Inference of Concise DTDs from XML Data

Geert Jan Bex Frank Neven
Hasselt University and
Transnational University of Limburg

Thomas Schwentick
Dortmund University

Karl Tuyls
Maastricht University and
Transnational University of Limburg

ABSTRACT

We consider the problem to infer a concise Document Type Definition (DTD) for a given set of XML-documents, a problem which basically reduces to learning of *concise* regular expressions from positive example strings. We identify two such classes: single occurrence regular expressions (SOREs) and chain regular expressions (CHAREs). Both classes capture the far majority of the regular expressions occurring in practical DTDs and are succinct by definition. We present the algorithm iDTD (infer DTD) that learns SOREs from strings by first inferring an automaton by known techniques and then translating that automaton to a corresponding SORE, possibly by 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 iDTD outperforms existing systems in accuracy, conciseness and speed. In a scenario where only a very small amount of XML data is available, for instance when generated by Web service requests or by answers to queries, iDTD produces regular expressions which 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. Finally, we discuss incremental computation, noise, numerical predicates, and the generation of XML Schemas.

1. INTRODUCTION

1.1 Motivation

XML is the lingua franca for data exchange on the Internet [2]. Within applications or communities, XML data is usually not arbitrary but adheres to some structure possibly imposed by a schema. The advantages offered by the presence of such a schema are numerous. The most direct application is of course automatic validation of the document structure. Input validation, for instance, not only facilitates automatic processing but also ensures soundness of the input data. Indeed, unvalidated input from web requests is considered as the number one vulnerability for web applications [1]. The presence of a schema allows for automation and optimization of search, integration, and processing of XML data (cf., e.g., [7, 17, 30, 31, 38, 50]). Various software development tools such as Castor[1] and SUN's JAXB[2] rely on schemas to perform object-relational mappings for persistence. Further, the existence of schemas is imperative when integrating (meta) data through schema matching [43] and in the area of generic model management [8, 32]. 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, a schema is not mandatory and many XML-documents do not possess one. Indeed, in a recent study, Barbosa et al. [6, 33] have shown that approximately half of the XML documents available on the web do not refer to a schema. In another study Bex et al. [9, 10] 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 [48], rendering them essentially useless for immedidate application. A similar observation was noted by Sahuguet [44] concerning DTDs.

Based on the above described advantages and 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 syntactic correct one, is present. The latter problem is also acknowledged by Florescu [22] 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*". In this paper, we describe two novel schema inference algorithms outperforming existing systems in accuracy, conciseness and speed.

Before we outline our approach, we give two applications of schema inference in situations where a schema is already available: schema cleaning and dealing with noise. Even when schemas do exist, it can be advantageous to derive one solely from the XML data at hand. Indeed, sometimes schemas can be too general with respect to the data they are to represent. This is, for instance, nicely illustrated by

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

VLDB '06, September 12-15, 2006, Seoul, Korea.

the following real-world example taken from the Protein Sequence Database DTD [34]. Consider the definition of the `refinfo`-element:

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

An analysis of the available XML corpus (683 megabyte of data) shows that the `refinfo`-element is better described by the following which is more strict than the original:

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

The latter emphasizes that `volume` and `month` do not occur together: one either specifies a journal articles month of publication, or the volume it appeared in, but not both. The latter definition for `refinfo` is derived by our algorithms, illustrating that they can be used to better understand the semantics of the data and adapt the schema when necessary. In general, schema inference can be used to restrict schemas to the relevant subset that is needed by the application at hand, thereby facilitating difficult tasks like schema matching and data integration. Indeed, as argued by Hinkelman [28], industry-level standards are generally too loosely defined, which can result in XML schemas in which many business structures are formally specified as being optional.

A final example illustrating the importance of schema inference is one in the context of noisy data. We investigated a corpus of XHTML documents gathered from the web and found that an astonishing 89 % of 2092 documents was not valid to the XHTML Transitional specification[3]. Inference of a schema from the data at hand and comparing it with the schema provided by the specification provides a uniform view of the kind of errors. Further, given that one often has no choice but to deal with such noisy data one may derive a schema from a subset and work with that rather than with the official specification to retain at least a minimal validation. We refer to Section 9 for a more thorough discussion.

1.2 Problem setting

Given a collection of XML documents, a schema should be inferred in an automatic way without intervention of the user and can therefore solely be based on the XML data at hand. Like for any effective inference algorithm, the generated schema should strike a good balance between (1) specialization, i.e., covering all XML documents in the sample in a minimal way; and, (2) generalization, i.e., covering all documents satisfying the target schema but which are not necessarily present in the sample.

In this respect, schema inference problems come in two flavors. First, there is the setting when only little XML data is present, for instance, when XML is returned as answers to queries or Web service requests [39, 40]. In such a case, a schema inference algorithm should balance more towards generalization than to specificity as it is unlikely that a rich class of schemas can be learned from only a limited number of data instances. The other scenario is one in which there is a huge amount of XML data available, for instance, when the data resides in a native XML databases or is generated in bulk from existing (say relational) data. In this case, there usually is enough information to derive a highly specific schema in a rich class and a learning algorithm should therefore favor specialization over generalization.

We consider the inference of concise Document Type Definitions (DTDs) in both of the above settings. We briefly motivate why we want to infer DTDs and not the much richer formalism of XML Schema [49]. The most important reason is that, as we explain in the related work section, DTD inference has not adequately been addressed yet: existing systems do not perform well when tested on real world or sparse XML data, and as a result, generate lengthy and long-winded regular expressions. Secondly, DTD inference is a subcase of XSD inference. Indeed, recent characterizations by Bex et al. [9] show that the structural core of XML Schema, that is, the pure sets of trees which are definable by XSDs, corresponds to DTDs extended with vertical regular expressions. Therefore, one cannot hope to successfully infer XSDs without good algorithms for the derivation of DTDs. So, the present work can be seen as a first necessary step towards XML Schema inference.

As DTDs can be abstracted by context-free grammars with regular expressions (REs) at their right-hand sides, DTD inference reduces to learning of REs from positive example strings. Indeed, for every element name we need to infer an RE describing all the strings occurring below that element name in the XML corpus. Unfortunately, a seminal result by Gold [25] shows that the class of *all* REs cannot be learned from positive data only. This means that no matter how many example strings from the target language are provided, no algorithm can infer every target RE. As the framework for schema inference from XML data is exactly such that only positive example strings are provided, it is unrealistic to develop inference algorithms for the class of all DTDs. One of the main challenges is therefore to identify subclasses of REs which include the far majority of REs occurring in practical DTDs, which are concise, and which can be learned efficiently from positive data only.

We present two such classes:

1. The class of **single occurrence REs (SOREs)**, these are REs in which every element name can occur at most once. For instance, $((b?(a+c))^+d)^+e$ is SORE while $a(a+b)^*$ is not as a occurs twice.

2. The class of **chain regular expressions (CHAREs)** which are those SOREs consisting of a sequence of factors $f_1 \cdots f_n$ where every factor is an expression of the form $(a_1+\cdots+a_k)$, $(a_1+\cdots+a_k)?$, $(a_1+\cdots+a_k)^+$, or, $(a_1+\cdots+a_k)^*$, where $k \geq 1$ 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.

Note that every SORE, and therefore, also every CHARE is deterministic (or one-unambiguous [12]) as required by the XML specification.

These classes certainly satisfy the relevance criteria mentioned above: an examination of the 819 DTDs and XSDs gathered from the Cover Pages[4] (including many high-quality XML standards) as well as from the web at large, reveals that more than 99% of the REs occurring in practical schema's are CHAREs (and therefore also SOREs) [10]. Furthermore, they are succinct by definition: every element name can occur only once. Hence, the size of the generated RE is always linear in the number of different element names occurring in the corpus.

In this paper, we show that these classes of REs can be efficiently learned, thereby deriving efficient algorithms for

[3] http://www.w3.org/TR/xhtml1/

[4] http://xml.coverpages.org/

the corresponding classes of DTDs. For REs outside these classes we derive a super-approximation within the class, that is, an RE which is a SORE or a CHARE containing all strings defined by the original RE (cf. example5 in Table 2). An immediate drawback of SOREs is that they can only count up to zero, one or more occurrences of an element name. In Section 9, we discuss extensions of SOREs which can express cardinality constraints like "there should be at least three a's".

In the next subsections, we give an overview of our approach.

1.3 Approach

1.3.1 Inferring SOREs: iDTD

Proposition 1 shows that the regular languages defined by SOREs form a subclass of the **2-testable regular languages**, a class for which it is well-known that an automaton can be learned efficiently from positive data only [23]. The details of that algorithm, called **2T-INF**, are explained in Section 4. An obvious approach would be to first derive an automaton which can then be translated to an equivalent RE. Unfortunately, as already hinted upon by Fernau [20, 21] and as we illustrate below, it is very difficult to get concise REs from an automaton representation. Indeed, consider for instance the automaton of Figure 1 which is generated by 2T-INF from a set of input strings as explained in Section 4. When the standard state elimination algorithm (cf., e.g., [29]) is applied, the following RE is generated:[5]

$$(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d) + (b + aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))(aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))^*(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d)))(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d) + (b + aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))(aa^*b + (c + aa^*c)(c + aa^*c)^*(b + aa^*b))^*(aa^*d + (c + aa^*c)(c + aa^*c)^*(d + aa^*d)))^*e, \qquad (\dagger)$$

which differs quite a bit from the equivalent SORE

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

A result by Ehrenfeucht and Zeiger [18] explains why it is impossible in general to generate concise REs from automata: there are automata, even of the restricted kind as generated by 2T-INF, for which the smallest equivalent RE contains an exponential number, in the size of the automaton, of occurrences of alphabet symbols. But, as by definition the number of occurrences of alphabet symbols in a SORE equals the number of states of the corresponding automaton, SOREs constitute a well-behaved concisely representable subset of the regular languages. It therefore makes sense to investigate how to generate a concise SORE rather than applying a transformation based on state elimination which results in long-winded REs.

We provide a polynomial time algorithm REWRITE which transforms an automaton to an equivalent SORE when one exists. The algorithm is of independent interest, as to the best of our knowledge, this is the first time a subclass of REs has been identified for which an efficient rewriting from automata exists.

Unfortunately, real world XML data does not always constitutes a representative sample with respect to the target schema. That is, when 2T-INF is executed some edges of the generated automaton might be missing. For instance, due to lack of data, it could be that the automaton of Figure 2 is inferred which is a subautomaton of the intended one in Figure 1. As that automaton does not correspond to a SORE anymore, REWRITE does not succeed in deriving an equivalent SORE.We present **iDTD** (**iDTD**), an adaptation of REWRITE with repair rules which derives a SORE defining a superset of the language accepted by the given automaton. We show that iDTD outperforms existing systems in accuracy, conciseness and speed both in the presence of abundant and moderately small data, as well as on real-world and generated data. For instance, iDTD still succeeds in deriving the corresponding intended RE $((b?(a+c))^+d)^+e$ when started with the automaton of Figure 2.

Why it is necessary to infer DTDs with REs when there is already an efficient algorithm to learn automata? One could indeed opt to stick with the automata representation and not to bother with a transformation to REs. Unfortunately, no popular schema language for XML is based on an automata encoding of regular languages. So, whenever a concrete schema has to be generated, either a DTD, an XSD or even a Relax NG schema [15], equivalent regular expressions have to be constructed. Therefore, in this transformation it is important to avoid any unnecessary blow-up in size. Secondly, manageable REs are also often much more readable than automata, giving users a clear idea about the semantics of the documents and helping them to specify meaningful queries (cf., e.g., the RE generated by state elimination (†) versus the equivalent SORE (‡)).

1.3.2 Inferring CHAREs: CRX

The class of SOREs on which iDTD focuses is too rich to be learned from very small data sets. To address DTD inference in that setting we switch to the subclass of CHAREs and present an algorithm **CRX** (**Chain Regular expression eXtractor**) which derives CHAREs directly without going through the intermediate automaton representation like for SOREs and, hence, bypassing a difficult automaton to RE translation.

1.4 Outline and Contributions

We summarize the contributions of this paper:

1. We introduce a novel polynomial time algorithm (REWRITE) to translate an automaton to an equivalent SORE, when one exists. This is the first time a subclass of the regular expressions is identified for which a linear size transformation from the corresponding automata exists. For the general class of REs, a resulting expression can be of exponential size. Furthermore, SOREs form a robust extension of the class of REs frequently encountered in practical DTDs, making it a relevant target class w.r.t. DTD inference. (Section 5)

2. We introduce the algorithm iDTD that first infers an automaton from a set of strings by applying 2T-INF and then rewrites the latter into an equivalent SORE when one exists and into a SORE that is a super-approximation otherwise. As the output of iDTD is restricted to the class of SOREs, the size of the produced regular expression is always linear in the number of different alphabet symbols. As every alphabet symbol needs to occur at least once, a SORE can be seen as the most concise representation. (Section 6)

3. We introduce the algorithm CRX that derives CHAREs

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

(a subclass of SOREs) directly without going through an automaton representation. Whereas iDTD derives more specific REs, the strength of CRX is its strong generalization ability. As a consequence only very small data sets are necessary to infer an optimal CHARE. (Section 7)

4. Although iDTD and CRX are complete for the target classes of SOREs and CHAREs, respectively, it remains to validate them on real world data, incorporating small and large data sets, and on real world DTDs containing REs in and outside the target classes. Our experiments show that iDTD and CRX outperform existing systems on such data. Further, we asses the strong generalization ability of CRX by establishing on average the minimal number of strings needed to derive optimal REs. (Section 8)

5. We discuss how to extend iDTD and CRX to incrementally compute the inferred RE when new data arrives, how to address noise, and how to deal with numerical predicates. We briefly explain how our inference algorithms can be used to infer simple XSDs. (Section 9)

In Section 3, we introduce the necessary definitions, and in Section 4 we recall the algorithm 2T-INF to learn automata from data. We conclude in Section 10.

2. RELATED WORK

Schema inference. Schemas for semi-structured data have been defined in [13, 19, 42] and their inference has been addressed in [26, 37, 36]. The methods in [37, 26] 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 [36], 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, [47] 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 [3]. The latter provides a method to translate one-unambiguous non-probabilistic string automata to REs, as given by Brüggemann-Klein and Wood [12], followed by a post-processing simplification step. Apart from several case analyses 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 [24, 35, 14]. Only one of them is available for testing: XTRACT [24]. In Section 8, 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 in [24] the XML data complying to these DTDs was generated. We report in Section 8, 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 [20]. 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 [10]. As a result, larger data sets result in larger regular expressions.

In [35] an adaptation of the XTRACT approach to a restricted class of regular expressions which form a subclass of SOREs is described. Although the system, according to the experiments conducted in [35], is outperforming 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 [3], the approach of [14] relies on the translation of Glushkov automata to regular expressions which, in general, can lead to an exponential blow-up.

Trang [46] 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 8.1.

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 [4, 41, 45]. As noted by Fernau [20] and argued in the previous Section, 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 [11] showed that REs without union can be approximately learned in polynomial time from a set of examples satisfying some criteria. Recently, Fernau [21] 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 [16, 27] 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. To the best of our knowledge the present paper proposes for the first time a rewrite algorithm for a subclass of the regular languages that results in concise regular expressions of linear size when one exists.

3. DEFINITIONS

In the rest of the paper Σ is a finite alphabet of symbols (also called element names). **Regular expressions** (REs) are recursively defined as follows: every alphabet symbol $a \in \Sigma$ is a regular expression. If r and r' are regular expressions, so are $r \cdot r'$, $r + r$, $r?$, r^+, and r^*. Note that ε (the empty string) and $\emptyset$ are not allowed as basic symbols. The language defined by a regular expression r is defined as usual and is denoted by $L(r)$. We denote by $\mathbf{RE(\Sigma)}$ the class of all regular expression over Σ. We abstract a DTD as a mapping from Σ-symbols to regular expressions: A **DTD** is a pair (d, s_d) where d is a mapping from Σ to RE(Σ) and $s_d \in \Sigma$ is the start symbol. The regular expression $d(a)$ is also referred to as the element definition or content model of a. The XML specification requires regular expressions to be *deterministic*. Although the latter is a strict subclass of the regular expressions [12], we do not go into details here as the most general class of REs we consider are SOREs which are by definition deterministic.

We deviate from the usual definition of automata: labels are placed at the states rather than on transitions. The underlying idea is that every edge carries the label of the state it points to. For a set S, an **S-labeled graph** G is a tuple $(V, E, \lambda, s_{\text{in}}, s_{\text{out}})$ where V is a finite set of nodes, $E \subseteq V \times V$ is the edge relation, $\lambda : V \setminus \{s_{\text{in}}, s_{\text{out}}\} \to S$ is the labeling function, and $s_{\text{in}}, s_{\text{out}} \in V$ are the source and sink, respectively. The latter nodes will play the role of the unique initial and final state, respectively. An **automaton** is then simply a Σ-labeled graph. A **single occurrence automaton (SOA)** is an automaton where every Σ-symbol is assigned to at most one state. An example is given in Figure 1. Note that no edges are labeled: all incoming edges in a state a are assumed to be labeled with a. We say that a SOA A is equivalent to an SORE when there exists an SORE r such that $L(A) = L(r)$.

4. INFERRING AUTOMATA

We provide some background on 2-testable regular languages. The class of 2-testable regular languages is defined in terms of allowable 2-grams. A **2-gram** of a string is the set of its substrings of length two. For instance, for the string $w = bacacdacde$ its set of 2-grams is $G^2_w = \{ba, ac, ca, cd, da, de\}$. A language $L \subseteq \Sigma^*$ is then **2-testable** when there is a set of start element names $I \subseteq \Sigma$, a set of final element names $F \subseteq \Sigma$, and a set of 2-grams S such that $w \in L$ iff the first symbol of w belongs to I, the last symbol of w belongs to F and $G^2_w \subseteq S$.

It is not so hard to see that every SORE is indeed 2-testable. Consider for instance the expression $r = (a+b)^+c$. The sets $I_r = \{a, b\}$ and $F_r = \{c\}$ are simply the symbols that can start and end a string defined by r, respectively. The set S_r equals the set of 2-grams $\{ab, aa, ba, bb, ac, bc\}$ which can be computed by inspecting which symbols can be succeeded by which other symbols in strings defined by r.

We now describe the algorithm **2T-INF** of [23] to infer the sets I, F and S, and therefore every 2-testable language, from a set of input strings $W = \{w_1, \ldots, w_n\}$. Then I_W and F_W are simply all first and last symbols, respectively, and $S_W = \bigcup_{i \leq n} G^2_{w_i}$. It remains to construct a corresponding finite automaton G_W: take a state for every element name and a separate initial and final state. Construct an edge from the initial state to every state in I_W and an edge from every state in F_W to the final state. Further, for every $ab \in S_W$ there is an edge from state a to b.

We illustrate the latter by means of an example. Consider the input strings *bacacdacde*, *cbacdbacde*, and *abccaadcde*. Then $I_W = \{a, b, c\}$, $F_W = \{e\}$, and $S_W = \{aa, ad, ac, ab, ba, bc, cb, cc, ca, cd, da, db, dc, de\}$. The automaton derived from I_W, F_W and S_W is given in Figure 1. Note that every 2-testable language is uniquely identified by its corresponding SOA and vice versa. The automaton generated by *bacacdacde* and *cbacdbacde* is given in Figure 2.

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

PROOF. (sketch) Given a SORE r, its Glushkov automaton [12] is an SOA. The result then follows as whenever $L(G) = L(G')$ for two SOAs then G and G' are isomorphic. □

As every SORE is 2-testable, the algorithm 2T-INF succeeds in efficiently generating the SOA equivalent to the target SORE when a representative sample of strings is provided. A set is representative w.r.t. a SORE when it contains all corresponding 2-grams.

In the next subsection we give an algorithm REWRITE to translate SOAs into SOREs. The latter will then be adapted in Section 6 to provide together with 2T-INF the inference algorithm iDTD.

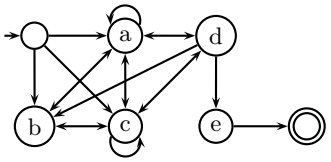


Figure 1: The automaton G_W corresponding to I_W, F_W and S_W for $W = \{bacacdacde, cbacdbacde, abccaadcde\}$. Every edge is assumed to be labeled by the label of the node it points to.

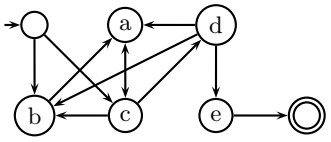


Figure 2: The automaton G_W derived from $W = \{bacacdacde, cbacdbacde\}$.

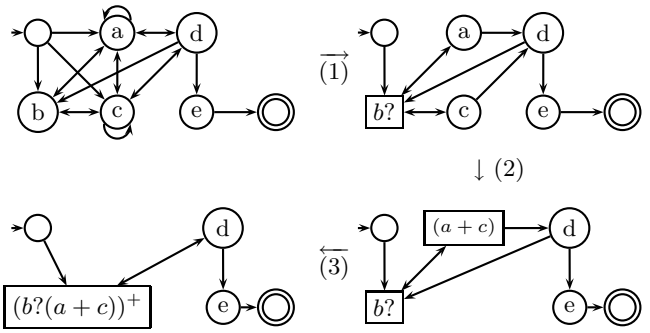


Figure 3: An execution of REWRITE **on the example automaton in Figure 1. Step (1) applies** OPTIONAL **to b. Step (2) applies** DISJUNCTION **to a and c. Step (3) applies** CONCATENATION **to $b?$ and $(a+c)$. Two further applications of concatenation lead to the resulting expression $((b?(a+c))^+d)^+e$. Note that** DISJUNCTION **could also be applied to the original automaton which would then result into the equivalent expression $((b?(a+c)^+)^+d)^+e$.**

5. FROM SOA TO SORE

In this section we employ a graph-rewriting approach to transform a SOA to a SORE. Our algorithm REWRITE is related to the usual state-elimination algorithm transforming an automaton to an RE (cf., e.g., [29]). However, in a transformation step, we do not replace a state by introducing edges but rather replace a suitable set of states by a single state. Further, there are rewriting steps which only remove edges.

Note that our rewrite algorithm never introduces the Kleene-star as in r^* but represent it like $(r^+)?$ or $(r?)^+$. This is no loss of generality as we can always rewrite the latter into the former during a post-processing step.

Just like the classical algorithm, we use automata with regular expressions as labels on the edges. In our setting, these labels occur at the states. Again, the semantics can be interpreted as if every edge carries the regular expression of the node the edge points to. Thus, a **generalized finite automaton (GFA)** is a RE(Σ)-labeled graph $G = (V, E, \lambda, s_{\text{in}}, s_{\text{out}})$. Such an automaton is **single occurrence (SO)** iff $\lambda(v)$ is an SORE for every $v \in V$ and different from ε, and every Σ-symbol occurs in at most one $\lambda(v)$. Clearly, each SOA is a single occurrence GFA. As every node carries a unique regular expression, we can identify a vertex v by its label $r = \lambda(v)$.

Let $G = (V, E, \lambda, s_{\text{in}}, s_{\text{out}})$ be a GFA. The **ε-closure** G^* of G is the graph (V, E^*), where E^* contains (i) all edges (r, r), where $r = s^+$ or $r = (s^+)?$ and (ii) all edges (r, r'), for which there is a path from r to r' in G which only passes (intermediate) nodes r'' with $\varepsilon \in L(r'')$. The **predecessor set** $\text{Pred}(r)$ of a node $r \in V$ contains all predecessors r' of r in G^*. Accordingly, we define the **successor set** of a node $r \in V$ denoted by $\text{Succ}(r)$.

The rewrite system consists of four rules, corresponding to the four operators used in (our) regular expressions. Two of them replace state sets by states, the other two remove edges. An example execution is shown in Figure 3.

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

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

3. SELF-LOOP**.** Precondition: $(r, r) \in E$.
   Action: Delete (r, r), relabel r by r^+.

4. OPTIONAL**.** Precondition: Every $r' \in \text{Pred}(r), \text{Succ}(r) \subseteq \text{Succ}(r')$. (Thus: every node that can be reached through r from a predecessor, can also be reached directly from that predecessor.)
   Action: Relabel r by $r?$, remove all edges (r', r'') such that $r' \in \text{Pred}(r)$ and $r'' \in \text{Succ}(r) \setminus \{r\}$.

The algorithm repeats the above rewrite rules until it ends up with a GFA which exists of one node r in addition to the source and the sink, with an edge from the source to r and an edge from r to the sink and no other edges. We say that such a GFA is **final**. Clearly, the language accepted by a final GFA is the language denoted by r. We denote such a GFA by G_r. The algorithm REWRITE is given in Algorithm 1.

Algorithm 1 REWRITE

Input: an SOA G
Output: a SORE r such that $L(r) = L(G)$

```
while a rewrite rule can be applied do
    let G' be obtained from G by applying a rewrite rule
    G := G'
if G is final then
    return the corresponding regular expression
else
    fail
```

Note that the complexity $\mathcal{O}(n^4)$ in the next theorem is not so bad as n actually refers to the number of different element names in the sample, not to the total number or total length of strings.

THEOREM 1. *Algorithm* REWRITE *transforms an SOA A into an equivalent SORE or reports that no such SORE exists. It works in time $\mathcal{O}(n^4)$ where n is the size of A.*

PROOF. Clearly, the size of the resulting SORE is linear in n. As every rewrite step introduces at least one operator and operators are never removed, there can be at most $\mathcal{O}(n)$ rewrite steps. It therefore suffices to show that, for each of the four rules, in time $\mathcal{O}(n^3)$ it can be checked, whether it is applicable and, if possible, the action can be taken. The precondition of SELF-LOOP can be checked in time $\mathcal{O}(n)$. For the precondition of OPTIONAL, for each pair

of nodes, $\mathcal{O}(n)$ steps are sufficient. To test the precondition of CONCATENATION, it has to be checked, for each node r_1, whether there are $r_2, \ldots, r_n$ fulfilling the requirements. Clearly, $\mathcal{O}(n^2)$ steps are sufficient. Finally, for DISJUNCTION, it suffices to first find two nodes r_1, r_2 fulfilling the precondition. This is possible in $\mathcal{O}(n^3)$ steps. Then in another $\mathcal{O}(n^2)$ it can be checked whether there are suitable $r_3, \ldots, r_n$.

By inspecting the four rules, soundness of the algorithm can be easily checked. We denote by $G \Rightarrow G'$ that G' is obtained from G by applying a rewrite rule. Then $\Rightarrow^*$ denotes the reflexive transitive closure of $\Rightarrow$. Thus, if $G \Rightarrow^* G_r$ then $L(G) = L(G_r) = L(r)$.

It remains to prove the completeness of the algorithm.

CLAIM 1. *For every SOA G that is equivalent to a SORE r, there is a SORE s such that $G \Rightarrow^* G_s$ and $L(r) = L(s)$.*

Let G and r be as stated. By applying the transformation rules $(s^+)^+ \to s^+$, $s?? \to s?$, and $(s?)^+ \to (s^+)?$ to all subexpressions of r we obtain a **normalized** equivalent SORE s without superfluous operators. We say that a GFA with only normalized SOREs is itself normalized. It can be shown that, whenever a GFA G contains at least one normalized SORE which is not an alphabet symbol, there is a GFA G', such that $G' \Rightarrow G$. By induction we obtain a "backward rewriting" of G_s ending in an SOA G'. As SOAs are unique up to isomorphism, we can conclude that $G \Rightarrow^* G_s$, thus proving the claim.

CLAIM 2. *If G is a GFA equivalent to a SORE r, then any order of rewrite steps leads to a SORE s.*

We call a set W fulfilling the precondition of DISJUNCTION or CONCATENATION a candidate. Likewise, a node r fulfilling the precondition of SELF-LOOP or OPTIONAL is a candidate node. Clearly, two candidates sets are always disjoint, in particular, a state to which DISJUNCTION can be applied can not be a candidate for CONCATENATION. Furthermore, a candidate node for OPTIONAL can not be in a candidate set for CONCATENATION. Finally, a candidate for SELF-LOOP is not suitable for CONCATENATION. Nevertheless, a SELF-LOOP candidate might be also a candidate for OPTIONAL and/or DISJUNCTION, and an OPTIONAL-candidate might be in a DISJUNCTION candidate set. However, applying SELF-LOOP does not change the candidate status of a state for the other two rules. Thus, the only non-trivial interference between rules is that an OPTIONAL-candidate might be in a candidate set for DISJUNCTION. But as the application of OPTIONAL does not change the closure of G,it does not affect the candidate sets for DISJUNCTION. To sum up, application of a rule never prevents the application of another rule.

All above claims taken together prove the theorem. □

6. INFERRING SORES: IDTD

Although REWRITE is complete, the algorithm only succeeds if the input SOA has an equivalent SORE. Otherwise it reports failure. In the context of incomplete data, this failure might be due to missing edges in the SOA G_W inferred from the set W of example strings by 2T-INF. In this section, we thus present an algorithm, iDTD (infer DTD), which is an adaptation of REWRITE which attempts to produce an SORE which describes a (as small as possible) superset of $L(G_W)$.

An outline of iDTD is shown in Algorithm 2. It runs REWRITE until no more rewrite rule can be applied (Line 1). Then it chooses one repair rule and restarts REWRITE on the result graph. We discuss the repair rules below.

Algorithm 2 iDTD

Input: SOA G_W
Output: SORE r_W such that, $L(G_W) \subseteq L(r_W)$
1: $G :=$ REWRITE(G_W)
2: $k := 1$
3: **while** G is non-final **do**
4: **while** no repair rule can be applied **do**
5: $k := k + 1$
6: Apply one repair rule on G
7: $G :=$ REWRITE(G)
8: r_W is the RE of the final SOA G

One repair rule enables application of DISJUNCTION the other enables OPTIONAL. The application of the latter might in turn enable CONCATENATION. The repair rules have a parameter k which allows to vary the degree of fuzziness. They are as follows.

1. ENABLE-DISJUNCTION**:** Action: Add, for some set $W = \{r_1, \ldots, r_n\}$ a (minimal) set of edges such that $\text{Pred}(r_i) = \text{Pred}(r_j)$ and $\text{Succ}(r_i) = \text{Succ}(r_j)$, for all i, j.
Preconditions: (a) for each i, j, $\text{Pred}(r_i) \cap \text{Pred}(r_j) \neq \emptyset$, $|\text{Pred}(r_i) \setminus \text{Pred}(r_j)| \leq k$, $|\text{Pred}(r_i) \setminus \text{Pred}(r_j)| \leq k$ and the corresponding conditions hold for the successor sets, or (b) for every $r \in W$, $W \subseteq \text{Pred}(r)$ and $W \subseteq \text{Succ}(r)$.

2. ENABLE-OPTIONAL**:** Action: Add, for some state r, all missing edges from states in $\text{Pred}(r)$ to states in $\text{Succ}(r)$.
Preconditions: (a) there is at least one edge from a predecessor of r to a successor of r, or (b) $\text{Pred}(r) = \{r'\}$, for some r' and $|\text{Succ}(r') \setminus \{r, r'\}| \leq k$

Note that after applying ENABLE-OPTIONAL, r is a candidate for OPTIONAL. The application of OPTIONAL will then remove all edges introduced by ENABLE-OPTIONAL. In case (a), at least one more edge is removed than inserted, in case (b) r, r' become candidates for CONCATENATION.

As an example, we consider the automaton of Figure 2. None of the rules of REWRITE can be applied to it. Nevertheless, $\{a, c\}$ fulfil Precondition (b) of ENABLE-DISJUNCTION. The minimal set of edges that need to be added to ensure $\text{Pred}(a) = \text{Pred}(c)$ and $\text{Succ}(a) = \text{Succ}(c)$ are the ones that are missing when comparing to Figure 1. So, from now on the rules of REWRITE can be applied.

THEOREM 2. *For an SOA A, iDTD always produces a SORE r with $L(A) \subseteq L(r)$.*

Because of efficiency reasons, our implementation considers rule 1(a) only for $n = 2$. Further, we set $k = 2$. Rule 1 and 2 are tried in this order, the latter only when the former can not be applied. Although, in our experiments iDTD always produced a SORE for these values of n and k, it is possible, for each fixed k and n, to construct SOAs where iDTD does not succeed. The unrestricted variant of iDTD always succeeds.

7. INFERRING CHARES: CRX

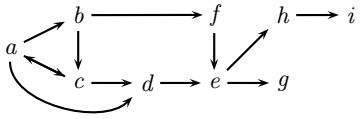
In this section, we present the algorithm CRX for the extraction of chain regular expressions (CHAREs) as defined in the introduction. We first present the core idea of CRX by means of an example. This idea is based on classifying positional relationships between occurrences of alphabet symbols in the input strings. Indeed, as every CHARE is also single occurrence, this means that every occurrence of an alphabet symbol in a string has to be generated by the same factor in the target CHARE.

EXAMPLE 1. *Consider the input strings $u = abd$, $v = bcdee$, and $w = cade$. Then, a occurs before b in u, b occurs before c in v, and c occurs before a in w. This means that a, b, and c belong to the same factor which can only be $(a+b+c)^+$ or $(a+b+c)^*$. As in every string a symbol of $\{a,b,c\}$ is present we take the factor $(a+b+c)^+$. Similarly, d and e form a factor by themselves. Whereas d occurs in every string once and is therefore presented by the factor d, e can occur zero, one or more times and is thus represented by the factor e^*. As 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 factors, orders them and uses that order to generate a CHARE. Of course, the order of factors is not necessarily linear. In that case, a linear order can be constructed by making classes optional. Some care has to be taken to generate factors which are disjunctions without repetitions.

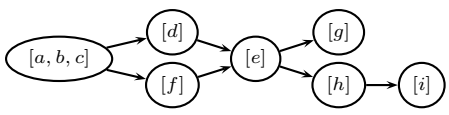
So, a set of strings induces a pre-order on the alphabet. We define this order formally: for $a, b \in \Sigma$, $a \rightarrow_W b$ iff there are a string $w = w_1 \cdots w_n \in W$ and some i such that $w_i = a$ and $w_{i+1} = b$.

EXAMPLE 2. *Let $W = \{abccde, cccad, bfegg, bfehi\}$. The graph induced by $\rightarrow_W$ is as follows.*



We define the relation $\approx_W \subseteq \Sigma \times \Sigma$ as follows: $a \approx_W b$ iff $a \rightarrow_W^* b$ and $b \rightarrow_W^* a$, where $\rightarrow_W^*$ is the reflexive, transitive closure of $\rightarrow_W$. Clearly, $\approx_W$ is an equivalence relation. Let Γ_W be the set of equivalence classes of $\approx_W$. We write equivalence classes in the form $[a_1, \ldots, a_n]$. A node of the form $[a]$ is called a singleton. Let $\preceq_W$ be the partial order on Γ_W which is induced by $\rightarrow_W^*$. For every $\preceq_W$, we denote by H_W its Hasse diagram, i.e., the graph of $\preceq_W$ without transitive edges. The successor set of a node γ in H_W, denoted by $\text{succ}(\gamma)$, is simply the set of nodes that can be reached by a single edge from γ. Conversely, the predecessor set of a node γ, denoted by $\text{pred}(\gamma)$, is the set of nodes from which γ can be reached by a single edge.

EXAMPLE 3. *The Hasse diagram H_W is as follows.*



Note that $\text{succ}([d]) = \text{succ}([f]) = \{[e]\}$ *and* $\text{pred}([d]) = \text{pred}([f]) = \{[a,b,c]\}$. □

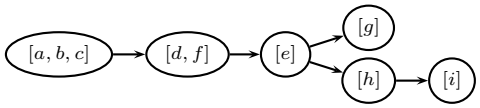
Algorithm 3 CRX

Input: set of strings W
Output: CHARE r_W such that $W \subseteq L(r_W)$

1: Compute the set Γ_W of equivalence classes of $\approx_W$
2: **while** a maximal set of singleton nodes $\gamma_1, \ldots, \gamma_\ell$ such that $\text{pred}(\gamma_1) = \cdots = \text{pred}(\gamma_\ell)$ and $\text{succ}(\gamma_1) = \cdots = \text{succ}(\gamma_\ell)$ exists **do**
3: Replace $\gamma_1, \ldots, \gamma_\ell$ by $\gamma := \cup_{j=1}^{\ell} \gamma_j$, redirect all incoming and outgoing edges of the γ_i to γ
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 string in W 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 string in W 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 string in W contains at least one of $a_1, \ldots, a_n$ and there is a string that contains at least two occurrences of symbols **then**
11: $r(\gamma_i) = (a_1 + \cdots + a_n)^+$
12: **else**
13: $r(\gamma_i) = (a_1 + \cdots + a_n)^*$

The CRX algorithm is shown in Algorithm 3 an illustrated in the following example.

EXAMPLE 4. *Step 3 can be applied to $[d]$ and $[f]$ resulting in:*



Although $\text{pred}([g]) = \text{pred}([h])$*, step 3 cannot be applied as* $\text{succ}([g]) \neq \text{succ}([h])$*. Similarly $[g]$ and $[i]$ share successors, i.e. $\emptyset$, but have different predecessors. 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, $r([a,b,c]) = (a+b+c)^+$ is the first factor; the second factor is $(d+f)$ 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 of $[h]$, $[i]$. Finally, $[g]$ occurs multiple times in a single string. Hence the simple regular expression derived by the algorithm is $(a+b+c)^+ \cdot (d+f) \cdot e? \cdot g^* \cdot h? \cdot i?$ which completes step 3.* □

Note that the order of the factors in the regular expression depends on the topological sort (cf., e.g., `genetics` in Table 1). It follows by inspection of the algorithm that the generated r_W is a CHARE.

THEOREM 3. *Given a set of strings W, CRX computes a CHARE r_W such that $W \subseteq L(r_W)$.*

On the class of CHAREs, CRX is complete:

THEOREM 4. *For each CHARE r there is a set of strings W_r, such that CRX infers r_W from W_r with $L(r_W) = L(r)$.*

The experiments in Section 8.2 show that the number of example strings in W_r needed in practice is very small.

Actually, the prime feature that makes CRX much more robust than iDTD for very small data sets is its strong generalization ability. Indeed, consider an expression of the

form $(a_1 + \cdots + a_n)^*$. While REWRITE requires all n^2 substrings of the form $a_i a_j$ for $i, j \in \{1, \ldots, n\}$ to be present, iDTD also still requires around $n^2 - n$ substrings. For CRX, the set $\{a_1a_2, a_2a_3, \ldots, a_{n-1}a_n, a_na_1\}$ of size $\mathcal{O}(n)$ will suffice. This point is illustrated in practice by `example3` and `example4` in Table 2 where n has a value of 41 and 56 respectively. Experiments illustrate that only $400 \ll 1682$ and $500 \ll 3136$ length 2 substrings are needed in the samples for CRX to learn `example3` and `example4`, respectively.

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

THEOREM 5. *For every set W, if Γ_W is linearly ordered then for every CHARE r such that $W \subseteq L(r)$ and $L(r) \subseteq L(r_W)$, we have $r_W = r$, i.e. r_W is syntactically equal to r up to commutativity of $+$.*

However, a simple example shows that this property does not hold when the partial order is not linear. Consider $W = \{abc, ade, abe\}$ that yields $r_W = a \cdot b? \cdot d? \cdot c? \cdot e?$ whereas the simple regular expression $a \cdot (b+d) \cdot (c+e)$ is better balanced between specificity and generalization.

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

8. EXPERIMENTAL EVALUATION

We assess the quality of our algorithms on real-world corpora and DTDs, and compare it with that of XTRACT [24] and Trang [46]. Next, we compare the generalization factor of iDTD and CRX by investigating how many example strings are needed to derive an optimal expression. Finally, we discuss the time performance of the algorithms.

8.1 Real-world examples

The number of publicly available XML corpora is rather limited. We employed the XML Data repository maintained by Miklau [34] as a testbed. Unfortunately, most of the listed XML corpora are either very small, lack a DTD or contain one with only trivial regular expressions. Nevertheless, two of them prove to be quite interesting for our experiments. Specifically, we compared XTRACT, iDTD, and CRX on the Protein Sequence Database and the Mondial corpus [34], a database of information on various countries. As 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 [10] containing a number of sophisticated REs outside the class of CHAREs.

Real-world data. Table 1 lists all non-trivial element definitions[6] in the above mentioned DTDs together with the results derived by the inference algorithms iDTD, 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 1 are learned quite satisfactory by iDTD and CRX w.r.t. the examples extracted from the XML corpus. The numbers in the first column refer to the number of used example strings. iDTD 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 corpus formed a representative sample. That is, in every single case iDTD 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 iDTD are more strict than the corresponding one in the DTD. This is due to the data present 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 element `authors`, 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 examples yields a complex expression of 185 token.

Real-world regular expressions. Table 2 lists the results of the algorithms on a number of more sophisticated regular expressions extracted from real-world DTDs discussed in [10]. As no data was available, we randomly generated some using ToXgene [5], taking care that all relevant examples where present to ensure the target expression could be learned. Again, we list the number of used strings 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 strings is not unreasonably large. Note that only the first three expressions in Table 2 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 iDTD 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 [46] 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 ${a_1}^+ + (a_2? {a_3}^+)$.

[6]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.

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

Table 1: Results of iDTD, 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/iDTD and sample size for XTRACT, respectively. The right column lists original DTD, inferred DTD by CRX/iDTD and the result of XTRACT, in that order.

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 for a formal model as the cornerstone of an implementation. Indeed, there is no paper 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 iDTD.

8.2 Generalization

We experimentally address the question of how many example strings are needed to learn a regular expression with CRX and iDTD. We do not address XTRACT as Table 1 already shows that even for small data sets XTRACT produces suboptimal results. We use the following approach: we start by generating a representative sample set for a regular expression. That is, when 2T-INF derives the corresponding SOA no edges are missing. Next, we compute from this data the target expressions r_{CRX} for CRX and r_{iDTD} for iDTD. Then, for both r_{CRX} and r_{iDTD}, the critical size, i.e., the smallest c such that from all tested samples of size c the target expression can be derived. This is done by generating 200 subsamples using reservoir sampling for each size and counting the number of those subsamples from which the target expression is successfully obtained. It is ensured that the subsamples contain all alphabet symbols of the target expressions for fair comparisons. For samples smaller than this critical size, the relative frequency of cases where the target expression can be successfully recovered decreases as is shown in Figure 4 for the expressions `example2`, `example4`, and $\left(a_1\left(a_2+\cdots+a_{12}\right)^+\left(a_{13}+a_{14}\right)\right)^+$. (‡)

The plots highlight CRX' strong generalization abilities: CRX needs far fewer example strings than iDTD to derive an optimal RE (depending on the example 2 to 10 times less). Note however that CRX is a super-approximation and that iDTD derives an approximation that is more precise. The plots also clearly illustrate that iDTD is able to infer r_{iDTD} in cases where REWRITE alone fails, emphasizing the effectiveness of the repair rules.

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

Table 2: Results of iDTD, CRX and XTRACT on non-simple real world DTDs and generated data. The left column gives element names, sample size for CRX, iDTD and XTRACT, respectively. The right column lists original DTD, inferred DTD by CRX, by iDTD and the result of XTRACT, in that order.

8.3 Performance

The performance of iDTD is adequate to deal with large data sets. `Example4` with 61 symbols in Table 2 is derived from 10000 example strings 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. 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 data sets with more than 1000 strings.

9. EXTENSIONS

Incremental computation. Especially in the setting of sparse data when over time more XML data gets generated, for instance, by answers to queries or results of calls to web services, it is desirable to update an already generated

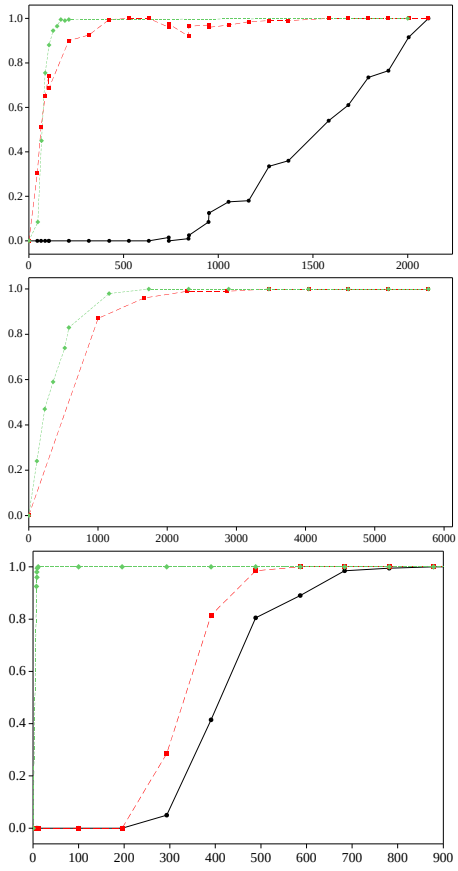


Figure 4: Fraction of runs resulting in target expression `example2` (top), `example4` (middle), and expression (‡) (bottom) as a function of the sample size for each of CRX (diamonds/dotted,green), iDTD (squares/dashed,red) and REWRITE (circles/solid, black).

schema based on the newly arrived XML data only. Such an approach is possible for both iDTD and CRX: as both algorithms make use of an internal representation (automata or partial orders), we only need to update that representation. So, for every element name we store the corresponding internal graph representation, which is only quadratic in the number of different element names, and we can forget about the XML data that generated it. Actually, for CRX, to assign the qualifiers ?, + and ∗, we also need to remember for each element name how it occurs (always exactly once, always more than once,...), but this is only a constant amount of information.

Noise. Like any real world data, XML data can contain noise. For instance, in XHTML the definition of a paragraph `<P>` element is a quite elaborate repeated disjunction $(a_1 + \cdots + a_k)^*$ where $k = 41$. Nevertheless, in the XHTML documents we examined (with a total number of more than 30000 occurrences of paragraph elements), a dozen of disallowed elements appeared (like `table`, `h1`, `h2`, ...) albeit in small numbers: on average in around 10 strings. An obvious way in dealing with noise is to consider the support of each element name and to simply disregard that element when the latter is less then a given threshold. For iDTD, a more sophisticated approach can be taken where a support is associated to each edge of the SOA generated by 2T+INF. As long as iDTD can apply the unmodified REWRITE rewrite rules these numbers are ignored, from the moment REWRITE get stuck and repair rules should be considered, it is checked whether removing some of the edges with low support allows to advance REWRITE.

Numerical predicates. An immediate drawback of SOREs is that they can not count. For instance, they can not express $aabb^+$ specifying that a string should start with two a's followed by any number of b's larger than 1. XML Schema even uses dedicated attributes for expressing the desired number of repetitions:

```
<xs:sequence>
  <xs:element name="a" minOccurs=2 maxOccurs=2/>
  <xs:element name="b" minOccurs=2 maxOccurs="unbounded"/>
</xs:sequence>
```

In the same way, REs can be extended by numerical predicates: when r is an RE and i is a natural number then $r^{\geq i}$ and $r^{=i}$ are also REs. They are semantically equivalent to $r^i r^*$ and r^i, respectively, where $r^i = r \cdot r \cdot \cdots \cdot r$ (i times). The above expression can then be expressed as $a^{=2}b^{\geq 2}$. To both iDTD and CRX a post-processing step can be added that rewrites + and ∗ to numerical values based on exact occurrences of element names in the XML data.

Generation of XSDs. The study in [9] shows that 85% of XSDs are structurally equivalent to a DTD. Generating such XSDs is merely a matter of using the correct syntax. Improvements to the derivation of built-in data types can be made by introducing heuristics to recognize times or dates, integers, doubles, nmtokens and strings hence extending the implementation for DTDs that was done for CRX. Extending schema derivation to XSDs with more expressive power than DTDs is the topic of current research.

10. DISCUSSION

We introduced two novel algorithms iDTD and CRX for the inference of concise DTDs. We show that the quality of inferred DTDs 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, iDTD and CRX always infer succinct DTDs by definition which can easily be interpreted by humans. Of independent interest, we introduced a new algorithm to generate REs from automata. In future work, we plan to investigate the inference of XML Schema Definitions, which by [9] can be abstracted by DTDs with vertical regular patterns.

Acknowledgments.. We thank the authors of [24], for making available XTRACT's source code.

11. REFERENCES

[1] Top ten most critical web application vulnerabilities. Technical report, Open Web Application Security Project, Jan. 2004.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 1999.

[3] H. Ahonen. Generating grammars for structured documents using grammatical inference methods. Report A-1996-4, Univ. of Finland, 1996.

[4] D. Angluin and C. H. Smith. Inductive inference: theory and methods. *ACM Computing Surveys*, 15(3):237–269, 1983.

[5] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: An extensible template-based data generator for XML. *WebDB*, 49–54, 2002.

[6] D. Barbosa, L. Mignet, and P. Veltri. Studying the XML Web: gathering statistics from an XML sample. *WWW*, 8(4):413–438, 2005.

[7] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *PODS*, 25–36, 2005.

[8] P. Bernstein. Applying model management to classical meta data problems. *CIDR*, 2003.

[9] W. Martens, F. Neven, and T. Schwentick and G.J. Bex. Expressiveness and Complexity of XML Schema. *ACM TODS*, 31(3), 2006.

[10] G.J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: a practical study. *WebDB*, 79–84, 2004.

[11] A. Brāzma. Efficient identification of regular expressions from representative examples. *COLT*, 236–242, 1993.

[12] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and computation*, 140(2):229–253, 1998.

[13] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. *ICDT*, LNCS 1186, 336–350, 1997.

[14] B. Chidlovskii. Schema extraction from XML: a grammatical inference approach. *KRDB*, 2001.

[15] J. Clark and M. Murata. *RELAX NG Specification*. OASIS, Dec. 2001.

[16] M. Delgado and J. Morais. Approximation to the smallest regular expression for a given regular language. *CIAA*, LNCS 3317, 312–314, 2004.

[17] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. *SIGMOD*, 431–442, 1999.

[18] A. Ehrenfeucht and P. Zeiger. Complexity measures for regular expressions. *Journal of computer and system sciences*, 12:134–146, 1976.

[19] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. *ICDE*, 14–23, 1998.

[20] H. Fernau. Extracting minimum length Document Type Definitions is NP-hard. *ICGI*, LNAI 3264, 277–278, 2004.

[21] H. Fernau. Algorithms for learning regular expressions. *ALT*, LNCS 3734, 297–311, 2005.

[22] D. Florescu. Managing semistructured data. *ACM Queue*, 3(8), Oct. 2005.

[23] P. Garcia and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *Patt. Anal. and Mach. Int.*, 12(9):920–925, 1990.

[24] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: learning document type descriptors from XML document collections. *Data mining and knowledge discovery*, 7:23–56, 2003.

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

[26] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *VLDB*, 436–445, 1997.

[27] Y.-S. Han and D. Wood. Shorter regular expressions from finite-state automata. In *CIAA*, 141–152, 2005.

[28] S. Hinkelman. Business Integration – Information Conformance Statements (BI-ICS). IBM, 2005. http://www.ibm.com/developerworks/xml/library/x-biics/

[29] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[30] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. *VLDB*, 228–239, 2004.

[31] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. *VLDB*, 241–250, 2001.

[32] S. Melnik. *Generic model management: concepts and algorithms*. Ph.D., LNCS 2967, Univ. of Leipzig, 2004.

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

[34] G. Miklau. XMLData Repository, Nov. 2002. http://www.cs.washington.edu/research/xmldatasets/

[35] J.K. Min, J.Y. Ahn, and C.W. Chung. Efficient extraction of schemas for XML documents. *Inf. Process. Lett.*, 85(1):7–12, 2003.

[36] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. SIGMOD, 295–306, 1998.

[37] S. Nestorov, J. D. Ullman, J. Wiener, and S. Chawathe. Representative objects: concise representations of semistructured, hierarchial data. *ICDE*, 79–90, 1997.

[38] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. *ICDT*, 315–329, 2003.

[39] A. Ngu, D. Rocco, T. Critchlow, and D. Buttler. Automatic discovery and inferencing of complex bioinformatics web interfaces. *WWW*, 463–493, 2005.

[40] P. Oaks and A. ter Hofstede. Guided Interaction: a language and method for incremental revelation of software interfaces for ad hoc interaction. *Business Process Management*, LNCS 3812, 3–17, 2005.

[41] L. Pitt. Inductive inference, DFAs, and computational complexity. *AII*, 18–44, 1989.

[42] D. Quass et al. LORE: a Lightweight Object Repository for semistructured data. *SIGMOD*, 549, 1996.

[43] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10, 2001.

[44] A. Sahuguet. Everything you ever wanted to know about DTDs, but were afraid to ask. *WebDB*, 2000.

[45] Y. Sakakibara. Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185(1):15–45, 1997.

[46] J. Clark. Trang. Multi-format schema converter. http://www.thaiopensource.com/relaxng/trang.html

[47] J. Sankey and R.K. Wong. Structural inference for semistructured data. *CIKM*, 159–166, 2001.

[48] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema part 1: structures*. 2001.

[49] E. van der Vlist. *XML Schema*. O'Reilly, 2002.

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