Solving String Constraints with Practical Regular Expressions

Anonymous Author(s)

Abstract

Text of abstract ....

Keywords keyword1, keyword2, keyword3

1 Introduction

Strings are a fundamental data type in virtually all programming languages.

One effective automatic testing method for identifying subtle programming errors is based on *symbolic execution* [13] and combinations with dynamic analysis called *dynamic symbolic execution* [5, 6, 12, 15, 16]. See [7] for an excellent survey.

Unlike purely random testing, which runs only *concrete* program executions on different inputs, the techniques of symbolic execution analyse *static* paths (also called symbolic executions) through the software system under test. Such a path can be viewed as a constraint φ (over appropriate data domains) and the hope is that a fast solver is available for checking the satisfiability of φ (i.e. to check the *feasibility* of the static path), which can be used for generating inputs that lead to certain parts of the program or an erroneous behaviour.

In this paper, we focus on two string operations with emphasis on practical usage of regular expressions. Rather than textbook style regular expressions, regular expressions used in programming languages are considerably more involved. On particular feature we consider is the capturing group. This is particularly useful for string pattern matching where it can be returned what subexpression matched which substring.

The *string-replace function*, which may be used to replace all occurrences of a string matching a pattern by another string.

The replace function (especially the replace-all functionality) is omnipresent in HTML5 applications [14, 18, 19].

A regular expression (shortened as regex) is a sequence of characters that define a search pattern. Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation.

The semantics of regular expressions with capturing groups and back references is rather involved. One of the reasons is that different languages may choose different semantics for a regex to match the string when the regex is served as a pattern.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

To capture the semantics, priority is introduced, giving rise to an extension of the standard finite-state automata. However, this is not sufficient for capturing string operations. For that purpose, we introduce a new transducer model, prioritized streaming string transducer (PSST) which is a combination of priority which is essential for modelling capturing groups and streaming transducers which are a highly expressive formalism for modelling string operations.

The new contribution:

- extending Streaming String Transducers: [10]
- universal solution
-

2 Motivating Example

We use the JavaScript program in Figure 1 as a motivating example to illustrate the main approach of this paper. The function "changeNameFormat" in Figure 1 transforms the name format of an author list from the ACM bibtex style to the DBLP bibtex style. For instance, if a paper is authored by Alice Brown and John Smith, then the author list in the ACM bibtex style is "Brown, Alice and Smith, John", while in the DBLP bibtex style it is "Alice Brown and John Smith".

The input of the function "changeNameFormat" is authorList, which should follow the pattern specified by the regular expression autListReg. Intuitively, autListReg stipulates that authorList joins the strings of the following structure, with the word "and" as the separator: concatenation of two sequences of alphabetic letters (denoted by \w), bar (denoted by –), dot (denoted by .), or the blank symbol (denoted by \s), with the comma in between. The symbols ^ and $ denote the beginning and end of an input.

The name format of each author, except for the last one, is specified by the regular expression nameReg in Figure 1, which describes the pattern of a surname (a sequence of alphabetic letters, – or .), followed by the comma, then a given name, finally the word "and" or $ denoting the end of the input, where \s represents the blank symbol and \w represents alphabetic letters, digits, or the underline symbol _. There are three capturing groups in nameReg, intuitively recording surname, given name, and the word "and" or $ respectively. Note that surnames or given names may comprise several words. The subexpression (?:\s+[\w-.]+) in nameReg denotes the non-capturing groups, i.e. matching \s+[\w-.]+ but not remembering the match. Notice that the global flag "g" is used in nameReg so that the name format of each author is transformed. TL: mention this is replaceAll? :LT The name format transformation is done by the replace function,

```
function changeNameFormat(authorList)
{
  var autListReg = /^([\w-\.\s]+,[\w-\.\s]+\sand\s)*[\w-\.\s]+,[\w-\.\s]+$/;
  if (autListReg.test(authorList)) {
    var nameReg = /([\w-\.]+(?:\s+[\w-\.]+)*),\s*([\w-\.]+(?:\s+[\w-\.]+)*)((\s+and\s+)|$)/g;
    return authorList.replace(nameReg, "$2 $1$3");
  }
  else return authorList;
}
```

Figure 1. Change the name format of an author list: A motivating example

i.e. authorList.replace(nameReg, "$2 $1 $3"), where $1, $2, $3 refer to the match of the first, second and third capturing group respectively.

For a symbolic execution of the JavaScript program in Figure 1, one needs to model the semantics of capturing groups as well as references. For this purpose, we introduce prioritized streaming string transducers (PSST, see Section 5). Then replace(nameReg, "$2 $1$3") can be captured by $\mathcal{T}_{\mathsf{nameReg}}$, where the priorities are used to model the greedy semantics of + or $*$ (see Definition 3.5 in Section 3) and the string variables are used to record the matches of capturing groups.

An intuitive invariant property of the output of changeNameFormat is that, if authorList matches autListReg, then none of its outputs contains the comma ",". The invariant property holds iff the following program in single static assignment form is infeasible (namely, there does not exist a value of authorList so that the program can execute to the end):

$$\begin{aligned} &\mathsf{assert}\,(\mathsf{authorList} \in \mathsf{autListReg})\,;\\ &\mathsf{ret} := \mathcal{T}_{\mathsf{nameReg}}(\mathsf{authorList});\\ &\mathsf{assert}\,(\mathsf{ret} \in .*,.*)\,, \end{aligned} \tag{1}$$

where assert (authorList ∈ autListReg) requires that authorList matches autListReg, while assert (ret ∈ .*, .*) requires that ret contains an occurrence of , (. can match any symbol, except the line breaker).

The path feasibility problem of the program in Equation (2) is solved by a "backward" reasoning as follows (see Section 6 for the details):

- At first, we compute $(\mathcal{T}_{\mathsf{nameReg}})^{-1}(.*,.*)$, i.e. the preimage of the language .*, .* under $\mathcal{T}_{\mathsf{nameReg}}$ (see Theorem 5.6), remove ret := $\mathcal{T}_{\mathsf{nameReg}}$(authorList), and add the assertion assert $\big(\mathsf{authorList} \in (\mathcal{T}_{\mathsf{nameREg}})^{-1}(.*,.*)\big)$, resulting into the following program,

$$\begin{aligned} &\mathsf{assert}\,(\mathsf{authorList} \in \mathsf{autListReg})\,;\\ &\mathsf{assert}\,(\mathsf{ret} \in .*,.*)\,;\\ &\mathsf{assert}\,\big(\mathsf{authorList} \in (\mathcal{T}_{\mathsf{nameREg}})^{-1}(.*,.*)\big)\,. \end{aligned} \tag{2}$$

- Then, we check the nonemptiness of the intersection of the regular languages autListReg and $(\mathcal{T}_{\mathsf{nameREg}})^{-1}(.*,.*)$. If the intersection is empty, then the invariant property holds, otherwise, it does not.

3 Preliminaries

Throughout the paper, $\mathbb{Z}^+$ denotes the set of positive integers, and $\mathbb{N}$ denotes the set of natural numbers. Furthermore, for $n \in \mathbb{Z}^+$, let $[n] := \{1, \ldots, n\}$.

We use Σ to denote a finite set of letters, called *alphabet*. A *string* over Σ is a finite sequence of letters from Σ. We use Σ^* to denote the set of strings over Σ and ε to denote the empty string. Moreover, for convenience, we use Σ^ε to denote $\Sigma \cup \{\varepsilon\}$. A string w' is called a *prefix* of w if $w = w'w''$ for some string w''. We use $\mathsf{Pref}(w)$ to denote the set of prefixes of w. For a prefix w_1 of w, let $w = w_1w_2$, then we use $w_1^{-1}w$ to denote w_2.

Definition 3.1 (Finite-state automata)**.** A *(nondeterministic) finite-state automaton* (FA) over a finite alphabet Σ is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times \Sigma^\varepsilon \times Q$ is the transition relation.

For an input string w, a *run* of $\mathcal{A}$ on w is a sequence $q_0\sigma_1q_1 \ldots \sigma_nq_n$ such that $w = \sigma_1 \cdots \sigma_n$ and $(q_{j-1}, \sigma_j, q_j) \in \delta$ for every $j \in [n]$. The run is said to be *accepting* if $q_n \in F$. A string w is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. In particular, the empty string ε is accepted by $\mathcal{A}$ if $q_0 \in F$. The set of strings accepted by $\mathcal{A}$, i.e., the language *recognised* by $\mathcal{A}$, is denoted by $\mathscr{L}(\mathcal{A})$. The *size* $|\mathcal{A}|$ of $\mathcal{A}$ is defined to be the cardinality of the set Q of states, which will be used when the computational complexity is concerned.

For convenience, for $a \in \Sigma$, we use $\delta^{(a)}$ to denote the relation $\{(q, q') \mid (q, a, q') \in \delta\}$.

3.1 Extended regular expressions

An (extended) regular expression with capturing group and back reference is defined as follows.

Definition 3.2 (Regular expressions with capturing group and back reference, ERegExp)**.**

$$e \stackrel{\text{def}}{=} \emptyset \mid \varepsilon \mid a \mid \$n \mid e + e \mid e \cdot e \mid e^* \mid e^{*?} \mid (e),$$

where $a \in \Sigma, n \in \mathbb{Z}^+$.

We abbreviate $e \cdot e^*$ as e^+ and $e \cdot e^{*?}$ as $e^{+?}$. Moreover, for $\Gamma = \{a_1, \ldots, a_k\} \subseteq \Sigma$, we write Γ for $a_1 + \cdots + a_k$ and thus $\Gamma^* \equiv (a_1 + \cdots + a_k)^*$.

Note that the parentheses in ERegExp are used for both precedence and capturing groups. TL: do we consider change the notation to [e]? :LT Parenthesis pairs are indexed according to the occurrence sequence of their left parentheses, and it is required that every back reference $\$n$ occurs after the n-th pair of parentheses. For instance, $((a + b)^*)c\$1$ is in ERegExp, where \$1 refers to the matching of the subexpression $(a + b)^*$. Intuitively, it denotes the set of strings of the form ucu, where u is a string of a and b.

We use ERegExp[CG] to denote the fragment of ERegExp excluding $\$n$, and REF to denote the set of expressions generated by $e \stackrel{\text{def}}{=} \varepsilon \mid a \mid \$n \mid e \cdot e$.

Definition 3.3 (Subexpression). For any two ERegExp e and r, we say r is a subexpression of e, if either $r = e$ or

- If $e = e_1e_2$ or $e_1 + e_2$ then r is a subexpression of e_1 or e_2
- If $e = e_1^*$, $e_1^{*?}$, or (e_1), then e_1 is a subexpression of e.

We use $S(e)$ to denote the set of all subexpressions of e.

We define the semantics of ERegExp, which uses a concept of match of $e \in$ ERegExp to string w defined as follows.

Definition 3.4 (Match of ERegExp to string). A **match** of ERegExp e to a string w is defined by a finite directed and ordered tree T, whose nodes are elements of $\Sigma^* \times S(e)$ satisfying the following constraints: Its root is (w, e), and for any node $\alpha = (w', e')$ in T, we have:

- If $e' = e'_1 \cdot e'_2$, then α has two children $\alpha_1 = (w'_1, e'_1)$ and $\alpha_2 = (w'_2, e'_2)$ where $w' = w'_1w'_2$.
- If $e' = e'_1 + e'_2$, then α has a single child $\alpha_1 = (w', e'_i)$ where $i \in \{1, 2\}$.
- If $e' = e_1'^*$ or $e_1'^{*?}$, then either $w' = \varepsilon$ in which case α is a leaf, or there is $k \geqslant 1$ such that α has k children $\alpha_1 = (w'_1, e'_1), \ldots, \alpha_k = (w'_k, e'_1)$ where $w' = w'_1 \ldots w'_k$ and for all $i \in [k]$, $w'_i \neq \varepsilon$ (even if $L(e'_1)$ could contain ε).
- If $e' = (e'_1)$, then α has a single child $\alpha_1 = (w', e'_1)$.
- If $e' = a$ (resp. $e' = \varepsilon$), then α is a leaf and $w' = a$ (resp. $w' = \varepsilon$).
- If $e' = \$n$, then α is a leaf of T, moreover, let $e'' \in S(e)$ be enclosed by the n-th pair of parentheses in e and $\beta = (w_1, e'')$ be the last node preceding α in T according to the left-to-right ordering of the nodes, then $w' = w_1$.

We use T_α to represent the subtree of T rooted at α. The notation $C(T)$ refers to the sequence of direct children of the root node of T (and thus all direct subtrees). We use $\mathscr{M}_w(e)$ to denote the set of all match trees of e to w. Moreover, for $L \subseteq \Sigma^*$, we use $\mathscr{M}_L(e)$ to denote the set of match trees of e to some $w \in L$. We also use $\mathscr{L}(e)$ to denote $\{w \in \Sigma^* \mid \mathscr{M}_w(e) \neq \emptyset\}$.

Definition 3.5 (Semantics of ERegExp). For any ERegExp e and a string w, we recursively define an order on $\mathscr{M}_{\text{Pref}(w)}(e)$, written $T >_{w,e} T'$ for $T, T' \in \mathscr{M}_{\text{Pref}(w)}(e)$, as follows:

- $e = \varepsilon$ or $e = a$ or $e = \$n$. There is only one match tree, thus the order $>_{w,e}$ is empty.
- $e = (e_1)$. Suppose that $C(T) = (w_1, e_1)$ and $C(T') = (w_2, e_1)$ for some $w_1, w_2 \in \text{Pref}(w)$. Then $T >_{w,e} T'$ iff $T_{(w_1,e_1)} >_{w,e_1} T'_{(w_2,e_1)}$.
- $e = e_1 + e_2$.
  - If $C(T) = (w_1, e_1)$ and $C(T') = (w_2, e_2)$ for some $w_1, w_2 \in \text{Pref}(w)$, then $T >_{w,e} T'$.
  - If $C(T) = (w_i, e_i)$ and $C(T') = (w'_i, e_i)$ for some $i \in \{1, 2\}$ and $w_i, w'_i \in \text{Pref}(w)$, then $T >_{w,e} T'$ iff $T_{(w_i,e_i)} >_{w,e_i} T'_{(w'_i,e_i)}$.
- $e = e_1 \cdot e_2$. Suppose $C(T) = (w_1, e_1)(w_2, e_2)$ and $C(T') = (w'_1, e_1)(w'_2, e_2)$, then $T >_{w,e} T'$ when either $T_{(w_1,e_1)} >_{w,e_1} T'_{(w'_1,e_1)}$, or $w_1 = w'_1$ and $T_{(w_2,e_2)} >_{w_1^{-1}w,e_2} T'_{(w'_2,e_2)}$.
- $e = e_1^*$.
  - If T' is a single node (ε, e), but T is not, then $T >_{w,e} T'$.
  - Otherwise, suppose $C(T) = (w_1, e_1) \ldots (w_k, e_1)$ and $C(T') = (w'_1, e_1) \ldots (w'_l, e_1)$, we have $T >_{w,e} T'$ iff either when $C(T')$ is a proper prefix of $C(T)$, or for the first index j such that $w_j \neq w'_j$, we have $T_{(w_j,e_1)} >_{(w_1 \cdots w_{j-1})^{-1}w,e_1} T'_{(w'_j,e_1)}$.
- $e = e_1^{*?}$.
  - If T is a single node (ε, e), but T' is not, then $T >_{w,e} T'$.
  - Otherwise, suppose $C(T) = (w_1, e_1) \ldots (w_k, e_1)$ and $C(T') = (w'_1, e_1) \ldots (w'_l, e_1)$, we have $T >_{w,e} T'$ iff either when $C(T)$ is a proper prefix of $C(T')$, or for the first index j such that $w_j \neq w'_j$, we have $T_{(w_j,e_1)} >_{(w_1 \cdots w_{j-1})^{-1}w,e_1} T'_{(w'_j,e_1)}$.

For $e \in$ ERegExp and $w \in \mathscr{L}(e)$, the *accepting match* of e to w, denoted by $M_e(w)$, is the supremum of $\mathscr{M}_w(e)$. Moreover, for $e \in$ ERegExp and $w \in \Sigma^*$, if there is $w \in \Sigma^*\mathscr{L}(e)\Sigma^*$, then the (first) *match* of e in w is defined as the accepting match of e to w' such that $w = w_1w'w_2$ for some $w_1, w_2 \in \Sigma^*$ and for every nonempty suffix w'_1 of w_1, $w'_1w'w_2 \notin \mathscr{L}(e)\Sigma^*$, otherwise, the match of e in w is undefined.

Example 3.6. Let $e = ((b^*) \cdot ((b \cdot a^*)|\varepsilon)) \cdot a^*$ and $w = baa$. Then $\mathscr{M}_w(e) = \{T_1, T_2, T_3\}$ as illustrated in Figure 2: (i), (ii), (iii). Because $(T_2)_{(\varepsilon,b^*)}$ is a single node, while $(T_1)_{(b,b^*)}$ is not, we have $(T_1)_{(b,b^*)} >_{w,b^*} (T_2)_{(\varepsilon,b^*)}$, thus $(T_1)_{(b,(b^*))} >_{w,(b^*)} (T_2)_{(\varepsilon,(b^*))}$. From the fact that $(T_1)_{(b,(b^*))}$ and $(T_2)_{(\varepsilon,(b^*))}$ are the left subtrees of $(T_1)_{(b,(b^*)\cdot((b\cdot a^*)|\varepsilon))}$ and $(T_2)_{(ba,(b^*)\cdot((b\cdot a^*)|\varepsilon))}$ respectively, we deduce that $(T_1)_{(b,(b^*)\cdot((b\cdot a^*)|\varepsilon))} >_{w,(b^*)\cdot((b\cdot a^*)|\varepsilon)} (T_2)_{(ba,(b^*)\cdot((b\cdot a^*)|\varepsilon))}$, thus $(T_1)_{(b,((b^*)\cdot((b\cdot a^*)|\varepsilon)))} >_{w,((b^*)\cdot((b\cdot a^*)|\varepsilon))} (T_2)_{(ba,((b^*)\cdot((b\cdot a^*)|\varepsilon)))}$. From this, we conclude that $T_1 >_{w,e} T_2$. Similarly, we have $T_1 >_{w,e} T_3$. Therefore, $M_e(w) = T_1$. Note that in T_1, the capturing group $((b^*) \cdot ((b \cdot a^*)|\varepsilon))$ does not acquire the longest match. Moreover, the match of e in $baabba$

is *baa*, where the matches of the first and second capturing group are b and ε respectively, while the match of the third capturing group is undefined.

Remark 1. *Intuitively, the semantics of* ERegExp *in Definition 3.5 is eager for $e_1 + e_2$ and greedy for e^*, which is adopted by the script languages Perl, PHP, and JavaScript etc [11]. In comparison, POSIX regular expressions require the leftmost and longest match, of which we leave the investigation as the future work.*

4 The string logic

In this section, we define the string-manipulating programming language SL considered in this paper.

The SL language is defined by

$$S \quad ::= \quad z := x \cdot y \mid y := \mathsf{extract}_{i,e}(x) \mid y := \mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x) \mid \mathsf{assert}\ (x \in e) \mid S; S$$

where

- $\cdot$ is the string concatenation operation which concatenates two strings,
- for the extract function, $i \in \mathbb{N}$, $e \in \mathsf{ERegExp}[\mathsf{CG}]$,
- for the replaceAll operation, $\mathsf{pat} \in \mathsf{ERegExp}[\mathsf{CG}]$, $\mathsf{rep} \in \mathsf{REF}$,
- for assertions, $e \in \mathsf{ERegExp}$.

The extract function models the regular-expression match function in programming languages, e.g. str.match(regexp) function in Javascript. Specifically, the $\mathsf{extract}_{i,e}(x)$ function extracts the match of the i-th capturing group in the match of e in x, if the match of e in x exists (otherwise, the return value of the function is undefined). Note that if $i = 0$, then $\mathsf{extract}_{i,e}(x)$ returns the match of e in x. For instance, let $e = ((b^*) \cdot ((b \cdot a^*)|\varepsilon)) \cdot a^*$, then $\mathsf{extract}_{1,e}(baabba) = b$ and $\mathsf{extract}_{2,e}(baabba) = \varepsilon$, as shown in Example 3.6.

The $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$ function is indexed by the *pattern* $\mathsf{pat} \in \mathsf{ERegExp}[\mathsf{CG}]$ and the *replacement* string $\mathsf{rep} \in \mathsf{REF}$. For a given input string x, the function identifies the first, second, ..., match of pat in x and replace them with the corresponding strings specified by the replacement string (where the references are replaced by the corresponding matches of the capturing groups). For instance, let $\mathsf{pat} = ((b^*) \cdot ((b \cdot a^*)|\varepsilon)) \cdot a^*$ and $\mathsf{rep} = \#\$1$, then $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(baabba) = \#b\#bb$.

Without loss of generality, we assume that all the SL programs are in single static assignment (SSA) form, that is, each variable x is assigned at most once, moreover, if it is assigned, then all its occurrences in the right-hand-sides of the assignment statements or in assertions are after the assignment statement of x.

For a SL program S, a variable x occurring in S is said to be an *input* variable if x does not occur in the left-hand-side of assignment statements. The *path feasibility* problem of a SL program is to decide whether there are valuations of the input variables so that the program can execute to the end.

It turns out that the path feasibility problem is undecidable, attributed to the the back references in assertion statements.

Proposition 4.1. *The path feasibility problem of* SL *is undecidable.*

The proof of Proposition 4.1 is obtained by an encoding of post correspondence problem (PCP). Let Σ be a finite alphabet such that $\# \notin \Sigma$ and $[n] \cap \Sigma = \emptyset$, $(u_i, v_i)_{i \in [n]}$ be a PCP instance with $u_i, v_i \in \Sigma^*$. A solution of the PCP instance is a string $i_1 \cdots i_m$ with $i_j \in [n]$ for every $j \in [m]$ such that $u_{i_1} \cdots u_{i_m} = v_{i_1} \cdots v_{i_m}$. We will use replaceAll to encode the generation of the strings $u_{i_1} \cdots u_{i_m}$ and $v_{i_1} \cdots v_{i_m}$ from $i_1 \cdots i_m$, then use a regular expression with capturing groups and back references to verify the equality of $u_{i_1} \cdots u_{i_m}$ and $v_{i_1} \cdots v_{i_m}$. Specifically, suppose $\# \notin \Sigma$, then the PCP instance is encoded by the following SL program,

$$\begin{array}{l}\mathsf{assert}\ (x_0 \in \{1, \cdots, n\}^+)\,; \\ x_1 := \mathsf{replaceAll}_{1,u_1}(x_0); \cdots; x_n := \mathsf{replaceAll}_{n,u_n}(x_{n-1}); \\ y_1 := \mathsf{replaceAll}_{1,v_1}(x_0); \cdots; y_n := \mathsf{replaceAll}_{n,u_n}(y_{n-1}); \\ z := x_n \# y_n; \mathsf{assert}\ (z \in (\Sigma^+)\#\$1)\,.\end{array}$$

We shall show that the path feasibility problem is decidable, if the uses of back references in assertion statements are forbidden, which turns out to be the situation in practice. In the sequel, we will use $\mathsf{SL}_{\mathsf{reg}}$ to denote the collection of SL programs where no back references occur in assertion statements. The decision procedure for $\mathsf{SL}_{\mathsf{reg}}$ utilizes a new model called prioritized streaming string transducers, which will be defined in the next section.

5 Prioritized Streaming String Transducers

In this section, we introduce prioritized streaming string transducers (PSST), which extend prioritized finite-state automata (PFA) proposed in [4]. We shall utilize PSST to model the eager and greedy semantics of ERegExp as well as the behavior of replaceAll functions where both capturing groups and back references occur.

For a finite set Q, let $\overline{Q} = \bigcup_{n \in \mathbb{N}} \{(q_1, \ldots, q_n) \mid \forall i \in [n], q_i \in Q \wedge \forall i, j \in [n], i \neq j \rightarrow q_i \neq q_j\}$. Intuitively, $\overline{Q}$ is the set of sequences of non-repetitive elements from Q. In particular, the empty sequence $\varepsilon \in \overline{Q}$. Note that the length of each sequence from $\overline{Q}$ is bounded by $|Q|$. For a sequence $P = (q_1, \ldots, q_n) \in \overline{Q}$ and $q \in Q$, we write $q \in P$ if $q = q_i$ for some $i \in [n]$. Moreover, for $P_1 = (q_1, \ldots, q_m) \in \overline{Q}$ and $P_2 = (q'_1, \ldots, q'_n) \in \overline{Q}$, we say $P_1 \cap P_2 = \emptyset$ if $\{q_1, \ldots, q_m\} \cap \{q'_1, \ldots, q'_n\} = \emptyset$.

Definition 5.1 (Prioritized Finite-state Automata)**.** A *prioritized finite-state automaton* (PFA) over a finite alphabet Σ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, \tau, q_0, F)$ where $\delta \in Q \times \Sigma \rightarrow \overline{Q}$ and $\tau \in Q \rightarrow \overline{Q} \times \overline{Q}$ such that for every $q \in Q$, if $\tau(q) = (P_1; P_2)$,

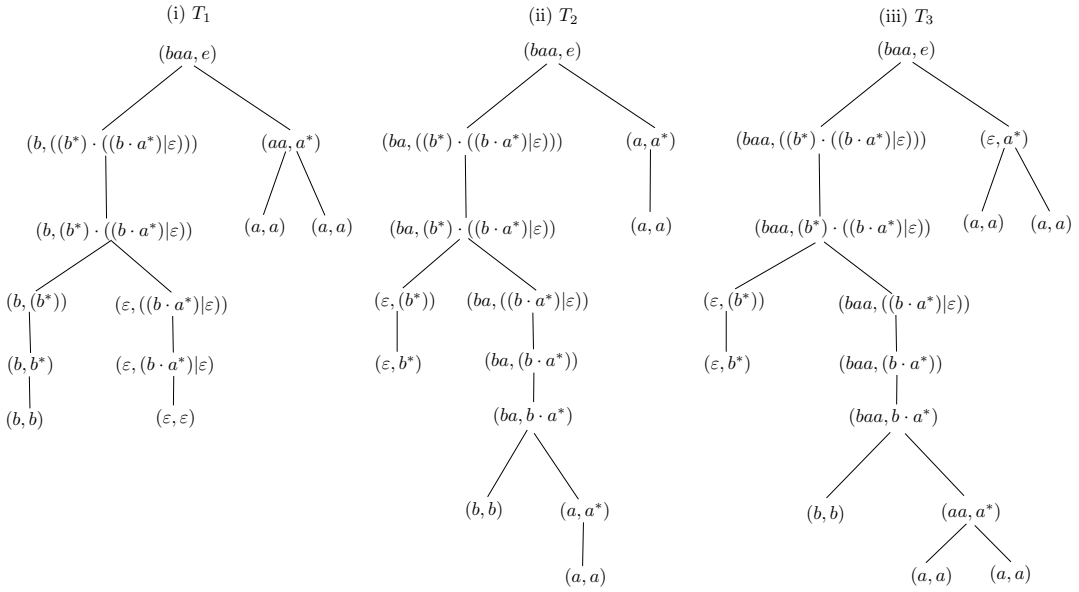


Figure 2. Match trees of $e = ((b^*) \cdot ((b \cdot a^*)|\varepsilon)) \cdot a^*$ to $w = baa$

then $P_1 \cap P_2 = \emptyset$. The definition of Q, q_0 and F is the same as ordinary FA.

For $\tau(q) = (P_1; P_2)$, we will use $\pi_1(\tau(q))$ and $\pi_2(\tau(q))$ to denote P_1 and P_2 respectively. With slight abuse of notation, we write $q \in (P_1; P_2)$ for $q \in P_1 \cup P_2$. Intuitively, $\tau(q) = (P_1; P_2)$ specifies the ε-transitions at q, with the intuition that the ε-transitions to the states in P_1 resp. P_2 have higher resp. lower priorities than the non-ε-transitions out of q.

A run of $\mathcal{A}$ on a string w is a sequence $q_0\sigma'_1q_1 \dots \sigma'_mq_m$ such that

- for any $i \in [m]$, either $\sigma'_i \in \Sigma$ and $q_i \in \delta(q_{i-1}, \sigma'_i)$, or $\sigma'_i = \varepsilon$ and $q_i \in \tau(q_{i-1})$
- $w = \sigma'_1 \cdots \sigma'_m$,
- for every subsequence $q_i\sigma'_{i+1}q_{i+1} \dots \sigma'_jq_j$ such that $i < j$ and $\sigma'_{i+1} = \cdots = \sigma'_j = \varepsilon$, it holds that for every $k, l : i \le k < l < j$, $(q_k, q_{k+1}) \neq (q_l, q_{l+1})$. (Intuitively, each transition occurs at most once in a sequence of ε-transitions.)

Note that it is possible that $\delta(q, \sigma) = ()$, namely, there is no σ-transition out of q. It is easy to observe that given a string w, the length of a run of $\mathcal{A}$ on w is $O(|w||\mathcal{A}|)$. For any two runs $p = q_0\sigma_1q_1 \dots \sigma_mq_m$ and $p' = q_0\sigma'_1q'_1 \dots \sigma'_nq'_n$ such that $\sigma_1 \dots \sigma_m = \sigma'_1 \dots \sigma'_n$, we say that p is of a higher priority over p' if

- either p' is a prefix of p (in this case, the transitions of p after p' are all ε-transitions),
- or there is an index j satisfying one of the following constraints:
  - $q_0\sigma_1q_1 \dots q_{j-1}\sigma_j = q_0\sigma'_1q'_1 \dots q'_{j-1}\sigma'_j$, $q_j \neq q'_j$, $\sigma_j \in \Sigma$, and $\delta(q_{j-1}, \sigma_j) = (\dots, q_j, \dots, q'_j, \dots)$,
  - $q_0\sigma_1q_1 \dots q_{j-1}\sigma_j = q_0\sigma'_1q'_1 \dots q'_{j-1}\sigma'_j$, $q_j \neq q'_j$, $\sigma_j = \varepsilon$, and either $\pi_1(\tau(q_{j-1})) = (\dots, q_j, \dots, q'_j, \dots)$, or $\pi_2(\tau(q_{j-1})) = (\dots, q_j, \dots, q'_j, \dots)$, or $q_j \in \pi_1(\tau(q_{j-1}))$ and $q'_j \in \pi_2(\tau(q_{j-1}))$,
  - $q_0\sigma_1q_1 \dots q_{j-1} = q_0\sigma'_1q'_1 \dots q'_{j-1}$, $\sigma_j = \varepsilon$, $\sigma'_j \in \Sigma$, $q_j \in \pi_1(\tau(q_{j-1}))$, and $q'_j \in \delta(q_{j-1}, \sigma'_j)$,
  - $q_0\sigma_1q_1 \dots q_{j-1} = q_0\sigma'_1q'_1 \dots q'_{j-1}$, $\sigma_j \in \Sigma$, $\sigma'_j = \varepsilon$, $q_j \in \delta(q_{j-1}, \sigma_j)$, and $q'_j \in \pi_2(\tau(q_{j-1}))$.

An *accepting* run of $\mathcal{A}$ on w is a run $q_0\sigma_1q_1 \dots \sigma_mq_m$ of $\mathcal{A}$ on w with the *highest* priority such that $q_m \in F$. (Note that a run $q_0\sigma_1q_1 \dots \sigma_mq_m$ of $\mathcal{A}$ with the highest priority may not be accepting, i.e. satisfy $q_m \in F$.) The language of $\mathcal{A}$, denoted as $\mathscr{L}(\mathcal{A})$, is the set of strings on which $\mathcal{A}$ has an accepting run.

Note that PFAs differ from FAs only in the way that a string is accepted; they both define regular languages.

Example 5.2. The PFAs corresponding to a^* and $a^{*?}$ respectively are illustrated in Figure 3: (i) and (ii), where the dashed line represents $\pi_2(\tau(q_0))$ (of lower priority than the a-transition), the thicker solid line represents $\pi_1(\tau(q_0))$ (of higher priority than the a-transition), and the doubly circled state q_1 is a final state.

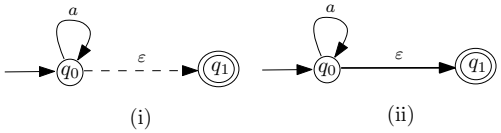


Figure 3. The PFAs for a^* and $a^{*?}$

The priorities of PFAs are used to model the eager and greedy semantics of ERegExp, as we shall see in Section 6.1.

We then introduce prioritized streaming string transducers, a new class of transducers that combine prioritized transducers [4] and streaming string transducers [1, 2].

Definition 5.3 (Prioritized Streaming String Transducers)**.** A *prioritized streaming string transducer* (PSST) is a tuple $\mathcal{T} = (Q, \Sigma, X, \delta, \tau, E, q_0, F)$, where Q a finite set of states, Σ is the input and output alphabet, X is a finite set of variables, $\delta \in Q \times \Sigma \to \overline{Q}$, $\tau \in Q \to \overline{Q} \times \overline{Q}$, E is a partial function from $Q \times \Sigma^\varepsilon \times Q$ to $X \to (X \cup \Sigma)^*$, i.e. the set of assignments, $q_0 \in Q$ is the initial state, and F is a partial function from Q to $(X \cup \Sigma)^*$.

A run of $\mathcal{T}$ on a string w is a sequence $q_0\sigma_1 s_1 q_1 \dots \sigma_m s_m q_m$ such that

- for each $i \in [m]$,
  - either $\sigma_i \in \Sigma$, $q_i \in \delta(q_{i-1}, \sigma_i)$, and $s_i = E(q_{i-1}, \sigma_i, q_i)$,
  - or $\sigma_i = \varepsilon$, $q_i \in \tau(q_{i-1})$ and $s_i = E(q_{i-1}, \varepsilon, q_i)$,
- for every subsequence $q_i\sigma_{i+1}s_{i+1}q_{i+1} \dots \sigma_j s_j q_j$ such that $i < j$ and $\sigma_{i+1} = \dots = \sigma_j = \varepsilon$, it holds that for every $k, l : i \le k < l < j$, $(q_k, q_{k+1}) \neq (q_l, q_{l+1})$.

For any pair of runs $p = q_0\sigma_1 s_1 \dots \sigma_m s_m q_m$ and $p' = q_0\sigma'_1 s'_1 \dots \sigma'_n s'_n q'_n$ such that $\sigma_1 \dots \sigma_m = \sigma'_1 \dots \sigma'_n$, the definition that p is of a higher priority over p' is similar to PFAs.

An accepting run of $\mathcal{T}$ on an input w is a run of $\mathcal{T}$ on w of the highest priority, say $q_0\sigma_1 s_1 \dots \sigma_m s_m q_m$, such that $F(q_m)$ is defined. The output of $\mathcal{T}$ on w, denoted by $\mathcal{T}(w)$, is defined as $\eta_m(F(q_m))$, where $\eta_0(x) = \varepsilon$ for each $x \in X$, and $\eta_i(x) = \eta_{i-1}(s_i(x))$ for every $1 \le i \le m$ and $x \in X$. Note that here we abuse the notation $\eta_m(F(q_m))$ and $\eta_{i-1}(s_i(x))$ by taking a function η from X to Σ^* as a function from $(X \cup \Sigma)^*$ to Σ^*, which maps each $\sigma \in \Sigma$ to σ and each $x \in X$ to $\eta(x)$. If there is no accepting run of $\mathcal{T}$ on w, then $\mathcal{T}(w) = \bot$, namely, the output of $\mathcal{T}$ on w is undefined. The string relation defined by $\mathcal{T}$, denoted by $\mathcal{R}_\mathcal{T}$, is $\{(w, \mathcal{T}(w)) \mid w \in \Sigma^*, \mathcal{T}(w) \neq \bot\}$.

Example 5.4. The PSST $\mathcal{T}_{\mathsf{nameReg}} = (Q, \Sigma, X, \delta, \tau, E, q_0, F)$ mentioned in Section 2 is illustrated in Figure 4, where $Q = \{q_0, \dots, q_{24}\}$, $X = \{x_1, x_2, x_3, x_4\}$ with x_1, x_2, x_3 recording the matches of the 1st, 2nd, 3rd capturing group, and x_4 recording the string after the replacements, $F(q_{24}) = x_4$ denotes the final output, and δ, τ, E are illustrated by the edges, where the dashed edges denote the ε-transitions of lower priorities than the non-ε-transitions and the symbol ℓ is used to denote the currently scanned input letter. For instance, $\delta(q_3, \backslash s) = (q_3)$, $\delta(q_3, \ell) = ()$ for every $\ell \in \Sigma \setminus \{\backslash s\}$, $\tau(q_3) = ((); (q_4))$, and $E(q_3, \backslash s, q_3)(x_1) = x_1 \backslash s$. Since the ε-transition has lower priority than the $\backslash s$-transition at the state q_3, whenever the currently scanned letter is $\backslash s$ at q_3, $\mathcal{T}_{\mathsf{nameReg}}$ will choose to go to q_3 greedily, until there is no more $\backslash s$. (In this case, it has to choose the ϵ-transition and goes to q_4.) Note that the identity assignments, e.g. $E(q_3, \backslash s, q_3)(x') = x'$ for $x' \in \{x_2, x_3, x_4\}$, are omitted in Figure 4, for readability.

Definition 5.5 (Pre-image)**.** For a string relation $R \subseteq \Sigma^* \times \Sigma^*$ and $L \subseteq \Sigma^*$, we define the *pre-image* of L under R as $R^{-1}(L) := \{w \in \Sigma^* \mid \exists w'.\ w' \in L \text{ and } (w, w') \in R\}$.

Theorem 5.6 (Pre-image of PSST)**.** *Given a* PSST $\mathcal{T} = (Q_T, \Sigma, X, \delta_T, \tau_T, E_T, q_{0,T}, F_T)$ *and an FA* $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$*, we can compute an FA* $\mathcal{B} = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$ *in exponential time such that* $\mathscr{L}(\mathcal{B}) = \mathcal{R}_\mathcal{T}^{-1}(\mathscr{L}(\mathcal{A}))$.

TL: another way is to first define B as a PFA, could make the construction a bit modular? :LT ZL: Add a counter example for this natural idea. :LZ

Let $\mathcal{T} = (Q_T, \Sigma, X, \delta_T, \tau_T, E_T, q_{0,T}, F_T)$ be a PSST and $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$ be an FA. Without loss of generality, we assume that $\mathcal{A}$ contains no ε-transitions.

To illustrate the intuition of the FA construction, let us start with the following natural idea of firstly constructing a PFA $\mathcal{B}$ for the pre-image: $\mathcal{B}$ simulates the run of $\mathcal{T}$ on w, and, for each $x \in X$, records an $\mathcal{A}$-abstraction of the string stored in x, that is, the set of state pairs $(p, q) \in Q_A \times Q_A$ such that starting from p, $\mathcal{A}$ can reach q after reading the string stored in x. Specifically, the states of $\mathcal{B}$ are of the form (q, ρ) with $q \in Q$ and $\rho \in (\mathcal{P}(Q_A \times Q_A))^X$. Moreover, the priorities of $\mathcal{B}$ inherit those of $\mathcal{A}$. The PFA $\mathcal{B}$ is then transformed to an equivalent FA by simply dropping all priorities. We refer to this FA as $\mathcal{B}'$.

Nevertheless, it turns out that this construction method is flawed: A string w is in $\mathcal{R}_\mathcal{T}^{-1}(\mathscr{L}(\mathcal{A}))$ iff the (unique) accepting run of $\mathcal{T}$ on w produces an output w' that is accepted by $\mathcal{A}$. However, a string w is accepted by $\mathcal{B}'$ iff there is a run of $\mathcal{T}$ on w, not necessarily of the highest priority, producing an output w' that is accepted by $\mathcal{A}$. The following example illustrates the flaw of the construction above.

Example 5.7. Let $\mathcal{T}$ be the PSST and $\mathcal{A}$ be the FA in Figure 5, that is,

- $\mathcal{T} = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{x_0\}, \delta_T, \tau_T, E_T, q_0, F_T)$, where $\delta_T(q_0, \sigma) = (q_0)$, $\delta_T(q_1, a) = (q_1)$, $\delta_T(q_2, \sigma) = (q_2)$, $\tau_T(q_0) = ((q_1); ())$, $\tau_T(q_1) = ((q_0, q_2); ())$, and $\tau_T(q_2) = ((); ())$, $E_T(q_0, \sigma, q_0)(x_0) = x_0\sigma$, $E_T(q_1, \varepsilon, q_0)(x_0) = x_0 c$, $E_T(q_1, \varepsilon, q_2)(x_0) = x_0 c$, $E_T(q_2, \sigma, q_2)(x_0) = x_0\sigma$, for $\sigma \in \{a, b\}$. Moreover, $F_T(q_2) = x_0$;
- $\mathcal{A} = (\{p_0\}, \{a, b, c\}, \delta_A, p_0, \{p_0\})$, where $\delta_A = \{(p_0, \sigma, p_0) \mid \sigma = b, c\}$.

Let us consider $w = a$. The accepting run of $\mathcal{T}$ on w is $q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow[x_0 := x_0 c]{\varepsilon} q_0 \xrightarrow[x_0 := x_0 a]{a} q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow[x_0 := x_0 c]{\varepsilon} q_2$, producing an output $cac \notin \mathscr{L}(\mathcal{A})$. Therefore, $a \notin \mathcal{R}_\mathcal{T}^{-1}(\mathscr{L}(\mathcal{A}))$. Nevertheless, if we consider the FA $\mathcal{B}'$ constructed from $\mathcal{T}$ and $\mathcal{A}$, it turns out that $\mathcal{B}'$ does accept w, witnessed by the run $(q_0, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_1, \{(p_0, p_0)\}) \xrightarrow{a} (q_1, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_2, \{(p_0, p_0)\})$. On the other hand, the run of $\mathcal{B}'$ corresponding to the accepting run of $\mathcal{T}$ on w, i.e. $(q_0, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_1, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_0, \{(p_0, p_0)\}) \xrightarrow{a} (q_0, \emptyset) \xrightarrow{\varepsilon} (q_1, \emptyset) \xrightarrow{\varepsilon}$

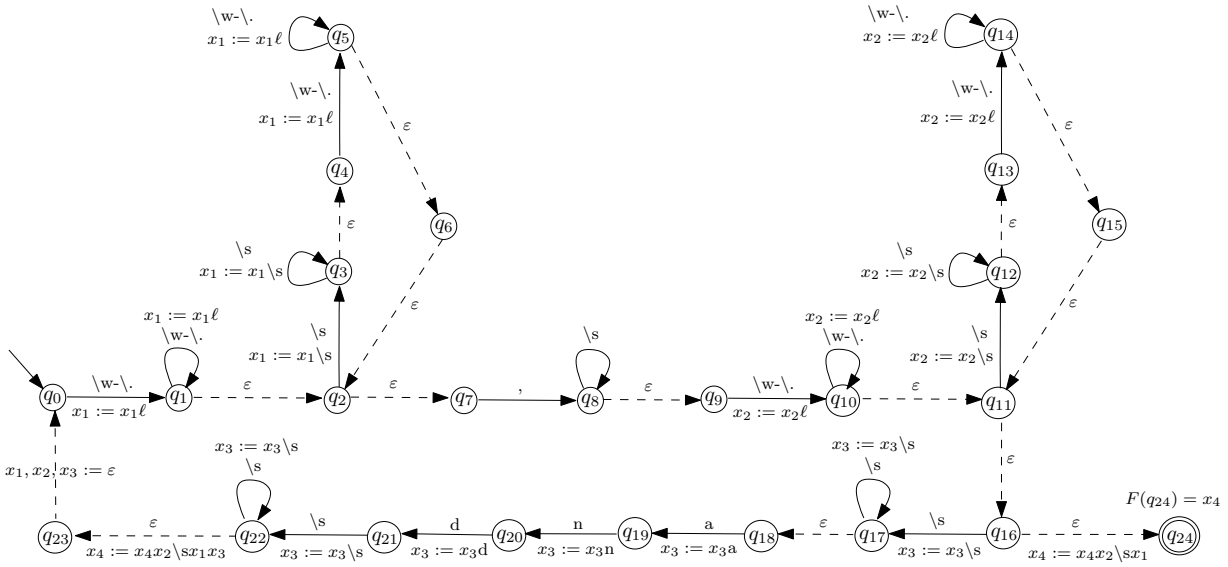


Figure 4. The PSST $\mathcal{T}_{\mathsf{nameReg}}$

$(q_2, \emptyset)$, is not accepting, where $\{(p_0, p_0)\}$ and $\emptyset$ are the $\mathcal{A}$-abstractions of x_0.

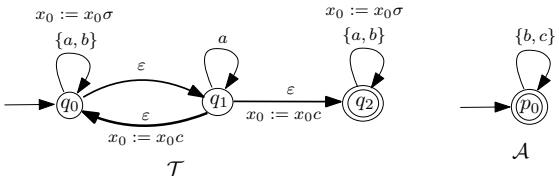


Figure 5. A counterexample to disprove the flawed pre-image construction method

Proof of Theorem 5.6. Let $\mathcal{T} = (Q_T, \Sigma, X, \delta_T, \tau_T, E_T, q_{0,T}, F_T)$ be a PSST and $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$ be an FA. For convenience, we use $\mathcal{E}(\tau_T)$ to denote $\{(q, q') \mid q' \in \tau_T(q)\}$.

Our goal is to construct an FA $\mathcal{B}$ that simulates the run of $\mathcal{T}$ on w, and, for each $x \in X$, records an $\mathcal{A}$-abstraction of the string stored in x, that is, the set of state pairs $(p, q) \in Q_A \times Q_A$ such that starting from p, $\mathcal{A}$ can reach q after reading the string stored in x. To simulate the runs of $\mathcal{T}$, it is necessary to record all the states accessible from a run of a higher priority to ensure the current run is an accepting run of $\mathcal{T}$ with the highest priority. Moreover, $\mathcal{B}$ also remembers the set of ε-transitions of $\mathcal{T}$ after the latest non-ε-transition to ensure that no transition occurs twice in a sequence of ε-transitions of $\mathcal{T}$ TL: need to revisit here :LT .

Specifically, each state of $\mathcal{B}$ is of the form (q, ρ, Λ, S), where $q \in Q_T$, $\rho \in (\mathcal{P}(Q_A \times Q_A))^X$, $\Lambda \subseteq \mathcal{E}(\tau_T)$, and $S \subseteq Q_T$. Note that when recording in S all the states accessible from a run of higher priority, we do not take the non-repetition of ε-transitions into consideration TL: the opposite? :LT since if a state is reachable by a sequence of ε-transitions where some ε-transitions are repeated, then there exists also a sequence of non-repeated ε-transitions reaching the state. Moreover, when simulating a σ-transition of $\mathcal{T}$ (where $\sigma \in \Sigma$) at a state (q, ρ, Λ, S), $\mathcal{B}$ should saturate the states in S by ε-transitions, that is, compute the set of states that are reachable from the states in S by a sequence of ε-transitions, then apply a σ-transition to them. For technical reasons, when constructing $\mathcal{B}$, we assume that this saturation happens when a state is added to S for the first time. Therefore, at a state (q, ρ, Λ, S), all the states reachable from the states in S by sequences of ε-transitions in $\mathcal{T}$ have already been in S.

Before the construction of $\mathcal{B}$, we introduce some notations.

- For $S \subseteq Q_T$, $\delta_T^{(ip)}(S, a) = \{q_1' \mid \exists q_1 \in S, q_1' \in \delta_T(q_1, a)\}$.
- For $q \in Q_T$, if $\tau_T(q) = (P_1, P_2)$, then $\tau_T^{(ip)}(\{q\}) = S$ such that $S = P_1 \cup P_2$. Moreover, for $S \subseteq Q_T$, we define $\tau_T^{(ip)}(S) = \bigcup_{q \in S} \tau_T^{(ip)}(\{q\})$. We also use $\left(\tau_T^{(ip)}\right)^*$ to denote the ε-closure of $\mathcal{T}$, namely, $\left(\tau_T^{(ip)}\right)^*(S) = \bigcup_{n \in \mathbb{N}} \left(\tau_T^{(ip)}\right)^n(S)$, where $\left(\tau_T^{(ip)}\right)^0(S) = S$, and for $n \in \mathbb{N}$, $\left(\tau_T^{(ip)}\right)^{n+1}(S) = \tau_T^{(ip)}\left(\left(\tau_T^{(ip)}\right)^n(S)\right)$.
- For $S \subseteq Q_T$ and $\Lambda \subseteq \mathcal{E}(\tau_T)$, we use $\left(\tau_T^{(ip)} \backslash \Lambda\right)^*(S)$ to denote the set of states reachable from S by sequences of ε-transitions where *no* transitions (q, ε, q') such that $(q, q') \in \Lambda$ are used.
- For $\rho \in (\mathcal{P}(Q_A \times Q_A))^X$ and $s \in X \to (X \cup \Sigma)^*$, we use $s(\rho)$ to denote ρ' that is obtained from ρ as follows: For each $x \in X$, if $s(x) = \varepsilon$, then $\rho'(x) = \{(p, p) \mid p \in Q_A\}$, otherwise, let $s(x) = b_1 \cdots b_\ell$ with $b_i \in \Sigma \cup X$ for each $i \in [\ell]$, then $\rho'(x) = \theta_1 \circ \cdots \circ \theta_\ell$, where $\theta_i = \delta_A^{(b_i)}$ if $b_i \in \Sigma$, and $\theta_i = \rho(b_i)$ otherwise.

We are ready to present the formal construction of $\mathcal{B} = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$.

- $Q_B = Q_T \times (\mathcal{P}(Q_A \times Q_A))^X \times \mathcal{P}(\mathcal{E}(\tau_T)) \times \mathcal{P}(Q_T)$,
- $q_{0,B} = (q_{0,T}, \rho_\varepsilon, \emptyset, \emptyset)$ where $\rho_\varepsilon(x) = \{(q, q) \mid q \in Q\}$ for each $x \in X$,
- δ_B comprises
  - the tuples $((q, \rho, \Lambda, S), \sigma, (q_i, \rho', \Lambda', S'))$ such that
    - $\sigma \in \Sigma$,
    - $\delta_T(q, \sigma) = (q_1, \ldots, q_i, \ldots, q_m)$,
    - $s = E(q, \sigma, q_i)$,
    - $\rho' = s(\rho)$,
    - $\Lambda' = \emptyset$, (Intuitively, Λ is reset.)
    - let $\tau_T(q) = (P_1, P_2)$, then $S' = \left(\tau_T^{(ip)}\right)^*(\{q_1, \ldots, q_{i-1}\} \cup \delta_T^{(ip)}\left(S \cup \left(\tau_T^{(ip)} \setminus \Lambda\right)^*(P_1'), \sigma\right))$, where $P_1' = \{q' \in P_1 \mid (q, q') \notin \Lambda\}$; (Note that according to the semantics of PSSTs, when computing the set of states reachable from q through an ε-transition to some $q' \in P_1$ first and a sequence of ε-transitions starting from q' next, the transitions (q'', ε, q''') with $(q'', q''') \in \Lambda$ should be excluded.)
  - the tuples $((q, \rho, \Lambda, S), \varepsilon, (q_i, \rho', \Lambda', S'))$ such that
    - $\tau_T(q) = ((q_1, \ldots, q_i, \ldots, q_m); \cdots)$,
    - $(q, q_i) \notin \Lambda$,
    - $s = E(q, \varepsilon, q_i)$,
    - $\rho' = s(\rho)$,
    - $\Lambda' = \Lambda \cup \{(q, q_i)\}$,
    - $S' = S \cup \left(\tau_T^{(ip)} \setminus \Lambda\right)^*(\{q_j \mid j \in [i-1], (q, q_j) \notin \Lambda\})$;
  - the tuples $((q, \rho, \Lambda, S), \varepsilon, (q_i, \rho', \Lambda', S'))$ such that
    - $\tau_T(q) = ((q_1', \ldots, q_n'); (q_1, \ldots, q_i, \ldots, q_m))$,
    - $(q, q_i) \notin \Lambda$,
    - $s = E(q, \varepsilon, q_i)$,
    - $\rho' = s(\rho)$,
    - $\Lambda' = \Lambda \cup \{(q, q_i)\}$,
    - $S' = S \cup \{q\} \cup \left(\tau_T^{(ip)} \setminus \Lambda\right)^*\left(\{q_j' \mid j \in [n], (q, q_j') \notin \Lambda\} \cup \{q_j \mid j \in [i-1], (q, q_j) \notin \Lambda\}\right)$. (Note that here we include q into S', since the non-ε-transitions out of q have higher priorities than the transition (q, ε, q_i).)
- Moreover, F_B is the set of states $(q, \rho, \Lambda, S) \in Q_B$ such that
  1. $F_T(q)$ is defined,
  2. for any $q' \in S$, $F_T(q')$ is not defined,
  3. if $F_T(q) = \varepsilon$, then $q_{0,A} \in F_A$, otherwise, let $F_T(q) = b_1 \cdots b_\ell$ with $b_i \in \Sigma \cup X$ for each $i \in [\ell]$, then $(\theta_1 \circ \cdots \circ \theta_\ell) \cap (\{q_{0,A}\} \times F_A) \neq \emptyset$, where for each $i \in [\ell]$, if $b_i \in \Sigma$, then $\theta_i = \delta_A^{(b_i)}$, otherwise, $\theta_i = \rho(b_i)$.

□

Note that the above construction does not utilize the so-called *copyless* property [1, 2], thus it works for general, or *copyful* PSST [10].

Example 5.8. Figure 6 gives the correct FA $\mathcal{B}$ encoding $\mathcal{R}_{\mathcal{T}}^{-1}(\mathcal{L}(\mathcal{A}))$ in Example 5.7 using the construction method introduced above. Only those states reachable from the initial state r_0 are shown. Table 1 shows the correspondence between the states in the figure and the states of $\mathcal{B}$, where $\rho_1(x) = \{(p_0, p_0)\}$ and $\rho_2(x) = \emptyset$.

Note that many states are redundant. For efficiency, we minimize the FA after the construction. See Section 7 for implementation details.

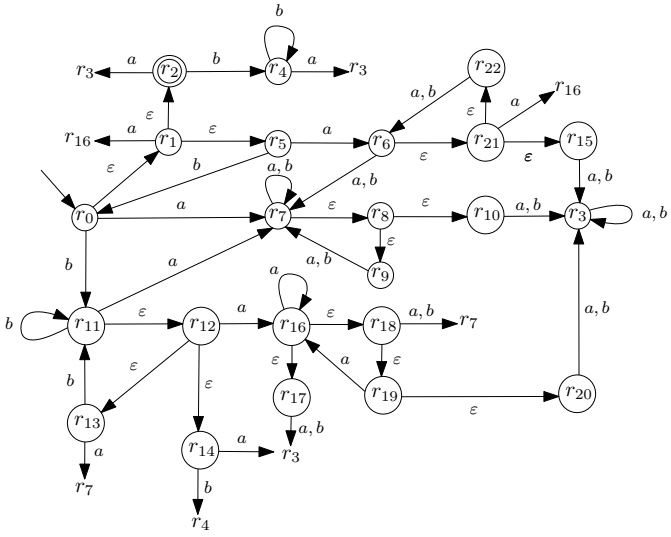


Figure 6. FA $\mathcal{B}$ that encodes $\mathcal{R}_{\mathcal{T}}^{-1}(\mathcal{L}(\mathcal{A}))$

6 Decision procedure

In this section, we show the main result of this paper.

Theorem 6.1. *The path feasibility of* $\mathsf{SL}_{\mathsf{reg}}$ *is decidable.*

The proof of Theorem 6.1 is obtained by first showing that both extract and replaceAll functions can be transformed into PSST, then computing the pre-images of regular languages under PSSTs and removing all the assignment statements, finally solving the nonemptiness of intersection of regular languages. The aforementioned procedure extends the approach of backward reasoning proposed in [8, 9] in the following sense: While [8, 9] used the standard one-way and two-way transducers, we introduce PSST, a new transducer model extract and replaceAll, where priorities are used to model the greedy and non-greedy semantics of capturing groups and string variables are used to model the back references. Moreover, as we have shown in Theorem 5.6, the pre-images of regular languages under PSSTs are still regular and can be computed effectively.

In the sequel, we shall show that semantically equivalent PSSTs can be effectively constructed from the extract and replaceAll functions.

Table 1. the actual $\mathcal{B}$ state in Figure 6

Symbol	State of $\mathcal{B}$
r_0	$(q_0, \rho_1, \emptyset, \emptyset)$
r_1	$(q_1, \rho_1, \{(q_0, q_1)\}, \emptyset)$
r_2	$(q_2, \rho_1, \{(q_0, q_1), (q_1, q_2)\}, \{q_0\})$
r_3	$(q_2, \rho_2, \emptyset, \{q_0, q_1, q_2\})$
r_4	$(q_2, \rho_1, \emptyset, \{q_0, q_1, q_2\})$
r_5	$(q_0, \rho_1, \{(q_0, q_1)(q_1, q_0)\}, \emptyset)$
r_6	$(q_0, \rho_2, \emptyset, \emptyset)$
r_7	$(q_0, \rho_2, \emptyset, \{q_0, q_1, q_2\})$
r_8	$(q_1, \rho_2, \{(q_0, q_1)\}, \{q_0, q_1, q_2\})$
r_9	$(q_0, \rho_2, \{(q_0, q_1)(q_1, q_0)\}, \{q_0, q_1, q_2\})$
r_{10}	$(q_2, \rho_2, \{(q_0, q_1)(q_1, q_2)\}, \{q_0, q_1, q_2\})$
r_{11}	$(q_0, \rho_1, \emptyset, \{q_0, q_1, q_2\})$
r_{12}	$(q_1, \rho_1, \{(q_0, q_1)\}, \{q_0, q_1, q_2\})$
r_{13}	$(q_0, \rho_1, \{(q_0, q_1)(q_1, q_0)\}, \{q_0, q_1, q_2\})$
r_{14}	$(q_2, \rho_1, \{(q_0, q_1)(q_1, q_2)\}, \{q_0, q_1, q_2\})$
r_{15}	$(q_2, \rho_2, \{(q_0, q_1)(q_1, q_2)\}, \{q_0\})$
r_{16}	$(q_1, \rho_2, \emptyset, \{q_0, q_1, q_2\})$
r_{17}	$(q_2, \rho_2, \{(q_1, q_2)\}, \{q_0, q_1, q_2\})$
r_{18}	$(q_0, \rho_2, \{(q_1, q_0)\}, \{q_0, q_1, q_2\})$
r_{19}	$(q_1, \rho_2, \{(q_1, q_0)(q_0, q_1)\}, \{q_0, q_1, q_2\})$
r_{20}	$(q_2, \rho_2, \{(q_1, q_0)(q_0, q_1)(q_1, q_2)\}, \{q_0, q_1, q_2\})$
r_{21}	$(q_1, \rho_2, \{(q_0, q_1)\}, \emptyset)$
r_{22}	$(q_0, \rho_2, \{(q_0, q_1)(q_1, q_0)\}, \emptyset)$

Lemma 6.2. *From* $\mathsf{extract}_{i,e}(x)$, *a PSST* $\mathcal{T}_{\mathsf{extract}_{i,e}}$ *can be constructed such that* $\mathcal{R}_{\mathcal{T}_{\mathsf{extract}_{i,e}}} = \{(w, w') \mid w' = \mathsf{extract}_{i,e}(w)\}$.

Lemma 6.3. *From* $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$, *a PSST* $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ *can be constructed such that* $\mathcal{R}_{\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}} = \{(w, w') \mid w' = \mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(w)\}$.

With Lemma 6.2-6.3, the path feasibility of $\mathsf{SL}_{\mathsf{reg}}$ is reduced to the path feasibility of string-manipulating programs that are a sequential composition of the statements of the form $z := x \cdot y$, $y := \mathcal{T}(x)$ and assert $(x \in \mathcal{A})$, where $\mathcal{T}$ is a PSST and $\mathcal{A}$ is an FA. Let us use $\mathsf{SL}'_{\mathsf{reg}}$ to denote the class of such programs. Then we can follow the backward reasoning approach of the OSTRICH solver proposed in [8, 9] and solve the path feasibility of $\mathsf{SL}'_{\mathsf{reg}}$ by repeating the following procedure, until no more assignment statements are left:
Let S the current $\mathsf{SL}'_{\mathsf{reg}}$ program.

- If the last assignment statement of S is $y := \mathcal{T}(x)$, then let assert $(y \in \mathcal{A}_1), \cdots,$ assert $(y \in \mathcal{A}_n)$ be an enumeration of all the assertion statements for y in S. Compute $\mathcal{R}_{\mathcal{T}}^{-1}(\mathcal{L}(\mathcal{A}_1))$ as an FA $\mathcal{B}_1, \cdots,$ and $\mathcal{R}_{\mathcal{T}}^{-1}(\mathcal{L}(\mathcal{A}_n))$ as $\mathcal{B}_n$. Remove the assignment $y := \mathcal{T}(x)$ and add the assertion statements assert $(x \in \mathcal{B}_1); \cdots;$ assert $(x \in \mathcal{B}_n)$.
- If the last assignment statement of S is $z := x \cdot y$, then let assert $(z \in \mathcal{A}_1), \cdots,$ assert $(z \in \mathcal{A}_n)$ be an enumeration of all the assertion statements for z in S. Compute $\cdot^{-1}(\mathcal{L}(\mathcal{A}_1))$, the pre-image of $\cdot$ under $\mathcal{L}(\mathcal{A}_1)$, as a collection of FA pairs $(\mathcal{B}_{1,j}, \mathcal{C}_{1,j})_{j \in [m_1]}, \cdots,$ and $\cdot^{-1}(\mathcal{L}(\mathcal{A}_n))$ as $(\mathcal{B}_{n,j}, \mathcal{C}_{n,j})_{j \in [m_n]}$ (c.f. [9]). Remove the assignment $z := x \cdot y$, nondeterministically choose the indices $j_1 \in [m_1], \cdots, j_n \in [m_n]$, and add the assertion statements assert $(x \in \mathcal{B}_{1,j_1})$; assert $(y \in \mathcal{C}_{1,j_1}); \cdots;$ assert $(x \in \mathcal{B}_{n,j_n})$; assert $(y \in \mathcal{C}_{n,j_n})$.

It remains to show Lemma 6.2 and Lemma 6.3. As a warm-up, we first present a cornerstone of the proof of the two lemmas, namely, the construction of PFAs from ERegExp[CG] expressions, which is adapted from the pNFA construction in [3], which in turn is a variant of the standard Thompson construction [17]. Afterwards, we prove the two lemmas.

6.1 From ERegExp[CG] to PFA

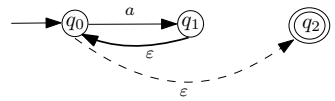
For any $e \in \mathsf{ERegExp}[\mathsf{CG}]$, a PFA $\mathcal{A}_e$ is constructed recursively in the sequel. The constructed PFA $\mathcal{A}_e$ satisfies that it has a unique initial state and a unique final state without outgoing transitions.

- If $e = \emptyset$, then $\mathcal{A}_e = (\{q_0, f_0\}, \Sigma, \delta, \tau, q_0, f_0)$, where $\delta(q_0, \sigma) = \delta(f_0, \sigma) = ()$ for every $\sigma \in \Sigma$, $\tau(q_0) = \tau(f_0) = ((); ())$.
- If $e = \varepsilon$, then $\mathcal{A}_e = (\{q_0, f_0\}, \Sigma, \delta, \tau, q_0, f_0)$, where $\delta(q_0, \sigma) = \delta(f_0, \sigma) = ()$ for every $\sigma \in \Sigma$, $\tau(q_0) = ((f_0); ())$, and $\tau(f_0) = ((); ())$.
- If $e = a$, then $\mathcal{A}_e = (\{q_0, f_0\}, \Sigma, \delta, \tau, q_0, f_0)$, where $\delta(q_0, a) = (f_0)$, $\delta(q_0, \sigma) = ()$ for every $\sigma \in \Sigma \setminus \{a\}$, $\tau(q_0) = ((); ())$, and $\tau(f_0) = ((); ())$.
- If $e = (e_1)$, then $\mathcal{A}_e = \mathcal{A}_{e_1}$.
- If $e = e_1 + e_2$, and suppose $\mathcal{A}_{e_1} = (Q_1, \Sigma, \delta_1, \tau_1, q_1, f_1)$ and $\mathcal{A}_{e_2} = (Q_2, \Sigma, \delta_2, \tau_2, q_2, f_2)$, then $\mathcal{A}_e = (Q_1 \cup Q_2 \cup \{q_0, f_0\}, \Sigma, \delta, \tau, q_0, f_0)$, where
  - $q_0, f_0 \notin Q_1 \cup Q_2$,
  - $\delta(q) = \delta_i(q)$ for every $q \in Q_i$ $(i = 1, 2)$, $\delta(q_0, \sigma) = \delta(f_0, \sigma) = ()$ for every $\sigma \in \Sigma$,
  - $\tau(q) = \tau_i(q)$ for every $q \in Q_i$ $(i = 1, 2)$, $\tau(q_0) = ((q_1, q_2); ())$, $\tau(f_1) = \tau(f_2) = ((f_0); ())$, and $\tau(f_0) = ((); ())$.
- If $e = e_1 \cdot e_2$, and suppose $\mathcal{A}_{e_1} = (Q_1, \Sigma, \delta_1, \tau_1, q_1, f_1)$ and $\mathcal{A}_{e_2} = (Q_2, \Sigma, \delta_2, \tau_2, q_2, f_2)$, then $\mathcal{A}_e = (Q_1 \cup Q_2, \Sigma, \delta, \tau, q_1, f_2)$, where
  - for every $q \in Q_i$, $\delta(q) = \delta_i(q)$ $(i = 1, 2)$,
  - for every $q \in Q_2$, $\tau(q) = \tau_2(q)$,
  - for every $q \in Q_1 \setminus \{f_1\}$, $\tau(q) = \tau_1(q)$, and $\tau(f_1) = ((q_2); ())$.
- If $e = e_1^*$, and suppose $\mathcal{A}_{e_1} = (Q_1, \Sigma, \delta_1, \tau_1, q_1, f_1)$, then $\mathcal{A}_e = (Q_1 \cup \{f_0\}, \Sigma, \delta, \tau, q_1, f_0)$, where
  - $f_0 \notin Q_1$,
  - for every $q \in Q_1$ and $\sigma \in \Sigma$, $\delta(q, \sigma) = \delta_1(q, \sigma)$,

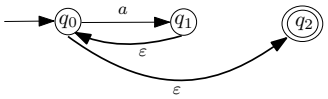
– for every $q \in Q_1 \setminus \{q_1, f_1\}$, $\tau(q) = \tau_1(q)$, moreover, $\tau(q_1) = (\pi_1(\tau_1(q_1)); (\pi_2(\tau_1(q_1)), f_0))$, $\tau(f_1) = ((q_1); ())$, and $\tau(f_0) = ((); ())$. (Intuitively, the ε-transitions from f_1 to q_1 and from q_1 to f_0 respectively are added, moreover, the ε-transition from q_1 to f_0 is of the lowest priority.)

- If $e = e_1^{*?}$, and suppose $\mathcal{A}_{e_1} = (Q_1, \Sigma, \delta_1, \tau_1, q_1, f_1)$, then $\mathcal{A}_e = (Q_1 \cup \{f_0\}, \Sigma, \delta, \tau, q_1, f_0)$, where
  – $q_0, f_0 \notin Q_1$,
  – for every $q \in Q_1$ and $\sigma \in \Sigma$, $\delta(q, \sigma) = \delta_1(q, \sigma)$,
  – for every $q \in Q_1 \setminus \{q_1, f_1\}$, $\tau(q) = \tau_1(q)$, moreover, $\tau(q_1) = ((f_0, \pi_1(\tau_1(q_1))); \pi_2(\tau_1(q_1)))$, $\tau(f_1) = ((q_1); ())$, and $\tau(f_0) = ((); ())$. (The ε-transition from q_1 to f_0 is of the highest priority.)

Example 6.4. Let $e_1 = a^*$ and $e_2 = a^{*?}$. Then the PFAs $\mathcal{A}_{e_1} = (Q_1, \Sigma, \delta_1, \tau_1, q_0, q_3)$ and $\mathcal{A}_{e_2} = (Q_2, \Sigma, \delta_2, \tau_2, q_0, q_3)$ are illustrated in Figure 7: (i), (ii), where thicker solid lines (resp. dashed lines) denote the ε-transitions of higher (resp. lower) priorities than non-ε-transitions. For instance, in $\mathcal{A}_{e_1}$, $\tau_1(q_0) = ((); (q_2))$ and $\tau_1(q_1) = ((q_0); ())$, while in $\mathcal{A}_{e_2}$, $\tau_2(q_0) = ((q_2); ())$ and $\tau_2(q_1) = ((q_0); ())$. Note that $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ are slightly different from those in Example 5.2.



(i) $\mathcal{A}_{e_1}$



(ii) $\mathcal{A}_{e_2}$

Figure 7. $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ for $e_1 = a^*$ and $e_2 = a^{*?}$

From the aforementioned recursive construction of $\mathcal{A}_e$, we know that for each subexpression e' of e, a PFA for e', which is isomorphic to $\mathcal{A}_{e'}$, is also constructed. Let us use $\mathsf{Sub}_{e'}[\mathcal{A}_e]$ to denote this PFA for e', which, roughly speaking, is a subgraph of $\mathcal{A}_e$.

6.2 Proof of Lemma 6.2-6.3

We first prove Lemma 6.2, namely, show how to construct a PPST $\mathcal{T}_{\mathsf{extract}_{i,e}}$ for $\mathsf{extract}_{i,e}$.

Construction of $\mathcal{T}_{\mathsf{extract}_{i,e}}$. Let e' be the subexpression corresponding to the i-th capturing group of e. In particular, if $i = 0$, then $e' = e$.

Suppose $\mathcal{A}_e = (Q, \Sigma, \delta, \tau, q_0, f_0)$. Then $\mathcal{T}_{\mathsf{extract}_{i,e}}$ is obtained from $\mathcal{A}_e$ by adding two fresh states q_0', f_0' such that (see Figure 8)

- $\mathcal{T}_{\mathsf{extract}_{i,e}}$ goes from q_0' to q_0 via an ε-transition of higher priority than the non-ε-transitions, in order to search for the first match of e,
- then $\mathcal{T}_{\mathsf{extract}_{i,e}}$ simulates $\mathcal{A}_e$ and stores the match of the i-th capturing group of e into a string variable x,
- when the first match of e is found, $\mathcal{T}_{\mathsf{extract}_{i,e}}$ goes from f_0 to f_0' via an ε-transition, then keeps idle, and finally output the content of x.

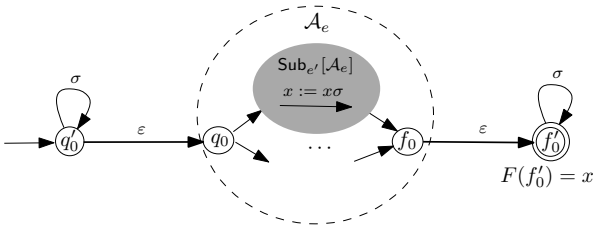


Figure 8. The PSST for $\mathsf{extract}_{i,e}$

Formally, $\mathcal{T}_{\mathsf{extract}_{i,e}} = (Q \cup \{q_0', f_0'\}, \Sigma, X, \delta', \tau', E, q_0', F)$ where

- $q_0', f_0' \notin Q$,
- $X = \{x\}$,
- $F(f_0') = x$, and $F(q')$ is undefined for every $q' \in Q \cup \{q_0'\}$,
- δ' and τ' are obtained from δ and τ as follows,
  – $\delta'(q_0', \sigma) = (q_0')$ for every $\sigma \in \Sigma$, and $\tau'(q_0') = ((q_0); ())$,
  – for every $q \in Q \setminus \{f_0\}$ and $\sigma \in \Sigma$, $\delta'(q, \sigma) = \delta(q, \sigma)$ and $\tau'(q) = \tau(q)$,
  – $\delta'(f_0, \sigma) = ()$ for every $\sigma \in \Sigma$ and $\tau'(f_0) = ((f_0'); ())$,
  – $\delta'(f_0', \sigma) = (f_0')$ for every $\sigma \in \Sigma$ and $\tau'(f_0') = ((); ())$,
- E is defined as follows,
  – for every transition (q, σ, q') with $\sigma \in \Sigma^\varepsilon$ in $\mathsf{Sub}_{e'}[\mathcal{A}_e]$, we have $E(q, \sigma, q')(x) = x\sigma$.
  – for all the other transitions (q, σ, q') with $\sigma \in \Sigma^\varepsilon$ in $\mathcal{T}_{\mathsf{extract}_{i,e}}$, we have $E(q, \sigma, q')(x) = x$

TL: tbc :LT

Construction of $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat,rep}}}$. Let $\$i_1, \cdots, \$i_k$ with $i_1 < \cdots < i_k$ be an enumeration of all the references in rep. Moreover, for every $j \in [k]$, let e'_{i_j} be the subexpression of pat corresponding to the i_j-th capturing group.

Suppose $\mathcal{A}_{\mathsf{pat}} = (Q, \Sigma, \delta, \tau, q_0, f_0)$. Then $\mathcal{T}_{\mathsf{replace}_{\mathsf{pat,rep}}}$ is obtained from $\mathcal{A}_{\mathsf{pat}}$ by adding a fresh states q_0' such that (see Figure 9)

- $\mathcal{T}_{\mathsf{replace}_{\mathsf{pat,rep}}}$ goes from q_0' to q_0 via an ε-transition of higher priority than the non-ε-transitions, in order to search the first match of pat starting from the current position,
- when $\mathcal{T}_{\mathsf{replace}_{\mathsf{pat,rep}}}$ stays at q_0', it keeps appending the current letter to the end of x_0,
- starting from q_0, $\mathcal{T}_{\mathsf{replace}_{\mathsf{pat,rep}}}$ simulates $\mathcal{A}_{\mathsf{pat}}$ and stores the matches of the $\$i_1$-th, …, $\$i_k$-th capturing groups of pat into the string variables $x_1, \cdots, x_k$ respectively,

- when the first match of pat is found, $\mathcal{T}_{\mathsf{replace}_{\mathsf{pat,rep}}}$ goes from f_0 to q_0' via an ε-transition, appends the current replacement string, which is $\mathsf{rep}[x_1/\$_{i_1}, \cdots, x_k/\$_{i_k}]$, to the end of x_0, and keeps searching for the next match of pat,

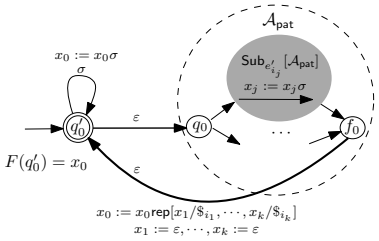


Figure 9. The PSST for $\mathsf{replace}_{\mathsf{pat,rep}}$

Formally, $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat,rep}}} = (Q \cup \{q_0'\}, \Sigma, X, \delta', \tau', E, q_0', F)$ where

- $q_0' \notin Q$,
- $X = \{x_0, x_1, \cdots, x_k\}$,
- $F(q_0') = x_0$, and $F(q')$ is undefined for every $q' \in Q$,
- δ' and τ' are obtained from δ and τ as follows,
  - $\delta'(q_0', \sigma) = (q_0')$ for every $\sigma \in \Sigma$, and $\tau'(q_0') = ((q_0); ())$,
  - for every $q \in Q \setminus \{f_0\}$ and $\sigma \in \Sigma$, $\delta'(q, \sigma) = \delta(q, \sigma)$ and $\tau'(q) = \tau(q)$,
  - $\delta'(f_0, \sigma) = ()$ for every $\sigma \in \Sigma$ and $\tau'(f_0) = ((q_0'); ())$,
- E is defined as follows,
  - for every transition (q, σ, q') with $\sigma \in \Sigma^\varepsilon$ in $\mathcal{A}_{\mathsf{pat}}$, $E(q, \sigma, q')(x_0) = x_0$,
  - for every transition (q, σ, q') with $\sigma \in \Sigma^\varepsilon$ and every $j \in [k]$, if (q, σ, q') occurs in $\mathsf{Sub}_{e'_{i_j}}[\mathcal{A}_{\mathsf{pat}}]$, then $E(q, \sigma, q')(x_j) = x_j\sigma$, otherwise, $E(q, \sigma, q')(x_j) = x_j$,
  - for every $\sigma \in \Sigma$, $E(q_0', \sigma, q_0')(x_0) = x_0\sigma$, and for every $j \in [k]$, $E(q_0', \sigma, q_0')(x_j) = x_j$,
  - $E(q_0', \varepsilon, q_0)(x_j) = x_j$ for every $j \in [k] \cup \{0\}$,
  - $E(f_0, \varepsilon, q_0')(x_0) = x_0\mathsf{rep}[x_1/\$i_1, \ldots, x_k/\$i_k]$, and for every $j \in [k]$, $E(f_0, \varepsilon, q_0')(x_j) = \varepsilon$,

ZL: stopped here :LZ ZLH: changed a little bit :HZL

6.3 Complexity

Proposition 6.5 (POPL'19). *The path feasibility problem of the following two fragments is non-elementary: SL with 2FTs, and SL with FTs+replaceAll.*

SL[conc, replaceAll, reverse, FFT] is expspace-complete (note that 2FTs in SL are restricted to be one-way and functional)

The main open question is the complexity of the SL fragment with replaceall function and prioritized streaming transducers. Note that PSST can simulate 2FT (adapting Matt's proof?), so we could obtain nonelementary lower bound for SL with PSST.

However, this variant of replaceall is quite different from the replaceall we had before ...

1. does copyless help?
2. how about SL with only this version of replaceall?

7 Implementation

8 Experiments

References

[1] Rajeev Alur and Pavol Cerný. 2010. Expressiveness of streaming string transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*. 1–12.

[2] Rajeev Alur and Jyotirmoy V. Deshmukh. 2011. Nondeterministic Streaming String Transducers. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II (Lecture Notes in Computer Science)*, Luca Aceto, Monika Henzinger, and Jirí Sgall (Eds.), Vol. 6756. Springer, 1–20.

[3] Martin Berglund, Frank Drewes, and Brink van der Merwe. 2014. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014 (EPTCS)*, Zoltán Ésik and Zoltán Fülöp (Eds.), Vol. 151. 109–123. https://doi.org/10.4204/EPTCS.151.7

[4] Martin Berglund and Brink van der Merwe. 2017. On the semantics of regular expression parsing in the wild. *Theoretical Computer Science* 679 (2017), 69 – 82. Implementation and Application of Automata.

[5] Cristian Cadar, Vijay , Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 322–335. https://doi.org/10.1145/1180405.1180445

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[7] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[8] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2018. What is decidable about string constraints with the ReplaceAll function. *PACMPL* 2, POPL (2018), 3:1–3:29.

[9] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision Procedures for Path Feasibility of String-Manipulating Programs with Complex Operations. *PACMPL* 3, POPL, Article Article 49 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290362

[10] Emmanuel Filiot and Pierre-Alain Reynier. 2017. Copyful Streaming String Transducers. In *Reachability Problems - 11th International Workshop, RP 2017, London, UK, September 7-9, 2017, Proceedings (Lecture Notes in Computer Science)*, Matthew Hague and Igor Potapov (Eds.), Vol. 10506. Springer, 75–86.

[11] Jeffrey E. F. Friedl and Andy Oram. 2002. *Mastering Regular Expressions* (2 ed.). O'Reilly & Associates, Inc., USA.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

[13] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[14] Anthony W. Lin and Pablo Barceló. 2016. String Solving with Word Equations and Transducers: Towards a Logic for Analysing Mutation XSS *(POPL '16)*. Springer, 123–136.

[15] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 488–498. https://doi.org/10.1145/2491411.2491447

[16] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, ESEC/SIGSOFT FSE 2005*. ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[17] Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

[18] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. Progressive Reasoning over Recursively-Defined Strings. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Springer, 218–240.

[19] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based Symbolic String Analysis for Vulnerability Detection. *Form. Methods Syst. Des.* 44, 1 (2014), 44–70.

A Appendix