Fast Learning of Restricted Regular Expressions and DTDs

Dominik D. Freydenberger [*]
Institut für Informatik, Goethe-Universität
60054 Frankfurt a.M.
freydenberger@em.uni-frankfurt.de

Timo Kötzing
Max-Planck-Institute for Informatics
66123 Saarbrücken
koetzing@mpi-inf.mpg.de

ABSTRACT

We study the problem of generalizing from a finite sample to a language taken from a predefined language class. The two language classes we consider are subsets of the regular languages and have significance in the specification of XML documents (the classes corresponding to so called *chain regular expressions*, CHAREs, and to *single occurrence regular expressions*, SOREs).

The previous literature gave a number of algorithms for generalizing to SOREs providing a trade off between quality of the solution and speed. Furthermore, a fast but non-optimal algorithm for generalizing to CHAREs is known.

For each of the two language classes we give an *efficient* algorithm returning a *minimal* generalization from the given finite sample to an element of the fixed language class; such generalizations are called *descriptive*. In this sense, both our algorithms are optimal.

Keywords

subregular language learning, single occurrence regular expression, chain regular expression, descriptive generalization

1. INTRODUCTION

The present paper follows and refines an approach for XML schema inference from positive examples that was introduced by Bex et al. [3]. The basic problem setting is as follows. Given a set of XML documents, generate a schema that describes these documents, while being compact and preferably human readable.

Bex et al. approach this problem by learning deterministic regular expressions from positive examples; i. e., they consider the following problem: Given a finite set S of positive examples from an unknown target language L, find a deterministic regular expression for L. These regular expressions can immediately be used as DTDs (Document Type Definitions), and while XSDs (XML Schema Documents) require additional effort, algorithms that infer regular expressions can also be used as a component of XSD inference algorithms (see [3, 4] for further explanations). In particular, as argued in [3], the results in [11] show that XSD inference requires deep insights into regular expression inference – as Bex et al. put it, “one cannot hope to successfully infer XSDs without good algorithms for inferring regular expressions”.

Using a classical technique from Gold [9], Bex et al. prove in [2] that even the class of deterministic regular expressions is too rich to be learnable from positive data. While, strictly speaking, the learnability criterion of *Gold-style learning* as defined in [9] (which is also called *learning in the limit from positive data* or *explanatory learning*) is different from the setting in [2, 3],[1] its non-learnability results still provide valuable insights into necessary restrictions.

In particular, Gold-style learning shows that, when learning from positive data, one has to balance the need for generalization (as in most cases, a regular expression that generates exactly the example is not considered a good hypothesis) with the need to avoid overgeneralization.

While there are numerous papers on restrictions on the class of regular languages that lead to learnability, apart from a few exceptions (e. g. [6]), most of these restrictions prior to [3] have been based on properties of automata. As explained in [3], this is problematic, as even under those restrictions, converting the inferred automaton to a regular expression can lead to an exponential size increase.

In order to achieve learnability of concise deterministic regular expression, Bex et al. propose *single occurrence regular expressions* (short SOREs), regular expressions where each *terminal letter* (or *element name*) occurs at most once. These SOREs are deterministic by definition, and as an additional benefit, this restriction ensures that the length of the inferred expressions is at most linear in the number of different terminal letters.

The corresponding SORE-inference algorithm `RWR` from [3] works as follows. First, it constructs a so-called *single occurrence automaton* (short SOA, as introduced by García and Vidal [8]). `RWR` then attempts to convert the SOA step by step into a SORE. As the class of SORE-languages is a proper subset of the class of SOA-languages, this conversion is not always possible. In these cases, `RWR` attempts to *repair* the

[*]This work was done while this author was visiting the Max-Planck-Institute for Informatics in Saarbrücken.

[1]Gold-style learning uses a growing set of samples and requires that the learner converges toward a correct hypothesis in finite time, while this setting uses only a single finite set for each inference instance.

SOA, and constructs a SORE that generates a generalization of the language of the SOA. In order to generalize as little as possible, [3] suggests different orderings on the set of repair rules, as well as the variant $\texttt{RWR}^2_\ell$, which uses additional heuristics and can have an exponential running time. Nonetheless, these variants may still infer SOREs that are not inclusion-minimal generalizations of the input sample (within the class of all SOREs).

In order to deal with insufficient data, Bex et al. proposed a further restriction on SOREs, the so-called *chain regular expressions* (short: CHAREs), and introduced the corresponding inference algorithm `CRX`. Analogously to `RWR`, `CRX` may infer CHAREs that are not inclusion-minimal generalizations.

The present paper focuses on inferring SOREs and CHAREs that are inclusion-minimal generalizations. This approach to regular expression inference is based on a slightly different angle than Gold-style learning, namely on the learning paradigm of *descriptive generalization* that was introduced by Freydenberger and Reidenbach [7].

While Gold-style learning assumes that an exact representation of the target language is present in the hypothesis space, and that the learner is provided with sufficient positive information to correctly recognize the target language, descriptive generalization views the hypothesis space and the space of target languages as distinct.

For a class $\mathcal{D}$ of language representation mechanisms (e. g., a class of automata, regular expressions, or grammars[2]), a language representation $\delta \in \mathcal{D}$ is called $\mathcal{D}$*-descriptive* of a sample S if $L(\delta)$ is an inclusion-minimal generalization of S, i. e., $S \subseteq L(\delta)$ and there is no $\gamma \in \mathcal{D}$ with $S \subseteq L(\gamma) \subset L(\delta)$.

This concept allows us to define $\mathcal{D}$-descriptive generalization as a natural extension of Gold-style learning: Instead of attempting to learn an exact representation of the target language L from a sample S, the learner has to infer a representation $\delta \in \mathcal{D}$ that is $\mathcal{D}$-descriptive of L. In other words, δ is a generalization of S that is as *inclusion-minimal* as possible within $\mathcal{D}$.

Descriptive generalization explicitly separates the hypothesis space from the class of target languages, while still providing a natural quality criterion for generalization from positive examples. In the present paper, we consider the class of SOREs and the class of CHAREs as hypothesis spaces $\mathcal{D}$, and examine the problem of inferring $\mathcal{D}$-descriptive generalizations from finite samples.

We approach this problem by first computing a SOA-descriptive SOA. As we shall see, this approach has the advantages that the descriptive SOA is uniquely defined, can be computed efficiently, and its language is included in the language of every descriptive SORE or CHARE.

The main contribution of the present paper are two algorithms, `Soa2Sore` and `Soa2Chare`, that can be used to transform any given SOA into a SORE (resp. CHARE) that is SORE-descriptive (resp. CHARE-descriptive) of the language of that SOA. That is, given a sample S, these algorithms can be used to compute a generalization of S that is inclusion-minimal (or, in the terminology of [3], *optimal*) within the class of SOREs or CHAREs (respectively).

In addition to this, `Soa2Chare` and `Soa2Sore` are efficient: `Soa2Chare` runs in time $O(m)$ (compared to $O(m + n^3)$ for `CRX`), `Soa2Sore` in time $O(nm)$ (compared to $O(n^5)$ for `RWR`), where m is the number of edges and n the number of nodes in the SOA.

The paper is structured as follows. Section 2 contains some mathematical preliminaries, followed by some informative properties of the language classes considered. Section 3 discusses `CRX` as well as `RWR` and its variants in the context of descriptive regular expressions. In particular, we show that for each of these algorithms, there are samples over small alphabets where the algorithm does not compute a descriptive CHARE or SORE. Sections 4 and 5 contain the algorithms `Soa2Chare` and `Soa2Sore`, respectively, as well as proofs of their correctness and running time. Finally, Section 6 concludes the paper. For space reasons, some of the proofs were omitted.

2. PRELIMINARIES

Let $\emptyset$ denote the *empty set*, let ε denote the *empty word*. With $|x|$, we denote the length of x if x is a word, or the number of elements in x if x is a set. We use $\subseteq$ (and $\subset$) to denote the inclusion (respectively proper inclusion) of sets. The *difference* of two sets A, B is denoted by $A \setminus B$ and defined as $\{a \in A \mid a \notin B\}$. A word v is a *factor* of a word $x \in \Sigma^*$ if there exist $u, w \in \Sigma^*$ such that $x = uvw$. A *2-gram* is a factor of length 2. Let $\mathrm{alph}(w)$ denote the set of all letters occurring in a word w, and extend this to languages by defining $\mathrm{alph}(L) := \bigcup_{w \in L} \mathrm{alph}(L)$.

2.1 Introducing SORE, CHARE, SOA

This section introduces the classes of regular expressions and automata that are used in the present paper. We mostly follow the notations introduced in [3]. In particular, we use the following variant of regular expressions.

DEFINITION 1. *Let Σ be a finite alphabet (the set of* terminal letters, *also called* element names*). Every letter $a \in \Sigma$ is a regular expression, as are ε and $\emptyset$, and $L(x) = \{x\}$ for $x \in \Sigma \cup \{\varepsilon\}$, while $L(\emptyset) = \emptyset$. If α is a regular expression, then α^+ and $\alpha?$ are regular expressions, where $L(\alpha^+) = (L(\alpha))^+$ and $L(\alpha?) = L(\alpha) \cup \{\varepsilon\}$. Furthermore, if α and β are regular expressions, then $\alpha \mid \beta$ and $\alpha \cdot \beta$ are also regular expressions, with $L(\alpha \mid \beta) = L(\alpha) \cup L(\beta)$ and $L(\alpha \cdot \beta) = \{uv \mid u \in L(\alpha), v \in L(\beta)\}$.*

For sake of convenience, we sometimes omit the concatenation operator (i. e., we write $\alpha\beta$ instead of $\alpha \cdot \beta$), and add or omit parentheses. For a regular expression α, we use $\mathrm{alph}(\alpha)$ to denote the set of terminal letters that occur in α. We call two regular expressions α, β *alphabet-disjoint* if $\mathrm{alph}(\alpha) \cap \mathrm{alph}(\beta) = \emptyset$. Two regular expressions α and β are *equivalent* if $L(\alpha) = L(\beta)$. For any set $A \subseteq \Sigma$, we use the notation $ALT(A)$ to denote the regular expression $ALT(A) := (a_1 \mid \cdots \mid a_n)$, with $ALT(\emptyset) = \varepsilon$ (ALT stands for *alternation*). In a strict sense, this definition requires an ordering on the letters to be sound, but for the purpose of this paper, this is of no concern, and we assume that $ALT(A) = ALT(B)$ if $A = B$.

The full class of regular expressions is too strong both for DTDs (which allow only *deterministic regular expressions*) and for learning from positive data (which requires language classes that are sufficiently sparse, cf. [9]). As proven in [2], even the class of deterministic regular expressions is still too large to be learnable from positive data. Hence, [3] proposes the following subclasses of deterministic regular expressions.

[2]The canonical class $\mathcal{D}$ is the class of *NE-patterns*, where *descriptive patterns* were introduced by Angluin [1] in the context of exact learning from positive data. See [12] for a survey on the influence of pattern languages in this area.

Definition 2 (Sore/Chare). *A* single occurrence regular expression *(or* Sore*) is a regular expression in which each terminal letter occurs (at most) once.*

A chain regular expression *(or* Chare*) is a* Sore *of the form $f_1 \cdot \ldots \cdot f_n$ $(n \geq 0)$, where each f_i is a* chain factor*, i. e., a* Sore *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 each a_j is a terminal letter.*

In other words, a Chare consists of a concatenation of alphabet-disjoint chain factors. We illustrate these definitions with a few short examples.

Example 3. *Consider the regular expressions given as $\alpha = (a)?(b\,|\,c)^+$, $\beta = (ab)^+$, and $\gamma = abaa$. Here, α is a* Chare *(and, hence, also a* Sore*), as it consists of two alphabet-disjoint chain-factors.*

On the other hand, β is a Sore *(every letter occurs only once), but not a* Chare *(as it is not composed of chain-factors). One can easily prove that $L(\beta)$ is not a* Chare *language: Assume there exists a* Chare *β' with $L(\beta') = L(\beta)$. By definition, β' must contain a and b. if a and b are not in the same chain-factor of β', then at least one of the 2-grams ab or ba cannot occur in any word of $L(\beta')$. But if a and b are in the same chain-factor of β', the same line of reasoning implies that this chain-factor must be followed by $^+$ or $^+?$. Therefore, there are words in $L(\beta')$ that contain the 2-grams aa or bb, which contradicts $L(\beta') = L(\beta)$.*

Finally, γ is not a Sore *(and therefore not a* Chare*), and one can prove that $L(\gamma)$ is not a* Sore*-language (this is best proven using techniques that shall be introduced right after this example, hence we defer the proof of this claim to Remark 7 further down in this section).*

While the focus of this paper is on learning regular expressions, most of our technical reasoning uses the following class of automata.

Definition 4 (Soa). *Let Σ be a finite alphabet, and let $\mathtt{snk}, \mathtt{src}$ be distinct symbols that do not occur in Σ. A* single occurrence automaton *(short:* SOA*) over Σ is a finite directed graph $A = (V, E)$ such that*

(1) $\{\mathtt{src}, \mathtt{snk}\} \in V$, and $V \subseteq \Sigma \cup \{\mathtt{src}, \mathtt{snk}\}$,

(2) $\mathtt{src}$ has only outgoing edges, $\mathtt{snk}$ has only incoming edges, and every $v \in V$ lies on a path from $\mathtt{src}$ to $\mathtt{snk}$.

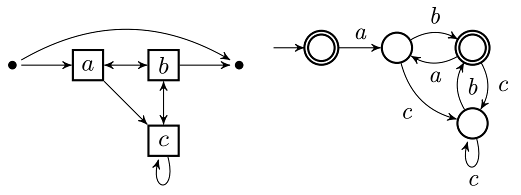
We call $\mathrm{alph}(A) := V \setminus \{\mathtt{src}, \mathtt{snk}\}$ *the set of* terminal letters *in A. We define the relation $\rightarrow_A$ on V by $\rightarrow_A := E$, and use $\rightarrow_A^+$ and $\rightarrow_A^*$ to denote the transitive and reflexive-transitive hull of $\rightarrow_A$. The language $L(A)$ that is accepted by A is the set of all words $w = a_1 \cdots a_n$ $(n \geq 0)$ such that $\mathtt{src} \rightarrow_A a_1 \rightarrow_A \cdots \rightarrow_A a_n \rightarrow_A \mathtt{snk}$.*

As usual, a *strongly connected component* of a Soa A is a non-empty and inclusion-maximal set C of vertices of A such that for all $a, b \in C$, $a \rightarrow_A^* b$ and $b \rightarrow_A^* a$ holds. A *strongly connected looped component* of a Soa A is a non-empty and inclusion-maximal set C of vertices of A such that for all $a, b \in C$, $a \rightarrow_A^+ b$ and $b \rightarrow_A^+ a$ holds. In other words, every strongly connected looped component contains exactly those vertices that are mutually reachable. Thus, a strongly connected component may be a singleton, while a singleton strongly connected *looped* component must have a self-loop.

By definition, all strongly connected looped components of a Soa are disjoint, and $\mathtt{src}$ and $\mathtt{snk}$ cannot be part of any strongly connected looped component.

Although their definition is somewhat different, it is easy to see that Soas are a subclass of DFAs. In particular, a Soa can be understood as a DFA where for each $a \in \Sigma$, there exists a characteristic state q_a such that $\delta(q, a) \in \{q_a, q_{trap}\}$ for all states $q \in Q$ (where q_{trap} is a trap state). This is illustrated by the following example.

Example 5. *In the picture below, we have a* Soa *on the left side, and the corresponding DFA to the right side.*



Both automata generate the same language as the regular expression $\alpha = ((ac^+?b)((ac^+?b) \,|\, (c^+b))^+?)?$. Note that α is not a Sore*. In fact, $L(\alpha)$ is not a* Sore*-language, but proving this using only techniques that have been introduced at this point requires considerable effort. (The most straightforward way to prove this is to use techniques that are introduced in Section 5: Apply the algorithm* `Soa2Sore` *to the* Soa*, which returns the* Sore *$(ab?c^+?)^+?$, which is not equivalent to α. By Theorem 25, this means that $L(\alpha)$ is not a* Sore *language.)*

In this paper, we frequently use Soas to approximate languages. For this, we rely on the following definition.

Definition 6. *For every $w \in \Sigma^*$, let* $\mathrm{first}(w)$ *and* $\mathrm{last}(w)$ *denote the first resp. last letter of w, and let $\mathrm{gram}_2(w)$ be the set of all 2-grams in w. We extend these functions on words to functions on languages by defining $\mathrm{first}(L) := \{\mathrm{first}(w) \mid w \in L\}$, $\mathrm{last}(L) := \{\mathrm{last}(w) \mid w \in L\}$, and $\mathrm{gram}_2(L) := \bigcup_{w \in L} \mathrm{gram}_2(w)$.*

For every language $L \subseteq \Sigma^*$, we define the Soa-approximation of *L, $\mathrm{SOA}(L)$, by $\mathrm{SOA}(L) := (V_L, E_L)$, where $V_L := \mathrm{alph}(L) \cup \{\mathtt{src}, \mathtt{snk}\}$, and E_L contains the edges*

- *$(\mathtt{src}, a)$ for every $a \in \mathrm{first}(L)$,*
- *$(a, \mathtt{snk})$ for every $a \in \mathrm{last}(L)$,*
- *(a, b) for all $a, b \in \Sigma$ with $ab \in \mathrm{gram}_2(L)$,*
- *$(\mathtt{src}, \mathtt{snk})$ if $\varepsilon \in L$.*

Using this terminology, the approach for Soa-learning presented in [8] can be summarized as follows. Given a finite set S, compute $\mathrm{SOA}(S)$. In [3], the resulting algorithm is called `2T-INF`. Furthermore, as computing $\mathrm{SOA}(L)$ is only as hard as computing $\mathrm{first}(L)$, $\mathrm{last}(L)$, and $\mathrm{gram}_2(L)$, note that $\mathrm{SOA}(L)$ can be constructed for language from classes that are larger than the classes of finite or regular languages, e. g., for context-free languages.

It is easy to see from the definition that $L(\mathrm{SOA}(L)) \supseteq L$ holds for every language L (in fact, we shall see in Proposition 14 that $L(\mathrm{SOA}(L))$ is always the least general approximation of L that is possible with Soa). This inclusion can be proper as follows.

REMARK 7. *Note that even for finite languages L, the equality $L(\mathrm{SOA}(L)) = L$ is not necessary; e. g., consider $L = \{abaa\}$ (from Example 3). Then $\mathrm{SOA}(L)$ contains an edge from* `src` *to a, from a to b, from b to a, from a to itself, and from a to* `snk`*. Hence, $aa \in L(\mathrm{SOA}(L))$, while $aa \notin L$. This also proves that L is not a* SOA*-language (and, as claimed in Example 3, not a* SORE*-language.)*

As Example 5 illustrates, there are SOA-languages that are not SORE-languages. On the other hand, we have that every SORE-language is a SOA-language (in other words, the SOA-approximation of a SORE-language is exact).

LEMMA 8 ([3], PROOF OF PROPOSITION 9). *Given any* SORE *α, we have $L(\mathrm{SOA}(L(\alpha)) = L(\alpha)$.*

It is easy to see that $\mathrm{SOA}(L(\alpha))$ can be derived from every SORE α (in fact, even every regular expression α) by deriving the sets of first letters, last letters, and 2-grams in $L(\alpha)$ from the expression α (we already did this in Remark 7).

Lemma 8 allows us to define $\mathrm{SOA}(\alpha)$ as a notational shorthand for $\mathrm{SOA}(L(\alpha))$. Similarly, we use $\rightarrow_\alpha$ to denote the relation $\rightarrow_{\mathrm{SOA}(\alpha)}$.

More importantly, we shall use Lemma 8 to develop a handy syntactic characterization of the inclusion for SOREs (and CHAREs), which is based on the inclusion of SOAs. We say that a SOA A *covers* a SOA B if A is a supergraph of B – in other words, $\mathrm{alph}(A) \supseteq \mathrm{alph}(B)$ holds, and $a \rightarrow_B b$ implies $a \rightarrow_A b$ for all $a, b \in \mathrm{alph}(B)$. This definition leads to the following characterization of SOA-inclusion.

LEMMA 9 ([8], THEOREM 3.1). *For every pair A, B of* SOA*s, $L(A) \subseteq L(B)$ holds if and only if A is covered by B.*

Although Lemma 9 is stated in [8] without proof (the authors just cite García's PhD thesis), it is easily proven considering the definition of $\mathrm{SOA}(L)$.

Combining Lemma 9 with Lemma 8, we are able to characterize inclusion of SOREs as follows.

LEMMA 10. *For every pair α, β of* SORE*s, $L(\alpha) \subseteq L(\beta)$ holds if and only if $\mathrm{SOA}(\alpha)$ is covered by $\mathrm{SOA}(\beta)$.*

This obviously implies that two SOREs (or CHAREs) are equivalent if their corresponding SOAs are equivalent. More importantly, Lemma 10 provides a simple syntactic and characteristic criterion for inclusion. While the algorithms in Sections 4 and 5 do not check for inclusion, their correctness proofs make heavy use of the fact that SORE-inclusion depends on the presence of edges in the corresponding SOA.

Before we introduce the other central definition of this paper in Section 2.2, we discuss some concepts which will be useful, although not quite as significant.

One can verify with little effort that the classes of SOA-, SORE-, or CHARE-languages are not closed under many of the operations that are commonly studied in formal language theory (e. g., concatenation, union, complementation, intersection with regular languages, morphism, inverse morphism). One of the few operations under which each of these classes is closed, and which we shall use, is projection.

Let Σ be an alphabet. A *projection* from Σ to $T \subseteq \Sigma$ is a morphism $\pi_T : \Sigma^* \rightarrow T^*$ that is defined by $\pi_T(x) := x$ for all $x \in T$, and $\pi_T(x) := \varepsilon$ for all $x \in \Sigma \setminus T$. We extended this to languages canonically, i. e., $\pi_T(L) := \{\pi_T(w) \mid w \in L\}$.

LEMMA 11. *The classes of* SORE-*,* CHARE-*, and* SOA-*languages are closed under projection.*

The proof was omitted for space reasons.

The main approach in the present paper (as well as in [3]) is converting SOAs into SOREs or CHAREs. During this process, it is occasionally convenient to work with a model that can be viewed as an intermediary step between a SOA and a regular expression.

DEFINITION 12. *A* generalized single occurrence automaton *(or* generalized SOA*) is a finite graph $A = (V, E)$ such that*

(1) $\{\mathtt{src}, \mathtt{snk}\} \subseteq V$, and all vertices in $V \setminus \{\mathtt{src}, \mathtt{snk}\}$ are pairwise alphabet-disjoint SORE*; and*

(2) the edge relation E is such that `src` *has only outgoing edges;* `snk` *has only incoming edges, and every $v \in V$ lies on a path from* `src` *to* `snk`*.*

The relations $\rightarrow_A$, $\rightarrow_A^*$, $\rightarrow_A^+$ on V are defined analogously to (non-generalized) SOA*. We extend* alph *to generalized* SOA*s by defining $\mathrm{alph}(A) := \bigcup_{v \in V \setminus \{\mathtt{src},\mathtt{snk}\}} \mathrm{alph}(v)$.*

The language $L(A)$ is defined to be the set of all $w \in \mathrm{alph}(A)^*$ for which there exist a $n \geq 0$, nodes $v_1, \ldots, v_n \in V \setminus \{\mathtt{src}, \mathtt{snk}\}$, and words $w_1, \ldots, w_n \in \mathrm{alph}(A)^*$ such that $\mathtt{src} \rightarrow_A v_1 \rightarrow_A \cdots \rightarrow_A v_n \rightarrow_A \mathtt{snk}$, $w = w_1 \cdots w_n$, and $w_i \in L(v_i)$ holds for every $1 \leq i \leq n$.

Note that generalized SOAs accept the same class of languages as SOAs.

2.2 Descriptivity

This section introduces the notion of descriptive expressions and automata, which is one of the central aspects of the present paper.

DEFINITION 13. *Let $\mathcal{D}$ be a class of regular expressions or finite automata over some alphabet Σ. A $\delta \in \mathcal{D}$ is called $\mathcal{D}$-*descriptive *of a non-empty language $S \subseteq \Sigma^*$ if $L(\delta) \supseteq S$, and there is no $\gamma \in \mathcal{D}$ such that $L(\delta) \supset L(\gamma) \supseteq S$.*

In other words, an expression or automation that is $\mathcal{D}$-descriptive of a language S generates a language that is a generalization of S that is $\subseteq$-minimal within languages described by elements of $\mathcal{D}$.

If the class $\mathcal{D}$ is clear from the context, we simply write *descriptive* instead of $\mathcal{D}$-descriptive. As stated in [8] (using quite different terminology), for every finite language S, $\mathrm{SOA}(S)$ is SOA-descriptive of S. This extends to infinite languages as well; for SOREs and CHAREs, we can also prove the existence of descriptive regular expressions:

PROPOSITION 14. *Let Σ be a finite alphabet. For every language $L \subseteq \Sigma^*$, $\mathrm{SOA}(L)$ is* SOA*-descriptive of L, and there exist a* SORE*-descriptive* SORE *δ_s and a* CHARE*-descriptive* CHARE *δ_c.*

The proof was omitted for space reasons.

In particular, this means that the algorithm `2T-INF` from [3] that was mentioned in the previous section can be used to compute SOA-descriptive SOAs for finite sample sets. Moreover, this shows that constructing a descriptive SOA for an arbitrary language L is as merely as hard as computing the sets $\mathrm{first}(L)$, $\mathrm{last}(L)$, and $\mathrm{gram}_2(L)$.

As we shall see, computing descriptive SOREs or CHAREs is less straightforward. First, note that the first part of the proof of Proposition 14 implies the following observation:

Class	num of languages	max num descriptive for sample	max num edges to add for descr
CHARE	$n!\, 2^{2n} \leq c(n) \leq n!\, 2^{3n}$	$\geq n!$	$\Theta(n^2)$
SORE	$n!\, 2^{3n-r\log n} \leq s(n) \leq n!\, 2^{7n}$	$\geq 2^n$	$\Theta(n^2)$
SOA	$2^{n^2+O(n)}$	1	$\times$

Table 1: A summary of the numbers presented in Proposition 16. For each of the classes of languages generated by Chares, Sores, and Soas, the table lists the number of different languages in the class, the maximum number of descriptive expressions or automata for a given sample $S \subset \Sigma^*$, and the maximum number of edges that need to be added to SOA(S) in order to obtain a Soa that corresponds to a descriptive Chare or Sore. In all cases, n denotes the size of Σ.

COROLLARY 15. *Let Σ be a finite alphabet, and let $L \subseteq \Sigma^*$. For every* SORE *(or* CHARE*) δ that is* SORE*-descriptive (resp.* CHARE*-descriptive) of L, $L(\delta) \supseteq L(\mathrm{SOA}(L))$ holds.*

Hence, if some SORE (or CHARE) is descriptive of a language L, it must be descriptive of $L(\mathrm{SOA}(L))$ as well. This allows us to compute descriptive SOREs and CHAREs not from a sample L, but from its SOA-approximation $\mathrm{SOA}(L)$.

Furthermore, if $L(\mathrm{SOA}(L))$ is not a SORE-language (or not a CHARE-language), a SOA for some SORE descriptive of L can be obtained in principle from $\mathrm{SOA}(L)$ by adding new edges. As only a finite number of edges need to be added, and SOA-inclusion can be decided easily (cf. Lemma 9), the main question is whether this can be done efficiently. But as it can be necessary to add a substantial number of new edges in order to turn a SOA into a SOA that corresponds to a descriptive expression (see Proposition 16 just below), a brute force approach is probably not advisable.

The next proposition lists these and other numbers about counting and descriptive SOREs and CHAREs. These results are summarized in Table 1. Recall that regular expressions are called equivalent if they accept the same language.

PROPOSITION 16. *Let n be the number of alphabet symbols. We have the following, for some constant r.*

(1) The number of pairwise non-equivalent CHARE*s is $c(n)$ with $n!\, 2^{2n} \leq c(n) \leq n!\, 2^{3n}$.*

(2) The number of pairwise non-equivalent SORE*s is $s(n)$ with $n!\, 2^{3n-r\log n} \leq s(n) \leq n!\, 2^{7n}$.*[3]

(3) There is a sample $S \subseteq \Sigma^*$ such that S has 2^n pairwise non-equivalent descriptive SORE*s.*

(4) There is a sample $S \subseteq \Sigma^*$ such that S has $n!$ pairwise non-equivalent descriptive CHARE*s.*

(5) There is a SOA *with $\Theta(n)$ edges such that a descriptive* SORE *with a minimal number of edges in the corresponding* SOA *has $\Theta(n^2)$ edges.*

(6) There is a SOA *with $\Theta(n)$ edges such that a descriptive* CHARE *with a minimal number of edges in the corresponding* SOA *has $\Theta(n^2)$ edges.*

The proof was omitted for space reasons.

In particular, note that Proposition 16 also demonstrates that a given sample can have numerous different descriptive SOREs (or CHAREs). Note that the number of different CHARE- and SORE-languages can be better approximated using more advanced tools from combinatorics. Finally, if we are only interested in the number of different such language *modulo renaming of the terminal letters*, then the same bounds without the factor $n!$ hold.

[3]Note that [2, Proof of Theorem 3.1] gives that any SORE-language has a SORE of length at most $10n-4$, which gives a bound of $2^{O(n\log n)}$.

3. DESCRIPTIVITY VS. CRX AND RWR

Proposition 16 demonstrates that the number of non-equivalent descriptive SOREs (or CHAREs) for a sample can be exponential in the size of the alphabet. Therefore, the present paper only examines the question how a single descriptive SORE (or CHARE) can be found for a sample, instead of looking for an enumeration of all these expressions.

As explained in Section 2.2 (in particular, Corollary 15), descriptive CHAREs and SOREs can be obtained from the descriptive SOA, and moreover, for every language L and every SORE α, $L(\alpha) \supseteq L(\mathrm{SOA}(L))$ must hold. This observation motivates our inference approach for SOREs and CHAREs: Given a sample S, first compute the SOA-descriptive single-occurrence automaton $\mathrm{SOA}(S)$, using `2T-INF`. As explained in [8], this can be done in time $O(ln)$, where $l := \sum_{s\in S} |s|$, and $n := |\,\mathrm{alph}(S)|$.

Using the algorithm `Soa2Chare` (Section 4) or `Soa2Sore` (Section 5), $\mathrm{SOA}(S)$ is then turned into a descriptive CHARE or SORE (respectively). Before we discuss these algorithms and the respective proofs in detail, we observe that the algorithms `CRX` and `RWR` and its variants from [3] do not always compute descriptive CHAREs or SOREs.

For the CHARE-algorithm `CRX`, this is quite easy to see: As pointed out in [3] (as a remark after Theorem 35), on the sample $S = \{abc, ade, abe\}$, the algorithm `CRX` returns the CHARE $a?b?c?d?e?$, while $\delta := a(b\,|\,d)(c\,|\,e)$ is a better approximation of S. (In fact, we shall be able to see that δ is not only better, but CHARE-descriptive. This can be verified by observing that δ is the output of `Soa2Chare` on $\mathrm{SOA}(S)$, and referring to Theorem 19 further down.)

The proofs for the non-descriptivity of the SORE-algorithm `RWR` and its variants require more effort, and can be found in the following section.

3.1 RWR-Variants and Descriptivity

In this section we give theorems regarding properties of `RWR`-variants (we refer the reader to [3] for details on all variants). In particular, we show that every variant fails to find a descriptive SORE on some input.

In [3, Algorithm 3] an Algorithm `RWR` ("Rewrite with Repairs") was given to turn a given SOA (derived in the canonical way from and input sample) into a generalizing SORE, by rewriting the SOA step by step. This algorithm was proven in [3] to turn any SOA in an equivalent SORE, if existent; if, at some point in the run of `RWR`, no rewrite rules are applicable, the algorithm will make a generalization step by applying a "repair rule". The four "repair rules" of RWR are as follows, given the current SOA A. For simplicity, we give a modification of the rules, where less edges are added. However, for the cases we use in this section, these rules are equivalent to the original set.

Repair $r \mid s$ If there are two nodes r and s of A which share a successor or a predecessor, add edges to A to make all successors of r or s successors of both r and s; similarly with the predecessors.

Repair $r \cdot s?$ If there are two nodes r and s of A such that r is the only predecessor of s, add edges to A to make all successors of r or s (except s) successors of both r and s.

Repair $r? \cdot s$ If there are two nodes r and s of A such that s is the only successor of r, add edges to A to make all predecessors of r or s (except r) predecessors of both r and s.

Repair $r? \cdot s?$ Let r and s be nodes of A such that s is a successor of r; add edges to A to make all successors of r or s successors of both r and s; similarly with the predecessors. Furthermore, for all predecessors u of r and all successors v of s, add an edge from u to v.

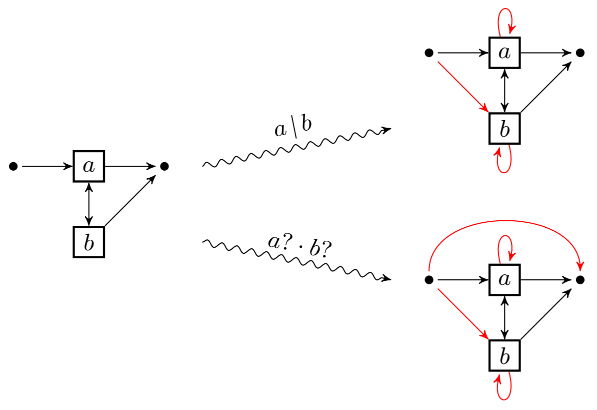
The authors of [3] prove that RWR (with the original repair rules) always terminates in $O(n^5)$ steps (where $n = |\Sigma|$) and gives a SORE which generalizes the input SOA. They also suggest that these rules are checked for applicability in the given order, but admit that different situations might call for different rules (in particular, they note that the outcome of RWR is not alway descriptive).

Next, we formally show that RWR does not always return a descriptive SORE.

THEOREM 17. *For Σ a finite alphabet with $|\Sigma| \geq 3$ and all orderings of the repair rules of RWR, there is a (finite) set of samples $S \subseteq \Sigma^*$ such that RWR on S produces a* SORE *which is not* SORE*-descriptive.*

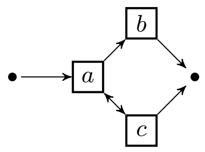
Proof. Let $a, b, c \in \Sigma$ be three different symbols from Σ.

First, consider the sample $\{aba, ab\}$. The corresponding SOA does not allow rewrite rules and requires repair; below this SOA is depicted, along with two possible repairs, corresponding to the two possible repairs "Repair $a \mid b$" and "Repair $a? \cdot b?$".



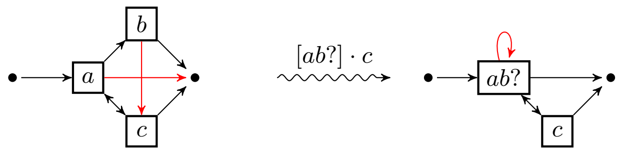
The SOAs resulting from the two repairs accept $(a \mid b)^+$ and $(a \mid b)^*$, respectively, which is not descriptive of $\{aba, ab\}$, as witnessed by $\delta_1 := (a(b?))^+$ (a SORE which accepts the given sample aba and ab, but not, for example, b, which is accepted by any of the SOAs derived from repair rules above).

Second, consider the sample $S = \{ab, ac, acac\}$. The corresponding SOA A is depicted as follows.



A descriptive SORE for S is $\delta_2 := (a(b \mid c))^+$, which we prove as follows. In comparison to A, the SOA that corresponds to δ_2 adds only a single edge, the edge from a to b. So the only possibility for a SORE-language $L(\gamma)$ with $L(A) \subseteq L(\gamma) \subset L(\delta_2)$ is $L(A)$ itself. However, $L(A)$ is not a SORE-language, which can be seen, just as in Proposition 16, by applying either the SORE-construction algorithm RWR from [3] or our algorithm Soa2Sore from Section 5 (which both compute a SORE equivalent to a given SOA, if existent) and observing a strict generalization. Hence, δ_2 is SORE-descriptive of S. (We note without proof that $(ac?)^+b?$ is another SORE that is descriptive of S. Necessarily, its language is incomparable to $L(\delta_2)$.)

An application of "Repair $a \cdot b?$" on A and then, after rewriting, of "Repair $[ab?] \cdot c?$" gives the following.



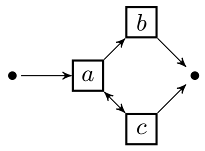
This SOA corresponds to the SORE $(ab?c?)^+$, and its language is a strict superset of $L(\delta_2)$ (for example abc is accepted by the former and not the latter).

Deceiving the rule "Repair $r? \cdot s$" is symmetric to deceiving "Repair $r \cdot s?$". $\square$

In [3], Bex et al. also propose a variant of RWR that is called RWR^2_ℓ, which uses a natural number ℓ as a branching parameter. The algorithm explores the (recursive) outcomes of the best ℓ candidates for a repair rule, choosing the ones that lead to a minimal number of words of length at most $2n$ ($= 2|\Sigma|$) in the language accepted by the resulting SORE.

THEOREM 18. *For all $\ell > 0$ there is a finite alphabet Σ with $|\Sigma| = 3\ell$ and a finite set of samples $S \subseteq \Sigma^*$ such that RWR^2_ℓ on S produces a* SORE *which is not* SORE*-descriptive.*

Proof. We first assume $\ell = 1$; consider again the sample $\{ab, ac, acac\}$ with the following corresponding SOA.



The three applicable repair rules are $b \mid c$, $a \cdot b?$ and $a \cdot c?$ (plus some rules of the type "$r? \cdot s?$", which explode the number of accepted words). This leads to the following SOAs.

RegEx α	$\lvert L(\alpha)^{\leq 6}\rvert$	exp growth basis	recurrence	base cases $n \in \{1,2,3\}$
$(ab \mid ac)^+$	14	$\sqrt{2} \approx 1.41$	$2f_\alpha(n-2)$	$0, 2, 0$
$(abc \mid ac)^+$	7	≤ 1.33	$f_\alpha(n-2) + f_\alpha(n-3)$	$0, 1, 1$
$(a \mid ac)^+ b?$	51	$(1+\sqrt{5})/2 \approx 1.62$	$f_\alpha(n-1) + f_\alpha(n-2)$	$1, 3, 5$

Table 2: Properties of the languages discussed in the proof of Theorem 18. For each regular expression α, $f_\alpha(n)$ denotes the number of words in $L(\alpha)$ of length n; given in the table are the number of words of length at most 6 accepted by α, the constant c such that f_α grows roughly as c^n, the recurrence relation for the $(f_\alpha(n))_{n\in\mathbb{N}}$, as well as $f_\alpha(n)$ for $n \in \{1,2,3\}$.

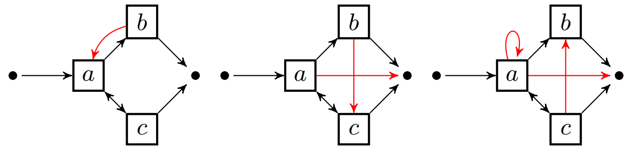
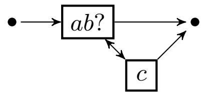


Table 2 gives an overview of the properties of these three SoAs.

Thus, we see that second possibility accepts a minimal number of words of length at most 6 ($= 2|\Sigma|$), which means that only this option will be explored, the first and the third will be discarded. After rewriting by `RWR`, this results in the following SoA.



The minimal repair for this results in $(ab?c?)^+$, which is *not* descriptive as witnessed by $(a(b \mid c))^+$ as in the proof of Theorem 17.

For $\ell > 1$, we use ℓ independent copies of the sample used for $\ell = 1$ (i.e., using different alphabet symbols). Thus, $\mathtt{RWR}^2_\ell$ will fail on at least one of these copies. □

4. DESCRIPTIVE CHARES

In this section, we give the first main algorithm of this paper, `Soa2Chare`, which efficiently computes descriptive CHAREs for given SoAs.

4.1 The CHARE algorithm

The algorithm `Soa2Chare` uses a number of subroutines, which are written with a dot-notation similar to some modern object oriented programming languages. For example "A.contract(U, ℓ)" denotes the application of the subroutine "contract" to the SoA A with parameters U and ℓ. For a given SoA A, we let A.`src` and A.`snk` denote the source and the sink of A, respectively. The following subroutines are used in `Soa2Chare`.

- "contract" on SoA A takes a subset U of vertices of A and a label ℓ. The procedure modifies A such that all vertices of U are contracted to a single vertex and labeled ℓ (edges are moved accordingly).
- "constructLevelOrder" on SoA $A = (V, E)$ assumes that A is acyclic and assigns a *level number* to every vertex $v \in V$, where the level number of a node $v \in V$ is defined to be the length of the longest path from A.`src` to v. Hence, A.`src` is on level number 0, and for every other node v, the level number is one more than the highest level number of the immediate successors of v.
- "isSkipLevel" on SoA A and a level number i returns `true` if level i is a *skip level*. A level i is a skip level if there exist a nodes $u, v \in V$ with (respective) level numbers $j_u < i$ and $j_v > i$ such that $u \to_A v$. In other words, one can skip level i by transitioning from u to v.

Algorithm 1: `Soa2Chare`

```
1  Input: SOA $A = (V, E)$;
2  while $A$ has a cycle do
3      Let $U$ be a strongly connected looped
       component of $A$;
4      $A$.contract$(U, ALT(U)^+)$;
5  $A$.constructLevelOrder();
6  result $\leftarrow \varepsilon$;
7  for $i$ = 1 to (level number of $A$.snk) $- 1$ do
8      $B \leftarrow$ all vertices with level number $i$ and $^+$;
9      $C \leftarrow$ all vertices with level number $i$ and no $^+$;
10     foreach $\alpha \in B$ do
11         if $A$.isSkipLevel(i) or $|B|+|C|>1$ then
               result $\leftarrow$ result $\cdot\, \alpha?$;
12         else result $\leftarrow$ result $\cdot\, \alpha$;
13     if $|C| > 0$ then
14         if $A$.isSkipLevel(i) or $|B|>0$ then
15             result $\leftarrow$ result $\cdot\, ALT(C)?$;
16         else result $\leftarrow$ result $\cdot\, ALT(C)$;
17 return result;
```

Note that the use of "contract" can turn the SoA into a generalized SoA. Intuitively speaking, the algorithm `Soa2Chare` works as follows:

(1) Replace each strongly connected looped component $A \subseteq V$ with a vertex that is labeled with the regular expression $ALT(A)^+$. This turns A into a (possibly generalized) SoA that is a DAG.

(2) Every node in the DAG is assigned a level number.

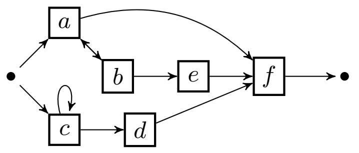
(3) Every level is turned into one or more chain-factors. If a level contains more than one non-letter node, or if a level is a skip level, ? is appended to every chain-factor on that level.

The following theorem states that `Soa2Chare` can be used to compute CHARE-descriptive CHAREs in a highly efficient manner.

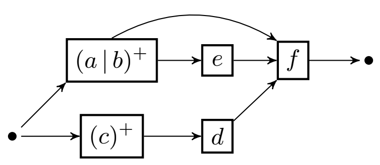
THEOREM 19. *For any given SoA A,* `Soa2Chare` *finds a CHARE that is CHARE-descriptive of $L(A)$ in time $O(m)$, where m is the number of transitions of A.*

Before we discuss the proof of Theorem 19 in Section 4.2, we illustrate the behavior of `Soa2Chare` with an Example.

EXAMPLE 20. *Let $S = \{abaf, abef, ccdf\}$. The corresponding* SOA*, $\mathrm{SOA}(S)$, is depicted as follows.*



First, `Soa2Chare` *removes all cycles by contracting strongly connected looped components. This leads to the following generalized* SOA.



Apart from the levels for A.`src` *and A.*`snk`*, this generalized* SOA *has three levels: The first level with the nodes $(a \mid b)^+$ and $(c)^+$, the second level with the nodes d and e, and the third level with the node f. As there is an edge between $(a \mid b)^+$ and f, the second level is a skip level. Thus, the levels lead to the respective* CHAREs *$(a \mid b)^+?(c)^+?$, $(e \mid d)?$, and f, which are concatenated to $(a \mid b)^+?(c)^+?(e \mid d)?f$. By Theorem 19, this* CHARE *is* CHARE*-descriptive of S.*

4.2 Proof of Theorem 19

Proof. We first prove termination and running time, followed by the proof of correctness. Note that in this Section, for simplicities sake and in contrast to Section 5, we do not distinguish between a node its label.

Termination and running time.

Termination is obvious, as the two loops (in lines 2 and 7) are executed only a bounded number of times.

Let n denote the number of vertices and m denote the number of edges in the input SOA. In the **while**-loop in line 2, the input SOA is transformed into an acyclic generalized SOA. Using Tarjan's algorithm (cf. [5]), this part can be realized in time $O(m + n)$.

Computing the level order and annotating, for each level, whether that level is a skip level, can also be done in time $O(m + n)$, analogously to a topological sorting.

Finally, each node in the generalized SOA is turned into a chain factor. This takes time $O(n)$. Hence, the individual steps sum up to a time of $O(m+n)$, which results in a total time of $O(m)$, as $n \leq m$ holds by definition.

Correctness.

First, it is quite easy to see that `Soa2Chare` computes a CHARE. Note that, in order to prove that this CHARE is descriptive of the sample S, we do not need to argue about every CHARE γ with $L(\gamma) \supseteq S$, but only about those with $L(\mathtt{Soa2Chare}(\mathrm{SOA}(S))) \supseteq L(\gamma) \supseteq S$.

This allows us to use Lemma 10 from two directions: On the one hand, every edge (and hence, every path) that is present in $\mathrm{SOA}(S)$ must be present in $\mathrm{SOA}(\gamma)$, on the other hand, $\mathrm{SOA}(\gamma)$ must not contain any edges that do not occur in $\mathrm{SOA}(\delta)$.

Before we consider the main part of the proof, we first develop some technical tools that deal with strongly connected looped components.

LEMMA 21. *Let α be a* CHARE*. A set $A \subseteq \mathrm{alph}(\alpha)$ is a strongly connected looped component in $\mathrm{SOA}(\alpha)$ if and only if α contains a chain factor of the form $ALT(A)^+$ or $ALT(A)^+?$.*

The proof was omitted for space reasons.

As `Soa2Chare` turns every strongly connected looped component A into a chain factor $ALT(A)$, we observe that `Soa2Chare` does not change these components.

COROLLARY 22. *Let Σ be an alphabet. For every finite and nonempty set $S \subseteq \Sigma^*$, and every set $A \subseteq \mathrm{alph}(S)$, the following holds. A is a strongly connected looped component in $\mathrm{SOA}(S)$ if and only if A is a strongly connected looped component in $\mathrm{SOA}(\mathtt{Soa2Chare}(SOA(S)))$.*

Finally, according to Lemma 10, this immediately leads to the following observation:

COROLLARY 23. *Let $S \subseteq \Sigma^*$ be a finite set, and let $\delta := \mathtt{Soa2Chare}(\mathrm{SOA}(S))$. For every* CHARE *$\gamma$ with $L(\delta) \supseteq L(\gamma) \supseteq S$, $\mathrm{SOA}(\gamma)$ must contain exactly the same strongly connected looped components as $\mathrm{SOA}(S)$ and $\mathrm{SOA}(\delta)$.*

We now posses all the tools we need to execute the main element of the proof of correctness of `Soa2Chare`.

LEMMA 24. *Let Σ be an alphabet, let $S \subseteq \Sigma^*$ be a nonempty set, and let $\delta := \mathtt{Soa2Chare}(\mathrm{SOA}(S))$. Then $L(\delta) = L(\gamma)$ holds for every* CHARE *γ with $L(\delta) \supseteq L(\gamma) \supseteq S$.*

The proof was omitted for space reasons.

Lemma 24 implies that there is no CHARE γ such that $L(\mathtt{Soa2Chare}(\mathrm{SOA}(S))) \supset L(\gamma) \supseteq S$. As we have, by definition, $L(\mathtt{Soa2Chare}(\mathrm{SOA}(S))) \supseteq S$, we get that the result of `Soa2Chare` on $\mathrm{SOA}(S)$ is CHARE-descriptive of S, which concludes the proof of correctness. $\square$

5. DESCRIPTIVE SORES

In this section, we give the second main algorithm of this paper, which efficiently computes descriptive SOREs for given SOAs.

5.1 SORE Algorithm

As in Section 4, we use dot-notation to denote the application of subroutines. As in Section 4, for a given SOA A, we let A.`src` and A.`snk` denote the source and the sink of A, respectively.

- "contract" on SOA A takes a subset U of vertices of A and a label ℓ. The procedure modifies A such that all vertices of U are contracted to a single vertex and labeled ℓ (edges are moved accordingly). The procedure returns the newly created vertex.
- "extract" on SOA A takes as argument a set of vertices U (of A); it does not modify A, but returns a new SOA with copies of all vertices of U as well as two new vertices for source and sink; all edges between vertices of U are copied, all vertices in U having an incoming

edge (in A) from outside of U have now an incoming edge from the new source, and all vertices in U having an outgoing edge (in A) to outside of U have now an outgoing edge to the new sink.

- "first" returns all vertices v such that the only predecessor of v is the source.
- "addEpsilon" on SOA A adds a new vertex labeled ε; all outgoing edges from the source to vertices that have more than one predecessor (vertices, that are not in the first-set) are redirected via this new vertex.
- "exclusive" on SOA A on argument v (a vertex of A) returns the set of all vertices u such that, on any path from the source to the sink that visits u, v is necessarily visited previously. Intuitively, the exclusive set of a vertex v is the set of all vertices exclusively reachable from v, not from any other vertex incomparable to v.
- Finally, the most difficult subroutine is called "bend" and is used to prepare the treatment of strongly connected looped components of the input SOA A. First, it computes the set W of all vertices reachable from the source without passing through (or ending with) vertices which are predecessors of the sink. Then it redirects (bends) all transitions directed from an element *outside* of W to a successor of the source to point to the sink instead. With other words, we redirect all transitions from an element c to an element $a \in A.\mathrm{succ}(A.\mathtt{src})$ to now transition to $A.\mathtt{snk}$ iff for all words u such that uc is a path in A, we have that uc contains an element $b \in A.\mathrm{pred}(A.\mathtt{snk})$. In particular, all elements of $A.\mathrm{pred}(A.\mathtt{snk})$ do not transition to elements from $A.\mathrm{succ}(A.\mathtt{snk})$. See Example 26 for an illustration.

Furthermore, we use the following three subroutines for the creation of labels.

- "plus" on label ℓ returns $(\ell)^+$.
- "concatenate" on labels ℓ and ℓ' returns $\ell \cdot \ell'$.
- "or" on labels ℓ and ℓ' returns $\ell \,|\, \ell'$.

The algorithm `Soa2Sore` is given in Algorithm 2. On a more intuitive level, the algorithm performs the following phases.

(1) Recurse on all strongly connected looped components; replace each with a vertex, labeled with the result of the recursion.

(2) After the SOA is a directed acyclic graph (DAG), focus on the set F of all vertices which can be reached from the source directly, but not via other vertices; make sure that there are no vertices which can be reached directly and via other vertices (if necessary, add an auxiliary node labeled ε).

(3) Recurse on the sets of vertices exclusively reachable from a vertex in F and contract these sets to vertices labeled with the result of the recursion.

(4) Combine vertices of F with "or," recurse again on what is exclusively reachable from this new vertex.

(5) Once only one item is left in F, split it off and recurse on the remainder.

Algorithm 2: `Soa2Sore`

```
Input: SOA $A = (V, E)$;
Output: SORE minimally generalizing $L(A)$;
if $|V| = 2$ then return $\varepsilon$;
else if $A$ has a cycle then
    Let $U$ be a strongly connected looped
    component of $A$;
    $B_0 \leftarrow A.\mathrm{extract}(U).\mathrm{bend}()$;
    $A.\mathrm{contract}(U, \mathrm{plus}(\mathtt{Soa2Sore}(B_0)))$;
else if $A.\mathrm{succ}(A.\mathtt{src}) \neq A.\mathrm{first}()$ then
    $A.\mathrm{addEpsilon}()$;
else if $|A.\mathrm{first}()| = 1$ then
    Let $v$ be the only successor of $\mathtt{src}$;
    $U \leftarrow V \setminus \{A.\mathtt{src}, v, A.\mathtt{snk}\}$;
    $\ell \leftarrow v.\mathrm{label}()$;
    $\ell' \leftarrow \mathtt{Soa2Sore}(A.\mathrm{extract}(U))$;
    return concatenate$(\ell, \ell')$;
else if $\exists v \in A.\mathrm{first}()$: $A.\mathrm{exclusive}(v) \neq \{v\}$ then
    Let $v$ be such that $A.\mathrm{exclusive}(v) \neq \{v\}$;
    $U \leftarrow A.\mathrm{exclusive}(v)$;
    $A.\mathrm{contract}(U, \mathtt{Soa2Sore}(A.\mathrm{extract}(U)))$;
else
    Let $u, v \in A.\mathrm{first}()$ with $u \neq v$
    s.t. $A.\mathrm{reach}(u) \cap A.\mathrm{reach}(v)$ is $\subseteq$-maximal;
    $A.\mathrm{contract}(\{u, v\}, \mathrm{or}(u.\mathrm{label}(), v.\mathrm{label}()))$;
return $\mathtt{Soa2Sore}(A)$;
```

Note that the algorithm introduces ? by way of constructing "or ε." This can be cleaned up by postprocessing the resulting SORE.

The following theorem states the correctness and the running time of the algorithm.

THEOREM 25. *The algorithm* `Soa2Sore`*, given a* SOA A *as input, finds a descriptive* SORE *for* $L(A)$ *in time* $O(nm)$*, where* n *is the number of alphabet symbols used in* A*, and* m *is the number of transitions in* A*. Furthermore, this algorithm produces a* SORE *such that the corresponding* SOA *has the same strongly connected components as the input* SOA*, and the same set of successors of the source.*

Before we get to the proof of Theorem 25, we give two examples of `Soa2Sore`. The first example illustrates how strongly connected looped components are treated. The second illustrates the use of "exclusive".

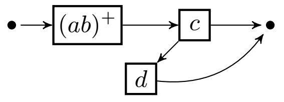
EXAMPLE 26. *Consider the following* SOA.

$$\begin{tikzcd}
 & & \boxed{b} \arrow[dr] & & \\
\bullet \arrow[r] & \boxed{a} \arrow[ur, leftrightarrow] & & \boxed{c} \arrow[r] \arrow[dl] & \bullet \\
 & & \boxed{d} \arrow[ul] & &
\end{tikzcd}$$

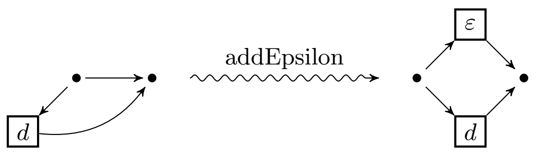
The labeled vertices of this SOA *consist of a single strongly connected looped component, an application of "bend" computes the set* $W = \{a, b\}$*, which leads to the following* SOA.

$$\begin{tikzcd}
 & & \boxed{b} \arrow[dr] & & \\
\bullet \arrow[r] & \boxed{a} \arrow[ur, leftrightarrow] & & \boxed{c} \arrow[r] \arrow[dl] & \bullet \\
 & & \boxed{d} \arrow[urr, bend right] & &
\end{tikzcd}$$

After resolving the strongly connected looped component containing a and b (all other are not "looped") and contract, we get the following.



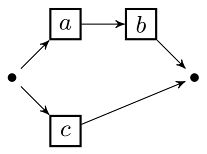
We can split off the first node twice now (as line 10 applies twice), recursing finally on the remaining Soa *as follows.*



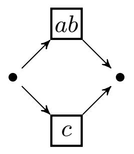
This results in $d \mid \varepsilon$, or, equivalently, $d?$. Going back through the recursions, we get

$$((ab)^+cd?)^+.$$

Example 27. *Consider now the following* Soa.



For this Soa, *line 16 applies and recurses on the upper arc; after contraction, this gives*



which results in $(ab) \mid c$ as desired (no generalizations were made).

5.2 Proof of Theorem 25

In this section we are concerned with proving Theorem 25. We start with a lemma which is used in its proof.

Lemma 28. *There is a function f on* Sore*s such that, for each* Sore *α, $L(f(\alpha)^+) = L(\alpha^+) \setminus \{\varepsilon\}$ and, for all $a \in \alpha.\text{succ}(\alpha.\texttt{src})$ and $c \in \alpha.\text{pred}(\alpha.\texttt{snk})$ we have $c \not\to_{f(\alpha)} a$.*

The proof was omitted for space reasons.

We are now ready to prove Theorem 25.

Proof. Let a Soa A be given. We proceed by first reasoning about *termination* and *running time*. After that, we will inductively show *correctness*, by assuming all recursive calls to be correct.

Termination and running time.

We refer to [5] for standard graph algorithms, such as finding strongly connected (looped) components.

As the algorithm never introduces self-loops, it is easy to see that the running time on a Soa A is at most the running of A with all self-loops removed plus n. Thus, it suffices to show that `Soa2Sore` has a running time of $O(nm)$ on self-loop free Soas.

We first bound the running time on acyclic Soas. We topologically sort the vertices of A (this takes $O(m)$ time). We will now iteratively construct and annotation of all the vertices of G with subsets of $A.\text{first}()$, corresponding to what vertices they are reachable from. We start by annotating each vertex of G that corresponds to a vertex $v \in A.\text{first}()$ with $\{v\}$ and all others with $\emptyset$ (in time $O(n)$). We now iterate through all vertices u from first to last in the topological sort of G and, for each successor w of u, we add to the current annotation of w the annotation of u (assuming unit time for this kind of set operations; overall, this will then take $O(m)$ time). This results in the desired annotation of A, in a total of $O(m)$ time.

Extracting the "exclusive" sets for all elements of $A.\text{first}()$ can now be done in $O(m)$ time. From these annotations we can also find a pair of vertices with $\subseteq$-maximal reach-sets in time $O(m)$.

Any two additions of ε-nodes are balanced in between by splitting off of a starting node, as given in line 10. As for all other operations, the algorithm can make at most n contractions; hence, there can be only $O(n)$ recursive calls. This results in an overall time of $O(nm)$ for acyclic Soas.

We now turn to the general case. Finding strongly connected looped components takes time $O(m)$, using well-known algorithms, for example Tarjan's algorithm. `Soa2Sore` first recurses on all strongly connected looped components, and then on the directed acyclic graph obtained by contracting all strongly connected looped components. The "bend" operation on a strongly connected looped component splits this component, as no vertex linked to the sink can now reach any of the elements of the "first" set. The running time is maximized when the recursions are as unbalanced as possible; this happens, when each "bend" operation splits off only one vertex, and the remaining Soa is still strongly connected. This results in splitting off n times, with a time of $O(m)$ for finding strongly connected looped components each time, plus the final work on acyclic Soas.

This shows that the overall running time is $O(nm)$.

Correctness.

The statements about strongly connected components and the successors of the source are straightforward. Furthermore, it is clear that the result is a Sore.

We show the following statement about `Soa2Sore` by induction (by termination, we assume the induction hypothesis to hold for all recursive calls). Let a generalized Soa A' be given, let A be a copy of the structure of A' where all labels are replaced with single distinct symbols. Suppose that the claim holds for all recursive calls that `Soa2Sore` makes on A. Let $\delta = \texttt{Soa2Sore}(A)$ and let γ be a Sore such that $L(A) \subseteq L(\gamma) \subseteq L(\delta)$.

We distinguish a number of different cases, depending on which clause was used for `Soa2Sore`(A). We will show $L(\delta) = L(\gamma)$ in each case.

Case 1: The clause in line 3 was used.
This case is trivial.

Case 2: The clause in line 4 was used.
Let U be as chosen in line 4. Let $B_0 = A.\text{extract}(U).\text{bend}()$; let z be a symbol not in $\text{alph}(A)$ and $B_1 = A.\text{contract}(U, z)$. Let $\hat{\delta_0} = \texttt{Soa2Sore}(B_0)$ and let $\delta_0 = \hat{\delta_0}^+$. We let δ_1 be `Soa2Sore`(B_1).

Let T be the syntax tree of γ. For each vertex x of T, we call x *plussed* iff inserting a $^+$ in T at x does not change the language accepted by T.

Claim 1. There is a plussed vertex x in T such that, for the subtree γ_0 rooted at x, we have $\text{alph}(\gamma_0) = \text{alph}(U)$. The proof was omitted for space reasons.

Let f be as shown existent in Lemma 28, and let x be the plussed vertex highest up in T such that $\text{alph}(x) = U$. Let $\hat{\gamma_0}$ be the subtree of γ rooted at x; let γ_1 be derived from γ by substituting the subtree at x with a leaf labeled z if $\varepsilon \notin L(\gamma_0)$ and $(z \,|\, \varepsilon)$ otherwise. Let $\gamma_0 = f(\hat{\gamma_0})$. Clearly, it suffices to show that $L(\gamma_0) = L(\delta_0)$ and $L(\gamma_1) = L(\delta_1)$.

Claim 2. $L(B_1) \subseteq L(\gamma_1) \subseteq L(\delta_1)$.
Proof of Claim 2. In order to avoid unnecessary case distinctions, we first introduce two new and distinct terminal symbols $\rhd$ and $\lhd$, where $\rhd$ is used as a word-start symbol, and $\lhd$ as a word-end symbol. To this end, we define $\gamma_1' := \rhd\gamma_1\lhd$ (δ_1', δ', and γ' are defined analogously). In addition to this, we define a Soa B_1' with $L(B_1') = \rhd L(B_1)\lhd$ and a Soa B' with $L(B') = \rhd L(B)\lhd$. (This is easily done by inserting new nodes label $\rhd$ or $\lhd$ between the source and its successors, or the sink and its predecessors, respectively).

We first prove $L(B_1') \subseteq L(\gamma_1') \subseteq L(\delta_1')$. After this is established, the claim follows by observing that projection preserves inclusion.

$\underline{L(B_1') \subseteq L(\gamma_1')}$: Let $a, b \in \text{alph}(B_1') \setminus \{z\}$ and suppose $a \rightarrow_{B_1'} b$. We have $a \rightarrow_{B'} b$, and, hence, $a \rightarrow_{\gamma'} b$. From the definition of γ_1' it is now easy to see that $a \rightarrow_{\gamma_1'} b$.

Let $a \in \text{alph}(B_1') \setminus \{z\}$ and suppose $a \rightarrow_{B_1'} z$. Thus, there is a $b \in U$ such that $a \rightarrow_{B'} b$, and, hence, $a \rightarrow_{\gamma'} b$. From the definition of γ_1' it is now easy to see that $a \rightarrow_{\gamma_1'} z$.

Let $b \in \text{alph}(B_1') \setminus \{z\}$ and suppose $z \rightarrow_{B_1'} b$. Thus, there is an $a \in U$ such that $a \rightarrow_{B'} b$, and, hence, $a \rightarrow_{\gamma'} b$. From the definition of γ_1 it is now easy to see that $z \rightarrow_{\gamma_1'} b$.

$\underline{L(\gamma_1') \subseteq L(\delta_1')}$: Let $a, b \in \text{alph}(\gamma_1') \setminus \{z\}$ and suppose $a \rightarrow_{\gamma_1'} b$. From the definition of γ_1' it is now easy to see that $a \rightarrow_{\gamma'} b$, and, hence, $a \rightarrow_{\delta'} b$. Thus, we get $a \rightarrow_{\delta_1'} b$.

Let $a \in \text{alph}(\gamma_1') \setminus \{z\}$ and suppose $a \rightarrow_{\gamma_1'} z$. Thus, there is a $b \in U$ such that $a \rightarrow_{\gamma'} b$, and, hence, $a \rightarrow_{\delta'} b$. We have now $a \rightarrow_{\delta_1'} z$.

Let $b \in \text{alph}(\gamma_1') \setminus \{z\}$ and suppose $z \rightarrow_{\gamma_1'} b$. Thus, there is an $a \in U$ such that $a \rightarrow_{\gamma'} b$, and, hence, $a \rightarrow_{\delta'} b$. We have now $z \rightarrow_{\delta_1'} b$.

Hence, $L(B_1') \subseteq L(\gamma_1') \subseteq L(\delta_1')$, which is equivalent to $\rhd L(B_1)\lhd \subseteq \rhd L(\gamma_1)\lhd \subseteq \rhd L(\delta_1)\lhd$. As inclusion is preserved under projection, this implies $\pi_T(L(B_1')) \subseteq \pi_T(L(\gamma_1')) \subseteq \pi_T(L(\delta_1'))$ which proves the claim (for $T := \Sigma \setminus \{\rhd, \lhd\}$).
$\square$ (for Claim 2)

Thanks to the claim we can now apply the induction hypothesis to see that $L(\gamma_1) = L(\delta_1)$.

Similarly, we now show γ_0 and δ_0 to be equivalent by showing $L(B_0) \subseteq L(\gamma_0) \subseteq L(\delta_0)$. From the induction hypothesis we know that $B_0.\text{succ}(B_0.\texttt{src}) = \delta_0.\text{succ}(\delta_0.\texttt{src})$; this shows that $\gamma_0.\text{succ}(\gamma_0.\texttt{src})$ has to coincide with these sets.

Claim 3. We have that

$$B_0.\text{pred}(B_0.\texttt{snk}) \subseteq \gamma_0.\text{pred}(\gamma_0.\texttt{snk}) \subseteq \delta_0.\text{pred}(\delta_0.\texttt{snk}).$$

The proof was omitted for space reasons.

Lastly, we turn to pairs of elements from U.

Claim 4. On $\text{alph}(U)$, $\rightarrow_{B_0}$ is a subrelation of $\rightarrow_{\gamma_0}$, which in turn is a subrelation of $\rightarrow_{\delta_0}$.
Proof of Claim 4. This is straightforward, using the properties of f taken from Lemma 28. $\square$ (for Claim 4)

This finishes showing $L(B_0) \subseteq L(\gamma_0) \subseteq L(\delta_0)$; thus, using the induction hypothesis, $L(\gamma_0) = L(\delta_0)$. This finishes the reasoning for this case.

<u>Case 3:</u> The clause in line 8 was used.
This case is trivial from the induction hypothesis, as the language is not changed by the addEpsilon() method.

<u>Case 4:</u> The clause in line 10 was used.
Let v be the only successor of $A.\texttt{src}$; let $a = v.\text{label}()$. Note that a is the only successor of $\gamma.\texttt{src}$. Let $U = \text{alph}(\delta) \setminus \{a\}$. As A does not have a strongly connected looped component, neither does Soa(γ); thus, we have $L(\gamma) = a \cdot \pi_U(L(\gamma))$. Let γ' equal γ with a replaced by ε and $\delta' = \texttt{Soa2Sore}(A.\text{extract}(U))$. Then we have $L(A.\text{extract}(U)) \subseteq L(\gamma') \subseteq L(\delta')$ and the claim follows by induction.

<u>Case 5:</u> The clause in line 16 was used.
We now know that A is cycle free and, thus, δ' does not contain a "$^+$". Therefore, without loss of generality, γ does not contain a "$^+$" either.

Let v be as chosen in line 16 and $a = v.\text{label}()$. Let $U = A.\text{exclusive}(v)$.

Let $B_0 = A.\text{extract}(U)$; let z be a symbol not in $\text{alph}(A)$ and $B_1 = A.\text{contract}(U, z)$. Let $\delta_0 = \texttt{Soa2Sore}(B_0)$ and let $\delta_1 = \texttt{Soa2Sore}(B_1)$. By the induction hypothesis, we have that $\delta_0.\text{first}() = \{a\}$. Thus, any word in $L(\gamma) \subseteq L(\delta)$ that contains an element of U has to start with an a.

Claim 5. There is a subtree γ_0 of γ such that $\text{alph}(\gamma_0) = U$. The proof was omitted for space reasons.

Let γ_0 be a subtree of γ such that $\text{alph}(\gamma_0) = U$; let γ_1 be derived from γ by substituting the γ_0 with a leaf labeled z. Note that $\varepsilon \notin L(\gamma_0)$ because of $A.\text{succ}(A.\texttt{snk}) = A.\text{first}()$.

We now clearly get $L(B_0) \subseteq L(\gamma_0) \subseteq L(\delta_0)$ and $L(B_1) \subseteq L(\gamma_1) \subseteq L(\delta_1)$. Thus, this case follows from the induction hypothesis, similarly to Case 2.

<u>Case 6:</u> The clause in line 20 was used.
In this case we know that $|A.\text{first}()| > 1$, as no other case applies. Furthermore, we will use without mention that A is cycle free.

Let u, v as chosen in line 20. Let z be a symbol not in $\text{alph}(A)$. Let $B = A.\text{contract}(\{u, v\}, z)$. Let δ_0 be $\texttt{Soa2Sore}(B)$.

Let $a = u.\text{label}()$ and $b = v.\text{label}()$. From $u, v \in \delta.\text{first}()$ we have that there is a subtree β of γ with "or" at the root and a and b are in different child trees.

Claim 6. $L(\beta)$ is a set of letters. The proof was omitted for space reasons.

From the claim we get, without loss of generality, that $(a \,|\, b)$ is a subexpression of γ; thus, $\beta = (a \,|\, b)$. Let γ_0 be derived from γ by substituting β with z. Clearly, we now have $L(B) \subseteq L(\gamma_0) \subseteq L(\delta_0)$. From the induction hypothesis we get $L(\gamma_0) = L(\delta_0)$; thus, $L(\gamma) = L(\delta)$. $\square$

6. CONCLUSIONS AND FURTHER WORK

This paper proposes a strategy for inferring descriptive SOREs and descriptive CHAREs: First, use `2T-INF` to compute a descriptive SOA, then use `Soa2Sore` or `Soa2Chare` to turn this automaton into a SORE or a CHARE.

In [3], Bex et al. state that their schema inference algorithms "outperform existing algorithms in accuracy, conciseness, and speed". Considering the results presented in Sections 3 to 5, the authors of the present paper feel confident to suggest that their new strategies outperform the algorithms from [3] at least with respect to both accuracy and speed. An experimental evaluation of the algorithms is planned for the near future. This will also give the opportunity to evaluate the quality of the results of the algorithms, for example with respect to different conciseness measures or how well they describe the target language.

We now discuss possible extensions, and possible directions for further work.

In order to overcome the problem that SOREs and CHAREs cannot count (beyond the trivial case of distinguishing between 0 and 1), Bex et al. [3] propose extending those models with numerical predicates, which can be obtained by post-processing. It is easy to see that this extension can also be adapted to the approaches in the present paper.

If one is willing to fix a probability distribution on the sample, the learning algorithms could be adapted to feature a variant of stochastic finite learning (introduced by Rossmanith and Zeugmann [13]). This could lead to inference algorithms that do not need to process the whole input, which might be interesting for very large datasets.

From the authors' point of view, the following problem is probably the most interesting: In [2], Bex et al. examine the inference of *k-occurence regular expressions* (short k-OREs); regular expressions where each terminal letter occurs at most k times. (Hence, SOREs are 1-OREs). Is it possible to extend `Soa2Sore` to deterministic k-OREs for some $k \geq 2$, or `Soa2Chare` to the corresponding extension of CHAREs (where letters are allowed to occur up to k times)?

It seems that one would need to develop not only a good generalization of SOAs, but also a "good" inclusion criterion, preferably syntactic. This conjecture is based on the following observation: While the results in the present paper make no direct use of the results and techniques that Freydenberger and Reidenbach [7] developed for descriptive generalization of pattern languages, both papers rely heavily on the fact that the inclusion problem for the respective language classes has a syntactic criterion for inclusion.

The proofs on descriptive generalization of pattern languages in [7] rely on the fact that inclusion for terminal-free E-pattern languages is characterized by the existence of a morphism which maps the pattern that generates the superlanguage to the pattern that generates the sublanguage. This criterion is a versatile tool to prove the nonexistence of a (pattern) language between the target language and the language of a descriptive pattern. While the proofs of the present paper cannot make any *direct* use of the proofs from [7], the approaches are similar *conceptually.* In particular, the line of reasoning in which the correctness proofs of `Soa2Chare` and `Soa2Sore` use the fact that the inclusion problem for SOREs (and CHAREs) is characterized by the covering of the respective SOAs is structurally similar to the proofs for pattern languages.

Moreover, although deciding whether such a pattern morphism exists is NP-complete, the techniques in [7] are not affected by the computational hardness. Hence, the hardness results on the decidability of the k-ORE-inclusion problem presented by Martens et al. [10] do not exclude the existence of such a criterion. This leaves room for hope that `Soa2Sore` can be extended to k-OREs with $k \geq 2$.

Acknowledgements

The authors wish to thank the anonymous referees for their helpful remarks.

7. REFERENCES

[1] D. Angluin. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21(1):46–62, 1980.

[2] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Transactions on the Web*, 4(4):14:1–14:32, 2010.

[3] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems*, 35(2):11:1–11:47, 2010.

[4] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *Proc. VLDB 2007*, pages 998–1009, 2007.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw Hill, 2nd edition, 2001.

[6] H. Fernau. Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4):521–541, 2009.

[7] D. D. Freydenberger and D. Reidenbach. Inferring descriptive generalisations of formal languages. In *Proc. COLT 2010*, pages 194–206, 2010.

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

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

[10] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing*, 39(4):1486–1530, 2009.

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

[12] Y. K. Ng and T. Shinohara. Developments from enquiries into the learnability of the pattern languages from positive data. *Theoretical Computer Science*, 397(1–3):150–165, 2008.

[13] P. Rossmanith and T. Zeugmann. Stochastic finite learning of the pattern languages. *Machine Learning*, 44(1–2):67–91, 2001.