Solving String Constraints with Real-World Regular Expressions

Abstract. Regular expressions (RE) are a classical concept in formal language theory. Real-world regular expressions (RWRE) in programming languages differ from REs in the non-standard semantics of operators (e.g. non-commutative union and greedy/lazy Kleene star), as well as additional features such as capturing groups and backreferences. While REs are supported by state-of-the-art string constraint solvers, RWREs are thus far unsupported. Recent work suggests that the mismatch between REs and RWREs makes it difficult for symbolic execution engines to handle RWREs, where approximation of RWREs is usually needed. In this paper, we propose an approach of natively supporting RWREs in string constraint solving. The key idea of our approach is to introduce a new automata model, called *prioritized streaming string transducers* (PSST), to model the string functions involving RWREs. PSSTs combine *priorities*, which have previously been introduced in prioritized finite-state automata to capture greedy/lazy semantics, with *string variables* as in streaming string transducers to model capturing groups. Based on PSSTs, we design a decision procedure for string constraints with RWREs and provide its implementation. We evaluate its performance on over 195 000 string constraints generated from RWREs in open-source programs. The experimental results show the efficacy of our approach, drastically improving the existing methods (via symbolic execution) in both precision and efficiency.

1 Introduction

Modern high-level programming languages such as JavaScript, Python, Java, and PHP natively support a variety of string operations, and use strings to store and process virtually all kinds of data or code. Applied string operations range from concatenation, splitting, and replacement, to complex functions such as regular expression matching and character encoding/decoding. As a result, string-manipulating programs are notoriously subtle, error-prone, and their potential bugs may bring severe security consequences. A typical example is cross-site scripting (XSS), which is among the OWASP Top 10 Application Security Risks. An effective and increasingly popular method for identifying such bugs in programs is symbolic execution, possibly in combination with dynamic analysis. This technique analyses a static path in a program, by viewing it as a constraint ϕ, whose feasibility can be checked by constraint solvers.

Regular expression matching is one of the most important string operations in programming languages [SAH+10,BvdM17a,LMK19,KGA+12]. Most state-of-the-art string constraint solvers (e.g. Z3, CVC4, Z3-str/2/3/4, ABC, Norn,

Trau, OSTRICH, S2S, Qzy, Stranger, Sloth, Slog, Slent, Gecode+S, G-Strings, HAMPI) therefore support *regular expression constraints*, e.g., matching a string with a regular expression, as we know it from formal language theory. Unfortunately, *Real-world Regular Expressions* (RWRE) in programming languages are dramatically different from *classical Regular Expressions* (RE) in formal language theory. Classical regular expressions are built from letters by the operators of concatenation, union, and Kleene star, and have nice compositional semantics. On the other hand, RWREs differ from classical ones mainly in the following two aspects: 1) non-standard semantics of operators, e.g., the non-commutative union, the greedy/lazy Kleene star, and 2) new features, e.g., capturing groups and backreferences. RWREs are in general more expressive than classical REs, e.g., it is known that with backreferences one can easily generate languages that are not even context-free (e.g. see [FS19,Aho90,BvdM17b]).

Example 1. Consider the RWRE `(\d+)(\d*)`. It has two capturing groups, each within a pair of opening/closing brackets and matching a string of digits (signified by `\d`). The second capturing group could be matched with an empty sequence of digits. Given a string of digits (e.g. `"2050"`), the entire string will always be matched by the first subexpression `(\d+)`, owing to the greedy semantics of Kleene plus.

Consider now the RWRE `(\d+)\1\1`. It contains two backreferences `\1`, each of which matches exactly on the contents of the first capturing group. It accepts precisely the set L of all the words www, where w is a nonempty sequence of digits, which is not a context-free language. □

To make matters worse, RWREs in real-world programs are also commonly used in combination with other string operations (e.g. match and replace(all) functions [LMK19]), which pose additional challenges to symbolic execution tools. On a given string s and a RWRE e, the match function allows one to extract the last match of a capturing group (e') with respect to the first match of e in s. For the replace function, on a given string s, a matching pattern RWRE e, and a replacement string t, it replaces the first match (or all matches, if the global flag is enabled) of e in s by t. Here t could contain references to the matches of various capturing groups in e.

Example 2. Consider the snippet

```
var namesReg = /([A-Za-z]+) ([A-Za-z]+)/g;
var newAuthorList = authorList.replace(nameReg, "$2, $1");
```

Assuming `authorList` is given as a list of `;`-separated author names — first name, followed by a last name — the above program would convert this to last name, followed by first name format. For instance, `"Don Knuth; Alan Turing"` would be converted to `"Knuth, Don; Turing, Alan"`. □

Since the state-of-the-art string solvers support only classical REs instead of RWREs, existing symbolic execution approaches that handle string-manipulating programs with RWREs apply workarounds. We mention Aratha [AAG+19] and

ExpoSE [LMK19], both of which are symbolic execution engines for JavaScript programs. Aratha performs a rough approximation to the non-standard semantics of regular expressions, e.g., a backreference is replaced by the regular expression Σ^* that accepts all words. On the other hand, ExpoSE attempts to exploit string equations and classical REs (as implemented in Z3 [dMB08]) supported by string solvers to capture the semantics of RWREs. Unfortunately, the semantics of RWREs cannot in general be fully captured by string constraints with REs. For this reason, ExpoSE attempts to approximate the semantics of RWREs in the style of CEGAR (counter-example guided abstraction and refinement). This results in a rather severe price in both precision and performance: the refinement process may not terminate and the symbolic execution of even a simple program with RWREs may need to be refined many times.

At a more theoretical level, there are no attempts to incorporate RWREs into a decidable string constraint language, e.g., word equations [Gut98]. Thus far, most decidability and complexity results regarding RWRE solely focus on standard decision problems (e.g. membership and emptiness being decidable and NP-complete [FS19,BvdM17b]). We conclude with two open questions:

(Q1) Incorporate RWREs into match and replace functions as primitives in a string constraint language and develop a fast string solver for them.

(Q2) Develop a reasonably expressive decidable string constraint language that supports the replace and match function with RWREs, as well as string concatenation.

Contributions. The main contributions of the paper are to answer both (Q1) and (Q2) in the positive for a reasonable fragment of RWREs. In particular, we consider the problem of path feasibility of a simple symbolic path constraint language that uses only string variables:

$$\begin{aligned} S \stackrel{\text{def}}{=} z := x \cdot y \mid y := \mathsf{extract}_{i,e}(x) \mid y := \mathsf{replace}_{\mathsf{pat},\mathsf{rep}}(x) \mid \\ y := \mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x) \mid \mathsf{assert}\,(x \in e) \mid S; S \end{aligned}$$

That is, assignments are allowed whose right hand side could use concatenation, the match function (extract), and the replace function (with/without the global flag). RWREs ($e, \mathsf{pat}, \mathsf{rep}$ in the syntax above) are allowed in the assertions, as well as in the match and the replace functions. A given path is feasible if there is an initialization of the string variables under which the above path can run from start to finish without violating any of the assertions. Our main result is the decidability of this problem for a reasonable class of RWREs. In particular, rep is a concatenation of string constants and backreferences, while e, pat are RWREs that allow non-commutative unions, greedy/lazy Kleene stars, and capturing groups, but *not* backreferences. An example of a symbolic path in this fragment is in Example 2. We complement this by proving undecidability when we permit backreferences in e or pat. Our decidable fragment of RWREs supports a significant portion of the frequently used features in RWREs (as indicated by the data analysis in [LMK19] across 415,487 NPM packages) including capturing groups

(∼39%), global flag (∼30%), and greedy/lazy Kleene stars (∼23%). Features such as backreferences turned out to be not so frequently used (∼0.8%). These statistics are also consistent with the RWRE usage statistics for Python across 3,898 projects [CS16], e.g., capturing groups are the most frequently used features of RWREs (∼53% out of the found RWREs), while backreferences are not frequently used (∼0.1%). Moreover, in a recent library of over 500,000 RWREs collected from open-source programs [DMIC+19], backreferences occur in less than 0.2% of them, and our decidable fragment is able to cover ∼80% of them.

Our decision procedure requires that we introduce a new automata model, called prioritized streaming string transducers (PSSTs), which extends and combines prioritized finite-state automata [BvdM17a] and streaming string transducers [AC10,AD11]. With PSSTs, we encode the non-standard semantics of regular expression operators by priorities and deal with capturing groups by string variables. The widely used string functions involving regular expressions, e.g. match and replace(all), can be easily transformed into PSSTs.

We then design a decision procedure for a class of string constraints with RWREs. The decision procedure extends the backward reasoning approach proposed in [CHL+19] to PSSTs. Specifically, we show that the pre-images of regular languages under PSSTs are regular and can be computed effectively.

We implement the decision procedure in our new solver EMU on top of the existing open-source solver OSTRICH [CHL+19], and carry out extensive experiments to evaluate the performance. For the benchmarks, we generate two collections of JavaScript programs (with 98,117 programs in each collection), from a library of real-world regular expressions [DMIC+19], by using two simple JavaScript program templates containing match and replace functions respectively. Then we generate all the four (resp. three) path constraints for each match (resp. replace) JavaScript program and put them into one SMT file. We run EMU on these SMT files. EMU is able to answer all four (resp. three) queries in 97.9% (resp. 97.6%) of the match (resp. replace) SMT files, with the average time 1.19 (resp. 1.48) seconds per file. For comparison, we also run ExpoSE on the JavaScript programs. ExpoSE covers 91.5% (resp. 63.2%) of feasible paths in the match (resp. replace) programs reported by EMU, with the average time 28.0 (resp. 55.0) seconds per program. The huge difference of the running time as well as the path coverage shows that our approach can reason about RWREs in a much more efficient and precise way than the CEGAR-based approach.

Organization. This paper is organized as follows: Section 2 proposes the motivating example. Section 3 defines RWREs. Section 4 introduces the string constraint language. Section 5 is devoted to PSSTs. Section 6 presents the decision procedure. Section 7 describes the implementation and experiments. Section 8 discusses the related work and concludes this work.

2 Motivating Example

We use the JavaScript program in Fig. 1 as a running example to illustrate our approach. The function `normalize` removes leading and trailing zeros from a

```
function normalize(decimal) {
  const decimalReg = /^(\d+)\.?(\d*)$/;
  var   decomp     = decimal.match(decimalReg);
  var   result     = "";
  if (decomp) {
    var integer    = decomp[1].replace(/^0+/, "");
    var fractional = decomp[2].replace(/0+$/, "");
    if (integer    !== "") result = integer; else result = "0";
    if (fractional !== "") result = result + "." + fractional;
  }
  return result;
}
```

Fig. 1. Normalize a decimal by removing the leading and trailing zeros

decimal string with the input `decimal`. For instance, `normalize("0.250") == "0.25"`, `normalize("02.50") == "2.5"`, `normalize("025.0") == "25"`, and finally `normalize("0250") == "250"`.

In the function body, the input `decimal` is matched to a regular expression `decimalReg` = `/^(\d+)\.?(\d*)$/`, which requires that the input comprises a digit sequence representing the integer part of the input, possibly followed by a dot symbol (the decimal point) as well as another digit sequence representing the fractional part. The anchors `^` and `$` denote the beginning and the end of the input, respectively. Note that `decimalReg` utilizes two capturing groups, namely, `(\d+)` and `(\d*)`, to record the integer and fractional part of the decimal. The expression `\.?` specifies that the dot symbol is optional, namely, it may not occur in the input. Moreover, the regular expression utilizes the *greedy* semantics of the quantifier `+` to enforce that `\d+` is matched by the whole string if the input does not contain any dots. For instance, if `decimal` = `"0250"`, then `(\d+)` is matched by `"0250"` and `(\d*)` is matched by the empty string. Note that the greedy semantics is crucial here, because with standard *non-deterministic* semantics the `(\d+)` could also (incorrectly) by matched by `"02"`, and `(\d*)` by `"50"`.

The result of the matching, which is an array of strings, is stored in the variable `decomp`. Then the leading zeros are trimmed by applying `replace(/^0+/, "")` to `decomp[1]` and the result is stored in the variable `integer`. Similarly, the trailing zeros are trimmed by `replace(/0+$/, "")` to `decomp[2]` and the result is stored in `fractional`. The greedy semantics of `0+` is used to trim *all* the leading/trailing zeros. If `integer` is empty, the return value gets a default value `"0"`. If `fractional` is empty, then the return value is `integer`. Otherwise, the return value joins `integer` and `fractional` with the dot symbol.

A natural post-condition of `normalize` is that the result contains neither leading nor trailing zeros. This post-condition has to be established by the function on *every* execution path. As an example, consider the path shown in Fig. 2, in which the branches taken in the program are represented as `assume` statements. The negated post-condition is captured by the regular expression in the

```
const decimalReg = /^(\d+)\.?(\d*)$/;
var decomp = decimal.match(decimalReg);
var result = "";
assume (decomp!==null);
var integer = decomp[1].replace(/^0+/, "");
var fractional  = decomp[2].replace(/0+$/, "");
assume (integer !== "");
result1 = integer;
assume (fractional !== "");
result2 = result1 + "." + fractional;
assume (/^0\d+.*|.*\.\d*0$/.test(result2));
```

Fig. 2. Symbolic execution of a path of the JavaScript program in Fig. 1

last `assume`. For this path, the post-condition can be proved by showing that the program in Fig. 2 is infeasible: there does not exist an initial value `decimal` so that no assumption fails and the program executes to the end.

To enable symbolic execution of the JavaScript programs like in Fig. 1, we need to model the greedy semantics of `+` and the matching of capturing groups. To this end, we propose the use of *prioritized streaming string transducers* (PSST, Section 5). The extraction of `decomp[1]` from `decimal`, namely `decimal.match(decimalReg)[1]`, can be modeled by a PSST $\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},1}}$, where the priorities are used to capture the greedy semantics of $+$ (see Definition 3 in Section 3) and the string variables are used to record the matches of capturing groups. The extraction of `decomp[2]` can be handled in a similar way. Moreover, the functions `replace(/^0+/, "")` and `replace(/0+$/, "")` can also be modeled by PSSTs $\mathcal{T}_{\texttt{replace(/\^0+/, "")}}$ and $\mathcal{T}_{\texttt{replace(/0+\$/, "")}}$.

After some further simplifications, we arrive at the following program in our string-manipulating language, capturing Fig. 2. In the program, $\mathcal{A}_{/.+/}$, $\mathcal{A}_{\texttt{decimalReg}}$ and $\mathcal{A}_{/\texttt{\^0\textbackslash d+.*|.*\textbackslash .\textbackslash d*0\$/}}$ denote finite-state automata corresponding to the regular expressions, and the assumptions are encoded using assert (using the same terminology as [CHL+19]):

$$
\begin{aligned}
&\mathsf{assert}\left(\texttt{decimal} \in \mathcal{A}_{\texttt{decimalReg}}\right);\\
&\mathsf{integer} := \mathcal{T}_{\texttt{replace(/\^0+/, "")}}(\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},1}}(\texttt{decimal}));\\
&\mathsf{fractional} := \mathcal{T}_{\texttt{replace(/0+\$/, "")}}(\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},2}}(\texttt{decimal}));\\
&\mathsf{assert}\left(\texttt{integer} \in \mathcal{A}_{/.+/}\right);\mathsf{assert}\left(\texttt{fractional} \in \mathcal{A}_{/.+/}\right);\\
&\texttt{result2} := \texttt{integer} \cdot \text{“.”} \cdot \texttt{fractional};\\
&\mathsf{assert}\left(\texttt{result2} \in \mathcal{A}_{/\texttt{\^0\textbackslash d+.*|.*\textbackslash .\textbackslash d*0\$/}}\right);
\end{aligned}
\tag{1}
$$

At first, we show that the pre-images of regular languages under PSSTs are regular (see Lemma 2). Then, path feasibility of the program in (1) can be checked by following the “backward” reasoning approach from [CHL+19]. First, we compute the pre-images of regular languages under the concatenation operation and remove the last assignment statement. Then, we compute the pre-images

of regular languages under the PSSTs $\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},2}}$ as well as $\mathcal{T}_{\texttt{replace(/0+\$/, "")}}$, and remove the second assignment statement. Similarly for the first assignment statement. In the end, a program containing only regular membership queries (but possibly including disjunctions) is obtained, whose feasibility is reduced to checking the nonemptiness of the intersection of regular languages, which is known to be decidable (PSPACE-complete). (See Appendix A.1 for more details.)

3 Real-World Regular Expressions

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 $w = w_1 w_2$, then we use $w_1^{-1} w$ to denote w_2.

Definition 1 (Real-world regular expressions, RWRE).

$$e \stackrel{def}{=} \emptyset \mid \varepsilon \mid a \mid [e^?] \mid [e^{??}] \mid \$n \mid [e + e] \mid [e \cdot e] \mid [e^*] \mid [e^+] \mid [e^{*?}] \mid [e^{+?}] \mid [e^{\{m_1,m_2\}}] \mid [e^{\{m_1,m_2\}?}] \mid (e)$$

where $a \in \Sigma$, $n \in \mathbb{Z}^+$, $m_1, m_2 \in \mathbb{N}$ *with* $m_1 \leq m_2$.

For $\Gamma = \{a_1, \ldots, a_k\} \subseteq \Sigma$, we write Γ for $[[\cdots[a_1 + a_2] + \cdots] + a_k]$ and thus $[\Gamma^*] \equiv [[[\cdots[a_1 + a_2] + \cdots] + a_k]^*]$. Similarly for $[\Gamma^{*?}]$, $[\Gamma^+]$, and $[\Gamma^{+?}]$. We write $|e|$ for the length of an RWRE e, i.e. the number of symbols occurring in e.

Note that square brackets [] are used for the operator precedence and the parentheses () are used for capturing groups. 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, $[[([[a + b]^*]) \cdot c] \cdot \$1]$ is in RWRE, 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.

The operator $[e^*]$ is the *greedy* Kleene star, meaning e should be matched as many times as possible. The operator $[e^{*?}]$ is the *lazy* Kleene star, meaning e should be matched as few times as possible. The Kleene plus operators $[e^+]$ and $[e^{+?}]$ are similar to $[e^*]$ and $[e^{*?}]$ but e should be matched at least once. Likewise, the optional operator has greedy and lazy variants $[e^?]$ and $[e^{??}]$, respectively.

We use $\mathsf{RWRE}_{\mathsf{reg}}$ to denote the fragment of RWRE excluding backreferences $\$n$ (where reg represents regular languages), and $\mathsf{RWRE}_{\mathsf{ref}}$ to denote the set of regular expressions generated by a concatenation of letters and backreferences, formally defined by $e \stackrel{\text{def}}{=} \varepsilon \mid a \mid \$n \mid [e \cdot e]$.

We shall define the formal semantics of RWRE, which uses the concepts of indexed RWREs, subexpressions and matches of RWREs to strings defined below.

For a RWRE e, we use $\mathsf{idx}(e)$ to denote the *indexed* RWRE obtained from e, namely, the expression obtained by adding indices to the parenthesis pairs, with the n-th parenthesis pair $(\cdots)$ changed to $({}_n\cdots)_n$. For instance, let $e = [([0^+]) \cdot ([0^*])]$, then $\mathsf{idx}(e) = [({}_1[0^+])_1 \cdot ({}_2[0^*])_2]$.

For two indexed RWREs e and e', we say e' is a *subexpression* of e, if one of the following conditions holds: 1) $e' = e$, 2) $e = [e_1 \cdot e_2]$ or $[e_1 + e_2]$, and e' is a subexpression of e_1 or e_2, 3) $e = [e^?], [e^{??}], [e_1^*]$, $[e_1^+]$, $[e_1^{*?}]$, $[e_1^{+?}]$, $e_1^{\{m_1,m_2\}}$, $e_1^{\{m_1,m_2\}?}$ or $({}_n e_1)_n$, and e' is a subexpression of e_1. We use $S(e)$ to denote the set of all subexpressions of e.

Definition 2 (Matches of RWRE to strings). *A **match** of a RWRE e to a string w is defined by a finite directed and ordered tree T, whose nodes are elements of $\Sigma^* \times S(\mathsf{idx}(e))$. The root of T is $(w, \mathsf{idx}(e))$, 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 $w' = w'_1 w'_2$.*
- *If $e' = [e'_1 + e'_2]$, then α has a single child $\alpha_1 = (w', e'_i)$ for some $i \in \{1, 2\}$.*
- *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 $\alpha_1 = (w'_1, e'_1), \ldots, \alpha_k = (w'_k, e'_1)$, $w' = w'_1 \cdots w'_k$, and $w'_i \neq \varepsilon$ for every $i \in [k]$.*
- *If $e' = [{e'_1}^+]$ or $[{e'_1}^{+?}]$, then there is $k \geqslant 1$ such that α has k children $\alpha_1 = (w'_1, e'_1), \ldots, \alpha_k = (w'_k, e'_1)$, $w' = w'_1 \cdots w'_k$, and $w'_i \neq \varepsilon$ for every $i \in \{2, \cdots, k\}$.*
- *If $e' = ({}_n e'_1)_n$, 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 $e' = [e^?]$ or $[e^{??}]$, then either α is a leaf and $w' = \varepsilon$, or α has a single child $\alpha_1 = (w', e)$ and $w' \neq \varepsilon$.*
- *If $e' = e^{\{m_1,m_2\}}$ or $e^{\{m_1,m_2\}?}$, then there is $m_1 \leqslant k \leqslant m_2$ such that α has k children $\alpha_1 = (w'_1, e'_1), \ldots, \alpha_k = (w'_k, e'_1)$, $w' = w'_1 \cdots w'_k$, and $w'_i \neq \varepsilon$ for every $i \in \{m_1 + 1, \cdots, k\}$.*
- *If $e' = \$n$, then α is a leaf of T. Moreover, let $({}_n e'')_n \in S(\mathsf{idx}(e))$ and $\beta = (w_1, e_1)$ be the last node preceding α in T such that $e_1 = ({}_n e'')_n$ (there may be multiple nodes preceding α satisfying this condition), 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 $\{w \in \Sigma^* \mid \mathscr{M}_w(e) \neq \emptyset\}$.

By a mutual induction on $|w|$ and $|e|$, we can show that $|\mathscr{M}_w(e)|$, the size of $\mathscr{M}_w(e)$, is at most $|w||e|$.

Example 3. Let $w = 0250$ and $e = [[([\Gamma^+]) \cdot .?] \cdot ([\Gamma^*])]$ where $\Gamma = \{0, 1, \cdots, 9\}$. Note that e is essentially `decimalReg` in the motivating example. Then $\mathscr{M}_w(e) = \{T_1, T_2, T_3, T_4\}$ as illustrated in Figure 3(i), (ii), (iii), and (iv), where the match trees rooted at $(0, \Gamma)$, $(2, \Gamma)$, and $(5, \Gamma)$ are omitted.

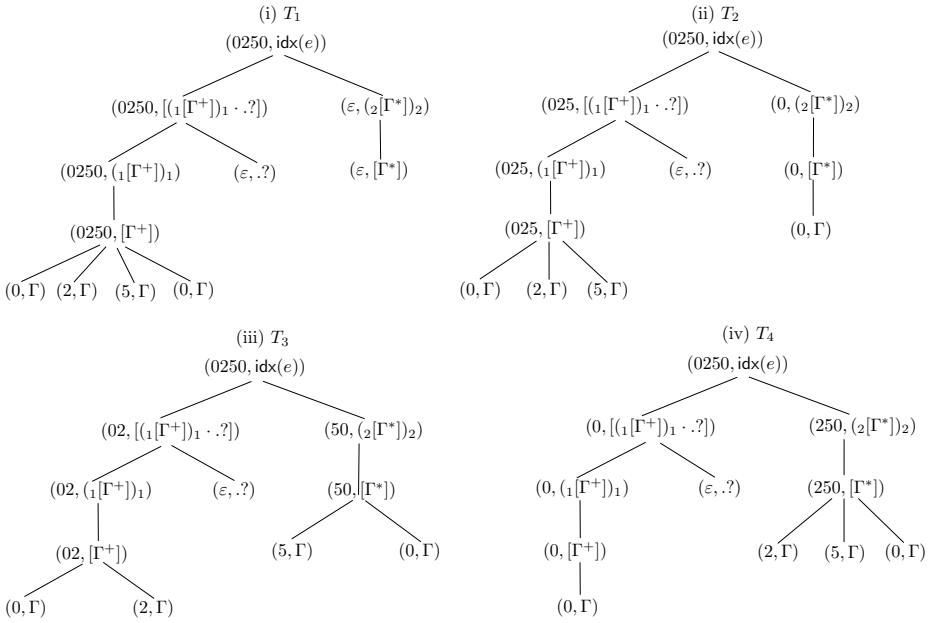

Fig. 3. Match trees of $e = [[([\Gamma^+]) \cdot .?] \cdot ([\Gamma^*])]$ to $w = 0250$

Definition 3 (Semantics of RWRE). *For a RWRE e and a string w, we recursively define a (strict) total order on $\mathscr{M}_{\mathsf{Pref}(w)}(e)$, written $T >_{w,\mathsf{idx}(e)} T'$ for $T, T' \in \mathscr{M}_{\mathsf{Pref}(w)}(e)$, as follows:*

- $\mathsf{idx}(e) = \varepsilon$, $\mathsf{idx}(e) = a$, *or* $\mathsf{idx}(e) = \$n$. *There is only one match tree, thus the order $>_{w,\mathsf{idx}(e)}$ is empty.*
- $\mathsf{idx}(e) = [e_1^?]$.
  - *If T' is a single node $(\varepsilon, \mathsf{idx}(e))$, but T is not, then $T >_{w,\mathsf{idx}(e)} T'$.*
  - *If $C(T) = (w_1, e_1)$ and $C(T') = (w_2, e_1)$ for some $w_1, w_2 \in \mathsf{Pref}(w)$, then $T >_{w,\mathsf{idx}(e)} T'$ iff $T_{(w_1,e_1)} >_{w,e_1} T'_{(w_2,e_1)}$.*
- $\mathsf{idx}(e) = [e_1^{??}]$.
  - *If T is a single node $(\varepsilon, \mathsf{idx}(e))$, but T' is not, then $T >_{w,\mathsf{idx}(e)} T'$.*
  - *If $C(T) = (w_1, e_1)$ and $C(T') = (w_2, e_1)$ for some $w_1, w_2 \in \mathsf{Pref}(w)$, then $T >_{w,\mathsf{idx}(e)} T'$ iff $T_{(w_1,e_1)} >_{w,e_1} T'_{(w_2,e_1)}$.*
- $\mathsf{idx}(e) = ({}_n e_1)_n$. *Suppose that $C(T) = (w_1, e_1)$ and $C(T') = (w_2, e_1)$ for some $w_1, w_2 \in \mathsf{Pref}(w)$. Then $T >_{w,\mathsf{idx}(e)} T'$ iff $T_{(w_1,e_1)} >_{w,e_1} T'_{(w_2,e_1)}$.*

– $\mathsf{idx}(e) = [e_1 + e_2]$.
  - *If* $C(T) = (w_1, e_1)$ *and* $C(T') = (w_2, e_2)$ *for some* $w_1, w_2 \in \mathsf{Pref}(w)$, *then* $T >_{w,\mathsf{idx}(e)} T'$.
  - *If* $C(T) = (w_i, e_i)$ *and* $C(T') = (w_i', e_i)$ *for some* $i \in \{1, 2\}$ *and* $w_i, w_i' \in \mathsf{Pref}(w)$, *then* $T >_{w,\mathsf{idx}(e)} T'$ *iff* $T_{(w_i,e_i)} >_{w,e_i} T'_{(w_i',e_i)}$.
– $\mathsf{idx}(e) = [e_1 \cdot e_2]$. *Suppose* $C(T) = (w_1, e_1)(w_2, e_2)$ *and* $C(T') = (w_1', e_1)(w_2', e_2)$ *(note that* w_1 *is a prefix of* w_1' *or vice versa), then* $T >_{w,\mathsf{idx}(e)} T'$ *when either* $T_{(w_1,e_1)} >_{w,e_1} T'_{(w_1',e_1)}$, *or* $w_1 = w_1'$ *and* $T_{(w_2,e_2)} >_{w_1^{-1}w,e_2} T'_{(w_2',e_2)}$.
– $\mathsf{idx}(e) = [e_1^*]$.
  - *If* T' *is a single node* $(\varepsilon, \mathsf{idx}(e))$, *but* T *is not, then* $T >_{w,\mathsf{idx}(e)} T'$.
  - *Otherwise, suppose* $C(T) = (w_1, e_1) \dots (w_k, e_1)$ *and* $C(T') = (w_1', e_1) \dots (w_l', e_1)$, *we have* $T >_{w,\mathsf{idx}(e)} T'$ *iff either* $k > l$ *and for every* $i \in [l]$ *we have* $T_{(w_j,e_1)} = T_{(w_j',e_1)}$, *or for the first index* j *such that* $T_{(w_j,e_1)} \neq T_{(w_j',e_1)}$, *we have* $T_{(w_j,e_1)} >_{(w_1 \cdots w_{j-1})^{-1}w,e_1} T'_{(w_j',e_1)}$.
– $\mathsf{idx}(e) = [e_1^+]$ *or* $[e_1^{\{m_1,m_2\}}]$. *Suppose* $C(T) = (w_1, e_1) \dots (w_k, e_1)$ *and* $C(T') = (w_1', e_1) \dots (w_l', e_1)$, *then* $T >_{w,\mathsf{idx}(e)} T'$ *iff either* $k > l$ *and for every* $i \in [l]$ *we have* $T_{(w_j,e_1)} = T_{(w_j',e_1)}$, *or for the first index* j *such that* $T_{(w_j,e_1)} \neq T_{(w_j',e_1)}$, *we have* $T_{(w_j,e_1)} >_{(w_1 \cdots w_{j-1})^{-1}w,e_1} T'_{(w_j',e_1)}$.
– $\mathsf{idx}(e) = [e_1^{*?}]$.
  - *If* T *is a single node* $(\varepsilon, \mathsf{idx}(e))$, *but* T' *is not, then* $T >_{w,\mathsf{idx}(e)} T'$.
  - *Otherwise, suppose* $C(T) = (w_1, e_1) \dots (w_k, e_1)$ *and* $C(T') = (w_1', e_1) \dots (w_l', e_1)$, *we have* $T >_{w,\mathsf{idx}(e)} T'$ *iff either* $k < l$ *and for every* $i \in [k]$ *we have* $T_{(w_j,e_1)} = T_{(w_j',e_1)}$, *or for the first index* j *such that* $T_{(w_j,e_1)} \neq T_{(w_j',e_1)}$, *we have* $T_{(w_j,e_1)} >_{(w_1 \cdots w_{j-1})^{-1}w,e_1} T'_{(w_j',e_1)}$.
– $\mathsf{idx}(e) = [e_1^{+?}]$ *or* $[e_1^{\{m_1,m_2\}?}]$. *Suppose* $C(T) = (w_1, e_1) \dots (w_k, e_1)$ *and* $C(T') = (w_1', e_1) \dots (w_l', e_1)$, *then* $T >_{w,\mathsf{idx}(e)} T'$ *iff either* $k < l$ *and for every* $i \in [k]$ *we have* $T_{(w_j,e_1)} = T_{(w_j',e_1)}$, *or for the first index* j *such that* $T_{(w_j,e_1)} \neq T_{(w_j',e_1)}$, *we have* $T_{(w_j,e_1)} >_{(w_1 \cdots w_{j-1})^{-1}w,e_1} T'_{(w_j',e_1)}$.

For $e \in \mathsf{RWRE}$ *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* $e \in \mathsf{RWRE}$ *and* $w \in \Sigma^*$, *if* $w \in \Sigma^*\mathscr{L}(e)\Sigma^*$, *then the (first)* match *of* e *in* w *is defined as the supremum of* $\mathscr{M}_{\mathsf{Pref}(w_1^{-1}w)}(e)$ *such that* w_1 *is the shortest prefix of* w *satisfying that* $w_1^{-1}w \in \mathscr{L}(e)\Sigma^*$. *If* $w \notin \Sigma^*\mathscr{L}(e)\Sigma^*$, *the match of* e *in* w *is undefined.*

Example 4. Let us continue Example 3. In Fig. 3, we have $(T_1)_{(0250,[\Gamma^+])} >_{(w,\mathsf{idx}(e))} (T_2)_{(025,[\Gamma^+])}$, since $(0,\Gamma)(2,\Gamma)(5,\Gamma)$ is a proper prefix of $(0,\Gamma)(2,\Gamma)(5,\Gamma)(0,\Gamma)$. Then we deduce that $(T_1)_{(0250,([\Gamma^+])_1)} >_{(w,\mathsf{idx}(e))} (T_2)_{(025,([\Gamma^+])_1)}$. Consequently, $(T_1)_{(0250,[([\Gamma^+])_1 \cdot .?])} >_{(w,\mathsf{idx}(e))} (T_2)_{(025,[([\Gamma^+])_1 \cdot .?])}$ and $T_1 >_{(w,\mathsf{idx}(e))} T_2$. Similarly, we have $T_2 >_{(w,\mathsf{idx}(e))} T_3$ and $T_3 >_{(w,\mathsf{idx}(e))} T_4$. Therefore, T_1 is the accepting match of e to w, where the first and second capturing group of e are matched to 0250 and ε respectively.

Remark 1. Our semantics of RWRE follows the 11th Edition of the ECMAScript specification (ES11 for short) [HS20], with a focus on the non-commutative union, the greedy/lazy semantics of Kleene star/plus, as well as capturing groups and backreferences. In comparison, POSIX regular expressions require the left-most and longest match of regular expressions, which we leave as future work.

4 The String Constraint Language

We define the string constraint language SL as follows.

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

where

- $\cdot$ is the string concatenation operation which concatenates two strings,
- for the extract function, $i \in \mathbb{N}$, $e \in \mathsf{RWRE}_{\mathsf{reg}}$,
- for the replace and replaceAll operation, $\mathsf{pat} \in \mathsf{RWRE}$, $\mathsf{rep} \in \mathsf{RWRE}_{\mathsf{ref}}$,
- for assertions, $e \in \mathsf{RWRE}$.

The extract function is used to model the regular-expression match function in programming languages. Specifically, the $\mathsf{extract}_{i,e}(x)$ function extracts the match of the i-th capturing group in the accepting match of e to x for $x \in \mathscr{L}(e)$ (otherwise, the return value of the function is undefined). Note that $\mathsf{extract}_{i,e}(x)$ returns x if $i = 0$. For instance, assuming $e = [[([\Gamma^+]) \cdot .?] \cdot ([\Gamma^*])]$, $\mathsf{extract}_{1,e}(0250) = 0250$ and $\mathsf{extract}_{2,e}(0250) = \varepsilon$, as shown in Example 4. Moreover, if the i-the capturing group of e is *not* matched, even if $x \in \mathscr{L}(e)$, then $\mathsf{extract}_{i,e}(x)$ returns null. For instance, when $[[a^+] + ([a^*])]$ is matched against the string aa, $[a^+]$, instead of $([a^*])$, will be matched, since $[a^+]$ precedes $([a^*])$. Therefore, $\mathsf{extract}_{1,[[a^+]+([a^*])]}(aa) = \mathsf{null}$.

Remark 2. The match function in programming languages, e.g. str.match(reg) function in JavaScript, finds the first match of reg in str. We can use extract to express the first match of reg in str by adding $[\Sigma^{*?}]$ and $[\Sigma^*]$ before and after reg respectively. More generally, the value of the i-th capturing group in the first match of a RWRE reg in str can be specified as $\mathsf{extract}_{i+1,\mathsf{reg}'}(\mathsf{str})$, where $\mathsf{reg}' = [[[\Sigma^{*?}] \cdot (\mathsf{reg})] \cdot [\Sigma^*]]$. The other string functions involving RWREs, e.g. exec and test, are similar to match, thus can be encoded by extract as well.

The $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$ function is parameterized by the *pattern* $\mathsf{pat} \in \mathsf{RWRE}$ and the *replacement* string $\mathsf{rep} \in \mathsf{RWRE}_{\mathsf{ref}}$. For an input string x, it identifies all matches of pat in x and replaces them with strings specified by the replacement string rep. (In rep, references may be used to refer to the corresponding matches of the capturing groups.) For instance, let $w = 2.5, 3.4$, $\mathsf{pat} = [[([\Gamma^+]) \cdot .?] \cdot ([\Gamma^*])]$ and $\mathsf{rep} = \$1$, then $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(w) = 2, 3$. The $\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}(x)$ function is similar to $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$, except that it replaces at most once.

Without loss of generality, we assume that all the SL programs are in single static assignment (SSA) form, that is 1) each variable x is assigned at most once; and 2) if x is assigned, all its occurrences on the right hand sides of the assignment statements or in assertions are after the assignment statement of x. For an SL program S, a variable x occurring in S is said to be an *input* variable if x does not occur on the left hand sides of assignment statements.

The *path feasibility* problem of an SL program is to decide whether there are valuations of the input variables so that the program can execute to the end. This problem turns out to be undecidable, this is because of the backreferences in assertion statements or in pattern parameters of the replace/replaceAll function.

Proposition 1. *The path feasibility problem of* SL *is undecidable.*

We shall show that the path feasibility problem becomes decidable, if the uses of backreferences in assertion statements and pattern parameters of the replace/replaceAll function are forbidden, which turns out to be the situation in practice, according to the statistics in the literature (see Section 1). In the sequel, we will use $\mathsf{SL}_{\mathsf{reg}}$ to denote the collection of SL programs which are free of backreferences in assertion statements as well as in pattern parameters of the replace/replaceAll function. Note that $\mathsf{SL}_{\mathsf{reg}}$ allows backreferences in replacement parameters of replace/replaceAll. The main result of this paper is as follows.

Theorem 1. *The path feasibility of* $\mathsf{SL}_{\mathsf{reg}}$ *is decidable.*

The decision procedure for $\mathsf{SL}_{\mathsf{reg}}$ utilizes a new automata 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), a new class of transducers that combine prioritized finite-state automata [BvdM17a] and streaming string transducers [AC10,AD11]. We shall utilize PSSTs to model greedy/lazy semantics of Kleene star/plus as well as the behavior of the extract and replaceAll functions.

Definition 4 (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 $q_0 a_1 q_1 \dots a_n q_n$ such that $w = a_1 \cdots a_n$ and $(q_{j-1}, a_j, q_j) \in \delta$ 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. The set of strings accepted by $\mathcal{A}$, i.e., the language *recognized* by $\mathcal{A}$, is denoted by $\mathscr{L}(\mathcal{A})$. The *size* $|\mathcal{A}|$ of $\mathcal{A}$ is the cardinality of the set Q of states.

For a finite set Q, let $\overline{Q} = \bigcup_{n\in\mathbb{N}}\{(q_1, \dots, q_n) \mid \forall i \in [n], q_i \in Q \wedge \forall i, j \in [n], i \neq j \rightarrow q_i \neq q_j\}$. 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 (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 $\delta \in Q \times \Sigma \to \overline{Q}$ and $\tau \in Q \to \overline{Q} \times \overline{Q}$ such that for every $q \in Q$, if $\tau(q) = (P_1; P_2)$, then $P_1 \cap P_2 = \emptyset$. The definition of Q, q_0 and F is the same as 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 $q_0 a'_1 q_1 \ldots a'_m q_m$ such that

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

Note that it is possible that $\delta(q, a) = ()$, that is, there is no a-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 $R = q_0 a_1 q_1 \ldots a_m q_m$ and $R' = q_0 a'_1 q'_1 \ldots a'_n q'_n$ such that $a_1 \ldots a_m = a'_1 \ldots a'_n$, we say that R is of a higher priority over R' if

- either R' is a prefix of R (in this case, the transitions of R after R' are all ε-transitions),
- or there is an index j satisfying one of the following constraints:
  - $q_0 a_1 q_1 \ldots q_{j-1} a_j = q_0 a'_1 q'_1 \ldots q'_{j-1} a'_j$, $q_j \neq q'_j$, $a_j \in \Sigma$, and $\delta(q_{j-1}, a_j) = (\ldots, q_j, \ldots, q'_j, \ldots)$,
  - $q_0 a_1 q_1 \ldots q_{j-1} a_j = q_0 a'_1 q'_1 \ldots q'_{j-1} a'_j$, $q_j \neq q'_j$, $a_j = \varepsilon$, and one of the following conditions holds: (i) $\pi_1(\tau(q_{j-1})) = (\ldots, q_j, \ldots, q'_j, \ldots)$, (ii) $\pi_2(\tau(q_{j-1})) = (\ldots, q_j, \ldots, q'_j, \ldots)$, or (iii) $q_j \in \pi_1(\tau(q_{j-1}))$ and $q'_j \in \pi_2(\tau(q_{j-1}))$,
  - $q_0 a_1 q_1 \ldots q_{j-1} = q_0 a'_1 q'_1 \ldots q'_{j-1}$, $a_j = \varepsilon$, $a'_j \in \Sigma$, $q_j \in \pi_1(\tau(q_{j-1}))$, and $q'_j \in \delta(q_{j-1}, a'_j)$,
  - $q_0 a_1 q_1 \ldots q_{j-1} = q_0 a'_1 q'_1 \ldots q'_{j-1}$, $a_j \in \Sigma$, $a'_j = \varepsilon$, $q_j \in \delta(q_{j-1}, a_j)$, and $q'_j \in \pi_2(\tau(q_{j-1}))$.

An *accepting* run of $\mathcal{A}$ on w is a run $R = q_0 a_1 q_1 \ldots a_m q_m$ of $\mathcal{A}$ on w satisfying that 1) $q_m \in F$, 2) R is of the *highest* priority among those runs satisfying $q_m \in F$. 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 the priorities in PFAs do not change the fact that a string is accepted, they only affect the way that a string is accepted. Therefore, PFAs still define the class of regular languages.

Example 5. The PFA corresponding to the RWRE $e = [[([\Gamma^+]) \cdot .?] \cdot ([\Gamma^*])]$ in Example 3 is illustrated in Fig. 4, where the dashed (resp. thicker solid) lines represent the ε-transitions of lower (resp. higher) priorities than non-ε transitions (if there is any), and the doubly circled states are final states. For instance, $\delta(q_1, \ell) = (q_1)$ for every $\ell \in \{0, \ldots, 9\}$, $\delta(q_1, .) = ()$, $\tau(q_1) = ((); (q_2))$. Since the ε-transition has lower priority than the ℓ-transition at the state q_1, whenever the currently scanned letter is $\ell \in \{0, \cdots, 9\}$ at q_1, the PFA will choose to go to q_1 greedily, until there is no more $\ell \in \{0, \cdots, 9\}$. (In this case, it has to choose the ϵ-transition and goes to q_2.)

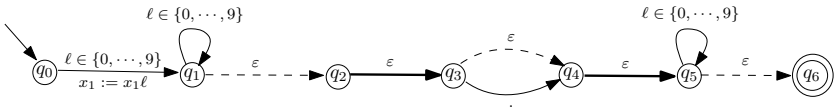

Fig. 4. The PFA for $e = [[([\Gamma^+]) \cdot .?] \cdot ([\Gamma^*])]$, where $\Gamma = \{0, \cdots, 9\}$

We are ready to define prioritized streaming string transducers. In the definition, the special symbol null is introduced to capture the situation that $\mathsf{extract}_{i,e}(x)$ returns null, i.e. $x \in \mathscr{L}(e)$ but the i-th capturing group of e is not matched.

Definition 6 (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 *is a finite set of states,* Σ *is the input and output alphabet such that* $\mathsf{null} \notin \Sigma$*,* 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* $X \to \{\mathsf{null}\} \cup (X \cup \Sigma)^*$*, 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 $q_0 a_1 s_1 q_1 \ldots a_m s_m q_m$ such that

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

For any pair of runs $R = q_0 a_1 s_1 \ldots a_m s_m q_m$ and $R' = q_0 a'_1 s'_1 \ldots a'_n s'_n q'_n$ such that $a_1 \ldots a_m = a'_1 \ldots a'_n$, the definition that R is of a higher priority over R' is similar to PFAs.

An accepting run of $\mathcal{T}$ on w is a run of $\mathcal{T}$ on w, say $R = q_0 a_1 s_1 \ldots a_m s_m q_m$, such that 1) $F(q_m)$ is defined, 2) R is of the highest priority among those runs satisfying 1). 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 $1 \leq i \leq m$ 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 $(\Sigma \cup \{\mathsf{null}\})^*$ as a function from $(X \cup \Sigma \cup \{\mathsf{null}\})^*$

to $(\Sigma \cup \{\mathsf{null}\})^*$, which maps each $x \in X$ to $\eta(x)$, each $a \in \Sigma$ to a, and null to null . If there is no accepting run of $\mathcal{T}$ on w, then $\mathcal{T}(w) = \bot$, that is, the output of $\mathcal{T}$ on w is undefined. The string relation defined by $\mathcal{T}$, denoted by $\mathcal{R}_\mathcal{T}$, is

$$\{(w, \mathcal{T}(w)) \mid w \in \Sigma^*, \mathcal{T}(w) \in \Sigma^* \cup \{\mathsf{null}\}\}.$$

Note that in the definition of $\mathcal{R}_\mathcal{T}$ above, the inputs of $\mathcal{T}$ whose outputs are in $(\Sigma \cup \{\mathsf{null}\})^* \setminus (\Sigma^* \cup \{\mathsf{null}\})$ are ignored.

Example 6. The PSST $\mathcal{T}_{\mathtt{extract}_{\mathtt{decimalReg},1}} = (Q, \Sigma, X, \delta, \tau, E, q_0, F)$ mentioned in Section 2 is obtained from the PFA in Fig. 4 by adding $x_1 := x_1\ell$ to each ℓ-transition going into q_1 (see Fig. 5). More specifically, in $\mathcal{T}_{\mathtt{extract}_{\mathtt{decimalReg},1}}$, we have $\Sigma = \{0, \cdots, 9, .\}$, $X = \{x_1\}$ with x_1 recording the matches of the 1st capturing group, $F(q_6) = x_1$ denotes the final output, and δ, τ, E are illustrated by the edges in Fig. 5, where the dashed/ticker solid edges denote the ε-transitions of lower/higher priorities than the non-ε-transitions and the symbol ℓ is used to denote the currently scanned input letter. Note that the identity assignments, e.g. $E(q_3, ., q_4)(x_1) = x_1$, are omitted in Fig. 5, for readability.

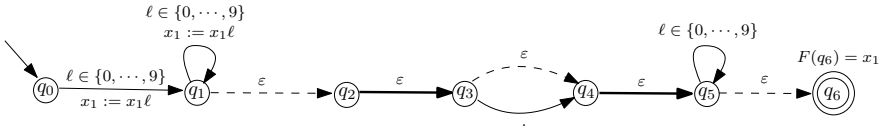

Fig. 5. The PSST $\mathcal{T}_{\mathtt{extract}_{\mathtt{decimalReg},1}}$

6 Decision Procedure for $\mathsf{SL}_{\mathsf{reg}}$

The goal of this section is to prove Theorem 1, i.e., the decidability of the path feasibility of $\mathsf{SL}_{\mathsf{reg}}$. We first show that semantically equivalent PSSTs can be effectively constructed from the extract, replace, and replaceAll functions.

Lemma 1. *For each string function $f = \mathsf{extract}_{i,e}$, $\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}$, or $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}$, a PSST $\mathcal{T}_f$ can be constructed such that $\mathcal{R}_f = \{(w, w') \mid w' = f(w)\}$.*

The proof is given in Appendix A.3. With Lemma 1, the path feasibility of $\mathsf{SL}_{\mathsf{reg}}$ reduces to path feasibility of string-manipulating programs consisting of a sequence of the statements of the form $z := x \cdot y$, $y := \mathcal{T}(x)$, and $\mathsf{assert}\,(x \in \mathcal{A})$, where $\mathcal{T}$ is a PSST and $\mathcal{A}$ is an FA. The class of programs is denoted by $\mathsf{SL}'_{\mathsf{reg}}$. We then follow the backward reasoning approach proposed in [CCH+18,CHL+19] to solve the path feasibility of $\mathsf{SL}'_{\mathsf{reg}}$, where the key is to show that the pre-images of regular languages under PSSTs are regular and can be computed effectively.

Definition 7 (Pre-image). *For a string relation $R \subseteq \Sigma^* \times (\Sigma^* \cup \{\mathsf{null}\})$ and $L \subseteq \Sigma^*$, we define the* pre-image *of L under R as $R^{-1}(L) := \{w \in \Sigma^* \mid \exists w'.\ w' \in L$ and $(w, w') \in R\}$.*

Lemma 2 (Pre-image of regular languages under PSSTs). *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* $\mathscr{L}(\mathcal{B}) = \mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}))$.

The proof of Lemma 2 is given in Appendix A.4. With Lemma 2, we solve the path feasibility of $\mathsf{SL}'_{\mathsf{reg}}$ by repeating the following procedure, until no more assignment statements are left. Let S be the current $\mathsf{SL}'_{\mathsf{reg}}$ program.

- If the last assignment statement of S is $y := \mathcal{T}(x)$, then let $\mathsf{assert}\,(y \in \mathcal{A}_1)$, $\cdots$, $\mathsf{assert}\,(y \in \mathcal{A}_n)$ be an enumeration of all the assertion statements for y in S. Compute $\mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}_1))$ as an FA $\mathcal{B}_1, \cdots$, and $\mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}_n))$ as $\mathcal{B}_n$. Remove the assignment $y := \mathcal{T}(x)$ and add the assertion statements $\mathsf{assert}\,(x \in \mathcal{B}_1)$; $\cdots$; $\mathsf{assert}\,(x \in \mathcal{B}_n)$.
- If the last assignment statement of S is $z := x \cdot y$, then let $\mathsf{assert}\,(z \in \mathcal{A}_1)$, $\cdots$, $\mathsf{assert}\,(z \in \mathcal{A}_n)$ be an enumeration of all the assertion statements for z in S. Compute $\cdot^{-1}(\mathscr{L}(\mathcal{A}_1))$, the pre-image of $\cdot$ under $\mathscr{L}(\mathcal{A}_1)$, as a collection of FA pairs $(\mathcal{B}_{1,j}, \mathcal{C}_{1,j})_{j\in[m_1]}$, $\cdots$, and $\cdot^{-1}(\mathscr{L}(\mathcal{A}_n))$ as $(\mathcal{B}_{n,j}, \mathcal{C}_{n,j})_{j\in[m_n]}$ (c.f. [CHL$^+$19]). 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 $\mathsf{assert}\,(x \in \mathcal{B}_{1,j_1})\,;\mathsf{assert}\,(y \in \mathcal{C}_{1,j_1})$; $\cdots$; $\mathsf{assert}\,(x \in \mathcal{B}_{n,j_n})\,;\mathsf{assert}\,(y \in \mathcal{C}_{n,j_n})$.

Let S' be the resulting $\mathsf{SL}'_{\mathsf{reg}}$ program containing no assignment statements. Then the path feasibility of S' can be solved by checking the nonemptiness of the intersection of regular constraints for the input variables, which is known to be PSPACE-complete [Koz77].

Complexity analysis. Because the pre-image computation for each PSST incurs an exponential blow-up, the aforementioned decision procedure has a non-elementary complexity in the worst-case. Nevertheless, since the number of PSSTs is usually small in the path constraints of string-manipulating programs, the performance of the decision procedure is actually good on the benchmarks we tested, with the average running time per query a few seconds (see Section 7).

Remark 3. The procedure above extends the backward-reasoning approach in [CHL$^+$19], since standard one-way and two-way finite-state transducers were considered therein and PSSTs, in particular priorities, are beyond them[1].

7 Implementation and Experiments

We have implemented our decision procedure for $\mathsf{SL}_{\mathsf{reg}}$ in the SMT solver EMU,[2] extending the open-source solver OSTRICH [CHL$^+$19]. As shown in Section 6,

[1] It is known that deterministic streaming string transducers are expressively equivalent to two-way deterministic finite-state transducers, which, nevertheless, is not the case for nondeterministic transducers [AC10,AD11].

[2] Name anonymized for doubly-blind review, and will be provided in the final version.

PSSTs satisfy the conditions required by the backward reasoning approach of OSTRICH, which enables us to integrate our logic with standard string theory. The resulting extended theory of strings is a conservative extension of the SMT-LIB theory of Unicode strings.[3]

7.1 Implementation

Our implementation extends classical regular expressions in SMT-LIB with indexed re.capture and re.reference operators, which denote capturing groups and references to them. We also add re.*?, re.+? and re.loop? for the lazy counterparts of Kleene star, plus operator and loop operator.

The three string operators that use these extended real world regular expressions are str.replace_cg, str.replace_cg_all, and str.extract. Operators str.replace_cg and str.replace_cg_all are counterparts of the standard str.replace_re and replace_re_all operators, and allow capturing groups in the match pattern and references in the replacement pattern. As an example, the following constraint swaps the first name and the last name (see Example 2),

```
(= w (str.replace_cg_all v (re.++
 ((_ re.capture 1)(re.+ (re.union (re.range "A" "Z")(re.range "a" "z"))))
 (str.to.re " ")
 ((_ re.capture 2)(re.+ (re.union (re.range "A" "Z")(re.range "a" "z")))))
 (re.++ (_ re.reference 2) (_ re.reference 1))))
```

The replacement string is written as a regular expression only containing the operators re.++, str.to_re, and re.reference. In contrast to the standard operators, using string variables in the replacement parameter is not allowed.

The indexed operator str.extract implements $\mathsf{extract}_{i,e}$ in $\mathsf{SL}_{\mathsf{reg}}$. For instance,

```
((_ str.extract 1)
      (re.++ (re.*? re.allchar)
             ((_ re.capture 1) (re.+ (re.range "a" "z")) re.all)) x)
```

extracts the left-most, longest sub-string consisting of only lower-case characters from a string x.

Our implementation is able to handle *anchors* as well. Due to space restrictions and concerns about simplicity, we did not introduce them as part of our formalism. Anchors are special symbols that match certain positions of a string without consuming any input characters. In most practical programming languages, it is quite common to use ^ and $ in regular expressions to signify the start and end of a string, respectively. We add re.begin-anchor and re.end-anchor for them. Our implementation correctly models the semantics of anchors and is able to solve constraints containing these operators.

The implementation revolves around PSSTs. Any of the three string operators mentioned above will be converted into an equivalent PSST (See Appendix A.3). EMU then iterates the dependency graph and repeatedly eliminates them. More specifically, we first use Lemma 2 to back-propagate regular

[3] http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml

```
(declare-fun x () String)
(define-fun  y () String
   (str.replace_cg_all x <re1> <repl>))
(push 1)
(assert (str.in.re x
          (re.++ re.all <re1> re.all)))
(assert (str.in.re y
          (re.++ re.all <re2> re.all)))
(check-sat) (get-model)
(pop 1) (push 1)
(assert (str.in.re x
          (re.++ re.all <re1> re.all)))
(assert (not (str.in.re y
          (re.++ re.all <re2> re.all))))
(check-sat) (get-model)
(pop 1) (push 1)
(assert (not (str.in.re x
          (re.++ re.all <re1> re.all))))
(check-sat) (get-model)
(pop 1)
```

```
function fun(x) {
   if(/<re1>/.test(x)) {
      var y = x.replace(/<re1>/g, <repl>);
      if(/<re2>/.test(y))
        console.log("1");
      else
        console.log("2");
   }
   else
       console.log("3");
}

var S$ = require("S$");
var x = S$.symbol("x", "");
fun(x);
```

Fig. 6. Harnesses with replace-all: SMT-LIB for EMU, and JavaScript for ExpoSE.

constraints on string variables and check satisfiability, and then use forward concrete evaluation to generate a model. We omit the details here due to space restrictions, and refer the readers to the baseline [CHL+19].

7.2 Experimental evaluation

Our experiments have the purpose of answering the following main questions:

R1: Are the RWREs defined in this paper suitable to encode regular expressions in programming languages, for instance ECMAScript regular expressions [HS20]?
R2: How does EMU compare to other solvers that can handle real-world regular expressions, e.g., greedy/lazy quantifiers and capturing groups?
R3: How does EMU perform in the context of symbolic execution, the primary application of string constraint solving?

*For **R1**:* We implemented a translator from ECMAScript 11th Edition (ES11 for short) regular expressions to RWREs, and integrated EMU into the symbolic execution tool Aratha [AAG+19]. We then ran Aratha+EMU on the regression test suite of ExpoSE [LMK17], as well as some other JavaScript programs containing match or replace functions extracted from Github. To verify the soundness of Aratha+EMU, we compared the results with those produced by ExpoSE; we also checked the correctness of models computed by EMU by concretely executing the JavaScript program under test on the generated inputs, to confirm that the concrete execution indeed follows the targeted path. The results are summarized in Table 1; no inconsistencies were observed in the experiments, showing that the semantics in this paper are indeed suitable for capturing ES11 semantics.

*For **R2**:* There are no standard string benchmarks involving RWREs, and we are not aware of other constraint solvers supporting capturing groups, neither among

	Aratha+EMU						ExpoSE+Z3					
	# paths covered within 60s						# paths covered within 60s					
	0	1	2	3	≥4	#T.O.	0	1	2	3	≥4	#T.O.
ExpoSE	14	9	9	2	15	0	15	8	9	2	15	1
(49 programs)	Average time: **6.49**s						Average time: 13.46s					
	Total #paths covered:124						Total #paths covered:120					
Match	3	7	12	6	0	0	3	8	12	5	0	6
(28 programs)	Average time: **4.30**s						Average time: 23.66s					
	Total #paths covered: 49						Total #paths covered: 47					
Replace	12	20	6	0	0	0	12	23	3	0	0	22
(38 programs)	Average time: **2.71**s						Average time: 41.73s					
	Total #paths covered: 32						Total #paths covered: 29					

Table 1. Results of Expose+Z3 and Aratha+EMU on Javascript programs for **R1** and **R3**. Experiments were done on an Intel-Xeon-E5-2690-@2.90GHz machine, running 64-bit Linux and Java 1.8. Runtime was limited to 60s wall-clock time. Average time is wall-clock time needed per benchmark, and counts timeouts as 60s. #T.O. is the number of timeouts. Note that some paths may have already been covered before T.O.

the SMT nor the CP solvers. The closest related work is the algorithm implemented in ExpoSE, which applies Z3 [dMB08] for solving string constraints, but augments it with a refinement loop to approximate the RWRE semantics.[4] For **R2**, we compared EMU with ExpoSE on 98,117 RWREs taken from [DMIC+19].

For each of the regular expressions, we created four harnesses: two in SMT-LIB, as inputs for EMU, and two in JavaScript, as inputs for ExpoSE. The two harnesses shown in Fig. 6 use one of the regular expressions from [DMIC+19] (`<re1>`) in combination with the replace-all function to simulate typical string processing; `<re2>` is the fixed pattern `[a-z]+`, and `<repl>` the replacement string `"$1"`. The three paths of the JavaScript function `fun` correspond to the three queries in the SMT-LIB script, so that a direct comparison can be made between the results of the SMT-LIB queries and the set of paths covered by ExpoSE. The other two harnesses are similar to the ones in Fig. 6, but use the match function instead of replace-all, and contain four queries and four paths, respectively.

The results of this experiment are shown in Table 2. EMU is able to answer all four queries in 96,126 of the match benchmarks (97.9%), and all three queries in 95,735 of the replace-all benchmarks (97.6%). The errors in 1,132 cases are due to back-references in `<re1>`, which are not handled by EMU. ExpoSE can on the match problems cover 228,888 paths in total (91.5% of the number of sat results of EMU), although the runtime of ExpoSE is on average 23x higher than that of EMU. For replace, ExpoSE can cover 173,007 paths (63.2%), showing that this class of constraints is harder; the runtime of ExpoSE is on average 37x higher than that of EMU. Overall, even taking into account that ExpoSE

[4] We considered replacing Z3 with EMU in ExpoSE for the experiments. However, ExpoSE integrates Z3 using its C API, and changing to EMU, with native support for capture groups, would have required the rewrite of substantial parts of ExpoSE.

	EMU						ExpoSE+Z3				
	# queries solved within 60s						# paths covered within 60s				
	0	1	2	3	4	#Err	0	1	2	3	4
Match	172	92	61	534	96,126	1132	3,333	9,274	36,916	48,594	
(98,117	Average time: 1.19s						Average time: 28.0s				
benchm.)	Total #sat: 249,975, #unsat: 136,345						Total #paths covered: 228,888				
Replace	445	229	576	95,735	—	1,132	5,281	18,221	69,059	5,556	—
(98,117	Average time: 1.48s						Average time: 55.0s				
bench.)	Total #sat: 273,927, #unsat: 14,659						Total #paths covered: 173,007				

Table 2. The number of queries answered by EMU, and number of paths covered by ExpoSE, in the **R2** experiments. Experiments were done on an AMD Opteron 2220 SE machine, running 64-bit Linux and Java 1.8. Runtime per benchmark was limited to 60s wall-clock time, 2GB memory, and the number of tests executed concurrently by ExpoSE to 1. Average time is wall-clock time per benchmark, timeouts count as 60s.

has to analyze JavaScript code, as opposed to the SMT-LIB given to EMU, the experiments show that EMU is a highly competitive solver for RWREs.

*For **R3**:* We compare Aratha+EMU with ExpoSE+Z3 on the benchmarks from **R1**. In Table 1, we can see that Aratha+EMU can within 60s cover slightly more paths than ExpoSE+Z3. Aratha+EMU can discover feasible paths more quickly than ExpoSE+Z3, however: on all three families of benchmarks, Aratha+EMU terminates on average in less than 10s, and it discovers all paths within 35s. ExpoSE+Z3 needs full 60s for 29 of the programs ("T.O." in the table), and it finds new paths until the end of the 60s. Since ExpoSE+Z3 handles the replace-all operation by unrolling, it is not able to prove infeasibility of paths involving such operations, and will therefore not terminate on some programs. Overall, the experiments indicate that EMU is more efficient than the CEGAR-augmented symbolic execution for dealing with RWREs.

8 Related Work and Conclusion

RWREs have been investigated in formal language theory. Regular expressions with capturing groups and backreferences were considered in [CSY03,CN09] and also more recently in [Fre13,Sch16,FS19], where the expressibility issues and decision problems were investigated. Nevertheless, some basic features of RWREs, namely, the non-commutative union and the greedy/lazy semantics of Kleene star/plus, were not addressed therein.

Prioritized finite-state automata and prioritized finite-state transducers were proposed in [BvdM17a]. Prioritized finite-state transducers add indexed brackets to the input string in order to identify the matches of capturing groups. It is hard, if not impossible, to use prioritized finite-state transducers to model replace(all) function in general, e.g. swapping the first and last name as in Example 2. In contrast, PSSTs store the matches of capturing groups into string

variables, which can then be referred to, thus allowing us to conveniently model the match and replace(all) function. Streaming string transducers were used in [ZAM19] to solve the straight-line string constraints with concatenation, finite-state transducers, and regular constraints.

RWREs have also received attention in the software engineering community. Some empirical studies were reported for RWREs recently, including portability across different programing languages [DMIC+19] and DDos attacks [SP18], as well as how programmers write RWREs in practice [MDD+19].

To conclude, in this paper, we proposed a novel approach for natively supporting real-world regular expressions (RWRE) in string constraint solving. We introduced prioritized streaming string transducers (PSSTs) to model the string functions involving real-world regular expressions. We showed that the pre-images of regular languages under PSSTs are regular and designed a decision procedure for string constraints with RWREs. We implemented the decision procedure and carried out extensive experiments. The experimental results showed that our approach significantly improves the CEGAR-based approach for RWREs in both precision and performance. To the best of our knowledge, this work represents the first string constraint solver that natively supports RWREs. For the future work, it is interesting to extend this work to deal with more advanced features of RWREs, e.g., lookahead and lookbehind. It is also desirable to support the string functions involving the integer data type, in addition to those involving RWREs.

References

[AAG+19] Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Constraint programming for dynamic symbolic execution of javascript. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2019.

[AC10] Rajeev Alur and Pavol Cerný. 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*, pages 1–12, 2010.

[AD11] Rajeev Alur and Jyotirmoy V. Deshmukh. Nondeterministic streaming string transducers. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[Aho90] Alfred V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 255–300. 1990.

[BDvdM14] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. In Zoltán Ésik and Zoltán Fülöp, editors, *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014*, volume 151 of *EPTCS*, pages 109–123, 2014.

[BvdM17a] Martin Berglund and Brink van der Merwe. On the semantics of regular expression parsing in the wild. *Theoretical Computer Science*, 679:69 – 82, 2017.

[BvdM17b] Martin Berglund and Brink van der Merwe. Regular expressions with backreferences re-examined. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, pages 30–41. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017.

[CCH+18] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *PACMPL*, 2(POPL):3:1–3:29, 2018.

[CHL+19] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *PACMPL*, 3(POPL), January 2019.

[CN09] Benjamin Carle and Paliath Narendran. On extended regular expressions. In Adrian-Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, volume 5457 of *Lecture Notes in Computer Science*, pages 279–289. Springer, 2009.

[CS16] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in python. In Andreas Zeller and Abhik Roychoudhury, editors,

Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, pages 282–293. ACM, 2016.

[CSY03] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[DMIC+19] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 443–454, New York, NY, USA, 2019. Association for Computing Machinery.

[FR17] Emmanuel Filiot and Pierre-Alain Reynier. Copyful streaming string transducers. In Matthew Hague and Igor Potapov, editors, *Reachability Problems - 11th International Workshop, RP 2017, London, UK, September 7-9, 2017, Proceedings*, volume 10506 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 2017.

[Fre13] Dominik D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory Comput. Syst.*, 53(2):159–193, 2013.

[FS19] Dominik D. Freydenberger and Markus L. Schmid. Deterministic regular expressions with back-references. *J. Comput. Syst. Sci.*, 105:1–39, 2019.

[Gut98] Claudio Gutiérrez. Solving equations in strings: On makanin's algorithm. In *LATIN*, pages 358–373, 1998.

[HS20] Jordan Harband and Kevin Smith. ECMAScript 2020 language specification, 11th edition. https://262.ecma-international.org/11.0/, 2020.

[KGA+12] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, 2012.

[Koz77] Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 254–266. IEEE Computer Society, 1977.

[LMK17] Blake Loring, Duncan Mitchell, and Johannes Kinder. Expose: practical symbolic execution of standalone javascript. In Hakan Erdogmus and Klaus Havelund, editors, *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, pages 196–199. ACM, 2017.

[LMK19] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 425–438. ACM, 2019.

[MDD+19] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 415–426. IEEE Press, 2019.

[SAH+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 513–528, 2010.

[Sch16] Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput.*, 249:1–17, 2016.

[SP18] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 361–376, USA, 2018. USENIX Association.

[Tho68] Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[ZAM19] Qizhen Zhu, Hitoshi Akama, and Yasuhiko Minamide. Solving string constraints with streaming string transducers. *Journal of Information Processing*, 27:810–821, 2019.

A Appendix

A.1 Backward Reasoning in the Motivating Example

The path feasibility problem of the program in Equation (1) is solved by "backward" reasoning as follows:

– At first, we compute the pre-image of $(\mathcal{A}_{\texttt{/\^0\textbackslash d+.*|.*\textbackslash .\textbackslash d*0\$/}})$ under the concatenation $\cdot$, which is a finite union of products of regular languages, remove $\texttt{result2} := \texttt{integer} \cdot \text{“.”} \cdot \texttt{fractional}$, select one disjunct of the union, say $(\mathcal{A}'_1, \mathcal{A}'_2)$, add the assertion $\mathsf{assert}\,(\texttt{integer} \in \mathcal{A}'_1)\,;\mathsf{assert}\,(\texttt{fractional} \in \mathcal{A}'_2)$, resulting into the following program,

$$
\begin{aligned}
&\mathsf{assert}\left(\texttt{decimal} \in \mathcal{A}_{\texttt{decimalReg}}\right);\\
&\texttt{integer} := \mathcal{T}_{\texttt{replace(/\^0+/, "")}}(\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},1}}(\texttt{decimal}));\\
&\texttt{fractional} := \mathcal{T}_{\texttt{replace(/0+\$/, "")}}(\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},2}}(\texttt{decimal}));\\
&\mathsf{assert}\,(\texttt{integer} \in \mathcal{A}_{\texttt{.+}})\,;\mathsf{assert}\,(\texttt{fractional} \in \mathcal{A}_{\texttt{.+}})\,;\\
&\mathsf{assert}\left(\texttt{result2} \in \mathcal{A}_{\texttt{/\^0\textbackslash d+.*|.*\textbackslash .\textbackslash d*0\$/}}\right);\\
&\mathsf{assert}\,(\texttt{integer} \in \mathcal{A}'_1)\,;\mathsf{assert}\,(\texttt{fractional} \in \mathcal{A}'_2)\,;
\end{aligned}
\tag{2}
$$

– Next, we compute the pre-image of $\mathcal{A}'_2$ under $\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},2}}$ (see Lemma 2), denoted by $\mathcal{B}_1$, then the pre-image of $\mathscr{L}(\mathcal{B}_1)$ under $\mathcal{T}_{\texttt{replace(/0+\$/, "")}}$, denoted by $\mathcal{B}'_1$. Similarly, we compute the pre-image of $\mathcal{A}_{\texttt{.+}}$ under $\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},2}}$ as well as $\mathcal{T}_{\texttt{replace(/\^0+/, "")}}$, and obtain a finite automaton $\mathcal{B}'_2$. Moreover, we remove the assignment statement for `fractional`, and add the assertions $\mathsf{assert}\,(\texttt{decimal} \in \mathcal{B}'_1)\,;\mathsf{assert}\,(\texttt{decimal} \in \mathcal{B}'_2)$. Finally, we compute the pre-images of $\mathcal{A}'_1$ and $\mathcal{A}_{\texttt{.+}}$ under $\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},1}}$ as well as $\mathcal{T}_{\texttt{replace(/\^0+/, "")}}$, and obtain finite automata $\mathcal{C}'_1$ and $\mathcal{C}'_2$ respectively. Then we remove the assignment for `integer`, and add $\mathsf{assert}\,(\texttt{decimal} \in \mathcal{C}'_1)\,;\mathsf{assert}\,(\texttt{decimal} \in \mathcal{C}'_2)$. In the end, we get the following program containing no assignment statements,

$$
\begin{aligned}
&\mathsf{assert}\left(\texttt{decimal} \in \mathcal{A}_{\texttt{decimalReg}}\right);\\
&\mathsf{assert}\,(\texttt{integer} \in \mathcal{A}_{\texttt{.+}})\,;\mathsf{assert}\,(\texttt{fractional} \in \mathcal{A}_{\texttt{.+}})\,;\\
&\mathsf{assert}\left(\texttt{result2} \in \mathcal{A}_{\texttt{/\^0\textbackslash d+.*|.*\textbackslash .\textbackslash d*0\$/}}\right);\\
&\mathsf{assert}\,(\texttt{integer} \in \mathcal{A}'_1)\,;\mathsf{assert}\,(\texttt{fractional} \in \mathcal{A}'_2)\,;\\
&\mathsf{assert}\,(\texttt{decimal} \in \mathcal{B}'_1)\,;\mathsf{assert}\,(\texttt{decimal} \in \mathcal{B}'_2)\,;\\
&\mathsf{assert}\,(\texttt{decimal} \in \mathcal{C}'_1)\,;\mathsf{assert}\,(\texttt{decimal} \in \mathcal{C}'_2)\,;
\end{aligned}
\tag{3}
$$

– Finally, we check the nonemptiness of the intersection of the regular languages for the input variable `decimal`, namely, $\mathscr{L}(\mathcal{A}_{\texttt{decimalReg}})$, $\mathscr{L}(\mathcal{B}'_1)$, $\mathscr{L}(\mathcal{B}'_2)$, $\mathscr{L}(\mathcal{C}'_1)$, and $\mathscr{L}(\mathcal{C}'_2)$. If the intersection is nonempty, then the invariant property does *not* hold.

A.2 Undecidability of SL

Proposition 1. *The path feasibility problem of* SL *is undecidable.*

Proof. The proof of Proposition 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 $\mathsf{replaceAll}$ 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 backreferences to verify the equality of $u_{i_1} \cdots u_{i_m}$ and $v_{i_1} \cdots v_{i_m}$. Specifically, the PCP instance is encoded by the following SL program,

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

Note that the above program uses backreferences in assertion statements. We can achieve the same reduction by replacing $\mathsf{assert}\,(z \in (\Sigma^+)\#\$1)$ in the above program with $z' := \mathsf{replace}(\Sigma^+\#\$1, \top); \mathsf{assert}\,(z' \in \top)$, where $\top \notin \Sigma$. Note that the program resulted from the replacement uses backreferences only in the pattern parameter of the $\mathsf{replace}$ function.

A.3 From string functions extract, replace, and replaceAll to PSST

At first, we can adapt the PFA construction in [BDvdM14], which in turn is a variant of the standard Thompson construction [Tho68], and show the following result.

Proposition 2. *For each* $\mathsf{RWRE}_{\mathsf{reg}}$ *e, a PFA* $\mathcal{A}_e$ *can be constructed in linear time such that*

- $\mathcal{A}_e$ *has a unique initial state without incoming transitions and a unique final state without outgoing transitions,*
- *for subexpression* e' *of* e, $\mathcal{A}_e$ *contains at least one isomorphic copy of* $\mathcal{A}_{e'}$ *(i.e. the PFA constructed for* e'*), denoted by* $\mathsf{Sub}_{e'}[\mathcal{A}_e]$.

Proof. For any $e \in \mathsf{RWRE}_{\mathsf{reg}}$, 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 without incoming transitions and each of its final states has no outgoing transitions,
- all the transitions out of the initial state are ε-transitions,
- the set of final states is divided into two disjoint subsets F_1, F_2 such that for each $w \in \Sigma^*$ satisfying that $q_0 \xrightarrow[\mathcal{A}_e]{w} q$ for some $q \in F_1$ (resp. $q \in F_2$), $w = \varepsilon$ (resp. $w \neq \varepsilon$).

Specifically, $\mathcal{A}_e$ is constructed as follows.

- If $e = \emptyset$, then $\mathcal{A}_e = (\{q_{\emptyset,0}\}, \Sigma, \delta, \tau, q_{\emptyset,0}, (\emptyset, \emptyset))$, where $\delta(q_{\emptyset,0}, a) = ()$ for every $a \in \Sigma$, $\tau(q_{\emptyset,0}) = ((); ())$.

– If $e = \varepsilon$, then $\mathcal{A}_e = (\{q_{\varepsilon,0}, f_{\varepsilon,0}\}, \Sigma, \delta, \tau, q_{\varepsilon,0}, (\{f_{\varepsilon,0}\}, \emptyset))$, where $\delta(q_{\varepsilon,0}, a) = \delta(f_{\varepsilon,0}, a) = ()$ for every $a \in \Sigma$, $\tau(q_{\varepsilon,0}) = ((f_{\varepsilon,0}); ())$, and $\tau(f_{\varepsilon,0}) = ((); ())$.
– If $e = a$, then $\mathcal{A}_e = (\{q_{a,0}, q_{a,1}, f_{a,0}\}, \Sigma, \delta, \tau, q_{a,0}, (\emptyset, \{f_{a,0}\}))$, where $\delta(q_{a,0}, b) = ()$ for every $b \in \Sigma$, $\delta(q_{a,1}, a) = (f_{a,0})$, $\delta(q_{a,1}, b) = ()$ for every $b \in \Sigma \setminus \{a\}$, $\tau(q_{a,0}) = ((q_{a,1}); ())$, $\tau(q_{a,1}) = ((); ())$, and $\tau(f_{a,0}) = ((); ())$.

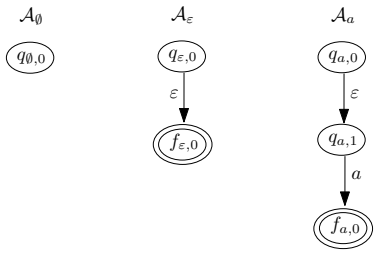

Fig. 7. The PFA $\mathcal{A}_\emptyset$, $\mathcal{A}_\varepsilon$, and $\mathcal{A}_a$

– If $e = (e_1)$, then $\mathcal{A}_e = \mathcal{A}_{e_1}$.
– If $e = [e_1 + e_2]$, let $\mathcal{A}_{e_1} = (Q_{e_1}, \Sigma, \delta_{e_1}, \tau_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$ and $\mathcal{A}_{e_2} = (Q_{e_2}, \Sigma, \delta_{e_2}, \tau_{e_2}, q_{e_2,0}, (F_{e_2,1}, F_{e_2,2}))$, then $\mathcal{A}_e = (Q_{e_1} \cup Q_{e_2} \cup \{q_{e,0}\}, \Sigma, \delta_e, \tau_e, q_{e,0}, (F_{e_1,1} \cup F_{e_2,1}, F_{e_1,2} \cup F_{e_2,2}))$ (see Fig. 8), where
  - $q_{e,0} \notin Q_{e_1} \cup Q_{e_2}$,
  - $\delta_e(q, a) = \delta_{e_i}(q, a)$ for every $i \in \{1, 2\}$, $q \in Q_{e_i}$ and $a \in \Sigma$, $\delta_e(q_{e,0}, a) = ()$ for every $a \in \Sigma$,
  - $\tau_e(q) = \tau_{e_i}(q)$ for every $q \in Q_{e_i}$ $(i = 1, 2)$, $\tau_e(q_{e,0}) = ((q_{e_1,0}, q_{e_2,0}); ())$.

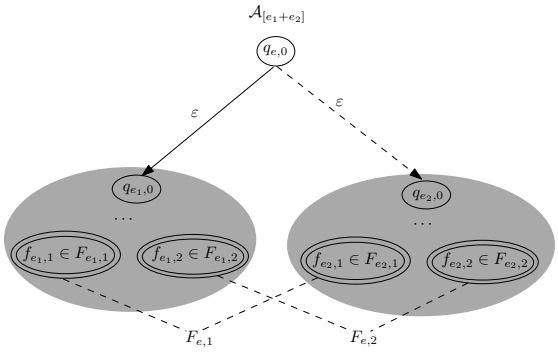

Fig. 8. The PFA $\mathcal{A}_{[e_1+e_2]}$

– If $e = [e_1^?]$, suppose $\mathcal{A}_{e_1} = (Q_{e_1}, \Sigma, \delta_{e_1}, \tau_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$, then $\mathcal{A}_e = (Q_{e_1} \cup \{q_\varepsilon\}, \Sigma, \delta_e, \tau_e, q_{e,0}, (\{q_\varepsilon\}, F_{e_1,2}))$ (see Fig. 9), where
  - $q_{e,0} \notin Q_{e_1}$

- $\delta_e(q,a) = \delta_{e_1}(q,a)$ for every $q \in Q_{e_1}$ and $a \in \Sigma$, $\delta_e(q_{e,0},a) = ()$ and $\delta_e(q_\varepsilon,a) = ()$ for every $a \in \Sigma$,
- $\tau_e(q) = \tau_{e_1}(q)$ for every $q \in Q_{e_1}$, $\tau_e(q_{e,0}) = ((q_{e_1,0}, q_\varepsilon); ())$.

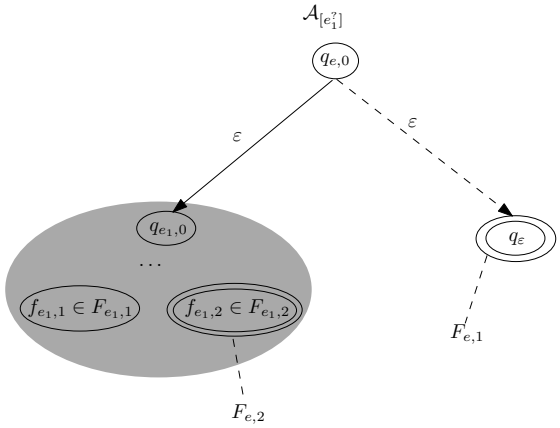

Fig. 9. The PFA $\mathcal{A}_{[e_1^?]}$

– If $e = [e_1^{??}]$, suppose $\mathcal{A}_{e_1} = (Q_{e_1}, \Sigma, \delta_{e_1}, \tau_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$, then $\mathcal{A}_e = (Q_{e_1} \cup \{q_\varepsilon\}, \Sigma, \delta_e, \tau_e, q_{e,0}, (\{q_\varepsilon\}, F_{e_1,2}))$ (see Fig. 10), where
  - $q_{e,0} \notin Q_{e_1}$
  - $\delta_e(q,a) = \delta_{e_1}(q,a)$ for every $q \in Q_{e_1}$ and $a \in \Sigma$, $\delta_e(q_{e,0},a) = ()$ and $\delta_e(q_\varepsilon,a) = ()$ for every $a \in \Sigma$,
  - $\tau_e(q) = \tau_{e_1}(q)$ for every $q \in Q_{e_1}$, $\tau_e(q_{e,0}) = ((q_\varepsilon, q_{e_1,0}); ())$.

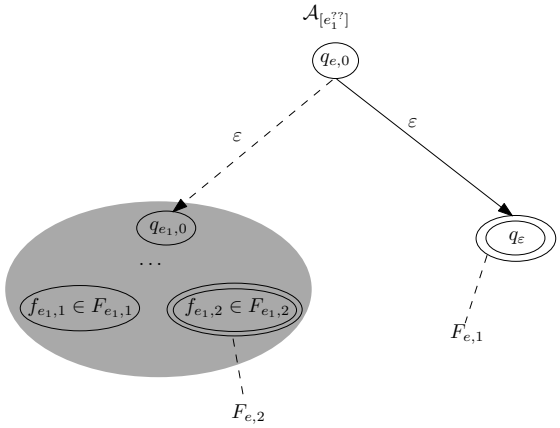

Fig. 10. The PFA $\mathcal{A}_{[e_1^{??}]}$

– If $e = [e_1 \cdot e_2]$, let $\mathcal{A}_{e_1} = (Q_{e_1}, \Sigma, \delta_{e_1}, \tau_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$, $\mathcal{A}_{e_2} = (Q_{e_2}, \Sigma, \delta_{e_2}, \tau_{e_2}, q_{e_2,0}, (F_{e_2,1}, F_{e_2,2}))$, and $\mathcal{A}'_{e_2} = (Q'_{e_2}, \Sigma, \delta'_{e_2}, \tau'_{e_2}, q'_{e_2,0}, (F'_{e_2,1}, F'_{e_2,2}))$ be a fresh copy of $\mathcal{A}_{e_2}$, then $\mathcal{A}_e = (Q_{e_1} \cup Q_{e_2} \cup Q'_{e_2}, \Sigma, \delta_e, \tau_e, q_{e_1,0}, (F_{e_2,1}, F_{e_2,2} \cup F'_{e_2,1} \cup F'_{e_2,2}))$ (see Fig. 11), where
  - for every $i \in \{1, 2\}$, $q \in Q_{e_i}$ and $a \in \Sigma$, $\delta_e(q, a) = \delta_{e_i}(q, a)$,
  - for every $q' \in Q'_{e_2}$ and $a \in \Sigma$, $\delta_e(q', a) = \delta'_{e_2}(q', a)$,
  - for every $q \in Q_{e_2}$, $\tau_e(q) = \tau_{e_2}(q)$ and $\tau_e(q') = \tau'_{e_2}(q')$,
  - for every $q \in Q_{e_1} \setminus (F_{e_1,1} \cup F_{e_1,2})$, $\tau_e(q) = \tau_{e_1}(q)$, for every $f_{e_1,1} \in F_{e_1,1}$, $\tau_e(f_{e_1,1}) = ((q_{e_2,0}); ())$, and for every $f_{e_1,2} \in F_{e_1,2}$, $\tau_e(f_{e_1,2}) = ((q'_{e_2,0}), ())$.

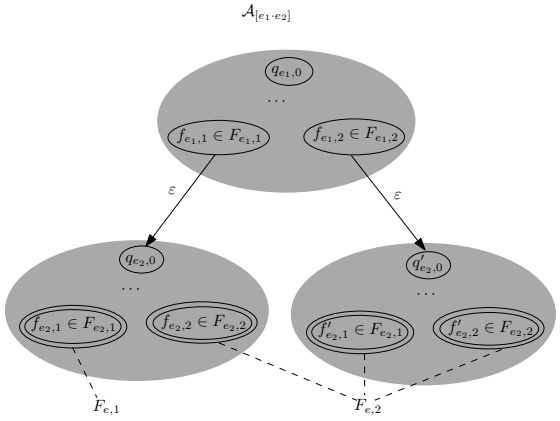

Fig. 11. The PFA $\mathcal{A}_{[e_1 \cdot e_2]}$

– If $e = [e_1^*]$, let $\mathcal{A}_{e_1} = (Q_{e_1}, \Sigma, \delta_{e_1}, \tau_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$, then $\mathcal{A}_e = (Q_{e_1} \cup \{q_{e,0}, f_{e,0}, f_{e,1}\}, \Sigma, \delta_e, \tau_e, q_{e,0}, (\{f_{e,0}\}, \{f_{e,1}\}))$, where
  - $q_{e,0}, f_{e,0} \notin Q_{e_1}$,
  - for every $q \in Q_{e_1}$ and $a \in \Sigma$, $\delta_e(q, a) = \delta_{e_1}(q, a)$,
  - for every $q \in Q_{e_1} \setminus (F_{e_1,1} \cup F_{e_1,2})$, $\tau_e(q) = \tau_{e_1}(q)$, moreover, $\tau_e(q_{e,0}) = ((q_{e_1,0}, f_{e,0}); ())$, $\tau_e(q) = ((q_{e_1,0}); ())$ for every $q \in F_{e_1,1}$, $\tau_e(q) = ((q_{e_1,0}, f_{e,1}); ())$ for every $q \in F_{e_1,2}$, $\tau_e(f_{e,0}) = \tau_e(f_{e,1}) = ((); ())$. (Intuitively, the ε-transitions from $q_{e,0}$ to $q_{e_1,0}$ and $f_{e,0}$, from each $q \in F_{e_1,1}$ to $q_{e_1,0}$, and from $q \in F_{e_1,2}$ to $q_{e_1,0}$ and $f_{e,1}$ respectively are added, moreover, the ε-transition from $q_{e,0}$ to $f_{e,0}$ and from $q \in F_{e_1,2}$ to $f_{e,1}$ are of the lowest priority.)

– If $e = [e_1^+]$, then $\mathcal{A}_e$ is constructed as $\mathcal{A}_{[e_1 \cdot [e_1^*]]}$.

– If $e = [e_1^{*?}]$, let $\mathcal{A}_{e_1} = (Q_{e_1}, \Sigma, \delta_{e_1}, \tau_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$, then $\mathcal{A}_e = (Q_{e_1} \cup \{q_{e,0}, f_{e,0}, f_{e,1}\}, \Sigma, \delta_e, \tau_e, q_{e,0}, (\{f_{e,0}\}, \{f_{e,1}\}))$ (see Fig. 12), where
  - $q_{e,0}, f_{e,0} \notin Q_{e_1}$,
  - for every $q \in Q_{e_1}$ and $a \in \Sigma$, $\delta_e(q, a) = \delta_{e_1}(q, a)$,

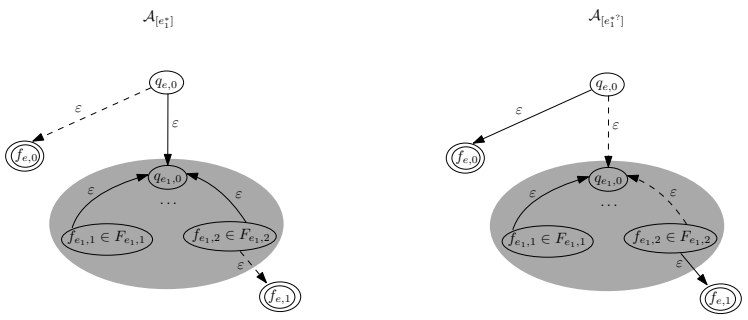

Fig. 12. The PFA $\mathcal{A}_{[e_1^*]}$ and $\mathcal{A}_{[e_1^{*?}]}$

- for every $q \in Q_{e_1} \setminus (F_{e_1,1} \cup F_{e_1,2})$, $\tau_e(q) = \tau_{e_1}(q)$, moreover, $\tau_e(q_{e,0}) = ((f_{e,0}, q_{e_1,0}); ())$, $\tau_e(q) = ((q_{e_1,0}); ())$ for every $q \in F_{e_1,1}$, $\tau_e(q) = ((f_{e,1}, q_{e_1,0}); ())$ for every $q \in F_{e_1,2}$, $\tau_e(f_{e,0}) = \tau_e(f_{e,1}) = ((); ())$. (Intuitively, the ε-transitions from $q_{e,0}$ to $f_{e,0}$ and $q_{e_1,0}$, from each $q \in F_{e_1,1}$ to $q_{e_1,0}$, and from each $q \in F_{e_1,2}$ to $f_{e,1}$ and $q_{e_1,0}$ respectively are added, moreover, the ε-transition from $q_{e,0}$ to $q_{e_1,0}$ and from $q \in F_{e_1,2}$ to $q_{e_1,0}$ are of the lowest priority.)

– If $e = [e_1^{+?}]$, then $\mathcal{A}_e$ is constructed as $\mathcal{A}_{[e_1 \cdot [e_1^{*?}]]}$.
– If $e = [e_1^{\{m_1,m_2\}}]$, then $\mathcal{A}_e$ is constructed as the concatenation of $\mathcal{A}_{e_1^{m_1}}$ and $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}}}$, where $\mathcal{A}_{e_1^{m_1}}$ is the PFA corresponding to consecutive concatenations of m_1 copies of e_1, and $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}}}$ is illustrated in Fig. 13, which consists of $m_2 - m_1$ copies of $\mathcal{A}_{e_1}$, say $(\mathcal{A}_{e_1}^{(i)})_{i \in [m_2-m_1]}$, as well as the ε-transition from $q_{e_1,0}^{(1)}$ to a fresh state f'_0 (of the lowest priority), and the ε-transitions from each $f_{e_1,2}^{(i)} \in F_{e_1,2}^{(i)}$ to $q_{e_1,0}^{(i+1)}$ (of the highest priority), and a fresh state f'_1 (of the lowest priority). The accepting states of $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}}}$ are $(\{f'_0\}, \{f'_1\})$. (Intuitively, each $\mathcal{A}_{e_1}^{(i)}$ accepts only nonempty strings, thus $f_{e_1,1}^{(i)} \in F_{e_1,1}^{(i)}$ contains no outgoing transitions in $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}}}$.)
– If $e = [e_1^{\{m_1,m_2\}^?}]$, then $\mathcal{A}_e$ is constructed as the concatenation of $\mathcal{A}_{e_1^{m_1}}$ and $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}^?}}$, where $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}^?}}$ is illustrated in Fig. 14, which is the same as $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}}}$ in Fig. 13, except that the priorities of ε-transition from $q_{e_1,0}^{(1)}$ to f'_0 has the highest priority and the priorities of ε-transitions from each $f_{e_1,2}^{(i)} \in F_{e_1,2}^{(i)}$ to f'_1 are reversed.

□

Lemma 1. *For each string function* $f = \mathsf{extract}_{i,e}$, $\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}$, *or* $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}$, *a PSST* $\mathcal{T}_f$ *can be constructed such that*

$$\mathcal{R}_f = \{(w, w') \mid w' = f(w)\}.$$

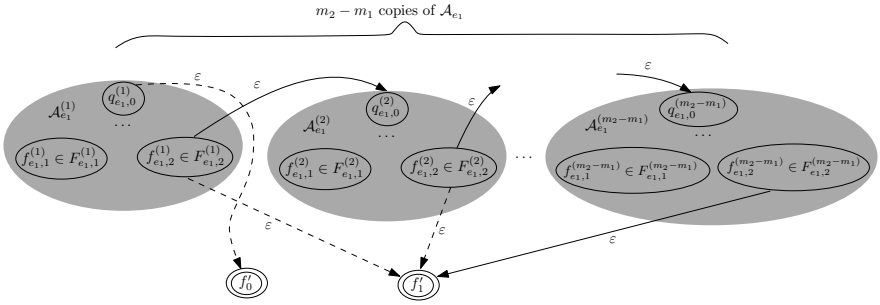

Fig. 13. The PFA $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}}}$

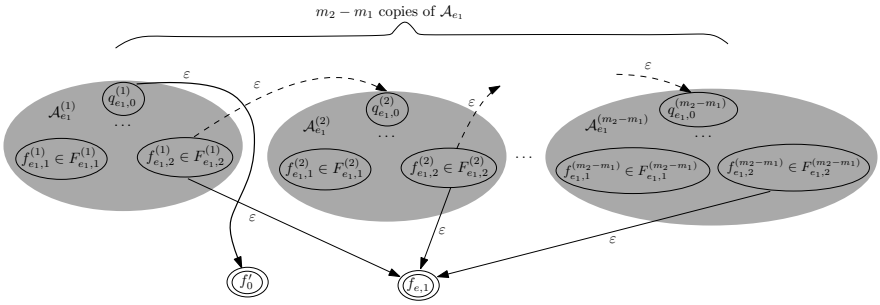

Fig. 14. The PFA $\mathcal{A}'_{e_1^{\{1,m_2-m_1\}?}}$

Proof. Let e' be the subexpression corresponding to the i-th capturing group of e. In particular, if $i = 0$, then $e' = e$. Suppose $\mathcal{A}_e = (Q, \Sigma, \delta, \tau, q_0, f_0)$, and $\mathsf{Sub}_{e'}[\mathcal{A}_e]$ is any isomorphic copy of $\mathcal{A}_{e'}$ in $\mathcal{A}_e$.

Intuitively, $\mathcal{T}_{\mathsf{extract}_{i,e}}$ (see Fig. 15)

- uses a string variable x to store the value of the ith capturing group,
- initially assigns null to x to denote the fact that the capturing group is not matched yet,
- then simulates $\mathcal{A}_e$ and stores letters into x when applying the transitions in $\mathsf{Sub}_{e'}[\mathcal{A}_e]$,
- finally outputs the value of x when $\mathcal{A}_e$ accepts.

Formally, $\mathcal{T}_{\mathsf{extract}_{i,e}} = (Q \cup \{q'_0\}, \Sigma, X, \delta', \tau', E', q'_0, F')$, where

- $q'_0 \notin Q$,
- $X = \{x\}$,
- $F'(f_0) = x$ and $F'(p)$ is undefined for all the other states $p \in Q \cup \{q'_0\}$,
- δ' and τ' are defined as follows,
  - $\tau'(q'_0) = ((q_0); ())$,

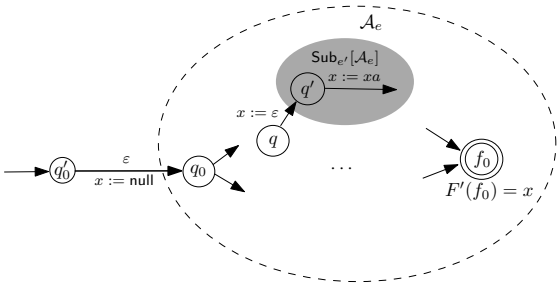

Fig. 15. The PSST $\mathcal{T}_{\mathsf{extract}_{i,e}}$

- δ' includes all the transitions in δ,
- τ' includes all the transitions in τ,

– E' is defined as follows,
- $E'(q'_0, \varepsilon, q_0)(x) = \mathsf{null}$,
- for each transition (q, a, q') in $\mathsf{Sub}_{e'}[\mathcal{A}_e]$ such that q is the inital state of $\mathsf{Sub}_{e'}[\mathcal{A}_e]$(Note in this case, the construction in Proposition 2 ensures a equals ε), $E'(q, a, q')(x) = \varepsilon$,
- for each other transition (q, a, q') in $\mathsf{Sub}_{e'}[\mathcal{A}_e]$, $E'(q, a, q')(x) = xa$,
- for all the other transitions t of $\mathcal{A}_e$, $E'(t)(x) = x$.

Next, we show how to construct the PSST for $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}$.

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, and $\mathsf{Sub}_{e'_{i_j}}[\mathcal{A}_{\mathsf{pat}}]$ be any isomorphic copy of $\mathcal{A}_{e'_{i_j}}$ in $\mathcal{A}_{\mathsf{pat}}$.

Suppose $\mathcal{A}_{\mathsf{pat}} = (Q, \Sigma, \delta, \tau, q_0, f_0)$. Then $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ is obtained from $\mathcal{A}_{\mathsf{pat}}$ by adding a fresh state q'_0 such that (see Fig. 16)

– $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ 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 $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ stays at q'_0, it keeps appending the current letter to the end of x_0, which stores the output of $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$,
– starting from q_0, $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ 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, $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ goes from f_0 to q'_0 via an ε-transition, appends the replacement string, which is $\mathsf{rep}[x_1/\$_{i_1}, \cdots, x_k/\$_{i_k}]$, to the end of x_0, resets the values of $x_1, \cdots, x_k$ to ε and keeps searching for the next match of pat.

Formally, $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}} = (Q \cup \{q'_0\}, \Sigma, X, \delta', \tau', E, q'_0, F)$ where

– $q'_0 \notin Q$,

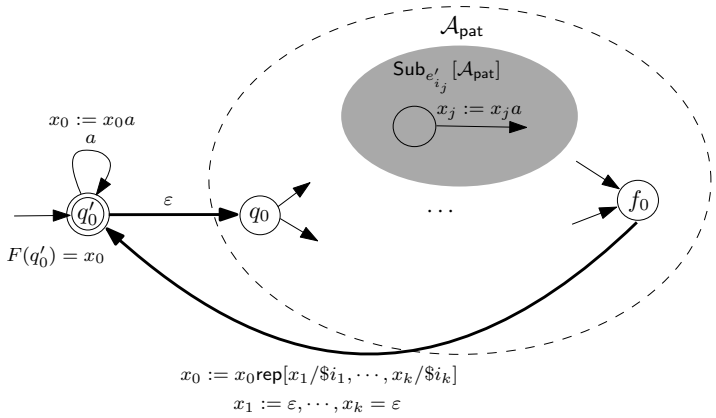

Fig. 16. The PSST $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$

- $X = \{x_0, x_1, \cdots, x_k\}$,
- $F(q'_0) = x_0$, and $F(q')$ is undefined for every $q' \in Q$,
- δ' and τ' are obtained from δ and τ as follows,
  - $\delta'(q'_0, a) = (q'_0)$ for every $a \in \Sigma$, and $\tau'(q'_0) = ((q_0); ())$,
  - for every $q \in Q \setminus \{f_0\}$ and $a \in \Sigma$, $\delta'(q, a) = \delta(q, a)$ and $\tau'(q) = \tau(q)$,
  - $\delta'(f_0, a) = ()$ for every $a \in \Sigma$ and $\tau'(f_0) = ((q'_0); ())$,
- E is defined as follows,
  - for every transition (q, a, q') with $a \in \Sigma^\varepsilon$ in $\mathcal{A}_{\mathsf{pat}}$, $E(q, a, q')(x_0) = x_0$,
  - for every transition (q, a, q') with $a \in \Sigma^\varepsilon$ and every $j \in [k]$, if (q, a, q') occurs in $\mathsf{Sub}_{e'_{i_j}}[\mathcal{A}_{\mathsf{pat}}]$, then $E(q, a, q')(x_j) = x_j a$, otherwise, $E(q, a, q')(x_j) = x_j$,
  - for every $a \in \Sigma$ and $j \in [k]$, $E(q'_0, a, q'_0)(x_0) = x_0 a$ and $E(q'_0, a, q'_0)(x_j) = x_j$,
  - $E(q'_0, \varepsilon, q_0)(x_j) = x_j$ for every $j \in [k] \cup \{0\}$,
  - $E(f_0, \varepsilon, q'_0)(x_0) = x_0\mathsf{rep}[x_1/\$i_1, \ldots, x_k/\$i_k]$, moreover, for every $j \in [k]$, $E(f_0, \varepsilon, q'_0)(x_j) = \varepsilon$, where $\mathsf{rep}[x_1/\$i_1, \ldots, x_k/\$i_k]$ denotes the string term obtained from rep by replacing every occurrence of $\$i_1, \cdots, \$i_k$ with $x_1, \cdots, x_k$ respectively.

□

The construction of the PSST for $\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}$ is similar and illustrated in Fig. 17. The details are omitted.

A.4 Construction of an FA $\mathcal{B}$ for $\mathcal{R}^{-1}_{\mathcal{T}}(\mathscr{L}(\mathcal{A}))$

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

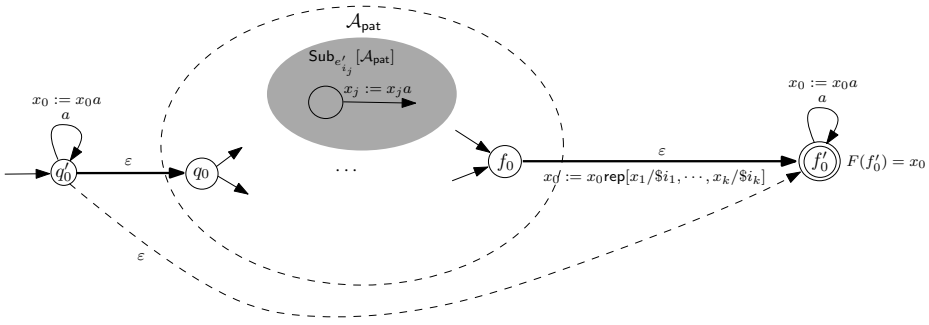

Fig. 17. The PSST $\mathcal{T}_{\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}}$

To illustrate the intuition of the proof of Lemma 2, let us start with the following natural idea of firstly constructing a PFA $\mathcal{B}$ for the pre-image: $\mathcal{B}$ simulates a 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 $\mathcal{T}$. 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 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 7. Let $\mathcal{T}_{\mathtt{extract}_{\mathtt{decimalReg},1}}$ be the PSST in Fig. 5 and $\mathcal{A}$ be the FA corresponding to the regular expression $\{1, \cdots, 9\}^*$, specifically, $\mathcal{A} = (\{p_0\}, \{0, \cdots, 9\}, \delta_A, p_0, \{p_0\})$, where $\delta_A = \{(q_0, \ell, q_0) \mid \ell = 1, \cdots, 9\}$.

Let us consider $w = 10$. The accepting run of $\mathcal{T}_{\mathtt{extract}_{\mathtt{decimalReg},1}}$ on w is $q_0 \xrightarrow[x_1 := x_1 1]{1} q_1 \xrightarrow[x_1 := x_1 0]{0} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{\varepsilon} q_3 \xrightarrow{\varepsilon} q_4 \xrightarrow{\varepsilon} q_5 \xrightarrow{\varepsilon} q_6$, producing an output $10 \notin \mathscr{L}(\mathcal{A})$. Therefore, $10 \notin \mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}))$. 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 $(q_0, \{(p_0, p_0)\}) \xrightarrow{1} (q_1, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_2, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_3, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_4, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_5, \{(p_0, p_0)\}) \xrightarrow{0} (q_5, \{(p_0, p_0)\}) \xrightarrow{\varepsilon} (q_6, \{(p_0, p_0)\})$, where $\{(p_0, p_0)\}$ is the $\mathcal{A}$-abstraction of the strings ε and 1. On the other hand, the run of $\mathcal{B}'$ corresponding to the accepting run of $\mathcal{T}$ on w, i.e. $(q_0, \{(p_0, p_0)\}) \xrightarrow{1} (q_1, \{(p_0, p_0)\}) \xrightarrow{0} (q_1, \emptyset) \xrightarrow{\varepsilon} (q_2, \emptyset) \xrightarrow{\varepsilon} (q_3, \emptyset) \xrightarrow{\varepsilon} (q_4, \emptyset) \xrightarrow{\varepsilon} (q_5, \emptyset) \xrightarrow{\varepsilon} (q_6, \emptyset)$, is not accepting, where $\{(p_0, p_0)\}$ is the $\mathcal{A}$-abstraction of ε as well as 1, and $\emptyset$ is the $\mathcal{A}$-abstraction of 10.

While the aforementioned natural idea does not work, we choose to construct an FA $\mathcal{B}$ that simulates the *accepting* 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 accepting run of $\mathcal{T}$, it is necessary to record all the states accessible through the runs of higher priorities to ensure the current run is indeed the accepting run of $\mathcal{T}$ (of 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 $\mathcal{T}$.

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 $S \subseteq Q_T$. For a state (q, ρ, Λ, S), our intention for S is that the states in it are those that can be reached in the runs of higher priorities than the current run, by reading the same sequence of letters and applying the ε-transitions as many as possible. Note that when recording in S all the states accessible through the runs of higher priorities, we do not take the non-repetition of ε-transitions into consideration 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 an a-transition of $\mathcal{T}$ (where $a \in \Sigma$) at a state (q, ρ, Λ, S), suppose $\delta_T(q, a) = (q_1, \cdots, q_m)$ and $\tau_T(q) = (P_1, P_2)$, then $\mathcal{B}$ nondeterministically chooses q_i and goes to the state $(q_i, \rho', \emptyset, S')$, where

- ρ' is obtained from ρ and $E_T(q, \sigma, q_i)$,
- Λ is reset to $\emptyset$,
- all the states obtained from S by applying an a transition should be *saturated by ε-transitions* and put into S', more precisely, all the states reachable from S by first applying an a-transition, then a sequence of ε-transitions, should be put into S',
- moreover, all the states obtained from $q_1, \cdots, q_{i-1}$ (which are of higher priorities than q_i) by saturating with ε-transitions should be put into S',
- finally, all the states obtained from those in $P_1' = \{q' \in P_1 \mid (q, q') \notin \Lambda\}$ (which are of higher priorities than q_i) by saturating with non-Λ ε-transitions first (i.e. the ε-transitions that do not belong to Λ), and applying an a-transition next, finally saturating with ε-transitions again, should be put into S', (note that according to the semantics of PSST, the ε-transitions in Λ should be avoided when defining P_1' and saturating the states in P_1' with ε-transitions).

The above construction does not utilize the so-called *copyless* property (i.e. for each transition t and each variable x, x appears at most once on the right-hand side of the assignment for t) [AC10,AD11], thus it works for general, or *copyful*, PSSTs [FR17].

For the formal construction of $\mathcal{B}$, we need some additional 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

$\big(\tau_T^{(ip)}\big)^*$ to denote the ε-closure of $\mathcal{T}$, namely, $\big(\tau_T^{(ip)}\big)^*(S) = \bigcup_{n\in\mathbb{N}} \big(\tau_T^{(ip)}\big)^n(S)$, where $\big(\tau_T^{(ip)}\big)^0(S) = S$, and for $n \in \mathbb{N}$, $\big(\tau_T^{(ip)}\big)^{n+1}(S) = \tau_T^{(ip)}\big(\big(\tau_T^{(ip)}\big)^n(S)\big)$.

- For $S \subseteq Q_T$ and $\Lambda \subseteq \mathcal{E}(\tau_T)$, we use $\big(\tau_T^{(ip)} \backslash \Lambda\big)^*(S)$ 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, and $\circ$ represents the composition of binary relations.

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 $((q, \rho, \Lambda, S), a, (q_i, \rho', \Lambda', S'))$ such that
    - $a \in \Sigma$,
    - $\delta_T(q, a) = (q_1, \ldots, q_i, \ldots, q_m)$,
    - $s = E((q, a, q_i))$,
    - $\rho' = s(\rho)$,
    - $\Lambda' = \emptyset$, (Intuitively, Λ is reset.)
    - let $\tau_T(q) = (P_1, P_2)$, then $S' = \big(\tau_T^{(ip)}\big)^*\big(\{q_1, \ldots, q_{i-1}\} \cup \delta_T^{(ip)}\big(S \cup \big(\tau_T^{(ip)} \setminus \Lambda\big)^*(P_1'), a\big)\big)$, where $P_1' = \{q' \in P_1 \mid (q, q') \notin \Lambda\}$;
  - the tuples $((q, \rho, \Lambda, S), \varepsilon, (q_i, \rho', \Lambda', S'))$ such that
    - $\tau_T(q) = ((q_1, \ldots, q_i, \ldots, q_m); \cdots)$,
    - $(q, q_i) \notin \Lambda$,
    - $s = E(q, \varepsilon, q_i)$,
    - $\rho' = s(\rho)$,
    - $\Lambda' = \Lambda \cup \{(q, q_i)\}$,
    - $S' = S \cup \big(\tau_T^{(ip)} \backslash \Lambda\big)^*(\{q_j \mid j \in [i-1], (q, q_j) \notin \Lambda\})$;
  - the tuples $((q, \rho, \Lambda, S), \varepsilon, (q_i, \rho', \Lambda', S'))$ such that
    - $\tau_T(q) = ((q_1', \ldots, q_n'); (q_1, \ldots, q_i, \ldots, q_m))$,
    - $(q, q_i) \notin \Lambda$,
    - $s = E(q, \varepsilon, q_i)$,
    - $\rho' = s(\rho)$,
    - $\Lambda' = \Lambda \cup \{(q, q_i)\}$,
    - $S' = S \cup \{q\} \cup \big(\tau_T^{(ip)} \backslash \Lambda\big)^*\big(\{q_j' \mid j \in [n], (q, q_j') \notin \Lambda\} \cup \{q_j \mid j \in [i-1], (q, q_j) \notin \Lambda\}\big)$. (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 every $q' \in S$, $F_T(q')$ 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, $\theta_i = \rho(b_i)$.

Example 8. Let us continue Example 7. Suppose $\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},1}} = (Q_T, \Sigma, X, \delta_T, \tau_T, E_T, q_{0,T}, F_T)$. Then the FA defining $\mathcal{R}^{-1}_{\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},1}}}(\mathscr{L}(\mathcal{A}))$ constructed by using the aforementioned procedure is illustrated in Fig. 18, where the final states are those doubly boxed states, moreover, the states reachable from the state $(q_2, \emptyset, \{(q_1, q_2)\}, \{q_1\})$ are omitted because no final states are unreachable from those states, which are therefore redundant. Let us exemplify the construction by considering the state $(q_5, \{(p_0, p_0)\}, \{(q_1, q_2), (q_2, q_3), (q_3, q_4), (q_4, q_5)\}, \{q_1, q_3\})$. For each letter $\ell \in \{0, \cdots, 9\}$, the state $(q_5, \{(p_0, p_0)\}, \emptyset, \{q_1, q_2, q_3, q_4, q_5, q_6\})$ is reached from it, since $\delta_T(q_5, \ell) = (q_5)$, $\delta_T(q_1, \ell) = (q_1)$ and $(\tau_T^{(ip)})^*(\{q_1\}) = \{q_1, q_2, q_3, q_4, q_5, q_6\}$. The state $(q_6, \{(p_0, p_0)\}, \{(q_5, q_6)\}, \{q_1, q_2, q_3, q_4, q_5, q_6\})$ is not a final state since q_6 is in $\{q_1, q_2, q_3, q_4, q_5, q_6\}$ and $F_T(q_6)$ is defined.

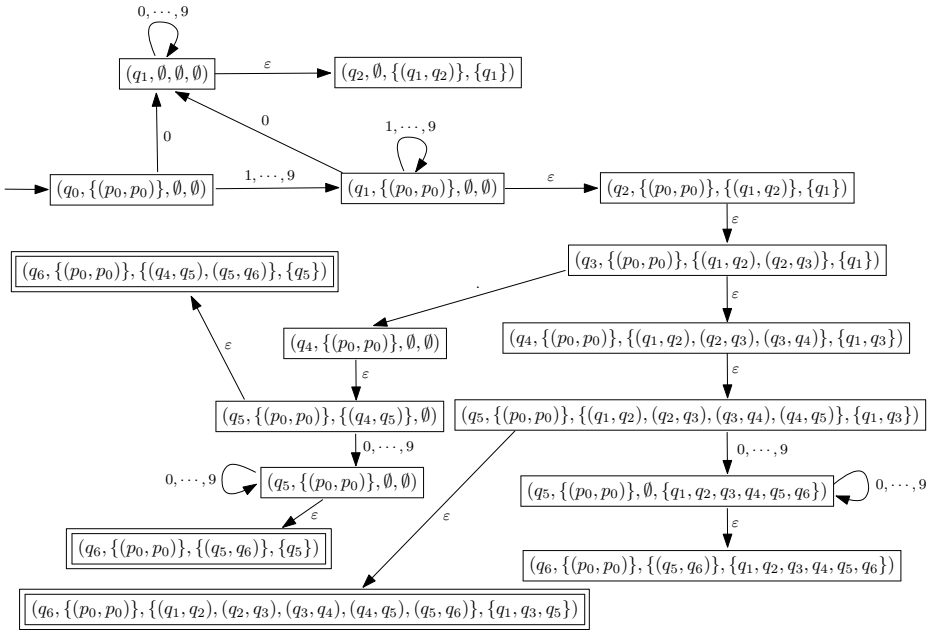

Fig. 18. The FA defining $\mathcal{R}^{-1}_{\mathcal{T}_{\texttt{extract}_{\texttt{decimalReg},1}}}(\mathscr{L}(\mathcal{A}))$

Remark 4. The computation of the pre-image of an FA $\mathcal{A}$ under a PSST $\mathcal{T}$ can be understood from a different angle: At first, $\mathcal{T}$ can be turned into an equivalent a streaming string transducer $\mathcal{T}'$, then the pre-image of $\mathcal{A}$ under $\mathcal{T}'$ is computed. Therefore, from the viewpoint of expressibility, PSSTs and SSTs are equivalent.

Nevertheless, PSSTs are exponentially more succinct than SSTs, moreover, the priorities in PSSTs are very convenient for modeling the greedy/lazy semantics of Kleene star.