefore first compute the *adapted content model* r'_t for each target type t in the XSD for XML Schema Definitions. This is the SOA we obtain by transforming the content model r of t into a SOA, and by subsequently removing all edges for which there is no subfragment of type t in $\mathcal{C}_{\mathrm{XSD}}$ that would actually add the edge (a, b). The similarity between the inferred content model and r should be better than between r and r'_t. Here, *similarity* is simply the number of edges that are present in the first SOA but not in the second plus the number of edges present in the second but not in the first (not taking supports into account).

The experiment for $k = 2$ then shows that the content models of the derived XSD are at least as good (38 out of 47) or better than the baseline (9 out of 47), in five cases by more than 10 %. The fact that iXSD exceeds the baseline expectations is due to the combined effect of REDUCE's smoothing and RWR^0's generalization capacity.

4.4.3 Sensitivity to parameters

An important consideration is the sensitivity of the algorithm with respect to the choice of the parameters, the context size k and the similarity threshold γ.

A low number of false positives implies good generalization, while a low number of false negatives is a mark of precision. Ideally, neither should occur, but it is obvious that trying to minimize the number of false positives will cause an increase of the number of false negatives and vice versa. The parameters fine tune iXSD's performance. For increasing context size k, the number of false positives will increase, the same effect occurs for decreasing similarity threshold γ.

If the target schema of the corpus is a DTD, iXSD produces no false positives for $k = 2$. This is the case for the real-world corpora and as well as the synthetic corpus mentioned in Section 4.4.1.

For context size $k = 2$, the XSD derived from $\mathcal{C}_{\mathrm{XSD}}$ has neither false positives, nor false negatives, which confirms the quality of the algorithm. For $k = 3$, 11 false positives crop up in the derived XSD. For example, as illustrated in Figure 4.10 for `restriction`, we have three types for $k = 2$, which are subsequently refined into false positives for $k = 3$. We note, however, that 2 of them can be identified as such since they are caused by 0.5 % or less of the examples that make up the corresponding type for $k = 2$ and hence are very unlikely. We consider this a good rule of thumb for the identification of false positives when the target schema is not known. So, iXSD can be run for increasing values of k until too large discrepancies are encountered.

The sensitivity of the algorithm with respect to the similarity threshold

$$\begin{tikzcd}
k=1 & & & & 1.000 \arrow[dll] \arrow[d] \arrow[dr] & \\
k=2 & & 0.971 \arrow[dl] \arrow[d] \arrow[dr] & & 0.013 \arrow[d] & 0.016 \arrow[d] \\
k=3 & 0.969 & 0.001 & 0.002 & 0.013 & 0.016
\end{tikzcd}$$

Figure 4.10: Distribution of the number of examples over types as a function of k for XSD's `restriction` element.

γ is illustrated in Table 4.1 which shows the number of false positives and false negatives as a function of γ. It is clear that the algorithm is not overly sensitive to its value: the target XSD is derived whenever $0.05 \leq \gamma \leq 0.15$. The results are similar for each of the synthetic corpora $\mathcal{C}_i$.

γ	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 4.1: Sensitivity of the algorithm with respect to the similarity threshold γ for an 3-local XSD.

4.4.4 Generalization

In order to assess *i*XSD's robustness with respect to missing data, we split one of the corpora $\mathcal{C}_i$ into two parts: the first is used to derive an XSD, the second to validate that XSD. The *generalization* is the number of XML documents in the validation set that is valid with respect to the inferred XSD. In Figure 4.11 we show the generalization as a function of the training set size. Here the corpora for two of the synthetic XSDs were used with $k = 2$, the second of which (denoted by $\times$'s in Figure 4.11) has a content model containing a term of the form $(a_1 + \cdots + a_{12})^+$ which is quite hard to derive. It is clear from the plot that the algorithm performs well for even a relatively small number of XML documents as training set, 50 and 200 in this case.

4.4.5 Runtime performance

Although the Java code is in a prototype stage and hence not optimized for speed, the algorithm runs quite fast. The process of parsing the 697 XSDs

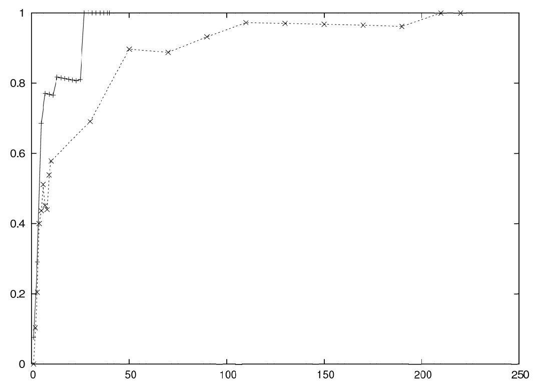


Figure 4.11: Generalization as function of the corpus size for two XSDs, one (×) with a large fan out.

of the W3C XML Schema Document corpus, a total of over 40 Mb, and the derivation of its XSD for $k = 2$ takes less than 15 seconds on an off the shelf laptop with a 1.73 GHz Pentium-M processor. Deriving a $k = 3$ XSD from the same set of data takes 17 seconds. Given that the number of distinct ancestor strings is 136 for $k = 2$ and 299 for $k = 3$, the algorithm's scales well with the complexity of the target XSD. Even non-optimized, the algorithm can be comfortably used for real-world applications.

4.5 Conclusions

We introduced two novel algorithms for the inference of concise XSDs. iLOCAL is theoretically complete in the sense that it derives any target local SOXSD given enough data. A second algorithm, iXSD , is the algorithm iLOCAL followed by a smoothing of the obtained XSD through REDUCE to compensate for the lack of data. We have shown that iXSD performance is excellent on both real-world and synthetic data. One of the main open issues in our framework is how to determine the best value of k. Although, we provide a rule of thumb which gave optimal results in our experiments, it would be worthwhile to look into machine learning techniques for parameter estimation. In future work, we want to extend our algorithms to larger classes of XSDs.

5 SchemaScope

The SchemaScope application was developed to address the issues mentioned in Chapter 1. SchemaScope supports (1) the automatic inference of Document Type Definitions (DTDs) and XML Schema Definitions (XSDs) from corpora of sample XML documents and (2) tools to visualize, clean, and refine existing or inferred schemas. It can be used to assist schema developers in three scenarios: (1) schema inference, (2) schema cleaning, and (3) schema refinement.

Schema inference The automatic inference of schemas from a corpus of XML documents is particularly useful in those situations when no existing schema is available. Although several command-line tools are available for this task [BNST06, BNV07, Cla03, GGR^{+}03] the quality of the inferred schema is always heavily dependent on the quality of the XML corpus: when the corpus does not completely cover the intended schema the inferred schema may be too specific; when the corpus contains errors the inferred schema may be too general. For this reason, SchemaScope not only allows automatic inference of DTDs and XSDs (using the algorithms presented in Chapters 2 and 4), but also provides appropriate visualization tools to allow a human expert to fine-tune the inferred schema based on the actual corpus. Furthermore, as more sample documents become available, the schema can be evolved to capture the corpus more precisely.

Schema cleaning A related, but distinct setting is the one where one has a corpus of XML documents, as well as an existing schema that is supposed to describe it, but for which some documents fail to validate. In order to use the schema to aid in further efficient processing or querying of the corpus, it is then desirable to clean the schema based on the corpus. SchemaScope supports such cleaning by allowing users to interactively relax the content models of the original schema, at each step showing the parts of the corpus that become valid during this relaxation, and measuring how many fragments in the corpus actually necessitate the relaxation. The latter helps in rejecting those schema changes which are due to errors in the corpus.

Schema refinement A final setting is where one has a corpus of XML documents, as well as an existing schema that describes the corpus, but where the schema is too general in the sense that some parts of its content models are never realized in a document. Since schemas that better describe the true structure of the data provide more information to be exploited with regard to storage and query optimization, it is in this case preferable to refine the existing schema. SchemaScope supports such refinement by visualizing the support each content model part has in the corpus, and by indicating those XML document fragments that become invalid when the user modifies the schema.

5.1 System components

SchemaScope consists of a number of interacting modules that provide the required functionality. A schematic overview of the application's modules and their interactions is shown in Figure 5.1.

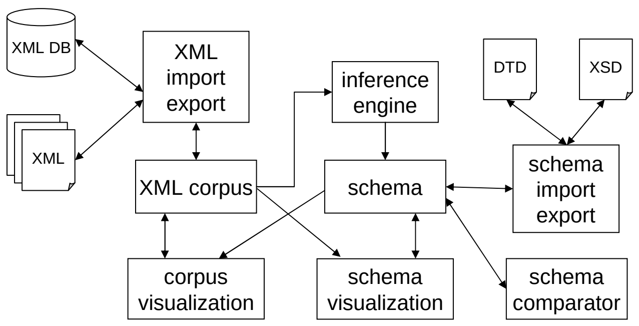


Figure 5.1: Schematic overview of the SchemaScope system's components. Figures 5.2 and 5.3 show details of the schema visualization component.

Import and export XML documents or fragments of such documents can be imported (depicted by "XML import" in Figure 5.1) from a range of data sources including files, URLs, and XML database query results. XML schemas in DTD or XSD syntax can be imported ("schema import" in Figure 5.1) and are converted to an internal representation, simply denoted by the term "schema". A derived or modified schema can be saved as a DTD or XSD (in the former case with potential loss of precision).

Schema inference engine The inference module generates a DTD or XSD from the imported XML corpus using the algorithms presented in previous work [BNST06, BNV07]. Although these algorithms cannot infer every possible target DTD or XSD (a classical result of Gold [Gol67] states that no such algorithm exists), they can infer those subclasses of DTDs and XSDs that are used in practice [BNST06, BNV07]. Furthermore, the inferred content models are always concise and human-readable.

The user actually has a choice between several algorithms for the inference of complex type content models. Some algorithms work well when only a small number of sample documents are available, while others yield better content models, but require more data. Simple type content models (like int, base64, string, ...) and types of attributes are inferred based on a number of heuristics.

Schema visualization Imported or inferred schemas can be visualized by the schema representation module. A textual, outline, or graph view (cf. Figure 5.2) is provided to inspect the ancestor relationship between the elements in the schema, while content models can be viewed as text, e.g.,

```
annotation?  (attribute | attributeGroup)* anyAttribute?
```

or as hierarchical graphs (cf. Figure 5.3). All views are annotated and color coded with frequency information calculated from the imported XML corpus. Parts of content models that are realized by only few members of the corpus are brightly rendered, so as to call attention to opportunities for schema cleaning or refinement. Using these views the schema can be edited, while the impact of the changes on the frequency information can be used as feedback and guidance. An additional view provides a number of metrics of the schema [Cho02, BNV04, MSbY04].

XML data visualization Individual imported documents or fragments can be viewed using the corpus visualization module. More importantly, the corpus as a whole can be visualized in a list layout based on its properties with respect

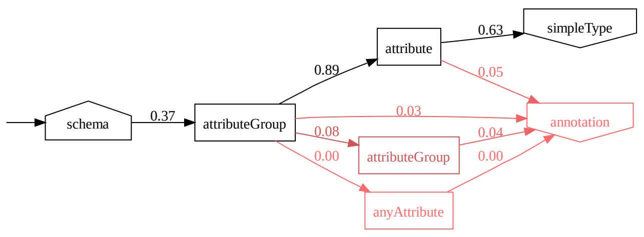


Figure 5.2: Schema view centered on the XML Schema for XSD's `attributeGroup` element, showing its parent (`schema`) and children (`annotation`, `attribute`, `attributeGroup`, and `anyAttribute`). Rectangular elements are fully expanded, upward pointing elements are partially expanded, and downward pointing elements are not expanded. Numbers and colors indicate the elements' support in the corpus.

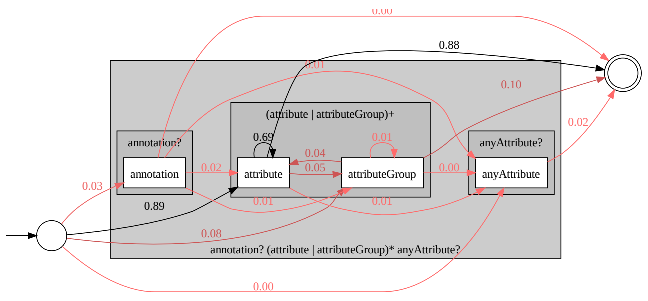


Figure 5.3: Content model view of the XML Schema for XSD's `attributeGroup` element, shown here as a fully expanded deterministic finite automaton. As in Figure 5.2, numbers and colors show the support for the syntactic structure of the model.

to the current schema. One can view the data that realizes a particular content model part in the schema or one can view and inspect those XML documents or fragments that are no longer valid after the user has modified the schema. The XML documents can also be ranked in the list according to one of several measures that quantify the degree of conformance to the schema. In this view a cut-off can be set so that one obtains subsets of the corpus. Those can subsequently be used to attempt to infer more appropriate schema for the individual subsets. An additional view provides metrics and statistics of the corpus.

Schema comparator During schema refinement or correction, it is important to be able to compare the obtained schema with its previous versions. This module allows semantic containment tests of individual content models and even whole schemas. An example is given in Figure 5.4 where two content models are compared; differences are shown using colors.

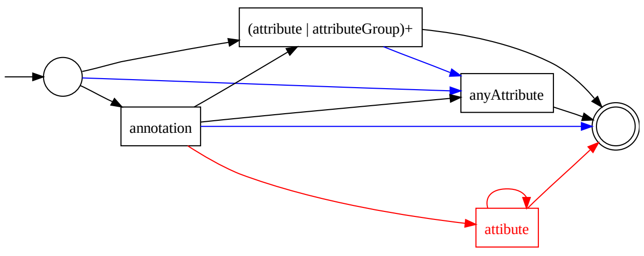


Figure 5.4: Schema comparator focused on two content models for `attributeGroup`, syntactic structures present in both models are shown in black, those only present in either one in red or blue.

XML generation Finally, we come full circle with the XML generator module that can synthesize a collection of XML documents valid to the specified schema. Such a corpus can be used, e.g., to test external applications that use the schema. The generation process is parametrized in order to allow fine-tuning to many requirements. It should be noted that the generator's expressive power complements that of ToXgene, the current state of the art [BM06]. Whereas ToXgene strictly respects the metrics specified by the user, thus potentially generating invalid documents, our implementation yields documents that are guaranteed to be valid. This component is described in more detail in Section A.5.

5.2 Demo overview

SchemaScope has been presented as a demonstration at SIGMOD 2008, and the scenario we used is presented in this Section. We focused on a number of real-world use cases that serve as motivating examples for the features implemented in SchemaScope and illustrate its versatility.

Schema inference For the purpose of demonstrating the automatic inference of schemas, we use a corpus of XSD documents; infer a schema from it; and compare this inferred schema with the actual XSD for XML Schema Definitions [TBMM01]. Aided by the various schema visualization views, we refine the inferred schema based on the semantics implied by the corpus' ontology and frequency information. Figure 5.5, for instance, shows the content model for the `attributeGroup` element inferred from incomplete and noisy data. Using this view of the content model, it is immediately clear to a human expert that the element `attribute` (versus `attribute`) is due to noise. Finally, we save the resulting schema as an XSD using the schema export component to illustrate that the result is concise and human readable.

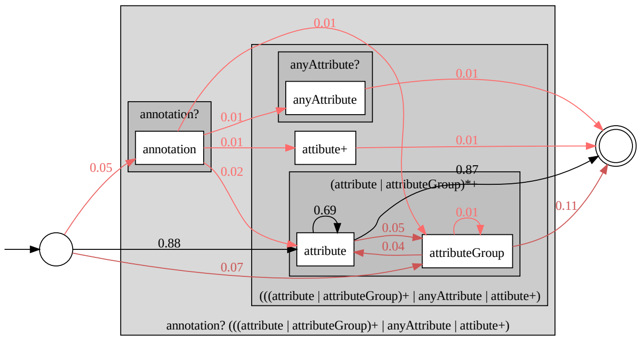

annotation? (((attribute | attributeGroup)+ | anyAttribute | attibute+)
(((attribute | attributeGroup)+ | anyAttribute | attibute+)
annotation?
annotation
anyAttribute?
anyAttribute
attibute+
(attribute | attributeGroup)*+
attribute
attributeGroup
0.01
0.01
0.01
0.01
0.01
0.02
0.05
0.87
0.69
0.05
0.04
0.01
0.11
0.88
0.07

Figure 5.5: Content model for `attributeGroup` derived from incomplete and noisy data.

Schema cleaning To demonstrate schema cleaning, we have harvested a corpus of real-world XHTML documents from the World Wide Web (all on a specific topic) that—although well-formed—are not all valid according to the specified XHTML DTD. Starting from this real-world corpus and the W3C XHTML DTD specification, we derive a more relaxed DTD that validates a larger fraction of the corpus, while still rejecting those documents that deviate too much from the specification. Crucial in this relaxation is the XML data visualization component that allows us to determine the balance between the number of valid documents and the precision of the schema. Furthermore, the XML schema visualization component (cf. Figure 5.5) helps in identifying noisy data.

Schema refinement For the purpose of demonstrating schema refinement, we consider Microsoft's WordML document format. WordML specifies the syntax of Microsoft Word documents in XML form. While developing a converter from some arcane and obsolete document format to WordML, we noted that some documents that were valid according to WordML's XSD were nevertheless incorrectly processed by Microsoft Word. The WordML XSD is in fact too general since a number of syntactic constraints are coded in the application's logic, rather than in the documents' specification. In this part of the demonstration, we therefore import a corpus of WordML documents and the XSD provided by Microsoft and gradually refine the schema to capture some of the logic that is imposed by the application, but that is not integrated into the original schema. The schema comparator is used to visualize the differences between the refined and the original content models.

6 Data collection

In the context of learning, data is very important. We will first analyze real-world Document Type Definitions and XML Schema Definitions to determine their characteristics. It will turn out that these schemas' properties can guide us to design effective learning algorithms. A description of how the real-world corpus was obtained and the results of the analysis are presented in Section 6.1.

Given that the learning process is based on XML document instances, one needs example documents for relevant schemas. For schemas such as those describing W3C's XHTML and XML Schema Definition, Microsoft's WordML and other important standards XML documents are easy to come by. However, it is fairly difficult or even impossible to find XML documents that are instances of less popular schemas that are nevertheless interesting from a technical point of view. Details on our XML document corpus are given in Section 6.2.

In this chapter, we use the convention that if we refer to the W3C standard for Document Type Definitions and XML Schema Definitions, we will use the term in full, while we reserve the use of the abbreviations DTD and XSD for specific instances.

To make this chapter more self-contained, we list the relevant definitions here, although they have already been introduced in previous chapters.

6.1 Schema corpus

6.1.1 Sources

Our research is driven by a real-world problem, i.e., deriving schemas from XML data, so the corpus should contain a sizable number of DTDs and XSDs as used in applications. We gathered schemas from various sources.

Standards Various high-quality schemas representing web standards such as those for RDF, SMIL, SVG, XHTML, XML Schema Definition were obtained from the W3C. Other standards, such as the schemas for office application file formats from Microsoft Office and OpenOffice were included as well.

Cover Pages The Cover Pages website [Cov03] is a valuable resource that contains a large collection of schemas developed for a wide range of applications such as, e.g., telecommunications, financial transactions and bioinformatics. The Cover Pages list approximately 600 DTDs and XSDs, but of those, only 109 DTDs and 93 XSDs could actually be retrieved. The latter illustrates the phenomenon of “link rot” that contributes substantially to the transient nature of the web [Koe02, Koe03]. The schemas were retrieved using a crawler.

Web To widen the scope further, DTDs and XSDs were obtained from the web at large using the results of the Google and Yahoo! web search engines. The engines were accessed using the respective APIs offered by the respective companies for that purpose. Both engines obviously support keyword queries, but also allow the specification of file types to restrict the search. To wrap this functionality and facilitate the extension to other web search engines, “Web-HunterGatherer”, a lightweight Java framework was developed that allows to first retrieve query results from one or more search engines, and subsequently download the documents using the provided URLs (see Appendix A.1). It is obvious that the quality of this sample falls short of that of the other sources. Whereas the schemas representing various standards have been developed by experts in the field, and most of those mentioned on the Cover Pages have at least been used in applications, no such guarantees exist for schemas retrieved from the web. Indeed, it turns out that approximately two out of every three XSDs is either not well formed, invalid with respect to the schema definition or contains errors according to IBM’s SQC [IBM03]. A similar observation was made by Sahuguet in his study of DTDs [Sah00].

All in all, we obtained 109, respectively 225 valid and correct DTDs and XSDs. The total number of XSDs is 819. The corpus has been available upon request since 2004 and has been provided to several researchers.

Synthetic schemas For a number of applications, a small number of XSDs has been crafted by hand since specific quantitative features were required. More importantly however, on a number of occasions regular expressions with specific properties were needed that are hard to find in the real-world content models. Software was developed to generate regular expressions of the types described in Section 6.1.3. This functionality is part of the Formal Languages Toolkit (FLT) that forms the foundation of most of our software and that is described in more detail in Appendix A.4.

6.1.2 XML Schema Definition's features quantified

A number of authors [LC00, Jel01, DuC02] discuss various drawbacks of Document Type Definitions, such as, among others,

- very few datatypes,
- no user defined types,
- almost no value constraints, only for attributes,
- no derived types
- no support for unordered sequences.
- lack of support for namespaces,
- very limited import facilities,
- very limited support for keys
- DTDs are not formulated in XML,

Although the Document Type Definition specification [BPSM$^+$06] offers limited support for a number of the desired features, the W3C's XML Schema Definition [TBMM01, BM01] has been developed to alleviate these shortcomings. Several other proposals have been formulated with various degrees of success [Cla01, CM01, VAG03, KMS02, Sch06], however, here we will limit ourselves to a discussion of XML Schema Definition, while a comparison of the feature set of six schema definition specifications was given by Lee and Chu [LC00]. Below a number of features are introduced only briefly since we merely wanted to determine how frequently they are used in real-world schemas. Given the complexity of some, a detailed explanation is beyond the scope of this chapter. Several sources [Fal01, vdV02] give a good introduction.

The goal of our analysis is to quantify the degree to which each of these features has percolated into the design of real-world schemas. Figures such as these may help guide development of schema related software, but also steer extension to existing schema languages or even proposals for new languages. The results on single type schemas have influenced the design of our algorithm for schema inference.

Modularity The Document Type Definition specification provides an import mechanism for element, attribute and entity definitions through the external entity definition. Although this allows for a limited form of modularization, reuse is hampered nevertheless. Element names have to be unique throughout the DTD and all its modules. The XML Schema Definition specification introduced the concept of namespaces for XML definitions, allowing for modular design of schemas. Definitions can be imported explicitly, adding to XML Schema Definition's modularity.

Basic type definitions Basic types in Document Type Definitions are limited to the most generic `#PCDATA` for element contents, and a slightly wider range for attributes (e.g., `ID`, `IDREF`, value enumerations, `NMTOKEN`, `CDATA`,...). This implies that one can not restrict the content of an element or attribute to, e.g., an integer or a date. In contrast, XML Schema Definition has an extensive set of basic types ("simple types" in XSD parlance).

Linking Although one can refer to elements in XML documents using Document Type Definition's `ID`/`IDREF` mechanism. An element can have an `ID` attribute that has to be unique in the document it occurs in, and that can be referred to by the `IDREF` or `IDREFS` attribute of other elements in that document. XML Schema Definition extends this notion to arbitrary attributes and elements, or even combinations thereof using the `key`/`keyref` mechanism. Moreover, it is possible to ensure that values of elements and attributes are unique in a specified scope. The selection mechanism for both features is very similar and is based on XPath expressions [CD99].

Interleaving When the content model of an element has to contain some subelements, but their order is irrelevant, specifying this in Document Type Definitions is cumbersome to say the least. One has no alternative but to enumerate all permutations of the subelements in a disjunction to list all orderings. XML Schema Definition reintroduces the interleaving operator (`all`) that is part of SGML Document Type Definitions. The operands of the `all`-operator must all occur, but may do so in any order.

Inheritance Development time and effort is greatly reduced by being able to adapt existing type definitions to varying requirements by extending or restricting the original definition. XML Schema Definition has borrowed this feature—that has no counterpart in Document Type Definition—from the object oriented programming paradigm. XML Schema Definition furthermore supports the notion of abstract types for elements and attributes. Using an abstract type in a content model implies that an instance document should specify the element's type using the `xsi:type` attribute. Hence the `abstract`-attribute enforces the use of a derived type. Conversely, the `block`-attribute ensures that an element of the original type is used in an instance document, rather than a derived type. It is possible to prevent derivation by restriction or extension of a particular type by specifying that it is final. Similarly, on the level of instance documents, one can force the content of an element or attribute to equal a specified value using the `fixed`-attribute. A substitution group is used to indicate that an element occurring in an instance document should conform to any of the types listed in the substitution group's definition.

Redefine XML Schema Definition allows to redefine existing type definitions, however, the W3C primer explicitly advises caution when using this feature.

Single type definitions The most interesting feature that sets XML Schema Definition apart from Document Type Definition is the notion of context of a type definitions. In Document Type Definitions, the type definition of an element is absolute in the sense that it is independent of the context it occurs in. However, in an XML Schema Definition, an element can be of a different type depending on its context. Martens et al. [MNSB06] have shown that the context of an element is uniquely determined by the path from the document root to that element. In contrast, the type of an element in a document described by a Document Type Definition depends only on its name.

Results of the analysis The results are summarized in Table 6.1. For an analysis of syntactic features the schemas need not necessarily be valid, so for all but the single type definition feature the full corpus of 819 XSDs has been used, while for the latter only the 225 correct one were taken into consideration. It is clear that simple types and derivations thereof are quite heavily used. For complex types on the other hand, only one in five schemas employs inheritance.
The majority of XSDs in our corpus are structurally equivalent to a DTD, i.e., the definition of content models is independent of the context they occur in.

However, this is not the case for 15 % of the schema definitions, indicating that this feature is deemed useful by schema developers. The most interesting examples showing this feature are W3C XSD for XML Schema Definition, and Microsoft's XSD for WordML, the file format for the Word text processing application.

feature	**XSDs (%)**
derivation	
simpleType extension	18.9
simpleType restriction	45.5
complexType extension	20.7
complexType restriction	3.6
abstract attribute	9.8
final attribute	0.9
block attribute	0.0
fixed attribute	6.4
substitutionGroup	6.4
redefine	1.0
interleaving	
`xs:all`	5.5
modularity	
namespaces	12.1
import	27.7
linking	
key/keyref	4.1
unique	2.9
structure	
true single type definitions	15.0

Table 6.1: XML Schema features use in the corpus

6.1.3 Regular expressions

Another interesting question we try to answer is how sophisticated regular expressions tend to be in real-world DTDs and XSDs. If simple expressions make up the vast majority of schema definitions, it is worthwhile to take this into account when developing implementations of XML related applications and fine-tune algorithms to take advantage of this simplicity whenever possible.

In the context of DTDs and XSDs, a regular expression over the alphabet Σ can be defined as follows:

Definition 6.1. $\emptyset$ and ε are regular expressions. For each alphabet symbol a is a non-trivial regular expression. If r and s are non-trivial regular expressions, so are $r?$, r^+, r^*, rs and $r + s$. Any non-trivial regular expression is a regular expression.

The regular expression $\emptyset$ represents the empty language, while the expression ε denotes the language that only contains the empty string. The semantics of $r?$, r^+ and r^* are customary: zero or one, one or more and zero or more repetitions of strings accepted by r respectively. The expressions rs and $r + s$ denote the concatenation and disjunction respectively. Although this definition deviates from the textbook formulation, we nevertheless prefer it since it corresponds to what can be expressed in Document Type Definition and XML Schema Definition. Note that XML Schema Definition's interleaving operator has not been represented in the definition since we will not consider it any further.

In order to facilitate the analysis some preprocessing was performed. For the DTDs parsed entities were resolved and conditional sections included/excluded as appropriate. Since we are only concerned with the schema structure, the DTD element definitions were extracted and converted to a canonical form, which abstracts away the actual node labels and replaces them by canonical names c_1, c_2, c_3, ... For example,

```
<!ELEMENT lib ((book | journal )*)>
```

is represented by a canonical form $(c_1 \mid c_2)^*$ to preserve only the information necessary to gauge the expression's complexity. Given the lack of a DTD parser at the time, we developed one that was adequate for our purposes, see Appendix A.2.

The XSDs were preprocessed using XSLT to the canonical representation mentioned above for DTDs. To capture multiplicity constraints, '?' is used, e.g. for an element a with `minOccurs="1", maxOccurs="3", \lstinlinea` (a a?)?— is substituted. This approach allows us to reuse much of the software developed to analyze DTDs for XSDs as well.

For all DTDs, there is a total of 11802 element definitions which reduce to 750 distinct canonical forms. The 1016 element definitions in the XSDs yield 138 distinct canonical forms, totaling 838 for both types of schemas combined. Note that for this analysis the XSDs retrieved from the web were not yet used.

Simple regular expressions

The majority of the regular expressions encountered in real-world DTDs and XSDs can be classified in one of the categories of "simple regular expressions", which are subclasses of the expressions studied by Martens, Neven, and Schwentick [MNS04].

Definition 6.2. 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.

The following is an example of a simple regular expression: $(a^* + b^*)(a + b)?b^*(a + b)^*$.

We introduce a uniform syntax to denote subclasses of simple regular expressions by specifying the allowed factors. We distinguish base symbols extended by ? or $*$. Further, we distinguish between factors with one disjunct or with arbitrarily many disjuncts; the latter is denoted by $(+\cdots)$. Finally, factors can again be extended by $*$ or ?. For example, we write $\mathrm{RE}((+\,a)^*, a?)$ for the set of regular expressions $e_1 \cdots e_n$ where every e_i is $(a_1 + \cdots + a_n)^*$ for some $a_1, \ldots, a_n \in \Sigma$ and $n \geq 1$, or $a?$ for some $a \in \Sigma$. Table 6.2 provides an overview.

factor	**abbr.**		**factor**	**abbr.**
a	a		$(a_1 + \cdots + a_n)^*$	$(+a)^*$
a^*	a^*		$(a_1 + \cdots + a_n)?$	$(+a)?$
$a?$	$a?$		$(a_1^* + \cdots + a_n^*)$	$(+a^*)$
$(a_1 + \cdots + a_n)$	$(+a)$		$(a_1^* + \cdots + a_n^*)^*$	$(+a^*)^*$

Table 6.2: Possible factors in simple regular expressions and how they are denoted $(a, a_1, \ldots, a_n \in \Sigma)$.

We analyze the DTDs and XSDs to characterize their content models according to the subclasses defined above. The result is represented in Table 6.3 that lists the non-overlapping categories of expressions having a significant population (i.e., more than 0.5%). Two prominent differences between DTDs and XSDs immediately catch the eye: XSDs have (1) more `simpleType` elements (denoted by `#PCDATA`) and (2) less expressions in the category $\mathrm{RE}(a, (+\,a)^*)$. The first difference is due to the fact that it pays to introduce more distinct `simpleType` elements in XSD since thanks to type restriction, it is now possible to fine tune the specification of an element's content (cfr. the discussion in Section 6.1.2). The second distinction is most probably due to the nature of the XSDs in the sample since those describing data are overrepresented with respect to those describing meta documents [Cho02]. The latter tend to have more complex recursive structures than the former.

To gauge the quality of our sample of XSDs, we compared DTDs and XSDs using several of the measures proposed by Choi [Cho02]. No significant differences between the two samples are observed, which is confirmed by an additional measure in Figure 6.1, the density of XSDs. The density of a schema is defined as the number of elements occurring in the right hand side of its rules divided by the number of elements. DTDs and XSDs do not fundamentally differ in this respect. Several other measures such as the width and depth of canonical forms viewed as expression trees show no significant differences.

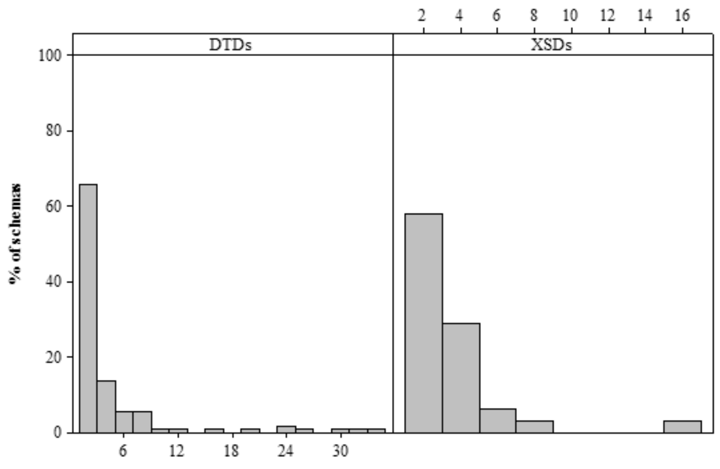


Figure 6.1: Fraction of DTDs (left) and XSDs (right) versus their density

	DTDs (%)	XSDs (%)
`#PCDATA`	34	48
`EMPTY`	16	10
`ANY`	1	0
RE(a)	5	5
RE($a, a?$)	2	10
RE(a, a^*)	8	10
RE($a, a?, a^*$)	1	4
RE($a, (+a)$)	3	3
RE($a, (+a)?$)	0	1
RE($a, (+a)^*$)	20	2
RE($a, (+a)?, (+a)^*$)	0	1
RE($a, (+a^*)^*$)	0	2
total simple expr.	92	97
non-simple expr.	8	3

Table 6.3: Relative occurrence of various types of regular expressions given in % of element definitions

More importantly though, it is clear that the vast majority of expressions are simple regular expressions, i.e., 92% and 97% of all element definitions in DTDs and XSDs respectively. Figure 6.2 shows the fraction of DTDs and XSDs versus the fraction of their simple content models: the majority of documents have 90% or more simple content models.

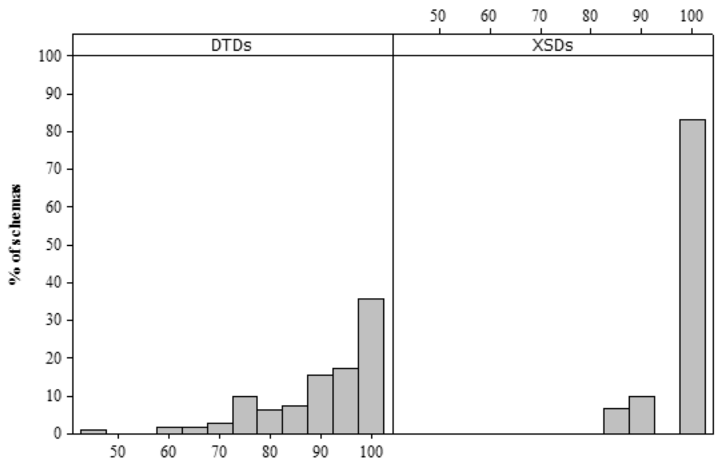


Figure 6.2: Fraction of DTDs (left) and XSDs (right) having a given % of simple expression content models

The relative simplicity of most DTDs and XSDs is further illustrated by the star height that is given in Table 6.4. The star height of a regular expression is the maximum nesting depth of Kleene stars occurring in the expression, e.g. 2 for the last example given below, 1 for all others. Content models with star height larger than 1 are very rare. No significant differences are observed between DTDs and XSDs, except for the star height but this is consistent with the relative abundance of $\mathrm{RE}(a, (+a)^*)$ type of expressions in DTDs with respect to XSDs.

star height	**DTDs (%)**	**XSDs (%)**
0	61	78
1	38	17
2	1	4
3	0	≈ 0

Table 6.4: Star height observed in DTDs and XSDs

In a sense this should not come as too great a surprise: DTDs and XSDs model data that reflect real-world entities. Mostly those entities are subject to simple relations among one another such as `is-a`, or `is-part-of` relations that are very often quite simple to express.

Here we show some randomly chosen examples of non-simple regular expressions that we encountered:

$$\begin{array}{l} c_1{}^+ \mid (c_2?c_3{}^+) \\ (c_1c_2?c_3?)?c_4?(c_5 \mid \ldots \mid c_{18})^* \\ c_1?(c_2c_3?)?(c_4 \mid \ldots \mid c_{44})^*c_{45}{}^+ \\ c_1?c_2c_3?c_4?(c_5{}^+ \mid ((c_6 \mid \ldots \mid c_{61})^+c_5{}^*)) \\ c_1(c_2 \mid c_3)^*(c_4, (c_2 \mid c_3 \mid c_5)^*)^* \end{array}$$

k-occurrence expressions

Regular expressions defining content models in real-world DTDs and XSDs exhibit an additional feature that turns out to be very useful. We define a Single Occurrence Regular Expression (SORE) as follows:

Definition 6.3. A regular expression r over the alphabet Σ is a single occurrence regular expression over Σ iff each alphabet symbol occurs at most once in the regular expression r that is of the form defined in Definition 6.1.

The regular expression $a(b+cd?)^+(ef)?$ is a SORE, while $a(b+cd?)^+(af)?$ that has exactly the same syntactic structure is not, since the symbol a occurs twice.

An analysis of the DTDs and XSDs in our corpus reveals that 98 % of all regular expressions are in fact SOREs, a feature that will turn out to be quite convenient.

More generally, we can define the set of k-occurrence regular expressions.

Definition 6.4. A regular expression r over the alphabet Σ is a k-occurrence regular expression over Σ iff each alphabet symbol occurs at most k times in the regular expression r that is of the form defined in Definition 6.1.

The expression $(a + b^+)c(baa)^+$ is a 3-occurrence regular expression since the alphabet symbol a occurs 3 times. If we denote the set of k-occurrence regular expressions by RE_k, then obviously $RE_1 \subset RE_2 \subset \cdots \subset RE_k$ for any k. In the set of all regular expressions obtained from the real-world DTDs, the number of regular expressions that are proper members of RE_k is given in Table 6.5. As mentioned previously, single occurrence regular expressions make up to vast majority of regular expression, while expressions with k-values over 3 are exceedingly rare, and none are found with $k > 5$.

maximal k	expressions (%)
1	98.3
2	1.25
3	0.32
4	0.03
5	0.01

Table 6.5: Number of regular expressions that are proper k-occurrence regular expressions in real-world DTDs

However, the complexity of a regular expression in terms of the symbols that occur in it is not only determined by the k-value, but also by the number of symbols that occurs multiple times. To this end, we define the quantity κ.

Definition 6.5. Given a regular expression that has alphabet symbols in Σ, the symbol density is defined as the total number of alphabet symbols in the expression, divided by the size of the alphabet $|\Sigma|$.

For example, the symbol density of the 3-occurrence regular expression $(a + b^+)c(baab)^+$ is $\kappa = 7/3 = 2.33$. By definition, a single occurrence regular expression has $\kappa = 1$. In Figure 6.3 the distribution of the κ-values computed for the regular expressions occurring in real-world DTDs is shown. Again, it is clear that sophisticated expression are very rare in the real-world.

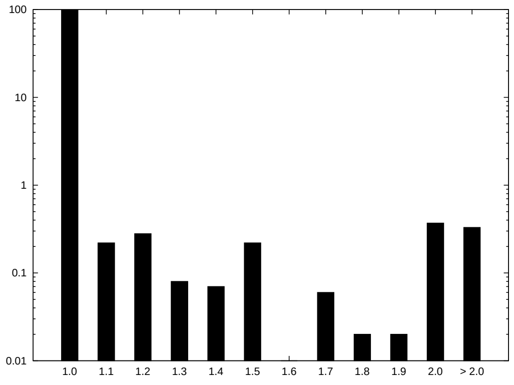


Figure 6.3: number of regular expression (%) versus symbol density κ in a real-world DTD corpus

Another way to characterize k-occurrence regular expressions is by the number of symbols that occur multiple times. The relative symbol multiplicity is defined as follows:

Definition 6.6. Given a regular expression that has alphabet symbols in Σ, the symbol multiplicity is defined as the number of alphabet symbols that occur more than once in the expression, divided by the size of the alphabet $|\Sigma|$.

For single occurrence regular expressions, every symbol occurs once, and hence this measure is zero. For the regular expression $(a + b^+)c(baab)^+$, the symbol multiplicity is $2/3 = 0.67$ since the symbols a and b occur multiple times in the expression. The distribution of the symbol multiple for expression from real-world DTDs is shown in Figure 6.4.

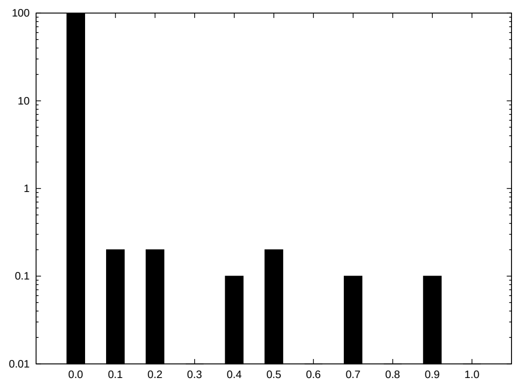


Figure 6.4: number of regular expression (%) versus symbol multiplicity in a real-world DTD corpus

Chain regular expressions

We consider one more subclass of regular expressions that bears a relationship with both simple and single occurrence regular expressions. A CHAin Regular Expression (CHARE) over the alphabet Σ is defined as:

Definition 6.7. $\emptyset$ and ε are chain regular expressions. The expressions f, $f?$, f^+ and f^* are factors, where f is a disjunction of alphabet symbols. A concatenation of one or more factors is a chain regular expression iff the expression is single occurrence.

The chain regular expression are a proper subset of both the single occurrence regular expressions and the simple regular expressions. The expression

$(a+b)^*c?(d+e)$ is a CHARE, while $(a+b)^*c?(d+e^*)$ is not since the concatenation's last operand in not a factor.

Although this subclass of regular expressions may seem construed, it is nevertheless quite interesting. It is very common in real-world schema definitions: 90 % of all regular expressions in DTDs and XSDs are CHAREs.

6.1.4 Expression size

A measure for the complexity of a regular expression representing a content model in a schema is the number of alphabet symbols in that expression. The number of transitions in an automaton representing this expression can roughly scale quadratic with the number of symbols.

Definition 6.8. The size of a regular expression over the alphabet Σ is defined as the number of symbols of Σ that occur in the expression.

Hence the expression $(a+b^+)c(baab)^+$ has size 7. A distribution of the regular expressions in real-world DTDs is shown in Figure 6.5. Most expressions have just a single alphabet symbol, while only 39 % have multiple symbols. Approximately 86 % of the expressions have 10 symbols or less, about 95 % of expressions have 30 symbols or less, while the largest expression occuring in the corpus has 239. One can conclude that most expressions are fairly small, but nevertheless some very large ones occur in practice.

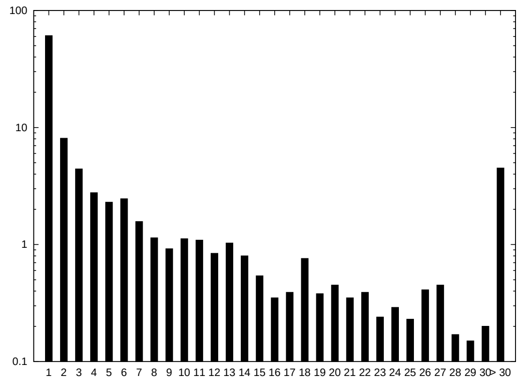


Figure 6.5: number of regular expression (%) versus expression size in a real-world DTD corpus

6.1.5 Language density

A useful measure to characterize regular expressions is the size of the language they represent.

Definition 6.9. Given a regular language L, the set of strings in L with length ℓ is denoted by L^ℓ.

Definition 6.10. The language density of a regular language L over the alphabet Σ is defined as

$$1 + \frac{1}{\sum_{\ell=0}^{\ell_{\max}} \log(1 + |\Sigma^\ell|)} \sum_{\ell=0}^{\ell_{\max}} \log \frac{1 + |L^\ell|}{1 + |\Sigma^\ell|}$$

The language Σ^* has size 1, while finite languages have values close to 0. Figure 6.6 shows the distribution of the language size over the corpus of DTD regular expressions. It is immediately clear that the regular expression Σ^* is used quite frequently, while the finite language a with size 0.2 is extremely common at 53 %.

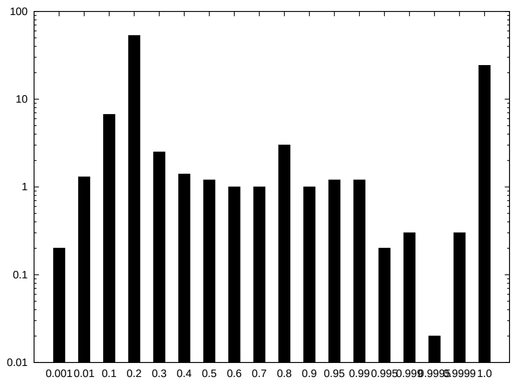


Figure 6.6: number of regular expression (%) versus language density in a real-world DTD corpus

6.2 XML corpus

Whereas the corpus of XML schemas is a prerequisite to design appropriate learning algorithms in the context of real-world applications, a corpus of XML documents and subsets of regular languages are required as training and test data. For a number of schemas such as W3C's XHTML and XML Schema Definition, Microsoft's WordML it is relatively easy to obtain a corpus of documents. We reused the software developed to obtain schema definitions from the web in order to search and retrieve appropriate XML documents.

XHTML All in all, we gathered 2092 XML documents from the web, or 48.4 MB of data. Remarkably though, a staggering 89 % of these documents is, although well-formed, nevertheless invalid with respect to the XHTML DTD [BNST06]. An observation such as this underscores the need for schema inference algorithms [BNV08].

XML Schema Definition W3C's XML Schema Definition is expressed in XML format. Hence each XSD is an XML document so that our corpus of real-world XSDs can actually be co-opted as an XML corpus for learning. We assembled $\mathcal{C}_{\mathrm{XSD}}$ from XSD documents found on the the Cover Pages [Cov03], as well as from the web at large using the Google and Yahoo! search engines, bringing the total number of fragments in $\mathcal{C}_{\mathrm{XSD}}$ to 697.

WordML The WordML corpus was not obtained directly, rather a corpus of Microsoft Word documents was retrieved from the web. Care was taken to diversify the documents' subjects and the natural language by adding additional search keywords such as "analysis", "beispiel" or "student". These documents were subsequently automatically converted to XML, so the documents are well-formed and valid by construction. In total, we obtained 11018 XML documents.

Synthetic XML Unfortunately, XML documents can not be obtained that easily for all schema definitions. We developed software to generate XML documents given a schema definition. Although ToXgene [BM06] serves a similar purpose, it does not suite our particular needs for recursive schema definitions. A more detailed description is given in Appendix A.5. Often however, generating a sample of strings that is a subset of a regular language described by some expression is all that is needed. This functionality that is part of the XML generation process can also be used in isolation.

Bibliography

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the web*. Morgan Kaufmann Publishers, 1999.

[Aho96] Helena Ahonen. Generating grammars for structured documents using grammatical inference methods. Technical report A-1996-4, University of Helsinki, Finland, 1996.

[AS83] Dana Angluin and Carl H. Smith. Inductive inference: theory and methods. *ACM Comput. Surv.*, 15(3):237–269, 1983.

[AV06] Pieter Adriaans and Paul Vitányi. The power and perils of MDL. Submitted to IEEE Transactions, December 2006.

[BDFS97] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In Foto N. Afrati and Phokion G. Kolaitis, editors, *Proceedings of International Conference on Database Theory (ICDT)*, volume 1186 of *Lecture Notes in Computer Science*, pages 336–350, Delphi, Greece, January 1997. Springer.

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems. In *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, 2003.

[BFG05] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. In Chen Li, editor, *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 25–36, Baltimore, MD, 2005. ACM Press.

[BGNV08] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *Proceedings of World Wide Web*

Conference (WWW), pages 825–834, Beijing, China, April 2008. ACM Press.

[BK93] Anne Brüggeman-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.

[BKW98] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Inf. Comput.*, 140(2):229–253, 1998.

[BM01] Paul V. Biron and Ashok Malhotra. *XML Schema part 2: datatypes.* W3C, May 2001.

[BM06] Denilson Barbosa and Alberto O. Mendelzon. Declarative generation of synthetic XML data. *Softw., Pract. Exper.*, 36(10):1051–1079, 2006.

[BMNS05] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *Proceedings of World Wide Web Conference (WWW)*, pages 712–721, Chiba, Japan, May 2005.

[BMV05] Denilson Barbosa, Laurent Mignet, and Pierangelo Veltri. Studying the XML Web: gathering statistics from an XML sample. *World Wide Web*, 8(4):413–438, 2005.

[BNST06] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of Conference on Very Large Databases (VLDB)*, pages 115–126, Seoul, South-Korea, September 2006. ACM Press.

[BNV04] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML Schema: a practical study. In *Proceedings of WebDB*, pages 79–84, Paris, France, June 2004.

[BNV07] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML Schema Definitions from XML data. In Christoph Koch, Johannes Gehrke, Minos Garofalakis, Divesh Srivastava, et al., editors, *Proceedings of Conference on Very Large Databases (VLDB)*, pages 998–1009, Vienna, Austria, September 2007.

[BNV08] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. SchemaScope: a system for inferring and cleaning XML schemas. In Jason Tsong-Li Wang, editor, *Proceedings of SIGMOD Conference*, pages 1259–1262, Vancouver, Canada, June 2008. ACM Press.

[BPSM$^+$06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0*. W3C, 4 edition, September 2006. Recommendadtion.

[Brā93] Alvis Brāzma. Efficient identification of regular expressions from representative examples. In *Proceedings of Conference on Computational Learning Theory (COLT)*, pages 236–242, Santa Cruz, CA, July 1993. ACM Press.

[CD99] James Clark and Steve DeRose. *XML Path language (XPath) Version 1.0*. W3C, November 1999.

[Chi01] Boris Chidlovskii. Schema extraction from XML: a grammatical inference approach. In Maurizio Lenzerini, Daniele Nardi, Werner Nutt, and Dan Suciu, editors, *Proceedings of Workshop on Knowledge Representation meets Databases (KRDB)*, volume 45 of *CEUR Workshop Proceedings*. Technical University of Aachen (RWTH), 2001.

[Cho02] Byron Choi. What are *real* DTDs like? In *Proceedings of WebDB*, pages 43–48, Madison, WI, June 2002.

[Cla01] James Clark. *TREX — tree regular expressions for XML: language specification*, February 2001.

[Cla03] James Clark. Trang: Multi-format schema converter based on RELAX NG. `http://www.thaiopensource.com/relaxng/trang.html`, June 2003.

[CLN04] Julien Carme, Aurélien Lemay, and Joachim Niehren. Learning node selecting tree transducer from completely annotated examples. In Paliouras and Sakakibara [PS04], pages 91–102.

[CM01] James Clark and Makoto Murata. *RELAX NG specification*. OASIS, December 2001.

[Cov03] Robin Cover. The Cover Pages. `http://xml.coverpages.org/`, 2003.

[CZ00] Pascal Caron and Djelloul Ziadi. Characterization of Glushkov automata. *Theor. Comput. Sci.*, 233(1–2):75–90, 2000.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings of SIGMOD Conference*, pages 431–442, Philadelphia, PA, June 1999. ACM Press.

[DM04] Manuel Delgado and José Morais. Approximation to the smallest regular expression for a given regular language. In Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu, editors, *Proceedings of Conference on Implementation and Application of Automata (CIAA)*, volume 3317 of *Lecture Notes in Computer Science*, pages 312–314, Kingston, Canada, July 2004. Springer.

[DuC02] Bob DuCharme. Filling in the DTD gaps with Schematron. O'Reilly xml.com, May 2002.

[EZ76] Andrzej Ehrenfeucht and Paul Zeiger. Complexity measures for regular expressions. *J. Comput. Syst. Sci.*, 12:134–146, 1976.

[Fal01] David C. Fallside. *XML Schema part 0: primer.* W3C, May 2001.

[Fer02] Henning Fernau. Learning tree languages from text. In Jyrki Kivinen and Robert H. Sloan, editors, *Proceedings of Conference on Compuational Learning Theory (COLT)*, volume 2375 of *Lecture Notes in Computer Science*, pages 153–168, Sydney, Australia, July 2002. Springer.

[Fer04] Henning Fernau. Extracting minimum length Document Type Definitions is NP-hard. In Paliouras and Sakakibara [PS04], pages 277–278.

[Fer05] Henning Fernau. Algorithms for learning regular expressions. In Sanjay Jain, Hans-Ulrich Simon, and Etsuji Tomita, editors, *Proceedings of Conference on Algorithmic Learning Theory (ALT)*, volume 3734 of *Lecture Notes in Computer Science*, pages 297–311, Singapore, October 2005. Springer.

[Flo05] Daniela Florescu. Managing semi-structured data. *ACM Queue*, 3(8), October 2005.

[FMSB$^+$06] Robert D. Finn, Jaina Mistry, Benjamin Schuster-Böckler, Sam Griffiths-Jones, et al. Pfam: clans, web tools and services. *Nucleic Acids Research*, 34:D247–D251, 2006.

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[Fra06] Jean-Marc François. Jahmm. `http://www.run.montefiore.ulg.ac.be/~francois/software/jahmm/`, April 2006.

[FS98] Mary F. Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 14–23, Orlando, FL, February 1998. IEEE Computer Society.

[GGR$^+$03] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim:. XTRACT: learning document type descriptors from XML document collections. *Data Min. Knowl. Discov.*, 7:23–56, 2003.

[GH08] Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Proceedings of Colloquium Automata, Languages and Programming (ICALP 2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 39–50, Reykjavik, Iceland, July 2008. Springer.

[GN08] Wouter Gelade and Frank Neven. Succinctness of the complement and intersection of regular expressions. In Susanne Albers and Pascal Weil, editors, *Proceedings of Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 08001 of *Dagstuhl Seminar Proceedings*, pages 325–336, Bordeaux, France, February 2008. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, May 1967.

[Gol96] Charles F. Goldfarb. The roots of SGML– a personal recollection. `http://www.sgmlsource.com/history/roots.htm`, 1996.

[GV90] Pedro García and Enrique Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(9):920–925, 1990.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *Proceedings of Conference on Very Large Databases (VLDB)*, pages 436–445, Athens, Greece, August 1997. Morgan Kaufmann.

[Hin05] Scott Hinkelman. Business Integration – Information Conformance Statements (BI-ICS). Technical report, IBM DeveloperWorks, October 2005.

[HNW06] Jan Hegewald, Felix Naumann, and Melanie Weis. XStruct: efficient schema extraction from multiple and large XML documents. In Roger S. Barga and Xiaofang Zhou, editors, *Proceedings of International Conference on Database Engineering Workshops (ICDE Workshops)*, page 81, Atlanta, GA, April 2006. IEEE Computer Society.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley series in computer science. Addison-Wesley, Reading, MA, 1979.

[Hue80] Gérard Huet. Confluent reductions: abstract properties and applications to term rewriting systems: abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

[HW05] Yo-Sub Han and Derick Wood. Shorter regular expressions from finite-state automata. In Jacques Farré, Igor Litovsky, and Sylvain Schmitz, editors, *Proceedings of Conference on Implementation and Application of Automata (CIAA)*, volume 3845 of *Lecture Notes in Computer Science*, pages 141–152, Sophia Antipolis, France, June 2005. Springer.

[IBM03] IBM corp. *XML Schema Quality Checker*, 2003.

[Jel01] Rick Jelliffe. The current state of the art of schema languages for XML. presentation at XML Asia Pacific, Sidney, Australia, 2001.

[KMS02] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. The DSD schema language. *Autom. Softw. Eng.*, 9(3):285–319, 2002.

[Koe02] Wallace Koehler. Web page change and persistence — a four-year longitudinal study. *JASIST*, 53(2):162–171, 2002.

[Koe03] Wallace Koehler. A longitudinal study of Web pages continued: a consideration of document persistence. *Inf. Res.*, 9(2), 2003.

[KSSS04] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *Proceedings of Conference on Very Large Databases (VLDB)*, pages 228–239, Toronto, Canada, August 2004. Morgan Kaufmann.

[LC00] Dongwon Lee and Wesly W. Chu. Comparative analysis of six XML schema languages. *ACM SIGMOD Record*, 29(3), 2000.

[MAC03] Jun-Ki Min, Jae-Yong Ahn, and Chin-Wan Chung. Efficient extraction of schemas for XML documents. *Inf. Process. Lett.*, 85(1):7–12, 2003.

[MBV03] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML web: a first study. In *Proceedings of World Wide Web Conference (WWW)*, pages 500–510, Budapest, Hungary, May 2003. ACM Press.

[ME95] Eve Maler and Jeanne El Andaloussi. *Developing SGML DTDs: from text to model to markup.* Prentice Hall, 1995.

[Mel04] Sergey Melnik. *Generic model management: concepts and algorithms.* Ph.D. Dissertation, Lecture Notes in Computer Science 2967, Springer, University of Leipzig, 2004.

[MFK01] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries on heterogeneous data sources. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *Proceedings of Conference on Very Large Databases (VLDB)*, pages 241–250, Roma, Italy, September 2001. Morgan Kaufmann.

[Mik02] Gerome Miklau. XMLData repository. `http://www.cs.washington.edu/research/xmldatasets/`, November 2002.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[MN07] Wim Martens and Joachim Niehren. On the minimization of XML Schemas and tree automata for unranked trees. *J. Comput. Syst. Sci.*, 73(4):550–583, 2007.

[MNS04] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for simple regular expressions. In J. Fiala, V. Koubek, and J. Kratochvil, editors, *Proceedings of Mathematical Foundations of Computer Science (MFCS)*, volume 3153 of *Lecture Notes in Computer Science*, pages 889–900. Springer, August 2004.

[MNSB06] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML Schema. *ACM TODS*, 31(3):770 – 813, September 2006.

[MSbY04] Andrew McDowell, Chris Schmidt, and Kwok bun Yue. Analysis and metrics of XML Schema. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of Conference on Software Engineering Research and Practice (SERP)*, volume 2, pages 538–544, Las Vegas, NV, June 2004. CSREA Press.

[NAM98] Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting schema from semistructured data. In Laura M. Haas and Ashutosh Tiwary, editors, *Proceedings of SIGMOD Conference*, pages 295–306, Seattle, CA, June 1998. ACM Press.

[NS03] Frank Neven and Thomas Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Proceedings of International Conference on Database Theory (ICDT)*, volume 2572 of *Lecture Notes in Computer Science*, pages 315–329, Siena, Italy, January 2003. Springer.

[NUWC97] Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. Representative objects: concise representations of semistructured, hierarchial data. In W. A. Gray and Per-Åke Larson, editors, *Proceedings of International Conference on Database Engineering (ICDE)*, pages 79–90, Birmingham, UK, April 1997. IEEE Computer Society.

[Ohl98] Enno Ohlebusch. Church-Rosser theorems for abstract reduction modulo an equivalence relation. In *Proceedings of Conference on Rewriting Techniques and Applications (RTA)*, volume 1379 of *Lecture Notes in Computer Science*, pages 17–31, Tsukuba, Japan, March 1998. Springer.

[OWA04] Top ten most critical web application vulnerabilities. Technical report, Open Web Application Security Project, January 2004.

[PAA$^+$02] Steven Pemberton, Daniel Austin, Jonny Axelsson, Tantek Çelik, et al. *XHTML 1.0 The Extensible HyperText Markup Language.* W3C, second edition edition, August 2002.

[Pit89] Leonard Pitt. Inductive inference, DFAs, and computational complexity. In Klaus P. Jantke, editor, *Proceedings of Workshop on Analogical and Inductive Inference (AII)*, volume 397 of *Lecture Notes in Computer Science*, pages 18–44, Reinhardsbrunn Castle, GDR, October 1989. Springer-Verlag.

[PS04] Georgios Paliouras and Yasubumi Sakakibara, editors. *Grammatical Inference: Algorithms and Applications, 7th International Colloquium, ICGI 2004, Athens, Greece, October 11-13, 2004, Proceedings*, volume 3264 of *Lecture Notes in Computer Science*, Athens, Greece, October 2004. Springer.

[QWG$^+$96] Dallan Quass, Jennifer Widom, Roy Goldman, Kevin Haas, et al. LORE: a Lightweight Object REpository for semistructured data. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of SIGMOD Conference*, page 549, Montreal, Canada, June 1996. ACM Press.

[Rab89] Lawrence R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proc. IEEE*, 77(2):257–286, 1989.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[RBV05] Stefan Raeymaekers, Maurice Bruynooghe, and Jan Van den Bussche. Learning (k, l)-contextual tree languages for information extraction. In João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo, editors, *Proceedigs of European Conference on Machine Learning (ECML)*, volume 3720 of *Lecture Notes in Computer Science*, pages 305–316, Porto, Portugal, October 2005. Springer.

[Sah00] Arnaud Sahuguet. Everything you ever wanted to know about DTDs, but were afraid to ask. In *Proceedings of WebDB*, pages 69–74, Dallas, TX, May 2000.

[Sak97] Yasubumi Sakakibara. Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185(1):15–45, 1997.

[Sch06] Information technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation — Schematron. International Standard ISO/IEC 19757-3, ISO/IEC, June 2006.

[SW01] Jason Sankey and Raymond K. Wong. Structural inference for semistructured data. In *Proceedings of Conference on Information and Knowledge Management (CIKM)*, pages 159–166, Atlanta, GA, November 2001. ACM Press.

[TBMM01] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema part 1: structures.* W3C, May 2001.

[VAG03] Fabio Vitali, Nicola Amorosi, and Nicola Gessa. Datatype- and namespace-aware DTDs: a minimal extension. In *Proceedings of Extreme Markup Languages*, Montreal, Canada, August 2003.

[vdV02] Eric van der Vlist. *XML Schema.* O'Reilly, Cambridge, 2002.

[WLY$^+$03] Guoren Wang, Mengchi Liu, Jeffrey Xu Yu, Bing Sun, Ge Yu, Jianhua Lv, and Hongjun Lu. Effective schema-based XML query optimization techniques. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS)*, pages 230–235, Hong Kong, China, July 2003. IEEE Computer Society.

Software

Here we give a short overview of some of the software that has been developed over the years to facilitate the experiments reported upon. A packages such as WebHunterGatherer, see Section A.1, can be of interest to any researcher who needs to search for and collect data from the web. Experimentor can be useful to any researcher doing experimental work on algorithms (see Section A.3). The Formal Language Toolkit (see Section A.4) may be useful to researcher working with formal languages, but it can also serve as the basis for XML related software since it is a library, not an application. XMLGenerator (see Section A.5 is build upon FLT and can be used to generate XML documents from a given schema.

A.1 WebHunterGatherer

WebHunterGatherer is a Java framework to perform queries and retrieve information from the World Wide Web. The package defines an interface for a search engine that is currently implemented for Google and Yahoo encapsulating the APIs provided by these companies. Subsequently, the results returned by the search engine can be retrieved easily. A web crawler component can extract links from a web page and retrieve the gathered URLs. The `LinkExtractor` is an implementation of the `Extractor` interface, so other extractors can easily be developed and plugged into the framework. The framework has proved to be quite useful when searching and gathering XML corpora on the web.

A.2 DTDParser

In order to perform our study of regular expressions that occur as content models in Document Type Definitions, real-world DTDs had to be parsed. However, DTD features such as internal and external entities make this task non trivial. At the time, no parser for DTDs was available, motivating us to develop our own implementation. To this end JavaCC[1], a Java parser generator, was used to develop a parser that supports the complete DTD syntax. DTDParser was used to extract the content models from the corpus of real-world DTDs described in Section 6.1.1. However, it can easily be adapted for other purposes as well.

A.3 Experimentor

The Experimentor Java framework has been developed in an attempt to support a rigorous setup for computer experiments. A series of experiments is described by an agenda, an XML file that defines the classes to be used and the methods that code the experiments. It also specifies the parameters to be used in the experiments, hence providing a complete description of a set of concrete experiments.

An actual experiment is implemented as a method in a class that extends `AbstractExperiment`. This base class provides methods to handle input and output. The framework provides support for timing and logging. It owes considerably to ideas developed in the context of unit testing, popularized by the Extreme Programming community [Fow99]. As such, it follows a similar pattern as, e.g., the JUnit testing framework.

When used properly, Experimentor offers considerable time savings, but its primary benefit is in the way that an agenda XML file provides documentation for the experiments that one carries out.

A.4 Formal Language Toolkit

The Formal Language Toolkit is a Java library developed to deal with formal languages, primarily with regular languages. It traces its history to a students' project in 2003 when it was developed as a reference platform to assess the difficulties involved in the project from a software engineering point of view, as well as a baseline to evaluate the students' work and track bugs in their implementation.

Although the initial scope was fairly limited, reuse ability has always been a primary design objective. And indeed, the library has been used in every research project reported here. This implies that FLT has been extended on several occasions to meet fairly diverse requirements. Since this was an almost painless process from a software engineering point of view, the initial design seems to have been adequate, even if it proved to be far from ideal. Hence it

[1] `https://javacc.dev.java.net/`

seems a useful exercise to cast a look over once shoulder on the road traveled and assess the good, the bad and the ugly.

Since this text discusses the implementation of a particular library, it is not of general interest. It may be convenient to read it with FLT's source code at hand. Although the latter has not been released, it is available from the author upon request.

Currently FLT's main strength is in the domain of regular languages. It is centered around the NFA class that implements non-deterministic finite state automata.
The class has a basic constructor and automata can be build by adding transitions, setting the initial and final states. Since a transition is a three-tuple of an alphabet symbol and two states, adding one to the automaton will automatically update its alphabet and set of states. However, symbols and states can be added or removed at any time just like transitions, and care is taken to maintain correctness, e.g., removing a state implies the removal of all of its incoming and outgoing transitions. To aid debugging, a string representation of the NFA can be computed at any time.
Several composition methods such as concatenation, union, intersection,... are available to create NFAs as well. True to its origins, these are implemented using the Thompson construction, starting from basic automata. Given that the intersection is essentially based on the product of automata, this operation was implemented as its basis and can be used independently. All relevant composition operators are n-ary rather than binary since their arguments are arrays of NFAs. An NFA can be converted to a DFA or a minimal automaton and its complement can be computed. Finally, a simplification method removes unused alphabet symbols, states that can't be reached from the initial states, and states from which no final state can be reached.
Several methods are provided to query the structure of the automaton such as testing whether a symbol is part of its alphabet, whether it has a given state, whether a state is initial or final, what state is initial,... Basic graph queries are possible as well: what states can be reached from a given state, or vice versa in a single transition with a specific or any alphabet symbol. The basic `NFA` class also implements some qualitative queries: one can test the equivalence of NFAs; whether an NFA is deterministic, or compute the shortest accepted strings.
String matching logic, i.e., checking whether a given string is a member of the automaton's language is also hosted in this class. A nice feature is that one can "step through" the matching process in the sense that a run of the automaton can be performed one symbol at the time.

`Regex` is a factory class that creates an `NFA` using the Thompson construction from a given regular expression. A default syntax for prefix regular expressions is defined, but this can be overridden by setting the appropriate properties. Regular expression can be serialized and deserialized to and from XML. Another factory class `Glushkov` has been designed to facilitate the construction of an `NFA` from a regular expression using the Glushkov construction. Additionally it provides a test for the ambiguity of regular expressions. Conversely, the class `NFAGraph` implements a generalized NFA and can convert an NFA to a regular expression using the naive state elimination algorithm.

A number of extensions of the basic `NFA` class are implemented. `CostNFA` can be used to compute the MDL of a given NFA and a set of strings. `AnnotatedNFA` provides facilities to associate information with the automaton's states and transitions. The implementation uses Java 5's generics, making it quite versatile. `GeneratingNFA` provides logic to generate a subset of the language recognized by the automaton, either in the form of a random sample, or of all strings up to a given length.

Finally, an `NFAWriter` interface provides an API that is implemented by the `DotWriter` class. This is a very useful class since it computes a dot representation of an NFA so that its structure can be visualized using the excellent GraphViz package[2]. If desired, `NFAWriter` could be implemented to provide an XML representation of an `NFA` for persistence since it is complemented by a `NFAReader` interface. Classes implementing the latter read some textual representation to construct an `NFA`.

Several classes are provided to test properties of regular expressions and automata such as ambiguity, or measures such as coverage and language size. These classes implement the interfaces `LanguageTest` and `LanguageMeasure` respectively, so that it is easy to develop additional tests and measures.

Very basic support for context-free languages is provided: context-free grammars must be in Chomsky normal form and a CYK-parser has been provided.

Several utility classes implement some basic algorithms such as computing all permutations of a list; the carthesian product of a list of `Collection`; and an enumerator of all sequences of a given length based on a set of objects. All those operations have been implemented through lazy iterators to avoid memory consumption exponential in the size of the constituents.

FLT has proved to be a nice and versatile library to work with, although it would markedly benefit from a number of improvements, mainly in the area of defining more interfaces and a finer granularity of the implementation.

[2]`http://www.graphviz.org/`

A.5 XMLGenerator

Although ToXgene [BM06] is a very nice and polished tool for generating XML documents based on an annotated schema, it can generate documents that are invalid with respect to that schema. It is also limited in its options to deal with recursive schemas. We developed XMLGenerator to overcome these limitations. The generation process can be completely parametrized by specifying probability distributions for individual operators that allows a choice such as the disjunction, the zero-or-one, zero-or-more and one-or-more operators. The depth of generated documents can also be controlled by setting hard limits or a probability distribution. Values can either be generated randomly, or selected from user-defined dictionaries. A drawback of generator is that the schema specification is non-standard, although it would be feasible to adapt it to use annotated XSDs.

Samenvatting

Motivatie

XML (eXtensible Markup Language) is uitgegroeid tot de lingua franca voor data-uitwisseling via het internet. Het is op dit ogenblik waarschijnlijk het meest populaire formaat voor semi-gestructureerde data. XML-documenten kunnen om het even welke vorm aannamen, zodat gebruikersgemeenschappen en toepassingen bepaalde structurele beperkingen opleggen aan de documenten die uitgewisseld of verwerkt moeten worden. Deze beperkingen kunnen formeel gespecificeerd worden met behulp van een *schema*, dat opgesteld wordt in een schemataal zoals *Document Type Definition* (DTD) of *XML Schema Definition* (XSD) [TBMM01].

De beschikbaarheid van een volledig gespecificeerd schema heeft vele voordelen. Eerst en vooral kan men met behulp van een schema automatisch nagaan of de structuur van een document geldig is, hetgeen automatische verwerking toelaat, maar het verzekert ook — in zeker mate — dat de invoer betrouwbaar is. Niet-valide gegevens doorgestuurd naar web servers worden gezien als de kwetsbare plek bij uitstek van webtoepassingen [OWA04]. De aanwezigheid van een schema laat tevens automatisatie en optimalisatie van zoekfunctionaliteit, integratie and verwerking van XML-gegevens toe (zie bijvoorbeeld [BFG05, DFS99, KSSS04, MFK01, NS03, WLY$^+$03]). Er zijn bovendien een aantal software-ontwikkelingswerktuigen zoals Castor[3] en SUNs JAXB[4] die steunen op schema's voor het opstellen van object-relationele vertalingen voor opslag in gegevensbanken. Het bestaan van schema's is onontbeerlijk wanneer men (meta)data integreert door gebruik te maken van overeenkomsten tussen schema's [RB01], zowel als op het terrein van generisch beheer van datamodellen [Ber03, Mel04]. Een laatste belangrijk voordeel van schema's is dat het betekenis toekent aan de gegevens. Het biedt de gebrui-

[3]`http://www.castor.org/`

[4]`http://java.sun.com/webservices/jaxb/`

ker een concrete semantiek voor het document en helpt dus bij het opstellen van zinvolle queries voor de XML-gegevens. Hoewel de hier vermelde voorbeelden slechts een greep vertegenwoordigen uit het geheel van toepassingen, onderstrepen ze duidelijk het belang van de aanwezigheid van schema's bij XML-gegevens.

Ondanks de hierboven vermelde voordelen is de aanwezigheid van een schema bij XML-documenten niet vereist, sterker nog, voor veel XML-documenten is er geen schema gegeven. Recent onderzoek van Barbosa et al. [BMV05, MBV03] toont aan dat ongeveer de helft van de XML-documenten beschikbaar op het web niet refereren naar een schema. In een andere studie hebben Bex et al. [BNV04, MNSB06] opgemerkt dat ongeveer tweederde van de XSDs in schema verzamelingen en op het web niet valide zijn ten opzichte van de XML Schema specificatie van het W3C. Dit betekent dat deze schema's onbruikbaar zijn voor toepassingen. Sahuguet [Sah00] observeerde een gelijkaardig fenomeen voor DTDs. Gegeven dit gebrek aan (bruikbare) schema's is het essentieel algoritmen te ontwikkelen die een schema kunnen afleiden uit een verzameling XML-documenten wanneer hiervoor geen syntactisch correct, of zelfs helemaal geen schema beschikbaar is.

Het is belangrijk op te merken dat zelf wanneer een schema beschikbaar is, er toch omstandigheden zijn waarin het afleiden van een schema nuttig kan zijn. Een dergelijke situatie is *schema verbetering*: soms is een gegeven schema te algemeen ten opzichte van de XML-gegevens dat het moet beschrijven. In een dergelijk geval kan het nuttig zijn een nieuw schema af te leiden dat uitsluitend gebaseerd is op de beschikbare gegevens. Dit wordt goed geïllustreerd aan de hand van het volgende voorbeeld uit de Protein Sequence Database DTD [Mik02]. Deze geeft de volgende definitie voor het `refinfo`-element:

```
authors, citation, volume?, month?, year, pages?,
  (title | description)?, xrefs?
```

Een analyse van het beschikbare XML-corpus (683 megabyte gegevens) met behulp van ons algoritme voor het afleiden van schema's genereert de volgende uitdrukking:

```
authors, citation, (volume | month), year, pages?,
  (title | description)?, xrefs?
```

Merk op dat deze strikter is dan deze uit het originele schema aangezien ze benadrukt dat het `volume`- en het `month`-element niet samen voorkomen voor een gegeven tijdschriftartikel. Men specificeert hetzij het volumenummer, hetzij de maand, maar niet beide. Dit voorbeeld illustreert dat algoritmes voor het afleiden van schema's gebruikt kunnen worden om de betekenis van de XML-gegevens beter te begrijpen, zodat het mogelijk wordt bestaande schema's aan te passen als dat nodig blijkt. Meer algemeen kan het afleiden van schema's gebruikt worden om schema's te beperken tot een relevant deel van de gegevens die nodig zijn voor een toepassing. Hierdoor kunnen moeilijke taken zoals het zoeken van overeenkomsten tussen schema's en data-integratie vergemakkelijkt worden. Hinkelman [Hin05] merkt in deze context op dat industriële standaarden vaak te onnauwkeurig gedefinieerd zijn. Dit uit zich in schema's waarin veel business-structuren formeel als optioneel gespecificeerd worden.

Een tweede situatie waarin het afleiden van schema's nuttig kan zijn, zelfs wanneer een schema beschikbaar is, is de aanwezigheid van fouten in de gegevens. Door deze fouten kan een groot gedeelte van de te verwerken gegevens geweigerd worden door het bestaande schema. We hebben een corpus van XHTML documenten van het web verzameld en bestudeerd. Het blijkt dat een verbazend groot percentage (89 %) van deze 2092 documenten niet valide was ten opzichte van de XHTML Transitional specificatie [PAA$^+$02]. In dit geval kan een afgeleid schema gebaseerd op het corpus en de vergelijking ervan met de officiële specificatie een overzicht geven van de fouten die men maakt. Verder heeft men vaak weinig keus: de foutieve data moet verwerkt worden. Men kan dan een nieuw schema afleiden uit het corpus, met weglating van die documenten waarin onacceptabele fouten gemaakt worden, om zo te kunnen werken met het nieuwe schema eerder dan met het originele om zo toch een minimale validatie te kunnen verzekeren.

Probleemstelling en bijdrage

Gezien deze observaties, lijkt het essentieel algoritmes te ontwikkelen die automatisch een DTD of een XSD kunnen afleiden uit een gegeven verzameling van XML-documenten.

Een DTD is een afbeelding van namen van XML-elementen naar reguliere expressies over namen van elementen. Een XML-document is valide ten opzichte van een DTD wanneer de kinderen van elk element voldoen aan de reguliere expressie geassocieerd met dat element.

Dit betekent dat om een DTD af te leiden uit een verzameling XML-documenten, het volstaat om voor elk element e in een document een reguliere expressie op te stellen waaraan de woorden, gevormd door de elementen onder e, voldoen. Het afleiden van een DTD is dus gereduceerd tot het afleiden van een reguliere expressie uit een verzameling woorden.

Het afleiden van XSDs is meer gecompliceerd dan het afleiden van DTDs, maar een recente karakterisatie [MNSB06] toont aan dat de structurele kern van XML Schema Definition (d.w.z. de boomtalen die gedefinieerd kunnen worden door XSDs) overeenkomen met DTDs, uitgebreid met verticale reguliere expressies. Dit illustreert dat het niet mogelijk is XSDs af te leiden zonder een goed algoritme voor het afleiden van reguliere expressies.

De klasse van alle reguliere expressies is echter te groot voor onze doeleinden: DTDs zowel als XSDs eisen dat de reguliere expressies die ze bevatten *deterministisch* zijn. Er bestaan niet-deterministische reguliere expressies die niet herschreven kunnen worden in een deterministische vorm die dezelfde taal beschrijft [BKW98]. De klasse van de deterministische reguliere expressies is dus een strikte deelklasse van de reguliere expressies.

In de context van het afleiden van DTDs uit een verzameling XML-documenten zoeken we dus een algoritme dat deterministische reguliere expressies kan afleiden uit positieve voorbeelden. Er bestaat echter een beroemd resultaat van Gold [Gol67] dat een dergelijk algoritme voor algemene reguliere expressies niet bestaat. Ook voor deterministische reguliere expressies bestaat een dergelijk algoritme niet. Hoewel het dus niet mogelijk is het probleem algemeen op te lossen, blijkt uit een analyse van een groot aantal DTDs en XSDs dat de reguliere expressies die hierin voorkomen voldoen aan de volgende eigenschap: elk alfabetsymbool komt ten hoogste k keer voor in de reguliere expressie. Het blijkt zelfs dat meer dan 98 % van de gevallen $k = 1$.

We beperken ons dus tot de deelklasse van deterministische reguliere expressies waarin een alfabetsymbool ten hoogste k maal voorkomt. Voor deze deelklasse zijn we in staat algoritmes te formuleren die een reguliere expressie uit een verzameling van voorbeelden kunnen afleiden. Voor $k = 1$ wordt een familie algoritmes gegeven in hoofdstuk 2, voor $k > 1$ in hoofdstuk 3.

Zoals ook hoger reeds aangegeven is het afleiden van XSDs een meer complex probleem dan het afleiden van DTDs. Waar bij DTDs de inhoud van een element uitsluitend bepaald wordt door de naam van dat element, is dit voor XSDs niet het geval. De inhoud van een element in een XSD wordt namelijk bepaald door de context waarin het voorkomt, d.w.z., de elementen waarin het zelf voorkomt tot aan het element dat de wortel van het document vormt [MLMK05, MNSB06]. Het is precies deze extra uitdrukkingskracht die het afleiden van XSDs bemoeilijkt in vergelijking met DTDs.

Aangezien elke DTD kan uitgedrukt worden door een equivalente XSD, en het in het algemeen niet mogelijk is een willekeurige DTD af te leiden, is het dus ook niet mogelijk elke willekeurige XSD af te leiden. Net als voor DTDs zullen we dus een deelklasse van de XSDs identificeren die afgeleid kan worden, en hiervoor een algoritme geven.

Wanneer men XSDs analyseert die in de praktijk gebruikt worden merken we op dat de context die de inhoud van een element bepaalt in de overgrote meerderheid van de gevallen vrij beperkt is. Deze hangt af van ten hoogste enkele voorouder-elementen, meestal enkel het element zelf, het vader- of — zelden —het grootvader-element. We noemen een XSD k-lokaal indien de inhoud van elk element afhangt van ten hoogste *k* voorouders. We beperken de klasse van XSDs die we beschouwen verder door op te leggen dat een alfabetsymbool slechts eenmaal mag voorkomen in een reguliere expressie die de inhoud van een element beschrijft. Voor deze deelklasse kunnen we een algoritme opstellen dat dergelijke XSDs afleidt uit een verzameling XML-documenten, zie hiervoor hoofdstuk 4.

De studie van DTDs en XSDs die in de praktijk gebruikt worden heeft ons veel geleerd, en vooral ons geholpen zinvolle deelklasses van DTDs en XSDs te identificeren die (1) belangrijk zijn in de praktijd, en (2) waarvoor we algoritmes kunnen formuleren die DTDs en XSDs afleiden uit een gegeven verzameling XML-documenten. Deze studie wordt besproken in hoofdstuk 6. Hierin worden ook nuttige eigenschappen van reguliere expressies bestudeerd, zoals de grootte van de talen die ze representeren.

Tenslotte, rekening houdend met de scenario's uiteengezet in de sectie over de motivatie van dit werk, werd SchemaScope ontwikkeld. Dit is een programma dat ontwikkelaars van XML-schema's kan bijstaan in hun werk. Het laat toe DTDs of XSDs af te leiden uit een verzameling XML-documenten, een bestaand schema te verbeteren, te verfijnen, of te veralgemenen zodat het gebruikt kan worden voor XML-documenten met fouten. SchemaScope wordt beschreven in hoofdstuk 5.