Noname manuscript No.
(will be inserted by the editor)

Fast Learning of Restricted Regular Expressions and DTDs

Dominik D. Freydenberger · **Timo Kötzing**

Received: date / Accepted: date

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 gives 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 of descriptivity, 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. [4]. 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.

D. D. Freydenberger
Johann-Wolfgang-Goethe-Universität, Frankfurt am Main, Germany
E-mail: freydenberger@em.uni-frankfurt.de

T. Kötzing
Friedrich-Schiller-Universität, Jena, Germany
E-mail: timo.koetzing@uni-jena.de

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 element type declarations in 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 [4,5] for further explanations). In particular, as argued in [4], the results in [16] 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 [12], Bex et al. prove in [3] 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 [12] (which is also called *learning in the limit from positive data* or *explanatory learning*) is different from the setting in [3, 4],[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. [7]), most of these restrictions prior to [4] have been based on properties of automata. As explained in [4], 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 [4] works as follows. First, it constructs a so-called *single-occurrence automaton* (short SOA, as introduced by García and Vidal [11]). `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 SOA, and constructs a SORE that generates a generalization of the language of the SOA. In order to generalize as little as possible, [4] suggests different orderings on the set of repair rules, as well as the variant $\mathtt{RWR}^2_\ell$, which uses additional heuristics and can have an exponential running time. Nonetheless, these variants may still infer SOREs that are not

[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.

inclusion-minimal generalizations of the input sample (within the class of all SOREs).

In order to deal with insufficient data, Bex et al. propose a further restriction on SOREs, the so-called *chain regular expressions* (short: CHAREs), and introduce 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 [10].

While Gold-style learning usually 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), a language representation $\delta \in \mathcal{D}$ is called *$\mathcal{D}$-descriptive* of a sample S if its language $\mathcal{L}(\delta)$ is an inclusion-minimal generalization of S, i. e., $S \subseteq \mathcal{L}(\delta)$ and there is no $\gamma \in \mathcal{D}$ with $S \subseteq \mathcal{L}(\gamma) \subset \mathcal{L}(\delta)$. To the authors' knowledge, the first class $\mathcal{D}$ for which the existence of descriptive representations was examined is the class of *NE-patterns*, where Angluin [1] introduces the notion of *descriptive patterns* in the context of exact learning from positive data (see [19] for a survey on the influence of pattern languages in this area). While the other mentioned example mechanisms are probably more familiar to many readers, most research on descriptive representations has focused on various classes of pattern languages (see [9, 10] for some examples).

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.

As in [4], 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 [4], *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 vertices 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. In Section 6, we discuss the use of these algorithms for less restricted language classes, while Section 7 contains some example DTDs that were generated by a prototype implementation of the algorithms. Finally, Section 8 concludes the paper.

A preliminary version of this article appeared as [8]. Apart from some minor changes, the present version was improved as follows.

- All proofs that were omitted from [8] have been included.
- A mistake in the algorithm for finding descriptive SOREs (more precisely: in its subroutine "bend") was fixed.
- Sections 6 and 7 were added.

2 Preliminaries

Let $\emptyset$ denote the *empty set* and 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 *digram* is a factor of length 2. Let $\mathrm{term}(w)$ denote the set of all letters occurring in a word w, and extend this to languages by defining $\mathrm{term}(L) := \bigcup_{w \in L} \mathrm{term}(w)$.

The concatenation operator is extended to languages by defining $L_1 \cdot L_2 := \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$ for languages L_1 and L_2. For every language L, we define $L^0 := \{\varepsilon\}$, $L^{n+1} := L \cdot L^n$ for all $n \geq 0$, and $L^+ := \bigcup_{n \geq 1} L^n$.

2.1 Introducing SORE, CHARE, SOA

This section introduces the classes of regular expressions and automata used in this paper. We mostly follow the notations of [4].

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 and defines the* *language $\mathcal{L}(a) := \{a\}$. Furthermore, ε and $\emptyset$ are regular expressions, with languages $\mathcal{L}(\varepsilon) := \{\varepsilon\}$ and $\mathcal{L}(\emptyset) := \emptyset$. If α is a regular expression, then α^+ and $\alpha?$ are regular expressions, where $\mathcal{L}(\alpha^+) := (\mathcal{L}(\alpha))^+$ and $\mathcal{L}(\alpha?) := \mathcal{L}(\alpha) \cup \{\varepsilon\}$. Furthermore, if α and β are regular expressions, then $\alpha \,|\, \beta$ and $\alpha \cdot \beta$ are also regular expressions, with $\mathcal{L}(\alpha \,|\, \beta) := \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$ and $\mathcal{L}(\alpha \cdot \beta) := \{uv \mid u \in \mathcal{L}(\alpha), v \in \mathcal{L}(\beta)\}$.*

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 $\mathrm{term}(\alpha)$ to denote the set of terminal letters that occur in α. We call two regular expressions α, β *alphabet-disjoint* if $\mathrm{term}(\alpha) \cap \mathrm{term}(\beta) = \emptyset$. Two regular expressions α and β are *equivalent* if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$. For any set $A = \{a_1, \dots, a_n\} \subseteq \Sigma$ $(n \geq 1)$, we use the notation $ALT(A)$ to denote the regular expression $ALT(A) := (a_1 \,|\, \cdots \,|\, a_n)$, 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. [12]). As proven in [3], even the class of deterministic regular expressions is still too large to be learnable from positive data. Hence, [4] proposes the following subclasses of deterministic regular expressions.

Definition 2 (Sore/Chare) *Let Σ be a finite alphabet. A* single-occurrence regular expression *(or* Sore*) is a regular expression over Σ in which each terminal letter occurs (at most) once.*

A chain regular expression *(or* Chare*) is a* Sore *over Σ 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.*

A language L is called a Sore *language (or a* Chare *language) if there exists a* Sore *(or a* Chare*) α with $\mathcal{L}(\alpha) = L$.*

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 $\alpha := (a)?(b \,|\, c)^+$, $\beta := (ab)^+$, 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). In fact, not only is β not a* Chare*, one can also prove that $\mathcal{L}(\beta)$ is not a* Chare *language.*[2]

[2] The proof uses a kind of pumping argument. Assume there exists a Chare β' with $\mathcal{L}(\beta') = \mathcal{L}(\beta)$. 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 digrams ab or ba cannot occur in any word of $\mathcal{L}(\beta')$. 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 $\mathcal{L}(\beta')$ that contain the digrams aa or bb, which contradicts $\mathcal{L}(\beta') = \mathcal{L}(\beta)$.

Finally, γ is not a SORE *(and therefore not a* CHARE*), and one can prove that $\mathcal{L}(\gamma)$ is not a* SORE *language*[3]*.*

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 $\mathrm{term}(A) := V \setminus \{\mathtt{src}, \mathtt{snk}\}$ *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 $\mathcal{L}(A)$ that is accepted by A is the set of all words $w = a_1 \cdots a_n$ ($n \geq 0$) such that $\mathtt{src} \rightarrow_A a_1 \rightarrow_A \cdots \rightarrow_A a_n \rightarrow_A \mathtt{snk}$. A language L is called a* SOA *language if there exists a* SOA *A $\mathcal{L}(A) = L$. Two* SOA*s A and B are* equivalent *if $\mathcal{L}(A) = \mathcal{L}(B)$.*

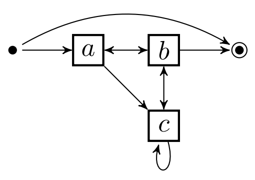
In order to ease understanding, we use specific language for vertices v in a SOA depending on the context: If the context is that of automata, we refer to it as a *state*; in the context of graph operations as a *vertex*.

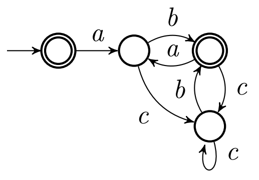
A *strongly connected component* of a SOA A is a non-empty and inclusion-maximal set C of vertices of A such that for all $a, b \in C$, $a \rightarrow_A^* b$ and $b \rightarrow_A^* a$ holds. A *strongly connected looped component* of a SOA A is a non-empty and inclusion-maximal set C of vertices of A such that for all $a, b \in C$, $a \rightarrow_A^+ b$ and $b \rightarrow_A^+ a$ holds. In other words, if $\rightarrow_A^+$ is interpreted as the reachability relation in A, every strongly connected looped component contains exactly those vertices that are mutually reachable. Thus, a strongly connected component may be a singleton without an edge, 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 converted into a DFA by labeling every edge with the state that it points to. Further, every state that has an edge to the sink is made a final 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. Note that, for depicting* SOA*s, we use the symbol "$\bullet$" as a symbol for the source, and "$\circledcirc$" as a symbol for the sink.*

[3] This is best proven using techniques that shall be introduced further down: In Remark 8, we observe that $\mathcal{L}(\mathrm{SOA}(\mathcal{L}(\gamma))) \neq \mathcal{L}(\gamma)$. According to Corollary 16, this implies that $\mathcal{L}(\gamma)$ is not a SOA language. As every SORE language is a SOA language (according to Lemma 9), $\mathcal{L}(\gamma)$ is not a SORE language.





Both automata generate the same language as the regular expression $\alpha := ((ac^+?b)((ac^+?b) \mid (c^+b))^+?)?$.

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* $\text{first}(w)$ *and* $\text{last}(w)$ *denote the first resp. last letter of* w, *and let* $\text{gram}_2(w)$ *be the set of all digrams in* w. *We extend these functions on words to functions on languages by defining* $\text{first}(L) := \{\text{first}(w) \mid w \in L\}$, $\text{last}(L) := \{\text{last}(w) \mid w \in L\}$, *and* $\text{gram}_2(L) := \bigcup_{w \in L} \text{gram}_2(w)$.

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

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

Using this terminology, the approach for SOA learning presented in [11] can be summarized as follows. Given a finite set S, compute $\text{SOA}(S)$. In [4], the resulting algorithm is called `2T-INF`. The following observation follows immediately from Definition 6.

Corollary 7 *For every language* L, *computing* $\text{SOA}(L)$ *is only as hard as computing* $\text{first}(L)$, $\text{last}(L)$, *and* $\text{gram}_2(L)$.

Hence, $\text{SOA}(L)$ can be constructed for languages from classes that are larger than the classes of finite or regular languages, e. g., for context-free languages; see Section 6.2 for a detailed discussion.

It is easy to see from the definition that $\mathcal{L}(\text{SOA}(L)) \supseteq L$ holds for every language L (in fact, we shall see in Proposition 15 that $\mathcal{L}(\text{SOA}(L))$ is always the least general approximation of L that is possible with a SOA language). This inclusion can be proper as follows.

Remark 8 *Note that even for finite languages* L, *the equality* $\mathcal{L}(\text{SOA}(L)) = L$ *is not necessary; e. g., consider* $L = \{abaa\}$. *Then* $\text{SOA}(L)$ *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,* $aa \in \mathcal{L}(\text{SOA}(L))$, *while* $aa \notin L$.

There are SOA languages that are not SORE languages. One example is the language $\mathcal{L}(\alpha)$ from Example 5, but proving this using only techniques that have been introduced at this point requires considerable effort.[4] 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 9 ([4], proof of Proposition 9) *Given any* SORE *α, we have* $\mathcal{L}(\mathrm{SOA}(\mathcal{L}(\alpha))) = \mathcal{L}(\alpha)$.

Moreover, according to Corollary 7, $\mathrm{SOA}(\mathcal{L}(\alpha))$ can be derived directly from every SORE α.

Lemma 9 allows us to define $\mathrm{SOA}(\alpha)$ as a notational shorthand for $\mathrm{SOA}(\mathcal{L}(\alpha))$. Similarly, we use $\rightarrow_\alpha$ to denote the relation $\rightarrow_{\mathrm{SOA}(\alpha)}$.

More importantly, we shall use Lemma 9 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, $\mathrm{term}(A) \supseteq \mathrm{term}(B)$ holds, and $a \rightarrow_B b$ implies $a \rightarrow_A b$ for all $a, b \in \mathrm{term}(B)$. This definition leads to the following characterization of SOA inclusion.

Lemma 10 ([11], Theorem 3.1) *For every pair A, B of* SOA*s, $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ if and only if A is covered by B.*

Although Lemma 10 is stated in [11] without proof (the authors cite García's PhD thesis), it is easily proven considering the definition of $\mathrm{SOA}(L)$.

Combining Lemma 10 with Lemma 9, we are able to characterize inclusion of SOREs as follows.

Lemma 11 *For every pair α, β of* SORE*s, $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\beta)$ 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 11 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.

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 all these classes is closed 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 12 *The classes of* SORE-*,* CHARE-*, and* SOA *languages are closed under projection.*

[4] 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 27, this means that $\mathcal{L}(\alpha)$ is not a SORE language.

Proof. Let $T \subseteq \Sigma$. Regarding SOREs, consider an arbitrary SORE α over Σ. For every letter $a \in \mathrm{term}(\alpha)$ with $a \notin T$, replace a with ε, and call the resulting expression α'. As α was a SORE, α' is a SORE as well, and it is easily seen that $\mathcal{L}(\alpha') = \pi_T(\mathcal{L}(\alpha))$. Hence, $\pi_T(\mathcal{L}(\alpha))$ is a SORE language.

Regarding CHAREs, consider an arbitrary CHARE β over Σ. By definition, there exist pairwise alphabet-disjoint chain factors $f_1, \dots, f_n$ $(n \geq 0)$ with $\beta = f_1 \cdot \ldots \cdot f_n$. We now define $A_i := \mathrm{term}(f_i)$ and $T_i := A_i \cap T$ for each $1 \leq i \leq n$. For each $1 \leq i \leq n$, we define chain factors f'_i and f''_i in the following way: If $T_i = \emptyset$, let $f'_i := f''_i := \varepsilon$. Otherwise, let

$$f'_i := \begin{cases} ALT(T_i) & \text{if } f_i = ALT(A_i) \text{ or } f_i = ALT(A_i)?, \\ ALT(T_i)^+ & \text{if } f_i = ALT(A_i)^+ \text{ or } f_i = ALT(A_i)^+?. \end{cases}$$

and

$$f''_i := \begin{cases} f'_i? & \text{if } A_i \not\subseteq T \text{ or } \varepsilon \in \mathcal{L}(f_i), \\ f'_i & \text{if } A_i \subseteq T \text{ and } \varepsilon \notin \mathcal{L}(f_i). \end{cases}$$

Finally, let $\beta'' := f''_1 \cdot \ldots \cdot f''_n$, and remove all chain factors $f''_i = \varepsilon$ from β''. Again, it is easy to see that $\mathcal{L}(\beta'') = \pi_T(\mathcal{L}(\beta))$, which means that $\pi_T(\mathcal{L}(\beta))$ is a CHARE language.

Finally, regarding SOAs, consider an arbitrary SOA A over Σ. We construct a SOA A' from A by iteratively removing letters; i. e., in each step, a letter $a \in (\mathrm{term}(A) \setminus T)$ and its associated edges are deleted, and for every pair of vertices u and v such that $u \to_A a$ and $a \to_A v$ holds, an edge (u, v) is added. Then $\mathcal{L}(A') = \pi_T(\mathcal{L}(A))$; hence, $\pi_T(\mathcal{L}(A))$ is a SOA language. □

The main approach in the present paper (as well as in [4]) 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 13 *Let Σ be a finite alphabet, and let $\mathtt{snk}, \mathtt{src}$ be distinct symbols that do not occur in Σ. A* generalized single-occurrence automaton *(or* generalized SOA*) over Σ is a finite directed 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 SOREs; and
(2) the edge relation E is such that $\mathtt{src}$ has only outgoing edges; $\mathtt{snk}$ has only incoming edges, and every $v \in V$ lies on a path from $\mathtt{src}$ to $\mathtt{snk}$.

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

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

Class	num of languages	descriptive	edges to add
Chare	$n!\,2^{2n} \le c(n) \le n!\,2^{3n}$	$\ge n!$	$\Theta(n^2)$
Sore	$n!\,2^{3n-r\log n} \le s(n) \le n!\,2^{7n}$	$\ge 2^n$	$\Theta(n^2)$
Soa	$2^{n^2+O(n)}$	1	$\times$

Table 1 A summary of the numbers presented in Proposition 18. 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.

Note that generalized Soas accept the same class of languages as Soas: As the Sores are required to be alphabet-disjoint, every generalized Soa can be transformed into a Soa by replacing each Sore with its Soa.

Just as for Soas, we again use specific language for vertices of generalized Soas depending on context. In addition to the words *state* and *vertex*, in the context of manipulating the Sore v (since every vertex is a Sore), we talk about the Sore as the *label of* v.

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 14 *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 $\mathcal{L}(\delta) \supseteq S$, and there is no $\gamma \in \mathcal{D}$ such that $\mathcal{L}(\delta) \supset \mathcal{L}(\gamma) \supseteq S$.*

In other words, an expression or automaton 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 [11] (using different terminology), for every finite language S, 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. Note that this proof is non-constructive; in later sections we will be concerned with efficiently finding descriptive Chares and Sores.

Proposition 15 *Let Σ be a finite alphabet. For every language $L \subseteq \Sigma^*$, SOA(L) is Soa-descriptive of L, and there exist a Sore-descriptive Sore δ_s and a Chare-descriptive Chare δ_c.*

Proof. We begin with the claim for Soas. First, note that every edge in SOA(L) corresponds to a first letter, a last letter, or a digram of a word in L. Hence, these edges must occur in every Soa A with $\mathcal{L}(A) \supseteq L$. By Lemma 10, this means that for every such Soa A, $\mathcal{L}(A) \supseteq \mathcal{L}(\text{SOA}(L)) \supseteq L$. In particular, there is no Soa A with $\mathcal{L}(\text{SOA}(L)) \supset \mathcal{L}(A) \supseteq L$, as this would imply $\mathcal{L}(\mathrm{SOA}(L)) \supset \mathcal{L}(A) \supseteq \mathcal{L}(\mathrm{SOA}(L))$. Therefore, $\mathrm{SOA}(L)$ is descriptive of L.

Regarding the second claim, let $\mathcal{D} \in \{\textsc{Chare}, \textsc{Sore}\}$ and assume that there is a language L over some finite alphabet Σ such that no expression $\alpha \in \mathcal{D}$ is $\mathcal{D}$-descriptive of L. This implies that there is an infinite sequence $(\beta_i)_{i \geq 0}$ of expressions from $\mathcal{D}$ with $\alpha = \beta_0$, and $\mathcal{L}(\beta_i) \supset \mathcal{L}(\beta_{i+1}) \supseteq L$ for all $i \geq 0$. This contradicts the fact that there is only a finite number of non-equivalent SOREs (and, hence, CHAREs) over Σ. Hence, for every language L, a CHARE-descriptive CHARE and a SORE-descriptive SORE must exist. $\square$

An immediate consequence of Proposition 15 is the following observation.

Corollary 16 *For every language L, $\mathcal{L}(\mathrm{SOA}(L)) = L$ iff L is a* SOA *language.*

More importantly, Proposition 15 implies that the algorithm `2T-INF` from [4] that was mentioned in the previous section can be used to compute SOA-descriptive SOAs for finite sample sets. Moreover, together with Corollary 7, this shows that constructing a descriptive SOA for an arbitrary language L is 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 15 implies the following observation.

Corollary 17 *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, $\mathcal{L}(\delta) \supseteq \mathcal{L}(\mathrm{SOA}(L))$ holds.*

Hence, if some SORE (or CHARE) is descriptive of a language L, it must be descriptive of $\mathcal{L}(\mathrm{SOA}(L))$ 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 $\mathcal{L}(\mathrm{SOA}(L))$ is not a SORE language (or not a CHARE language), a SOA for some SORE that is descriptive of L can be obtained as follows: iterate through all sets of missing edges in a $\subseteq$-increasing way; for each set, check whether adding these edges turns the SOA into accepting a SORE language. The main question is whether this approach is efficient: 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 18 just below), such a brute force approach is probably not advisable.

The next proposition lists these and other numbers about counting and descriptive SOREs and CHAREs; it is also of independent interest in order to understand the classes of SOREs and CHAREs better. From [3, Proof of Theorem 3.1] we know that any SORE language has a SORE of length at most $10n - 4$, which gives a bound of $2^{O(n \log n)}$ on the number of different SORE languages. Our results are summarized in Table 1. Recall that regular expressions are called equivalent if they generate the same language.

Proposition 18 *Let Σ be a finite alphabet of n 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) 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.*

Proof. We start with showing (1). We have $n!\ 2^{2n} \leq c(n)$, as any sequence of all and only the elements from $\{a_i \mid i \in \mathbb{N}\}$, each with one of the four possibilities of adding $^+$ (for repetition) or ? (for optional), give non-equivalent CHAREs. On the other hand, we can bound the number of syntactically different CHAREs as follows. There are $n!$ different choices for the order of the terminal symbols. For a given order of the terminal letters, we associate any binary string x of length $n-1$ with the CHARE that uses the given order of terminal letters, and, for all $i \leq n$, adds a new chain factor for the $i+1$th letter iff x at place i has a 1 (and use the same chain factor otherwise). For a CHARE with k chain factors, there are 4^k different choices for the annotation of the chain factors (none, ?, $^+$, or $^+$?). Thus, there are at most

$$\sum_{k=1}^{n-1} n!\ 4^k \binom{n-1}{k} \leq n!\ 4^n \sum_{k=0}^{n} \binom{n}{k} = n!\ 2^{3n}$$

possibilities for syntactically distinct CHAREs, which gives the upper bound.

We now discuss (2). We view a SORE as a binary tree with any internal node labeled by "$\mid$" (for the disjunction) or "$\cdot$" (for concatenation), and all leaves labeled by distinct alphabet symbols; furthermore, any node (including the leaves) can be additionally labeled by "$^+$" (for repetition) and/or "?" (for optional). We use the intuitive translation of such trees as syntax trees of SOREs; clearly, any SORE on n alphabet symbols can be equivalently written as such a tree. The number of different binary trees with k leaves is known as the sequence of *Catalan numbers* and is asymptotically $2^{2k-\Theta(\log k)}$. A much more precise bound is known; however, this bound will suffice for our rough estimates. With two different choices for the label of all internal nodes and four different choices for each node for the "$^+$" and/or "?" label, we get a maximal number of

$$n!\ 2^{2n-\Theta(\log n)} 2^n 4^{2n} \leq n!\ 2^{7n}$$

many different non-equivalent SOREs. Regarding the lower bound, consider now all such syntax tree with all internal nodes labeled "$\cdot$" and "$^+$", and all leaves possibly labeled by "$^+$". It is easy to check that this corresponds to pairwise non-equivalent SOREs. This gives a lower bound of

$$n!\ 2^{2n-\Theta(\log n)}2^n = n!\ 2^{3n-\Theta(\log n)}.$$

Regarding (3), we note that the sample $S = \{ab_1ab_2ab_3ab_4 \ldots ab_na\}$ has 2^n pairwise non-equivalent descriptive SOREs as follows. For any partition of $\{b_i \mid 1 \leq i \leq n\}$ into two disjoint (but possibly empty) sets B_1 and B_2, we have that $(ALT(B_1)?aALT(B_2)?)^+$ is a SORE-descriptive of S (recall that $ALT(\emptyset) = \varepsilon$). We see that these SOREs are descriptive by applying either the SORE construction algorithm from [4] (which finds a SORE equivalent to a given SOA, if existent) or ours from Section 5 and observing a strict generalization.

Regarding (4), we note that, for all $n \geq 2$, the sample $\{a_i^2 \mid 1 \leq i \leq n\}$ has $n!$ pairwise non-equivalent descriptive CHAREs (namely all CHAREs of the form $a_{p(1)}? \ldots a_{p(n)}?$, where p is a permutation of $\{1, \ldots, n\}$).

Regarding (5), suppose $n \geq 1$ and consider the sample $S = \{a_ia_{i+1} \mid 1 \leq i < n\} \cup \{a_1, a_n\}$. The SOA corresponding to S has $\Theta(n)$ edges. Let α be a SORE-descriptive of S; we will show that, for all i, j with $i < j < n$, we have $a_i \to_\alpha a_j$. Let x be the least common ancestor of a_i and a_j in the syntax tree of α. As there is a word in $\mathcal{L}(\alpha)$ which starts with a_j, the child of x which contains a_j generates words starting with a_j symmetrically, the child of x which contains a_i generates a word ending with a_i. Thus, we are done if x is labeled by concatenation and a_i is in the left child of x (and a_j in the right). It is straightforward to see that the other cases will similarly necessitate $a_i \to_\alpha a_j$, by using $a_i \to_S^+ a_j$. This also implies that $a_1? \ldots a_n?$ is the only SORE that is descriptive of S (modulo equivalence; as equivalent SOREs like $(a_1? \ldots a_n?)?$ are also descriptive of S).

We have that (6) follows from the proof of (5), as the only SORE-descriptive SORE for the sample in that case is also a CHARE. □

In particular, note that Proposition 18 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 (including a more precise counting, for example with the inclusion-exclusion principle, and the use of sophisticated bounds for known number sequences, such as the Bell numbers). Finally, if we are only interested in the number of different such languages *modulo renaming of the terminal letters*, then the same bounds without the factor $n!$ hold.

3 Descriptivity versus CRX and RWR

Proposition 18 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 17), descriptive CHAREs and SOREs can be obtained from the descriptive SOA, and moreover, for every language L and every SORE α, $\mathcal{L}(\alpha) \supseteq \mathcal{L}(\mathrm{SOA}(L))$ 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 [11], this can be done in time $O(ln)$, where $l := \sum_{s\in S} |s|$, and $n := |\,\mathrm{term}(S)|$.

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 their variants from [4] do not always compute descriptive CHAREs or SOREs.

For the CHARE algorithm `CRX`, this is quite easy to see: As pointed out in [4] (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 $\delta := a(b \mid d)(c \mid e)$ 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 21 further down.

The proofs for the non-descriptivity of the SORE algorithm `RWR` and its variants require more effort; we proceed by giving a description of `RWR`, followed by a proof of their non-desriptivity in Section 3.2.

3.1 Description of RWR

In this section we describe the algorithm `RWR` from [4] in some detail. For a more formal definition of `RWR` and the rules it uses, as well as any variants, we refer to [4]. This algorithm takes a SOA A as input and, by step-by-step modifications, turns it into smaller and smaller generalized SOAs, until a generalizing SORE can be read off the only remaining vertex (apart from source and sink).

The modifications concern either one or two states of the SOA; two types of rules are distinguished: *rewrite rules* and *repair rules*. First, rewrite rules are applied, as long as any are possible. From [4] we know that these rules will turn any SOA into an equivalent SORE, if possible. If, at some point, none of the rules are applicable, then we know that the input SOA is not equivalent to a SORE, and the repair rules are used to generalize (which will then enable rewrite rules to continue). Let A be a SOA used as input to `RWR`. The rewrite rules concerning only one state are as follows.

– Iteration r^+: If there is a vertex labeled r with a self-loop, remove the self-loop and change the label of the vertex to r^+.
– Optional $r?$: If there is a vertex labeled r with the source as predecessor, the sink as a successor, and there is an edge from source to sink, remove the edge from source to sink and change the label of the vertex to $r?$.

The rewrite rules concerning two states have similar syntactic criteria. For clarity, we give semantic criteria only. Let r and s be labels of two vertices of

A. We try contracting these two vertices (in a graph-theoretic sense, i.e., with appropriate edge modifications) into a vertex labeled one of

$$r \mid s, \quad r \cdot s, \quad r? \cdot s, \quad r \cdot s?, \quad r? \cdot s?$$

and possibly with a self loop. If one of these leads to a generalized SOA accepting the same language, then we keep the contraction, and continue from this generalized SOA. In [4] equivalent rules are stated, but with syntactic criteria which are easy to check (but tedious to state).

When no more rewrite rules are applicable, then either only one state apart from source and sink remains (which allows us to read off the SORE), or the original SOA was not equivalent to a SORE and we need to generalize. `RWR` uses the following (syntactic) *repair rules* which generalize the SOA.

– Repair $r \mid s$: If there are two vertices 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. If there is an edge between r and s, add an edge in the opposite direction as well; add self loops on both r and s unless where the label of the vertex has a "$^+$" at the root of the syntax tree.
– Repair $r \cdot s?$: If there are two vertices 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; add a self-loop on r only if the label of r does not have a "$^+$" at the root of its syntax tree.
– Repair $r? \cdot s$: If there are two vertices 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; add a self-loop on s only if the label of s does not have a "$^+$" at the root of its syntax tree.
– Repair $r? \cdot s?$: Let r and s be vertices 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; for both r and s, introduce self-loops only on vertices which do not have a label with "$^+$" at the root of its syntax tree. Furthermore, for all predecessors u of r and all successors v of s, add an edge from u to v.

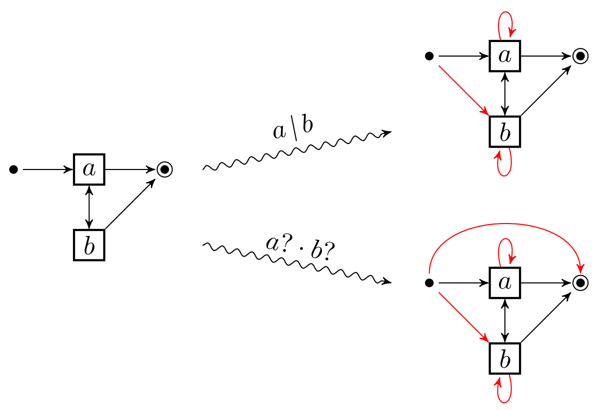
The repair rules are applied only if none of the rewrite rules is applicable, and the first applicable rule from the list above is applied. After each application of a repair rule, rewrite rules are used again as long as possible. This terminates with a generalized SOA with only a single state different from source and sink; its label is the output of `RWR`.

3.2 RWR-Variants and Descriptivity

In this section we give theorems regarding properties of `RWR`-variants. In particular, we show that every variant fails to find a descriptive SORE on some input. First, we formally show that `RWR` does not always return a descriptive SORE.

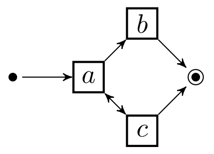
Theorem 19 *For Σ a finite alphabet with $|\Sigma| \geq 3$ and all orderings of the repair rules of* `RWR`*, there is a (finite) set of samples $S \subseteq \Sigma^*$ such that* `RWR` *on S produces a* SORE *which is not* SORE*-descriptive.*

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.

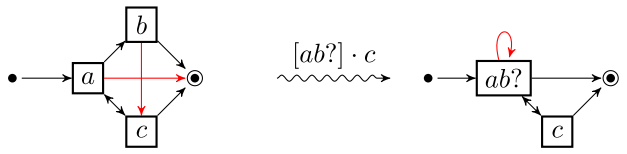


A descriptive SORE for S is $\delta_2 := (a(b \mid c))^+$, which we prove as follows. In comparison to A, the SOA that corresponds to δ_2 adds only a single edge, the edge from a to b. So the only possibility for a SORE language $\mathcal{L}(\gamma)$ with $\mathcal{L}(A) \subseteq \mathcal{L}(\gamma) \subset \mathcal{L}(\delta_2)$ is $\mathcal{L}(A)$ itself. However, $\mathcal{L}(A)$ is not a SORE language, which can be seen, just as in Proposition 18, by applying either the SORE construction algorithm `RWR` from [4] 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 $\mathcal{L}(\delta_2)$.)

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

RegEx α	$\lvert\mathcal{L}(\alpha)^{\leq 6}\rvert$	exp growth basis	recurrence	base cases
$(ab \mid ac)^+$	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 20. For each regular expression α, $f_\alpha(n)$ denotes the number of words in $\mathcal{L}(\alpha)$ of length n; given in the table are the number of words of length at most 6 generated 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 $n \in \{1, 2, 3\}$.

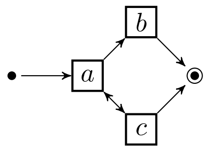


This SOA corresponds to the SORE $(ab?c?)^+$, and its language is a strict superset of $\mathcal{L}(\delta_2)$ for the (descriptive) SORE $\delta_2 = (a(b|c))^+$ (for example abc is generated by the former and not the latter). Deceiving the rule "Repair $r? \cdot s$" is symmetric to deceiving "Repair $r \cdot s?$". ☐

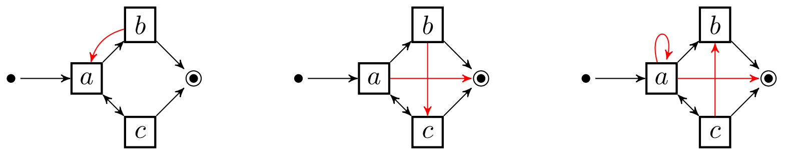
In [4], Bex et al. propose a variant of RWR that is called $\mathtt{RWR}^2_\ell$, which uses a natural number ℓ as a branching parameter. The algorithm explores the (recursive) outcomes of the best ℓ candidates for a repair rule, choosing the ones that lead to a minimal number of words of length at most $2n$ $(= 2|\Sigma|)$ in the language generated by the resulting SORE.

Theorem 20 *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 $\mathtt{RWR}^2_\ell$ on S produces a* SORE *which is not* SORE*-descriptive.*

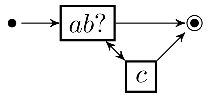
Proof. We first assume $\ell = 1$; consider again the sample $\{ab, ac, acac\}$ with the following corresponding SOA.



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.



In order to determine which of the rules RWR will choose, we analyse the properties of these three SOAs, which we summarise in Table 2. 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 $(a(b\,|\,c))^+$ as in the proof of Theorem 19.

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 from 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 vertex $v \in V$ 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 vertex 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 vertices $u, v \in V$ 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:

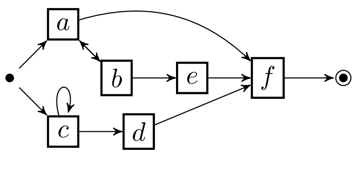
(1) Replace each strongly connected looped component $A \subseteq V$ with a vertex that is labeled with the regular expression $ALT(A)^+$. This turns A into a (possibly generalized) Soa that is a DAG.
(2) Every vertex in the DAG is assigned a level number.
(3) Every level is turned into one or more chain factors. If a level contains more than one non-letter vertex, 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 an efficient manner.

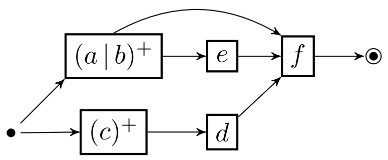
Theorem 21 *For any given* Soa *A, `Soa2Chare` finds a* Chare *that is* Chare*-descriptive of $\mathcal{L}(A)$ in time $O(m)$, where m is the number of transitions of A.*

Before we discuss the proof of Theorem 21 further down, we illustrate the behavior of `Soa2Chare` with an Example.

Example 22 *Let $S = \{abaf, abef, ccdf\}$. The corresponding* Soa, 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 vertices $(a \mid b)^+$ and $(c)^+$, the second level with the vertices d and e, and the third level with the vertex 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* Chare*s $(a \mid b)^+?(c)^+?$, $(e \mid d)?$, and f, which are concatenated to $(a \mid b)^+?(c)^+?(e \mid d)?f$. By Theorem 21, this* Chare *is* Chare*-descriptive of S.*

Proof of Theorem 21. We first prove termination and running time, followed by the proof of correctness.

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 for finding strongly connected components (cf. [6, Section 22.5]), 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 vertex 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 $\mathcal{L}(\gamma) \supseteq S$, but only about those with $\mathcal{L}(\texttt{Soa2Chare}(\mathrm{SOA}(S))) \supseteq \mathcal{L}(\gamma) \supseteq S$.

This allows us to use Lemma 11 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 23 *Let α be a* Chare*. A set $A \subseteq \mathrm{term}(\alpha)$ 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 $ALT(A)^+?$.*

Proof. We begin with the *if* direction. Assume that some Chare α contains a chain factor α' with $\alpha' = ALT(A)^+$ or $\alpha' = ALT(A)^+?$ for some finite and non-empty set $A \subseteq \Sigma$. By definition, $a \rightarrow_\alpha b$ holds for all letters $a, b \in A$. Furthermore, let $\alpha = \alpha_1 \alpha' \alpha_2$ for appropriate (possibly empty) CHAREs α_1, α_2. For every vertex $v \in \mathrm{term}(\alpha_1) \cup \{A.\mathtt{src}\}$ and every $a \in A$, we observe $v \rightarrow_\alpha^+ a$, but not $a \rightarrow_\alpha^+ v$. Likewise, for every vertex $v \in \mathrm{term}(\alpha_2) \cup \{A.\mathtt{snk}\}$ and every $a \in A$, we observe $a \rightarrow_\alpha^+ v$, but not $v \rightarrow_\alpha^+ a$. Hence, A is a strongly connected looped component in $\mathrm{SOA}(\alpha)$.

For the *only if* direction, we assume that there is a (non-empty) set $A \subseteq \mathrm{term}(\alpha)$ that is a strongly connected looped component in $\mathrm{SOA}(A)$. Now, let $\alpha = \alpha_1 \dots \alpha_n$ for some n, where each α_i is a chain factor. As A is a strongly connected looped component, $a \rightarrow_\alpha^+ b$ holds for all $a, b \in A$. Therefore, there is an k with $A_k := \mathrm{term}(\alpha_k) \supseteq A$ (otherwise, there would exist $a, b \in A$, $a \in \mathrm{term}(\alpha_i)$, $b \in \mathrm{term}(\alpha_j)$, $i < j$, which would imply $a \rightarrow_\alpha^+ b$, but not $b \rightarrow_\alpha^+ a$). Moreover, as $a \rightarrow_\alpha^+ a$ holds for all $a \in A$, α_k is equal to $ALT(A_k)^+$ or $ALT(A_k)^+?$. Finally, if $A_k \supset A$, there would exist an $b \in A_k \setminus A$ with $b \rightarrow_\alpha a \rightarrow_\alpha b$ for all $a \in A$. As $b \notin A$, this would imply that A is not a strongly connected looped component, and contradict our initial assumption. Hence, $A_k = A$, and $\alpha_k \in \{ALT(A)^+, ALT(A)^+?\}$. □

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 24 *Let Σ be an alphabet. For every finite and non-empty set $S \subseteq \Sigma^*$, and every set $A \subseteq \mathrm{term}(S)$, 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 $\mathrm{SOA}(\mathtt{Soa2Chare}(\mathrm{SOA}(S)))$.*

Finally, according to Lemma 11, this immediately leads to the following observation:

Corollary 25 *Let $S \subseteq \Sigma^*$ be a finite set, and let $\delta := \mathtt{Soa2Chare}(\mathrm{SOA}(S))$. For every CHARE γ with $\mathcal{L}(\delta) \supseteq \mathcal{L}(\gamma) \supseteq S$, $\mathrm{SOA}(\gamma)$ 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 26 *Let Σ be an alphabet, let $S \subseteq \Sigma^*$ be a non-empty set, and let $\delta := \mathtt{Soa2Chare}(\mathrm{SOA}(S))$. Then $\mathcal{L}(\delta) = \mathcal{L}(\gamma)$ holds for every CHARE γ with $\mathcal{L}(\delta) \supseteq \mathcal{L}(\gamma) \supseteq S$.*

Proof of Lemma 26. We shall prove Lemma 26 by proving a stronger claim, namely that $\delta = \gamma$ holds. This claim only holds if we allow for a slight abuse of the $=$ symbol; as for the remainder of this proof, we shall interpret $\alpha = \beta$ to mean that α and β are identical modulo reordering of the terminals symbols *inside* the chain factors (i. e., $(a \mid b) = (b \mid a)$ holds, but $(a)(b) \neq (b)(a)$).

Before we proceed to the actual proof, we begin with a preliminary observation. In line 5, the algorithm `Soa2Chare` partitions the vertices of $\mathrm{SOA}(S)$ into levels 0 to n (with $n \geq 1$). Note that with the exception of levels 0 and n

(which contain only $A.\mathtt{src}$ and $A.\mathtt{snk}$, respectively), each level i leads to a sub-CHARE δ_i, which consists of one or more chain factors. Hence, `Soa2Chare` implicitly defines a factorization $\delta = \delta_1 \dots \delta_{n-1}$, where each δ_i was derived from level i. Note that $\delta = \varepsilon$ holds for the special case of $n = 1$.

We prove Lemma 26 using induction on this n, i. e., the level number assigned to $A.\mathtt{snk}$ (which can also be understood as the number of levels after $A.\mathtt{src}$, and $n-1$ is the number of factors in the factorization $\delta_1 \dots \delta_{n-1}$ as given above). More specifically, we prove the following claim for every $n \geq 1$.

Claim 1. Let Σ be an alphabet, let $S \subseteq \Sigma^*$ be a non-empty set for which the level construction step of `Soa2Chare` creates levels 0 to n, and let $\delta := \mathtt{Soa2Chare}(\mathrm{SOA}(S))$. Then $\delta = \gamma$ holds for every CHARE γ with $\mathcal{L}(\delta) \supseteq \mathcal{L}(\gamma) \supseteq S$.

Base case. For $n = 1$, $\mathrm{SOA}(S)$ contains only the vertices $A.\mathtt{src}$ and $A.\mathtt{snk}$, and $S = \{\varepsilon\}$ must hold. As $\delta = \varepsilon$ and $\mathcal{L}(\delta) = S$, $\mathcal{L}(\delta) \supseteq \mathcal{L}(\gamma) \supseteq S$ implies $\gamma = \varepsilon = \delta$ for every CHARE γ. □ (BASE CASE)

Inductive step. Now assume that Claim 1 holds for some $n \geq 1$. Let S be a set for which the level construction step of `Soa2Chare` creates levels 0 to $n+1$, let $\delta := \mathtt{Soa2Chare}(\mathrm{SOA}(S))$ with $\delta = \delta_1 \dots \delta_n$, where each δ_i was derived from level i, and let γ be a CHARE such that $\mathcal{L}(\delta) \supseteq \mathcal{L}(\gamma) \supseteq S$ with $\gamma = \gamma_1 \dots \gamma_m$ ($m \geq 1$), where each γ_i is a chain factor. W. l. o. g. $\mathrm{term}(\delta) = \mathrm{term}(\gamma) = \mathrm{term}(S) = \Sigma$.

We define the CHARE $\delta' := \delta_1 \dots \delta_{n-1}$ (with $\delta' := \varepsilon$ if $n = 1$), and the set $S' := \pi_{\mathrm{term}(\delta')}(S)$. In other words, δ' is obtained by removing level n from the level construction for S; thus, $\delta' = \mathtt{Soa2Chare}(\mathrm{SOA}(S'))$ holds by definition. The proof is based on the following claim.

Claim 2. There exists some i with $1 \leq i \leq m$ such that $\delta_n = \gamma_i \cdots \gamma_m$.
Proof of Claim 2. When building δ_n, `Soa2Chare` constructs the sets B and C for level n, where B contains all vertices on level n that are labeled with $^+$ (and, hence, represent some strongly connected looped component that was contracted in lines 2–4), while C contains all vertices with level number n that represent single letters. Note that at most one of the sets B and C may be empty. Hence, exactly one of the following cases holds:

(1) $B = \emptyset$, $C \neq \emptyset$, level n is not a skip level,
(2) $B = \emptyset$, $C \neq \emptyset$, level n is a skip level,
(3) $|B| = 1$, $C = \emptyset$, level n is not a skip level,
(4) $|B| = 1$, $C = \emptyset$, level n is a skip level,
(5) $|B| \geq 2$, $C = \emptyset$,
(6) $B \neq \emptyset$, $C \neq \emptyset$.

Case (1): If $B = \emptyset$ and $C \neq \emptyset$, and level n is not a skip level, $\delta_n = ALT(B)$ holds. Then level n of the construction contains exactly the vertices in C, and for all vertices $v \in \Sigma \cup \{\mathtt{snk}\}$ and all $c \in C$, $c \rightarrow_S v$ if and only if $v = \mathtt{snk}$. Furthermore, as level n is not a skip level, there is no vertex $v \in (\Sigma \cup \{\mathtt{src}\}) \setminus C$

with $v \rightarrow_S \texttt{snk}$. By definition, these observations still hold if $\rightarrow_S$ is replaced with $\rightarrow_\delta$.

According to Lemma 11, SOA(δ) covers SOA(γ), which in turn covers SOA(S). Hence, for all vertices $v \in (\Sigma \cup \{\texttt{src}\})$, $v \rightarrow_\gamma \texttt{snk}$ holds if and only if $v \in C$. Therefore, $\text{term}(\gamma_m) = C$ and $\varepsilon \notin \mathcal{L}(\gamma_m)$ must hold. This is only satisfied if $\gamma_m = \mathit{ALT}(C)$ or $\gamma_m = \mathit{ALT}(C)^+$. We can exclude the latter case, as C is not a strongly connected looped component in SOA(S) or SOA(δ), and Corollary 25 applies. Hence, $\gamma_m = \mathit{ALT}(C) = \delta_n$ follows.

Case (2): If $B = \emptyset$ and $C \neq \emptyset$, and level n is a skip level, $\delta_n = \mathit{ALT}(C)?$. As in Case (1), we can observe that for all vertices $v \in \Sigma \cup \{\texttt{snk}\}$ and all $c \in C$, $c \rightarrow_S v$ (and $c \rightarrow_\delta v$) if and only if $v = \texttt{snk}$. But as n is a skip level, this only allows us to conclude that all elements of C must be placed in the last chain factor of γ; i. e., $C \subseteq \text{term}(\gamma_m)$. Furthermore, as the elements of C do not belong to any strongly connected looped component, Corollary 25 yields that $\gamma_m \neq \mathit{ALT}(\text{term}(\gamma_m))^+$ and $\gamma_m \neq \mathit{ALT}(\text{term}(\gamma_m))^{+}?$.

In order to prove $\text{term}(\gamma_m) = C$, assume that $C \subset \text{term}(\gamma_m)$, i. e., there exists a letter $a \in \text{term}(\gamma_m) \setminus C$. (Note that this is only possible if $n \geq 2$, otherwise, we can conclude that $\text{term}(\gamma_m) = C$ and skip to the next paragraph.) As γ_m cannot contain $^+$ or $^+?$, we observe that for all vertices $v \in \Sigma \cup \{\texttt{snk}\}$, $a \rightarrow_\gamma v$ holds iff. $v = \texttt{snk}$. Therefore, there can be no $c \in C$ with $a \rightarrow_\gamma^+ c$, and as SOA(γ) covers SOA(S), there is also no $c \in C$ for which $a \rightarrow_S^+ c$ holds. In other words, all paths from $\texttt{src}$ to vertices of C must lead through other vertices than a; hence, there is a vertex v on level $n - 1 \geq 1$ with $v \rightarrow_S c$ for some $c \in C$. Hence, we have $v \rightarrow_\gamma a$, but not $v \rightarrow_\delta a$, which contradicts $\mathcal{L}(\delta) \supseteq \mathcal{L}(\gamma)$. We conclude $\text{term}(\gamma_m) = C$.

We now know that $\gamma_m \in \{\mathit{ALT}(C), \mathit{ALT}(C)?\}$. As level n is a skip level, there exists a vertex $v \in \{\texttt{src}\} \cup (\Sigma \setminus C)$ with $v \rightarrow_S \texttt{snk}$. Hence, $\mathit{ALT}(\text{term}(C))? = \delta_n$ must hold.

Case (3): If $|B| = 1$, $C = \emptyset$, and level n is not a skip level, then $\delta_n = \mathit{ALT}(B_m)^+$ for some set $B_m \subseteq \Sigma$ with $B = \{\mathit{ALT}(B_m)^+\}$. By definition of $\texttt{Soa2Chare}$, this is only possible if B_m is a strongly connected looped component in SOA(S), and for all vertices $v \in \Sigma \cup \{\texttt{src}\}$, $v \rightarrow_S \texttt{snk}$ holds only if $v \in B_m$.

If $\text{term}(\gamma_m) \neq B_m$, then SOA($\gamma$) does not contain the same strongly connected looped component as SOA(S) and SOA(δ), or there is some letter $c \notin B_m$ with $c \rightarrow_\gamma \texttt{snk}$. In either case, there is a contradiction to $\mathcal{L}(\gamma) \supseteq S$ or $\mathcal{L}(\gamma) \subseteq \mathcal{L}(\delta)$ and Corollary 25. Hence, $\text{term}(\gamma_m) = B_m$ must hold.

Furthermore, if $\gamma_m \in \{\mathit{ALT}(B_m), \mathit{ALT}(B_m)?\}$, B_m is also not a strongly connected looped component in SOA(γ) (a contradiction to $\mathcal{L}(\gamma) \supseteq S$). Hence, either $\gamma_m = \mathit{ALT}(B_m)^+$ or $\gamma_m = \mathit{ALT}(B_m)^{+}?$ holds. Assume for the sake of the argument that $\varepsilon \in \mathcal{L}(\gamma_m)$. Then there exists a vertex $v \in \text{term}(\gamma_1 \dots \gamma_{m-1}) \cup \{\texttt{snk}\}$ with $v \rightarrow_\gamma \texttt{snk}$, but $v \rightarrow_\delta \delta$ does not hold (otherwise, n would be a skip level). We conclude $\gamma_m = \mathit{ALT}(B_m)^+ = \delta_n$.

Case (4): If $|B| = 1$, $C = \emptyset$, and level n is a skip level, then $\delta_n = \mathit{ALT}(B_m)^{+}?$ for some set $B_m \subseteq \Sigma$ with $B = \{\mathit{ALT}(B_m)^+\}$. As in Case (3), we are able to derive that either $\gamma_m = \mathit{ALT}(B_m)^+$ or $\gamma_m = \mathit{ALT}(B_m)^+?$. Taking the skip level into account as in Case (2), we conclude $\gamma_m = \mathit{ALT}(B_m)^+? = \delta_n$.

Case (5): If $|B| \geq 2$ and $C = \emptyset$, there exist a $k \geq 2$ and k disjoint non-empty sets $B_{m-k+1}, \ldots, B_m \subseteq \Sigma$ with $B = \{\mathit{ALT}(B_{m-k+1})^+, \ldots, \mathit{ALT}(B_m)^+\}$, and $\delta_n = \mathit{ALT}(B_{m-k+1})^+? \ldots \mathit{ALT}(B_m)^+?$. In order to increase readability, let $i := (m-k+1)$.

According to the definition of `Soa2Chare`, every B_j is a strongly connected looped component in $\mathrm{SOA}(S)$, and each B_j was placed in level n of the construction. Therefore, due to Corollary 25, for every j with $i \leq j \leq m$, γ contains a chain factor $\mathit{ALT}(B_j)^+$ or $\mathit{ALT}(B_j)^+?$. Hence, for all j with $i \leq j \leq m$, $\mathrm{term}(\gamma_j) = B_j$ and $\gamma_j \in \{\mathit{ALT}(B_j)^+, \mathit{ALT}(B_j)^+?\}$ must hold – otherwise, $\mathrm{SOA}(\gamma)$ would contain edges to or from the letters of B_j that do not occur in $\mathrm{SOA}(\delta)$, a contradiction to Lemma 11.

Finally, as all vertices in B are on level n of the level construction, there exist vertices $u \in (\Sigma \cup \{\mathtt{src}\}) \setminus (\bigcup_{j=i}^{m} B_j)$ and $b \in B_m$ with $u \to_S b$ as well as a vertex $b' \in B_i$ with $b' \to_S \mathtt{snk}$. In order to ensure these reachabilities in γ, each chain factor γ_j must be able to generate ε. Hence, $\gamma_j = \mathit{ALT}(B_j)^+?$ holds for all j $(i \leq j \leq m)$, and we conclude $\delta_n = \gamma_i \cdots \gamma_m$.

Case (6): If $B \neq \emptyset$ and $C \neq \emptyset$, there exist a $k \geq 1$ and sets $B_{m-k}, \ldots, B_{m-1}$ with $B = \{\mathit{ALT}(B_{m-k})^+, \ldots, \mathit{ALT}(B_{m-1})^+\}$, and

$$\delta_n = \mathit{ALT}(B_{m-k})^+? \ldots \mathit{ALT}(B_{m-1})^+? \mathit{ALT}(C)?.$$

Using reasoning that is analogous to Case (5), we are able to conclude that $\gamma_j = \mathit{ALT}(B_j)^+?$ for all j with $(m-k) \leq j \leq (m-1)$.

All that remains to do is proving $\gamma_m = \mathit{ALT}(C)?$. Once again according to the reasoning we used in all previous cases, $\mathrm{term}(\gamma_m) = C$ must hold, as otherwise, we would introduce edges not present in $\mathrm{SOA}(\delta)$, or lose edges present in $\mathrm{SOA}(S)$. Due to Corollary 25, we know that $\gamma_m \notin \{\mathit{ALT}(C)^+, \mathit{ALT}(C)^+?\}$; and due to the same reachability argument as for the γ_j in Case (5), $\varepsilon \in \mathcal{L}(\gamma_m)$ must hold. We conclude $\gamma_m = \mathit{ALT}(C)?$ and, hence, $\delta_n = \gamma_{m-k} \cdots \gamma_m$.

□ (FOR CLAIM 2)

As we now know that there is an i with $1 \leq i \leq m$ such that $\delta_n = \gamma_i \ldots \gamma_m$, we can combine $\delta = \delta' \delta_n$ and $\mathcal{L}(\delta) \supseteq \mathcal{L}(\gamma) = \mathcal{L}(\gamma_1 \ldots \gamma_m)$ to

$$\mathcal{L}(\delta' \delta_n) \supseteq \mathcal{L}(\gamma_1 \ldots \gamma_{i-1} \delta_n).$$

By splitting off δ_n, we conclude

$$\mathcal{L}(\delta') \supseteq \mathcal{L}(\gamma_1 \ldots \gamma_{i-1}).$$

Due to our induction assumption, this implies $\delta' = \gamma_1 \ldots \gamma_{i-1}$. Hence, $\delta = \delta' \delta_n = \gamma_1 \ldots \gamma_m$ holds, which completes the proof. □ (INDUCTIVE STEP)

As Claim 1 holds for all $n \geq 1$, $\mathcal{L}(\delta) \supseteq \mathcal{L}(\gamma) \supseteq S$ implies $\delta = \gamma$ (with the caveat that terminals inside chain factors of γ and δ might have a different order). Hence, $\mathcal{L}(\delta) = \mathcal{L}(\gamma)$, and Lemma 26 follows immediately.

$\square$ (FOR LEMMA 26)

According to Lemma 26, there is no CHARE γ for which $\mathcal{L}(\texttt{Soa2Chare}(\mathrm{SOA}(S))) \supset \mathcal{L}(\gamma) \supseteq S$ holds. As we have, by definition, $\mathcal{L}(\texttt{Soa2Chare}(\mathrm{SOA}(S))) \supseteq S$, we know that the result of `Soa2Chare` on SOA(S) is CHARE-descriptive of S, which concludes the proof of correctness.

$\square$ (FOR THEOREM 21)

5 Descriptive SOREs

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

5.1 SORE Algorithm

As in Section 4, we use dot-notation to denote the application of subroutines. Likewise, for a given SOA A, we let A.`src` and A.`snk` denote the source and the sink of A, respectively. We let V be the set of vertices in A and E the set of edges.

- For any vertex $v \in V$, we let $A.\mathrm{pred}(v)$ denote the set of all predecessors of v in A; similarly, $A.\mathrm{succ}(v)$ denotes the set of all successors.
- For any vertex $v \in V$, we let $A.\,\mathrm{reach}(v) := \{u \in V \mid v \to_A^* u\}$ be the set of all vertices reachable from v.
- "contract" on SOA A takes a subset U of vertices of A and a label ℓ. The procedure modifies A as follows. All vertices in U are removed; a new vertex labeled ℓ is added; for each edge $(v, u) \in E$ with $v \in V \setminus U$ and $u \in U$, we remove (v, u) and add an edge (v, ℓ); similarly, for each edge $(u, v) \in E$ with $v \in V \setminus U$ and $u \in U$, we remove (u, v) and replace with an edge (ℓ, v). We call this method whenever we identify a subset of vertices for which we can compute a descriptive generalization (the label is then this generalizing SORE).
- "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. We use this method whenever we identified a subpart of the SOA to recurse on.
- "first" returns all vertices v such that the only predecessor of v is the source. These are particularly interesting, since our algorithm will work on the SOA by starting from the source and progressing through the links.

In particular, in a cycle-free Soa, all other successors of the source are reachable via some element from $A.\mathrm{first}()$.

- "addEpsilon" on Soa A adds a new vertex labeled ε and an edge from $A.\mathtt{src}$ to this new vertex; let $U \subseteq V$ be the set of all successors of the source which are not in $A.\mathrm{first}()$; for each edge from $A.\mathtt{src}$ to an element $u \in U$, remove this edge and add an edge from the new ε vertex to u. This is used to be able to treat all successors of the sink equally (in particular, in a cycle-free Soa, this ensures that every node after the source is reachable from the source only by passing through an element from $A.\mathrm{first}()$; this additional vertex makes sure that $A.\mathrm{first}()$ is exactly the first layer of the Soa).
- "exclusive" on Soa A on argument v (a vertex of A) returns the set of all vertices u such that $A.\mathtt{src} \not\to^{+}_{A\backslash v} u$, where $A\backslash v$ is defined as the Soa A with the vertex v (and all incident edges) removed. Intuitively, the exclusive set of a vertex v is the set of all vertices which necessarily require v in order to be reached from the source. We will use this method to find sets of vertices to recurse on: whenever a part of the Soa can be exclusively reached via a fixed vertex, we can recurse on this set of exclusive vertices.
- 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. We let $U := A.\mathrm{succ}(A.\mathtt{src})$; with $A \setminus U$ we denote the graph which has vertices and edges as in A with all vertices (and incident edges) from U deleted. We let $W_1 := A.\mathrm{pred}(A.\mathtt{snk})$ and let W_2 be the set of all vertices $d \in V \setminus (A.\mathrm{succ}(A.\mathtt{src}) \cup A.\mathrm{pred}(A.\mathtt{snk}))$ such that there is $c \in W_1$ with $c \to^{+}_{A\setminus U} d$. We let $W := W_1 \cup W_2$. Intuitively, W is the set of all elements that can be reached in any number of steps from a predecessor of the sink without crossing a successor of the source. Then the subroutine replaces (bends) all edges from an element in W to a successor of the source by an edge from the same vertex in W to the sink. See Example 28 for an illustration. Note that the application of bend ensures that no element of $A.\mathrm{succ}(A.\mathtt{src})$ can be reached from any element of $A.\mathrm{pred}(A.\mathtt{snk})$.

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 $\ell \mid \ell'$.

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 vertex labeled ε).

Algorithm 2: `Soa2Sore`

```
Input: SOA $A = (V, E)$;
Output: SORE minimally generalizing $\mathcal{L}(A)$;
if $|E| = 0$ then return $\emptyset$;
else 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 src;
    $\ell \leftarrow v.\mathrm{label}()$;
    $A.\mathrm{contract}(\{A.\mathtt{src}, v\}, \mathtt{src})$;
    $\ell' \leftarrow \mathtt{Soa2Sore}(A)$;
    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 $\mathtt{Soa2Sore}(A)$;
```

(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, merge it with the sink and recurse on the remainder.

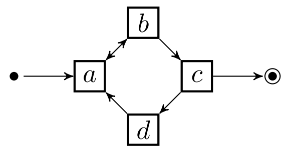
Note that the algorithm introduces ? by way of constructing "or ε." This can be cleaned up by postprocessing the resulting SORE.

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

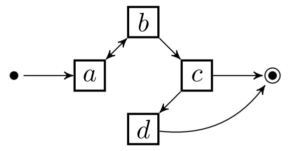
Theorem 27 *The algorithm `Soa2Sore`, given a* SOA *A as input, finds a descriptive* SORE *for $\mathcal{L}(A)$ in time $O(nm)$, where n is the number of alphabet symbols used in A, and m is the number of transitions in A. Furthermore, this algorithm produces a* SORE *such that the corresponding* SOA *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 27, we give two examples of `Soa2Sore`. The first example illustrates how strongly connected looped components are treated. The second illustrates the use of "exclusive".

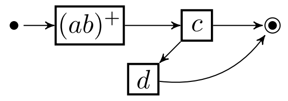
Example 28 *Consider the following* SOA*.*



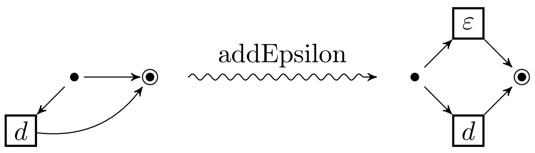
The labeled vertices of this Soa *consist of a single strongly connected looped component, an application of "bend" computes the set $W = \{c, d\}$ with $W_1 = \{c\}$ and $W_2 = \{d\}$, which leads to the following* Soa*.*



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



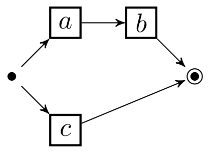
We can split off the first vertex twice now (as line 11 applies twice), recursing finally on the remaining Soa *as follows.*



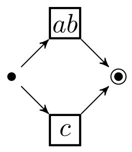
This results in $d \mid \varepsilon$, or, equivalently, $d?$. Going back through the recursions, we get

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

Example 29 *Consider now the following* Soa*.*



For this Soa*, line 17 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 27

In this section we are concerned with proving Theorem 27. We start with a lemma which is used in its proof. Intuitively, we use the function shown existent in the lemma to turn SOREs into a "canonical form", in order to ease the comparison of the computed SORE with the supposedly smaller descriptive SORE (see the proof for details).

Lemma 30 *There is a function f on* SORE*s such that, for each* SORE α*,* $\mathcal{L}(f(\alpha)^+) = \mathcal{L}(\alpha^+) \setminus \{\varepsilon\}$ *and, for all* $a \in \alpha.\mathrm{succ}(\alpha.\mathtt{src})$ *and* $c \in \alpha.\mathrm{pred}(\alpha.\mathtt{snk})$ *we have* $c \not\to_{f(\alpha)} a$*.*

Proof. We define f as a function on SOREs recursively as follows. For all symbols a and SOREs α_0, α_1, we let

$$\begin{aligned}
f(\varepsilon) &= \emptyset;\\
f(a) &= a;\\
f(\alpha_0^+) &= f(\alpha_0);\\
f(\alpha_0 \,|\, \alpha_1) &= \begin{cases} \emptyset, & \text{if } \mathcal{L}(\alpha_0) = \{\varepsilon\} = \mathcal{L}(\alpha_1);\\ f(\alpha_0), & \text{else if } \mathcal{L}(\alpha_1) = \{\varepsilon\};\\ f(\alpha_1), & \text{else if } \mathcal{L}(\alpha_0) = \{\varepsilon\};\\ f(\alpha_0) \,|\, f(\alpha_1), & \text{otherwise;} \end{cases}\\
f(\alpha_0 \cdot \alpha_1) &= \begin{cases} f(\alpha_0 \,|\, \alpha_1), & \text{if } \varepsilon \in \mathcal{L}(\alpha_0) \cap \mathcal{L}(\alpha_1);\\ \alpha_0 \cdot f(\alpha_1), & \text{else if } \varepsilon \in \mathcal{L}(\alpha_0);\\ f(\alpha_0) \cdot \alpha_1, & \text{else if } \varepsilon \in \mathcal{L}(\alpha_1);\\ \alpha_0 \cdot \alpha_1, & \text{otherwise.} \end{cases}
\end{aligned}$$

Let a SORE α be given. We omit the straightforward induction which shows $\mathcal{L}(f(\alpha)^+) = \mathcal{L}(\alpha^+) \setminus \{\varepsilon\}$.

Let $a \in \alpha.\mathrm{succ}(\alpha.\mathtt{src})$ and $c \in \alpha.\mathrm{pred}(\alpha.\mathtt{snk})$. We show the claim by induction on the syntax tree of $f(\alpha)$. Clearly, the root of $f(\alpha)$ is not labeled $^+$.

Suppose now the root is labeled "or." Then either a and c are in different subtrees of the root, in which case we have $c \not\to_{f(\alpha)} a$; or a and c are in the same subtree, in which case the claim follows by induction.

Suppose now the root is labeled with concatenation. We make the following two remarks. If a is in the right subtree of the root, then the left subtree allows ε (as $a \in \alpha.\mathrm{succ}(\alpha.\mathtt{src})$). Similarly, if c is in the left subtree of the root, then the right subtree allows ε (as $c \in \alpha.\mathrm{pred}(\alpha.\mathtt{src})$). We consider different cases as follows.

If a and c are both in the left subtree, then the right subtree allows ε, so the claim follows by induction. If a and c are both in the right subtree, then the left subtree allows ε, so the claim follows, again, by induction. If a is in the left subtree, and c in the right, then, trivially, $c \not\to_{f(\alpha)} a$. If c is in the left subtree, and a in the right, then both subtrees allow ε. Thus, the definition of $f(\alpha)$ gives immediately that $c \not\to_{f(\alpha)} a$. □

We are now ready to prove Theorem 27.

Proof of Theorem 27. 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 As for the termination, we first note that the algorithm starts by breaking up strongly connected looped components. As remarked at the end of the definition of the "bend" subroutine, the case of line 5 can only apply finitely often, as each "bend" operation breaks up a strongly connected looped component. Line 9 can never apply twice in a row, so it suffices to show that all other cases can only apply finitely often. All later cases contract vertices; this reduces the number of vertices, which can only increase by "addEpsilon". It is easy to see that for each application of "addEpsilon" at least three vertices are contracted before another application of "addEpsilon", which shows termination.

We refer to [6] for standard graph algorithms, such as finding strongly connected (looped) components.

As the algorithm never introduces self-loops, 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. Note that we use without further mention that $n < m$, which implies $O(n + m) = O(m)$.

We first bound the running time on acyclic Soas. We topologically sort the vertices of A (this takes $O(m)$ time). We now iteratively construct an annotation of all the vertices of G with subsets of $A.\operatorname{first}()$, corresponding to what vertices they are reachable from. We start by annotating each vertex of G that corresponds to a vertex $v \in A.\operatorname{first}()$ 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 $A.\operatorname{first}()$ 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-vertices are balanced in between by splitting off of a starting vertex, as given in line 11. 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: Strongly connected components are only produced by adding $^+$, and that is done exactly on SOREs for which the input has a strongly connected components; as for the successors of the source, no case of the algorithm introduces new ones. Furthermore, it is clear that the result is a SORE.

Let a generalized SOA A' be given, let A be a copy of A' where all labels are replaced with single distinct symbols. Let $\delta = \texttt{Soa2Sore}(A)$ and let γ be a SORE such that $\mathcal{L}(A) \subseteq \mathcal{L}(\gamma) \subseteq \mathcal{L}(\delta)$.

We argue by induction that $\mathcal{L}(\delta) = \mathcal{L}(\gamma)$ (i.e., we assume that Theorem 27 holds for all recursive calls that `Soa2Sore` makes on A). We distinguish a number of different cases, depending on which clause was used for $\texttt{Soa2Sore}(A)$.

Case 1: The clause in line 3 or the clause in line 4 was used.
This case is trivial.

Case 2: The clause in line 5 was used.
Let U be as chosen in line 6. Let $A_0 = A.\operatorname{extract}(U)$ and $B_0 = A.\operatorname{extract}(U).\operatorname{bend}()$; let z be a symbol not in $\operatorname{term}(A)$ and $B_1 = A.\operatorname{contract}(U, z)$. Let $\hat{\delta_0} = \texttt{Soa2Sore}(B_0)$ and let $\delta_0 = \hat{\delta_0}^+$. We let δ_1 be $\texttt{Soa2Sore}(B_1)$.

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 defined by T.

Claim 1. There is a plussed vertex x in T such that, for the subtree γ_0 rooted at x, we have $\operatorname{term}(\gamma_0) = \operatorname{term}(U)$.
Proof of Claim 1. Let u be the plussed vertex furthest down in T such that $\operatorname{term}(u)$ contains $\operatorname{term}(U)$; such a vertex has to exist in T, as $\rightarrow_\gamma$ is a superrelation of $\rightarrow_A$, where U is a strongly connected component.

Let $c \in \operatorname{term}(u)$ and let $a \in \operatorname{term}(U)$. Then $a \rightarrow_\gamma^+ c$ and $c \rightarrow_\gamma^+ a$; thus, $a \rightarrow_\delta^+ c$ and $c \rightarrow_\delta^+ a$, since $\rightarrow_\delta$ is a superrelation of $\rightarrow_\gamma$. As $\rightarrow_\delta$ has the same strongly connected components as $\rightarrow_A$, and U is the strongly connected component containing a, we get $c \in \operatorname{term}(U)$. □ (FOR CLAIM 1)

Let f be as shown existent in Lemma 30, and let x be the plussed vertex highest up in T such that $\operatorname{term}(x) = \operatorname{term}(U)$. 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 $\varepsilon \notin \mathcal{L}(\gamma_0)$ and $(z \mid \varepsilon)$ otherwise. Let $\gamma_0 = f(\hat{\gamma_0})$. Clearly, it suffices to show that $\mathcal{L}(\gamma_0) = \mathcal{L}(\delta_0)$ and $\mathcal{L}(\gamma_1) = \mathcal{L}(\delta_1)$.

Claim 2. $\mathcal{L}(B_1) \subseteq \mathcal{L}(\gamma_1) \subseteq \mathcal{L}(\delta_1)$.

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

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

$\underline{\mathcal{L}(B_1') \subseteq \mathcal{L}(\gamma_1')}$: Let $a, b \in \mathrm{term}(B_1') \setminus \{z\}$ and suppose $a \rightarrow_{B_1'} b$. We have $a \rightarrow_{B'} b$, and, hence, $a \rightarrow_{\gamma'} b$. Then $a \rightarrow_{\gamma_1'} b$ follows from the definition of γ_1'.

Let $a \in \mathrm{term}(B_1') \setminus \{z\}$ and suppose $a \rightarrow_{B_1'} z$. Thus, there is a $b \in \mathrm{term}(U)$ such that $a \rightarrow_{B'} b$, and, hence, $a \rightarrow_{\gamma'} b$. Then $a \rightarrow_{\gamma_1'} z$ follows from the definition of γ_1'.

Let $b \in \mathrm{term}(B_1') \setminus \{z\}$ and suppose $z \rightarrow_{B_1'} b$. Thus, there is an $a \in \mathrm{term}(U)$ such that $a \rightarrow_{B'} b$, and, hence, $a \rightarrow_{\gamma'} b$. Then $z \rightarrow_{\gamma_1'} b$ follows from the definition of γ_1.

$\underline{\mathcal{L}(\gamma_1') \subseteq \mathcal{L}(\delta_1')}$: Let $a, b \in \mathrm{term}(\gamma_1') \setminus \{z\}$ 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 $a \in \mathrm{term}(\gamma_1') \setminus \{z\}$ and suppose $a \rightarrow_{\gamma_1'} z$. Thus, there is a $b \in \mathrm{term}(U)$ such that $a \rightarrow_{\gamma'} b$, and, hence, $a \rightarrow_{\delta'} b$. We have now $a \rightarrow_{\delta_1'} z$.

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

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

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

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

$$\gamma_0.\mathrm{succ}(\gamma_0.\mathtt{src}) = B_0.\mathrm{succ}(B_0.\mathtt{src}) = A_0.\mathrm{succ}(A_0.\mathtt{src}). \tag{1}$$

Claim 3. We have that

$$B_0.\mathrm{pred}(B_0.\mathtt{snk}) \subseteq \gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk}) \subseteq \delta_0.\mathrm{pred}(\delta_0.\mathtt{snk}).$$

Proof of Claim 3. The statement $\gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk}) \subseteq \delta_0.\mathrm{pred}(\delta_0.\mathtt{snk})$ follows straightforwardly from the choice of γ_0.

Let $c \in B_0.\mathrm{pred}(B_0.\mathtt{snk})$. Suppose first that there is an element $d \in \mathrm{term}(A) \setminus \mathrm{term}(U)$ such that $c \rightarrow_A d$. Then $c \rightarrow_\gamma d$, and, thus, $c \in \gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk})$. Therefore, $A_0.\mathrm{pred}(A_0.\mathtt{snk}) \subseteq \gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk})$.

Suppose now, by way of contradiction, that there is a b with

$$b \in B_0.\mathrm{pred}(B_0.\mathtt{snk}) \ \wedge\ b \notin \gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk}). \tag{2}$$

Then $b \notin A_0.\mathrm{pred}(A_0.\mathtt{snk})$. Hence, the edge from b to $B_0.\mathtt{snk}$ was added by the bend routine, and we know that $b \in W_2$ must hold. Therefore,

$$b \notin A_0.\mathrm{pred}(A_0.\mathtt{snk}) \ \wedge\ b \notin A_0.\mathrm{succ}(A_0.\mathtt{src}). \tag{3}$$

Furthermore, there exist an a with

$$a \in A_0.\mathrm{succ}(A_0.\mathtt{src}) \ \wedge\ b \rightarrow_{A_0} a \tag{4}$$

and a c such that

$$b \in B_0.\mathrm{pred}(B_0.\mathtt{src}) \tag{5}$$

with the property $(*)$: b can be reached from c without crossing any elements of $A_0.\mathrm{succ}(A_0.\mathtt{src})$.

Let T_0 be the syntax tree of γ_0, and let x' be the lowest vertex below which all three letters a, b, and c occur. Let γ_0' be the subexpression of γ_0 that corresponds to the subtree that has x' as root. Due to the choice of x', this vertex must be labeled with a binary operator, which implies that it has a left and right subtree, to which we refer as T_L and T_R, respectively. We call the corresponding expressions γ_L and γ_R. Hence, γ_0' is either $(\gamma_L \cdot \gamma_R)$, or $(\gamma_L \mid \gamma_R)$. The following reasoning applies to both of these two cases.

As c is a predecessor of the sink in γ_0, we know that $\gamma_0'.\mathrm{pred}(\gamma_0'.\mathtt{snk}) \subseteq \gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk})$ must hold. This implies $\gamma_R.\mathrm{pred}(\gamma_R.\mathtt{snk}) \subseteq \gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk})$. Furthermore, if c occurs in γ_L, then $\gamma_L.\mathrm{pred}(\gamma_L.\mathtt{snk}) \subseteq \gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk})$ must hold as well. Analogously, we can observe that $\gamma_L.\mathrm{succ}(\gamma_L.\mathtt{src}) \subseteq \gamma_0.\mathrm{succ}(\gamma_0.\mathtt{src})$ holds; and if a occurs in γ_R, then $\gamma_R.\mathrm{succ}(\gamma_R.\mathtt{src}) \subseteq \gamma_0.\mathrm{succ}(\gamma_0.\mathtt{src})$. We conclude the proof of Claim 2 with the following case analysis.

Case 1: b and c occur in the same subexpression $\gamma' \in \{\gamma_L, \gamma_R\}$.
As x' is the lowest vertex above the three letters a, b, c, the letter a cannot occur in γ'. Hence, $b \in \gamma'.\mathrm{pred}(\gamma'.\mathtt{snk})$ must hold in order to allow $b \rightarrow_\gamma a$ (as given by (4)). But as c occurs in γ', this implies $b \in \gamma_0.\mathrm{pred}(\gamma_0.\mathtt{snk})$, in contradiction to (2).

Case 2: b occurs in γ_L and c in γ_R.
We have that b is only reachable from c if b is an element of $\gamma_L.\mathrm{succ}(\gamma_L.\mathtt{src})$, or by crossing an element of that set. But because of

$$\gamma_L.\mathrm{succ}(\gamma_L.\mathtt{src}) \subseteq \gamma_0.\mathrm{succ}(\gamma_0.\mathtt{src}) \underset{(1)}{=} A_0.\mathrm{succ}(A_0.\mathtt{src}),$$

the former contradicts (3) and the latter cannot happen due to $(*)$.

Case 3: b occurs in γ_R and c in γ_L.
If a occurs in γ_R, we can observe that the path from c to b (given by $(*)$) must

cross an element of $A_0.\mathrm{succ}(A_0.\texttt{src})$, a contradiction similar to the previous case. But if a occurs in γ_L, then $b \in \gamma_R.\mathrm{pred}(\gamma_R.\texttt{snk}) \subseteq \gamma_0.\mathrm{pred}(\gamma_0.\texttt{snk})$ follows from $b \to_A a$ given by (4), a contradiction to (2).

$\square$ (FOR CLAIM 3)

Lastly, we turn to pairs of elements from $\mathrm{term}(U)$.

Claim 4. On $\mathrm{term}(U)$, $\to_{B_0}$ is a subrelation of $\to_{\gamma_0}$, which in turn is a subrelation of $\to_{\delta_0}$.
Proof of Claim 4. This is straightforward, using the properties of f taken from Lemma 30. $\square$ (FOR CLAIM 4)

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

Case 3: The clause in line 9 was used.
This case is trivial from the induction hypothesis, as the language is not changed by the addEpsilon() method.

Case 4: The clause in line 11 was used.
Let v be the only successor of $A.\texttt{src}$; let $a = v.\mathrm{label}()$. Note that a is the only successor of $\gamma.\texttt{src}$. Let $U = \mathrm{term}(\delta) \setminus \{a\}$. As A does not have a strongly connected looped component, neither does $\mathrm{SOA}(\gamma)$; thus, we have $\mathcal{L}(\gamma) = a \cdot \pi_U(\mathcal{L}(\gamma))$. Let γ' equal γ with a replaced by ε and $\delta' = \texttt{Soa2Sore}(A.\mathrm{extract}(U))$. Then we have $\mathcal{L}(A.\mathrm{extract}(U)) \subseteq \mathcal{L}(\gamma') \subseteq \mathcal{L}(\delta')$ and the claim follows by induction.

Case 5: The clause in line 17 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 (the only $^+$ could be on ε or other terminal-free parts, which is unnecessary).

Let v be as chosen in line 17 and $a = v.\mathrm{label}()$. Let $U = A.\mathrm{exclusive}(v)$.

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

Claim 5. There is a subtree γ_0 of γ such that $\mathrm{term}(\gamma_0) = U$.
Proof of Claim 5. Let γ_0 be the smallest subtree such that $U \subseteq \mathrm{term}(\gamma_0)$. Suppose, by way of contradiction, there is $b \in \mathrm{term}(\gamma_0) \setminus U$. By the definition of U, we have that there is $b \in \gamma_0.\mathrm{first}() \setminus U$. From $a \in \gamma.\mathrm{first}()$ and $A.\mathrm{first}() = A.\mathrm{succ}(A.\texttt{src})$ we get $b \in \gamma.\mathrm{first}()$, and, thus, a and b cannot appear in the same word of $\mathcal{L}(\gamma)$. Thus, there is a subtree β of the syntax tree of γ_0 where the root is labeled with "or" and a and b are in different subtrees. In the child tree containing b there cannot be any elements of U, since all elements of U are reachable from a.

Thus, β cannot be all of γ_0, as γ_0 was chosen smallest. β cannot descend from the left child of γ_0, as then all elements of U in the right subtree are reachable via b (or γ_0 not smallest); similarly, β cannot descend from the right child of γ_0, as then all elements of U in the left subtree are not reachable from a (or γ_0 not smallest). $\square$ (FOR CLAIM 5)

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

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

Case 6: The clause in line 21 was used.
In this case we know that $|A.\mathrm{first}()| > 1$, as no other case applies. Furthermore, we will use without mention that A is cycle free.

Let u, v as chosen in line 21. Let z be a symbol not in $\mathrm{term}(A)$. Let $B = A.\mathrm{contract}(\{u, v\}, z)$. Let δ_0 be $\mathtt{Soa2Sore}(B)$.

Let $a = u.\mathrm{label}()$ and $b = v.\mathrm{label}()$. From $u, v \in \delta.\mathrm{first}()$ 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. $\mathcal{L}(\beta)$ is a set of letters.
Proof of Claim 6. Suppose, by way of contradiction, there is a word $w \in \mathcal{L}(\beta)$ of length $\neq 1$. From $A.\mathrm{first}() = A.\mathrm{succ}(A.\mathtt{src})$ we get $w \neq \varepsilon$; thus, the length of w is > 1. Let c be the last symbol of w. As Case 5 does not apply, we have that c is reachable from two different elements of $A.\mathrm{first}()$; let d_0, d_1 be two such elements.

Clearly, d_0 and d_1 are in the same subtree of β; without loss of generality, suppose they are in the same subtree as a. Thus, everything that is reachable in A from both a and b is also reachable from d_0 and d_1; furthermore, c is reachable in A from both d_0 and d_1 but not reachable from b (as c is not in the same subtree β as b). This is a contradiction to the minimality of u, v. $\square$ (FOR CLAIM 6)

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 $\mathcal{L}(B) \subseteq \mathcal{L}(\gamma_0) \subseteq \mathcal{L}(\delta_0)$. From the induction hypothesis we get $\mathcal{L}(\gamma_0) = \mathcal{L}(\delta_0)$; thus, $\mathcal{L}(\gamma) = \mathcal{L}(\delta)$. $\square$ (FOR THEOREM 27)

6 Beyond Single Occurrences

This section examines two other aspects of the descriptive generalization algorithms in the present paper. In Section 6.1, we consider a possible extension to restricted regular expressions that are not limited to a single number of occurrences of each terminal letter. Section 6.2 briefly discusses the descriptive generalization of language classes that are generated by mechanisms which are more powerful than SOAs, but use SOAs, SOREs, or CHAREs as hypotheses.

6.1 Learning k-OREs

While SOREs and SOAs allow only a single occurrence of each terminal letter, Bex et al. [3] introduced the more general concepts of *k-occurrence regular expressions (k-OREs)* and *k-occurrence automata (k-OAs)*. As might be expected, a k-ORE is a regular expression where every terminal symbol occurs at most k times. Analogously, while a SOA has only a single state for each terminal letter a, a k-OA allows up to k states $a^{(1)}, \dots, a^{(k)}$, where $k \geq 1$. Hence, SOAs are 1-OAs, and SOREs are 1-OREs.

A k-OA is called non-deterministic if it has a state that has two successor states with identical labels, and a k-ORE is called non-deterministic if its canonical k-OA (as defined in the next paragraph) is non-deterministic. Unlike SOAs and SOREs, which are deterministic by definition, the same does not hold for k-OAs and k-OREs (for further details, see [3]).

Similarly to the way SOREs can be translated into SOAs, any given k-ORE can be converted into a canonical k-OA: Given a k-ORE α over some alphabet Σ, we transform α into a SORE $\alpha^{(k)}$ over the *marked* alphabet $\Sigma^{(k)}$, which is defined by

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

In other words, every occurrence of some letter a is replaced with an occurrence that is marked with some number (the exact value of each of the marking i is irrelevant to our purposes, as long as every $a^{(i)}$ occurs at most once). We then transform $\alpha^{(k)}$ into a SOA $\mathcal{A}^{(k)} := \text{SOA}(\alpha^{(k)})$, and obtain the k-OA $\mathcal{A}$ over the alphabet Σ by *stripping* the markings from the letters in $\mathcal{A}^{(k)}$ (i. e., every $a^{(i)}$ is replaced with a). Note that $\mathcal{A}$ does not depend on the choice of the marking. More importantly, we observe that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\alpha)$, as $\mathcal{L}(\mathcal{A}^{(k)}) = \mathcal{L}(\alpha^{(k)})$.

Furthermore, note that the characteristic inclusion criterion for SOREs (and SOAs) becomes merely sufficient for deterministic k-OREs (and deterministic k-OAs). This is easily seen by considering the deterministic 2-ORE $\alpha := (ac \mid bc)$ and the deterministic SORE (and, hence, also 2-ORE) $\beta := (a \mid b \mid c)^*$. While $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\beta)$ holds, the canonical OA of β does not cover the canonical OA of α.

Bex et al. propose an algorithm $\mathtt{RWR}^2$ that transforms k-OAs into k-OREs. This algorithm can be paraphrased as follows: First, the input k-OA $\mathcal{A}$ is transformed into a SOA $\mathcal{A}^{(k)}$ over the marked alphabet $\Sigma^{(k)}$. Then $\mathtt{RWR}^2_1$ (i. e., $\mathtt{RWR}^2_\ell$ with $\ell = 1$) is used to compute a SORE $\alpha^{(k)}$ (also over $\Sigma^{(k)}$) for this SOA. In the last step, the markings are stripped from $\alpha^{(k)}$. The resulting k-ORE is called $\mathtt{RWR}^2(\mathcal{A})$.

Although $\mathcal{L}(\mathtt{RWR}^2(\mathcal{A})) \supseteq \mathcal{L}(\mathcal{A})$ holds, two problems might occur. Firstly, even if $\mathcal{A}$ is deterministic, $\mathtt{RWR}^2(\mathcal{A})$ is not necessarily deterministic; and secondly, even if $\mathcal{L}(\mathcal{A})$ is a k-ORE language, $\mathcal{L}(\mathtt{RWR}^2(\mathcal{A})) = \mathcal{L}(\mathcal{A})$ is not guaranteed (cf. Section 4.2 of [3]).

Nonetheless, for a large class of k-OAs, the transformation does not change the language, as shown just below (for the definition of Glushkov representations, see [3]).

Theorem 31 (Bex et al. [3]) *If a k-OA $\mathcal{A}$ is a Glushkov representation of a target k-ORE α, then $\mathtt{RWR}^2(\mathcal{A})$ is equivalent to α. Moreover, if α is deterministic, then so is $\mathtt{RWR}^2(\mathcal{A})$.*

Due to this result, it is possible to use $\mathtt{RWR}^2$ as a subroutine of a k-ORE inference algorithm called `iDREGEX`. Ignoring more technical aspects that are not relevant to the present paper, `iDREGEX` first infers k-OAs and then uses $\mathtt{RWR}^2$ to convert these into k-OREs (for a chosen k).

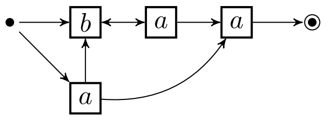
It is natural to ask what happens if the call of $\mathtt{RWR}^2_1$ in $\mathtt{RWR}^2$ is replaced with a call of `Soa2Sore`. We refer to the resulting algorithm as `Koa2Kore`. In other words, given a k-OA $\mathcal{A}$, `Koa2Kore`$(\mathcal{A})$ is defined as the result of applying `Soa2Sore` to $\mathcal{A}^{(k)}$ and then stripping the markings from `Soa2Sore`$(\mathcal{A}^{(k)})$.

We first observe that, similar to $\mathtt{RWR}^2$, `Koa2Kore` neither preserves determinism, nor does it guarantee descriptivity.

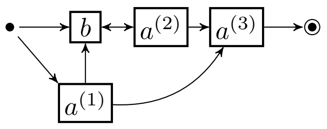
Theorem 32 *There exist deterministic k-OAs $\mathcal{A}_1, \mathcal{A}_2$ such that:*

(1) `Koa2Kore`$(\mathcal{A}_1)$ is not deterministic, and
(2) `Koa2Kore`$(\mathcal{A}_2)$ is not $\mathcal{D}$-descriptive of $\mathcal{L}(\mathcal{A}_2)$, where $\mathcal{D}$ is the class of deterministic k-OREs.

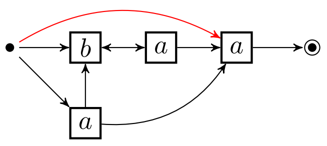
Proof. We begin with the first claim and define $\mathcal{A}_1$ to be the following 3-OA.



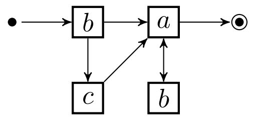
It is easily seen that $\mathcal{A}_1$ is deterministic. By marking the letters in $\mathcal{A}_1$, we obtain the following SOA $\mathcal{A}_1^{(k)}$.



Note that any other possible marking of the occurrences of a would suffice for our purpose. On this SOA, `Soa2Sore` returns the SORE $a^{(1)}?(ba^{(2)})^*a^{(3)}$, which, after stripping the markings, yields the 3-ORE $a?(ba)^*a$. This 3-ORE is not deterministic, which is easily seen by considering its canonical 3-OA (the single new edge is marked red).



To prove the second claim, we define the 2-Oa $\mathcal{A}_2$ as follows (this automaton is also used in [3] (Section 4.2) to prove that $\mathcal{L}(\texttt{RWR}^2(\mathcal{A})) \neq \mathcal{L}(\mathcal{A})$ can hold).



As pointed out in [3], $\mathcal{L}(\mathcal{A}_2)$ is generated by the deterministic 2-Ore $\delta := bc?a(ba)^*$. By applying `Soa2Sore` to any of the two possible marked version of $\mathcal{A}_2$, we obtain the deterministic 2-Ore $\alpha_2 := bc?(ab?)^+$. We observe that,

$$\begin{aligned}\mathcal{L}(\alpha_2) &= \mathcal{L}(bc?(ab?)^+)\\ &= \mathcal{L}(bc?(ab?)(ab?)^*)\\ &= \mathcal{L}(bc?a(b?a)^*b?)\\ &\supset \mathcal{L}(bc?a(ba)^*) = \mathcal{L}(\delta) = \mathcal{L}(\mathcal{A}_2)\end{aligned}$$

holds, which means that α_2 is not descriptive of $\mathcal{L}(\mathcal{A}_2)$. □

On a more positive side, an analogous result to Theorem 31 holds as well.

Theorem 33 *If a k-Oa $\mathcal{A}$ is a Glushkov representation of a target k-Ore α, then `Koa2Kore`$(\mathcal{A})$ is equivalent to α. Moreover, if α is deterministic, then so is `Koa2Kore`$(\mathcal{A})$.*

The proof is analogous to the proof of Theorem 4.8 in [3] (*mutatis mutandis*).

While Theorem 32 demonstrates that `Soa2Sore` cannot be generalized into an algorithm for descriptive k-Ores (at least not in a straightforward manner), Theorem 33 shows that that `Koa2Kore` can be used as a replacement of $\texttt{RWR}^2$ in the algorithm `iDREGEX` in [3].

Although `iDREGEX` can reliably identify target languages that are k-Ore-languages, Theorme 32 also proves that this replacement of $\texttt{RWR}^2$ does not lead to an algorithm for descriptive generalization with respect to deterministic k-Ores. In order to solve this problem, one would first need to properly extend `Soa2Sore` to k-Ores. Accordingly, the authors wish to highlight the following open problem.

Question 1 *Is there an efficient algorithm that, given a deterministic k-Oa $\mathcal{A}$, computes a deterministic k-Ore that is descriptive of $\mathcal{L}(\mathcal{A})$ (w. r. t. the class of deterministic k-Ores)?*

We briefly discuss a possible approach to this problem in Section 8.

As the k-Oa inference step in `iDREGEX` does not guarantee descriptivity, the following question is probably of equal importance:

Question 2 *Is there an efficient algorithm that, given a finite language S, computes a deterministic k-Oa that is descriptive of S (w. r. t. the class of deterministic k-Oas)?*

6.2 Approximation of Larger Language Classes

According to Bex et al. [2], a common difficulty in the creation of XML Schema Definitions is that many non-expert users struggle with the distinction between a regular expression and a deterministic regular expression (while this is not explicitly mentioned, the same reasoning also applies to DTDs). One potential solution that is examined in [2] is taking a user-specified non-deterministic regular expression α and transforming it into a deterministic regular expression δ such that (in the terminology of the present paper) δ is $\mathcal{D}$-descriptive of $\mathcal{L}(\alpha)$, where $\mathcal{D}$ is chosen to be the full class of deterministic regular expressions. As shown in Theorem 7 in [2], this choice of $\mathcal{D}$ is too large; as there are languages that do not have a descriptive deterministic regular expression.

In contrast to this, the main results of the present paper show that this approach is viable if one is willing to use CHAREs or SOREs instead of the full class of deterministic regular expressions. In fact, not only can one compute descriptive CHAREs or SOREs from non-deterministic regular expression, but from any class of language representation for which one can compute the descriptive SOA from the description of a language L. This includes a wide range of comparatively powerful classes of language description mechanisms (e. g., the class of pushdown automata, or the class of context-free grammars – cf. Hopcroft and Ullman [14], or almost any other introductory textbook). While it might not always be obvious whether these sets can be computed for some given class of descriptors, the following sufficient criterion might serve as first guidance.

Theorem 34 *Let $\mathcal{D}$ be a class of language description mechanisms. Then $\text{SOA}(\mathcal{L}(\delta))$ can be computed for every $\delta \in \mathcal{D}$ if there is an algorithm that, given any $\delta \in \mathcal{D}$ and any regular language R, decides whether $\mathcal{L}(\delta) \cap R = \emptyset$ holds.*

Proof. As the terminal alphabet of every $\delta \in \mathcal{D}$ is fixed, one can simply construct the regular languages for each possible first letter, each possible last letter, and each possible combination of 2-factors, and check whether these occur in $\mathcal{L}(\delta)$. Then $\text{SOA}(\mathcal{L}(\delta))$ can be constructed according to Corollary 7 by adding the appropriate edge for each language where the intersection with $\mathcal{L}(\delta)$ is non-empty. □

As an alternative to the condition from Theorem 34, one can require that $\mathcal{D}$ is effectively closed under intersection with regular languages (i. e., that a description for $\mathcal{L}(\delta) \cap R$ not only exists, but can be computed) and that the emptiness problem for $\mathcal{D}$ is decidable. As this implies that $\mathcal{L}(\delta) \cap R = \emptyset$ is decidable, Theorem 34 applies.

Of course, this approach is not without drawbacks. Considering the difference in expressive power, a SORE (or CHARE) that is descriptive of a context-free language L might only be a very rough approximation of L. But in addition to this, as the following theorem shows, it is not even possible to decide whether the descriptive expression generates the same language.

Theorem 35 *For any arbitrary CFG G, the three following questions are undecidable.*

(1) Is $\mathcal{L}(G)$ a Soa *language?*
(2) Is $\mathcal{L}(G)$ a Sore *language?*
(3) Is $\mathcal{L}(G)$ a Chare *language?*

Proof. We proof the theorem for all three cases at once. This proof is a slight modification of the proof of Theorem 8.11 in Hopcroft and Ullman [14][5] for the undecidability of the question whether $\mathcal{L}(G) = \Sigma^*$ holds for an arbitrary CFG G.

In that proof, Hopcroft and Ullman show that, given an arbitrary Turing machine M, one can effectively construct a CFG G_M with terminal alphabet $\Sigma = \Gamma \cup Q \cup \{\#\}$ such that $\mathcal{L}(G_M) = \Sigma^*$ holds if and only if $\mathcal{L}(M) = \emptyset$. In particular, their construction defines $\mathcal{L}(G_M)$ to be the set of invalid computations of M, which we shall refer to as I_M. Basically, I_M is the set of all strings that do not encode accepting runs of M (for the exact definition, see [14], Chapter 8.6).

By the definition of invalid computations, $\text{first}(I_M) = \text{last}(I_M) = \Sigma$ and $\text{gram}_2(I_M) = \Sigma^2$ must hold. Hence, $\textsc{Soa}(I_M) = \Sigma^*$ holds for every Turing machine M. Accordingly, I_M is a Soa language if and only if $I_M = \Sigma^*$ (otherwise, we would arrive at the contradictory observation that $\textsc{Soa}(I_M) = \Sigma^*$ is not Soa-descriptive of I_M).

Therefore, given an arbitrary Turing machine M, one can effectively construct a CFG G_M such that $\mathcal{L}(G_M)$ is a Soa language if and only if $\mathcal{L}(M) = \emptyset$. The question whether $\mathcal{L}(M) = \emptyset$ holds for an arbitrary Turing machine M is undecidable (again, cf. [14]); hence, the claim follows for Soa languages.

As a descriptive Sore (or a descriptive Chare) cannot be less general than the descriptive Soa, the claim immediately follows for Sore languages and Chare languages as well. □

Theorem 35 shows that, while it is possible to transform CFGs into each of Soas, Sores, and Chares with the guarantee of a minimal generalization, it is not possible to tell whether this step causes a proper generalization (this happens if the original language is not a Soa-, Sore-, or Chare language), or whether the language remains unchanged (if the original language is expressible in the respective model). Hence, it is not only impossible to decide how much information is lost during the transformation (or how many new words are introduced), but also whether there is any loss of information at all.

Note that this result can be adapted to all those language description mechanisms that can express I_M, or similarly constructed encodings of invalid computations of Turing machines.

[5] Note that the referenced material is not included in Hopcroft et al. [13] (the second edition of [14]).

7 Example DTDs

This section contains some example element type declarations that were obtained by running a prototype implementation of `Soa2Chare` and `Soa2Sore`[6] against a sample XML database, as well as a comparison to the declarations from the original DTD. These examples illustrate what kind of DTDs the algorithms generate, and what insights they might offer into the analyzed data.

The algorithms were tested against the version of the Mondial database [17] that, according to the website, has been revised in summer 2009 (the corresponding DTD states a revision date of April 2009). Note that this version of Mondial considerably differs from the older version provided by [18], which was used for the experimental evaluation by Bex et al. [4]. Most importantly, the XML file and the DTD from [17] are consistent with the data, which is not the case for [18] (as already pointed out by [4]).

First, note that with the single exception of `country`, all element type declarations in the Mondial DTD are Chares; and all are Sores. While it would have been interesting to examine the generalization process on an example where the provided DTD contains a declaration that is not a Sore, all examples that the authors were able to locate contained only single occurrences of element names.

As most of the other element type declarations are trivial, the test focussed on the following elements (listed with their respective number of occurrences in the XML file): `city` (3261), `country` (241), `desert` (62), `estuary` (220), `island` (281), `lake` (132), `mountain` (242), `organization` (152), `province` (1531), `river` (221), `sea` (40), and `source` (216).

The element `province` is the only of the examined elements for which the both computed Chare and the computed Sore are identical to the original declaration:

```
name,area?,population+,city*
```

For each of the elements `estuary`, `mountain`, `organization`, `sea`, and `source`, the computed Chare is identical to the computed Sore, which in turn is less general than the definition in the DTD. In each of the cases, the only difference between is that elements that are marked as optional in the DTD either always occur in the XML file, or do not occur at all (which means that in the former case, `?` is omitted or `*` is replaced with `+`, and in the latter case, optional elements from the original DTD do not appear in the inferred element type declaration).

The situation is similar, but more interesting, for the elements `island` and `river`. Here, the inferred Chares are identical to the declaration in the DTD, but each of the inferred Sores is less general. As an example, consider the element `island`.

– original declaration in DTD and inferred Chare:
  `name,islands?,located*,area?,elevation?,longitude?,latitude?`

[6] Available at `http://www.tks.informatik.uni-frankfurt.de/ddf/downloads`

– inferred SORE:
  `name,islands?,located*,area?,elevation?,(longitude,latitude)?`

In the original declaration, `longitude` and `latitude` are both optional. But as evidenced by the descriptive SORE, the two elements are not used independently – neither of them can occur without the other. Unlike CHAREs, SOREs are able to express such dependencies.

The results for the elements `city`, `desert`, and `lake` can be understood as a combination of the previous phenomena. Due to removal of options, the CHARE is less general than the original declaration; and the SORE is even less general due to dependencies as in the previous example.

The only case where the language that is generated by the inferred descriptive CHARE is incomparable to the language from the original declaration is for the element `country`. As already mentioned above, this declaration is the only declaration in the DTD that is not a CHARE.

```
name,population?,population_growth?,infant_mortality?,
gdp_total?,gdp_agri?,gdp_ind?,gdp_serv?,inflation?,
(indep_date|dependent)?,government?,encompassed*,ethnicgroups*,
religions*,languages*,border*,(province+|city+)
```

It is easily seen that the language that is described by this expression is not a CHARE language, due to the subexpression `(province+|city+)`.

Of course, the presence of such a non-CHARE expression in the DTD does not mean that it is necessary to describe the actual data in the XML file. For example, it might be possible that this subexpression is too general, and that the less general subexpression `(province|city)` is a better description of the data. But this is not the case, as the following descriptive CHARE shows.

```
name,population,population_growth?,infant_mortality?,
gdp_total?,gdp_agri?,gdp_ind?,gdp_serv?,inflation?,
(indep_date|dependent)?,government?,encompassed+,ethnicgroups*,
religions*,languages*,border*,province*,city*
```

Apparently, the non-CHARE subexpression is not too general for the actual data, as the descriptive CHARE replaces it with the more general expression `province*,city*`.

The only reason that the language of the inferred CHARE is incomparable to one of the original declaration (instead of being strictly more general) is the subexpression `encompassed+` instead of `encompassed*`.

In contrast to this, the inferred descriptive SORE is less general than the declaration from the DTD:

```
name,population,(population_growth,infant_mortality?)?,
(gdp_total,gdp_agri?,(gdp_ind,gdp_serv)?)?,inflation?,
(indep_date|dependent)?,government?,encompassed+,ethnicgroups*,
religions*,languages*,border*,(province+|city+)
```

Three subexpressions of this SORE are particularly noteworthy. First, note that `(province+|city+)` is present, as in the original declaration. Second, with `(population_growth,infant_mortality?)?`, we have a dependency in the actual XML data that was not expressed in the DTD (similar to `longitude` and `latitude` in the previous example): The elements `population_growth` and `infant_mortality` are both optional, but the latter never appears without the former. Finally, the subexpression `(gdp_total,gdp_agri?,(gdp_ind,gdp_serv)?)?` is another case of such a dependency.

Although the Mondial XML file might be considered comparatively small, and its DTD rather simple, the examples in the present section should provide some insights into the expressive power of CHAREs and SOREs. In particular, these examples illustrate that SOREs are able to express a certain kind of dependency for optional elements. Nonetheless, it should be mentioned that in order to be SORE-expressible, this dependency has to be local (e. g., as for `(longitude,latitude)?`).

In order to illustrate such an inexpressible dependency, consider the sample $S := \{abc, b\}$. While a is present if and only if c is present, the two letters are always separated by b. Using `Soa2Sore` on SOA(S) yields the descriptive SORE $a?bc?$, which does not express this dependency (and is, in fact, a CHARE).

8 Conclusions and Further Work

This paper introduces algorithms 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 [4], 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 [4] at least with respect to both accuracy and speed. In order to examine the potential practical value of these results, an extensive experimental evaluation of the algorithms would be very interesting. This would 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. [4] (Section 8) propose extending these models with numerical predicates; i. e., one could write $a^{\geq 1, \leq 3}$, with $\mathcal{L}(a^{\geq 1, \leq 3}) = \{a, aa, aaa\}$. With an additional post-processing step, the algorithms in [4] can be used to infer CHAREs and SOREs that are extended with counting. This extension can also be adapted to the approaches in the present paper. Basically, one replaces each $^+$ or $^+?$ with appropriate bound that describes how often the expression under the

$^+$ is repeated; e. g., the sample $\{a, aaa\}$ would lead to descriptive expression a^+, additional post-processing would turn this into $a^{\geq 1, \leq 3}$. Note that, in the form described in [4], this approach can only learn finite languages, as positive data does not allow do distinguish between a^+ and $a^{\leq n}$ for sufficiently large n. But this can be fixed by providing the post-processing algorithm with an additional threshold t, and removing those upper bounds $\leq n$ for which $n \geq t$ with unbounded; e. g., $a^{\geq 2, \leq n}$ with $n \geq t$ would become $a^{\geq 2}$.

On the topic of probabilistic learning, if one is willing to fix a set of probability distributions on the sample space, the learning algorithms could be adapted to feature a variant of stochastic finite learning (introduced by Rossmanith and Zeugmann [20]). It might be possible to derive algorithms which, with high probability, give descriptive generalizations from a very small set of (randomly chosen) examples. This could lead to inference algorithms that do not need to process the whole input, but only a random subset, which might be interesting for very large datasets.

From the authors' point of view, Questions 1 and 2 (cf. Section 6.1) are the most interesting. In other words: Is it possible to extend the inference algorithms discussed in the present paper from SOAs and SOREs to deterministic k-OAs and deterministic k-OREs? It seems that one would need to develop not only a good generalization of SOAs, but also a "good" inclusion criterion, preferably syntactic. This idea 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 [10] 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 [10] 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 [10], 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 [10] 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. [15] 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

This work was done while Dominik D. Freydenberger was visiting the Max-Planck-Institute for Informatics in Saarbrücken. The authors wish to thank the anonymous referees both of the conference version of this article and of the present version for their helpful remarks. Also we wish to thank Ping Lu for finding a mistake in an earlier version of the algorithm for finding descriptive SOREs.

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, W. Martens, and F. Neven. Simplifying XML schema: effortless handling of nondeterministic regular expressions. In *Proc. ACM SIGMOD International Conference on Management of Data, SIGMOD 2009*, pages 731–744, 2009.
3. 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.
4. 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.
5. G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *Proc. 33rd International Conference on Very Large Data Bases, VLDB 2007*, pages 998–1009, 2007.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 2001.
7. H. Fernau. Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4):521–541, 2009.
8. D. D. Freydenberger and T. Kötzing. Fast learning of restricted regular expressions and DTDs. In *Proc. 16th International Conference on Database Theory, ICDT 2013*, pages 45–56, 2013.
9. D. D. Freydenberger and D. Reidenbach. Existence and nonexistence of descriptive patterns. *Theoretical Computer Science*, 411(34–36):3274–3286, 2010.
10. D. D. Freydenberger and D. Reidenbach. Inferring descriptive generalisations of formal languages. *Journal of Computer and System Sciences*, 79:622–639, 2013.
11. 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.
12. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
13. J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley Publishing Company, second edition edition, 2001.
14. J. Hopcroft and J. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley Publishing Company, 1979.
15. 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.
16. 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.
17. W. May. Information extraction and integration with FLORID: The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://dbis.informatik.uni-goettingen.de/Mondial`.
18. G. Miklau. XMLData repository, 2002. Available from `http://www.cs.washington.edu/research/xmldatasets`.

19. 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.
20. P. Rossmanith and T. Zeugmann. Stochastic finite learning of the pattern languages. *Machine Learning*, 44(1–2):67–91, 2001.