An Efficient String Solver for String Constraints with Complex Regex-Counting and String-Length

Denghang Hu, Zhilin Wu

State Key Laboratory of Computer Science,Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract

Regular expressions (regex for short) and string-length function are widely used in string-manipulating programs. Counting is a frequently used feature in regexes that counts the number of matchings of sub-patterns. The state-of-the-art string solvers are incapable of solving string constraints with regex-counting and string-length eiciently, especially when the counting and length bounds are large. In this work, we propose an automata-theoretic approach for solving such class of string constraints. The main idea is to symbolically model the counting operators by registers in automata instead of unfolding them explicitly, thus alleviating the state explosion problem. Moreover, the string-length function is modeled by a register as well. As a result, the satisfiability of string constraints with regex-counting and string-length is reduced to the satisfiability of linear integer arithmetic, which the off-the-shelf SMT solvers can then solve. To improve the performance further, we also propose techniques to reduce the sizes of automata. We implement the algorithms and validate our approach on 48,843 benchmark instances. The experimental results show that our approach can solve more instances than the state-of-the-art solvers, at a comparable or faster speed, especially when the counting and length bounds are large.

1. Introduction

The string data type plays a crucial role in modern programming languages such as JavaScript, Python, Java, and PHP. String manipulations are error-prone and could even give rise to severe security vulnerabilities (e.g., cross-site scripting, aka XSS). One powerful method for identifying such bugs

is *symbolic execution*, which is possibly in combination with dynamic analysis. It analyses symbolic paths in a program by viewing them as constraints checked by constraint solvers. Symbolic execution of string manipulating programs has motivated the highly active research area of *string constraint solving*, resulting in the development of numerous string solvers in the last decade, e.g., Z3seq [1], CVC4/5 [2, 3], Z3str/2/3/4 [4, 5, 6, 7], Z3str3RE [8], Z3-Trau [9][10], OSTRICH [11], Slent [12], among many others.

Regular expressions (regex for short) and the string-length function are widely used in string-manipulating programs. According to the statistics from [13, 14, 15], regexes are used in about 3040% of Java, JavaScript, and Python software projects. Moreover, string-length occupies 78% of the occurrences of string operations in 18 Javascript applications, according to the statistics from [16]. As a result, most of the aforementioned string constraint solvers support both regexes and string-length function. Moreover, specialized algorithms have been proposed to solve such string constraints efficiently (see e.g. [17, 8]).

Counting (aka repetition) is a convenient feature in regexes that counts the number of matchings of sub-patterns. For instance, $a^{\{2,4\}}$ specifies that a occurs at least twice and at most four times, and $a^{\{2,\infty\}}$ specifies that a occurs at least twice. Note that the Kleene star and the Kleene plus operator are special cases of counting. For instance, a^* is equivalent to $a^{\{0,\infty\}}$ and a^+ is equivalent to $a^{\{1,\infty\}}$. Counting is a frequently used feature of regexes. According to the statistics from [13], Kleene star/plus occur in more than 70% of 1,204 Python projects, while other forms of counting occur in more than 20% of them. Therefore, an efficient analysis of string manipulating programs requires efficient solvers for string constraints containing regexes with counting[1] and string-length function at least.

Nevertheless, the aforementioned state-of-the-art string constraint solvers still suffer from such string constraints, especially when the counting and length bounds are large. For instance, none of the string solvers CVC5, Z3seq, Z3-Trau, Z3str3, Z3str3RE, and OSTRICH is capable of solving the following constraint within 120 seconds,

$$x \in (\Sigma \setminus a)^{\{1,60\}}(\Sigma \setminus b)^{\{1,60\}}(\Sigma \setminus c)^{\{0,60\}} \wedge x \in \Sigma^* c^+ \wedge |x| > 120. \qquad (1)$$

[1]In the rest of this paper, for clarity, we use counting to denote expressions of the form $e^{\{m,n\}}$ and $e^{\{m,\infty\}}$, but not e^* or e^+.

Intuitively, the constraint in (1) specifies that x is a concatenation of three strings x_1, x_2, x_3 where a (resp. b, c) does not occur in x_1 (resp. x_2, x_3), moreover, x ends with a nonempty sequence of c's, and the length of x is greater than 120. It is easy to observe that this constraint is unsatisfiable since on the one hand, $|x| > 120$ and the counting upper bound 60 in both $(\Sigma \setminus a)^{\{1,60\}}$ and $(\Sigma \setminus b)^{\{1,60\}}$ imply that x must end with some character from $\Sigma \setminus c$, that is, a character different from c, and on the other hand, $x \in \Sigma^* c^+$ requires that x has to end with c.

A typical way for string constraint solvers to deal with regular expressions with counting is to unfold them into those *without* counting using the concatenation operator. For instance, $a^{\{1,4\}}$ is unfolded into $a(\varepsilon + a + aa + aaa)$ and $a^{\{2,\infty\}}$ is unfolded into aaa^*. Since the unfolding incurs an exponential blow-up on the sizes of constraints (assuming that the counting in string constraints are encoded in binary), the unfolding penalizes the performance of the solvers, especially when the length bounds are also available.

Contribution. In this work, we focus on the class of string constraints with regular membership and string-length function, where the counting operators may occur (called RECL for brevity). We make the following contributions.

- We propose an automata-theoretical approach for solving RECL constraints. Our main idea is to represent the counting operators by cost registers in cost-enriched finite automata (CEFA, see Section 5 for the definition), instead of unfolding them explicitly. The string-length function is modeled by cost registers as well. The satisfiability of RECL constraints is reduced to the nonemptiness problem of CEFA w.r.t. a linear integer arithmetic (LIA) formula. According to the results from [18], an LIA formula can be computed to represent the potential values of registers in CEFA. Thus, the nonemptiness of CEFA w.r.t. LIA formulas can be reduced to the satisfiability of LIA formulas, which is then tackled by off-the-shelf SMT solvers.

- We propose techniques to reduce the sizes (i.e. the number of states and transitions) of CEFA, in order to achieve better performance.

- Combined with the size-reduction techniques mentioned above, the register representation of regex-counting and string-length in CEFA entails an efficient algorithm for solving RECL constraints. We implement the algorithm on top of OSTRICH, resulting in a string solver called OSTRICH$^{\text{RECL}}$.

- Finally, we utilize a collection of benchmark suites comprising 48,843 instances in total to evaluate the performance of OSTRICH$^{\text{RECL}}$. The experiment results show that OSTRICH$^{\text{RECL}}$ solves the RECL constraints more efficiently than the state-of-the-art string solvers, especially when the counting and length bounds are large (see Figure 1 and Table **??**). For instance, on 1,969 benchmark instances where the counting bounds are greater than or equal to 50 and the string lengths are required to be beyond 200, OSTRICH$^{\text{RECL}}$ solves at least 278 more instances than the other solvers, while spending only half or less time per instance on average.

Related work. We discuss more related work on regexes with counting, string-length function, and automata with registers/counters. Determinism of regexes with counting was investigated in [19, 20]. Real-world regexes in programming languages include features beyond classical regexes, e.g., the greedy/lazy Kleene star, capturing groups, and back references. Real-world regexes have been addressed in symbolic execution of Javascript programs [21] and string constraint solving [22]. Nevertheless, the counting operators are still unfolded explicitly in [22]. The Trau tool focuses on string constraints involving flat regular languages and string-length function and solves them by computing LIA formulas that define the Parikh images of flat regular languages [10]. The Slent tool solves the string constraints involving string-length function by encoding them into so-called length-encoded automata, then utilizing symbolic model checkers to solve their nonemptiness problem [12]. However, neither Trau nor Slent supports counting operators explicitly, in other words, counting operators in regexes should be unfolded before solved by them. Cost registers in CEFAs are different from registers in (symbolic) register automata [23, 24]: In register automata, registers are used to store input values and can only be compared for equality/inequality, while in CEFAs, cost registers are used to store integer-valued costs and can be updated by adding/subtracting integer constants and constrained by the accepting conditions which are LIA formulas. Therefore, cost registers in CEFAs are more like counters in counter automata/machines [25], that is, CEFAs can be obtained from counter machines by removing transition guards and adding accepting conditions. Counting-set automata were proposed in [26, 27] to quickly match a subclass of regexes with counting. Moreover, a variant of nondeterministic counter automata, called bit vector automata, was proposed recently in [28] to enable fast matching of a more

expressive class of regexes with counting. Nevertheless, the nonemptiness problem of these automata was not considered, and it is unclear whether these automata models can be used for solving string constraints with regex-counting and string-length.

Organization. The rest of this paper is structured as follows: Section 2 gives an overview of the approach in this paper. Section 3 introduces the preliminaries. Section 4 presents the syntax and semantics of RECL. Section 5 defines CEFA. Section 6 introduces the algorithm to solve RECL constraints. Section 8 shows the experiment results. Section 9 concludes this paper.

2. Overview

In this section, we utilize the string constraint in Equation (1) to illustrate the approach in our work.

At first, we construct a CEFA for the regular expression $(\Sigma \setminus a)^{\{1,60\}}(\Sigma \setminus b)^{\{1,60\}}(\Sigma \setminus c)^{\{0,60\}}$. Three registers are introduced, say r_1, r_2, r_3, to represent the three counting operators; the nondeterministic finite automaton (NFA) for $(\Sigma\setminus a)^*(\Sigma\setminus b)^*(\Sigma\setminus c)^*$ is constructed; the updates of registers are added to the transitions of the NFA; the counting bounds are specified by the accepting condition $1 \leq r_1 \leq 60 \wedge 1 \leq r_2 \leq 60 \wedge 0 \leq r_3 \leq 60$, resulting in a CEFA $\mathcal{A}_1$ illustrated in Figure 1(a). $r_1{+}{+}$ means that we increment the value of r_1 by one after running the transition. A string w is accepted by $\mathcal{A}_1$ if, when reading the characters in w, $\mathcal{A}_1$ applies the transitions to update the state and the values of registers, reaching a final state q in the end, and the resulting values of the three registers, say v_1, v_2, v_3, satisfy the accepting condition. In addition, we construct other two CEFAs $\mathcal{A}_2$ for $\Sigma^* c^+$ (see Figure 1(b)) and $\mathcal{A}_3$ for string length function (see Figure 1(c)). In $\mathcal{A}_3$, a register r_4 is used to denote the length of strings and the accepting condition is `true` (See Section 6.1 for more details about the construction of CEFA.) Note that we represent the counting operators symbolically by registers instead of unfolding them explicitly.

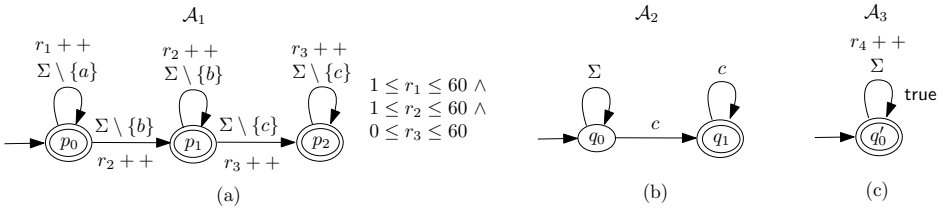

Figure 1: CEFA for $(\Sigma \setminus a)^{\{1,60\}}(\Sigma \setminus b)^{\{1,60\}}(\Sigma \setminus c)^{\{0,60\}}$, $\Sigma^* c^+$, and $|x|$

Next, we construct $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3$, that is, the intersection (aka product) of $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$, as illustrated in Figure 2(a), where the states can not reach the final states are removed. For technical convenience, we also think of the updates of registers in transitions as vectors (u_1, u_2, u_3, u_4), where $u_i \in \mathbb{Z}$ is the update on the register r_i for each $i \in [4]$. For instance, the transitions corresponding to the self-loop around (p_0, q_0, q_0') are thought as $((p_0, q_0, q_0'), a', (p_0, q_0, q_0'), (1, 0, 0, 1))$ with $a' \in \Sigma \setminus \{a\}$, since r_1 and r_4 are incremented by one in these transitions. After considering the updates of registers as vectors, the CEFA is like Figure 2(b).

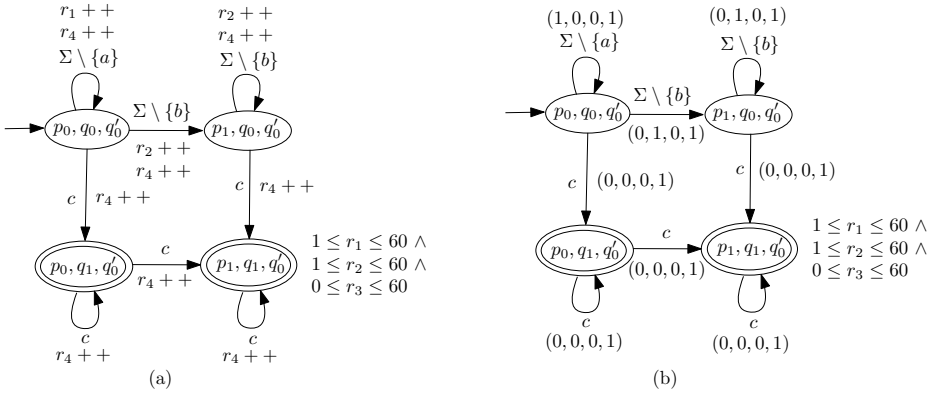

Figure 2: $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3$: Intersection of $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$

Finally, the satisfiability of the original string constraint is reduced to the nonemptiness of the CEFA $\mathcal{A} \equiv \mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3$ with respect to the LIA formula $\varphi \equiv r_4 > 120$, that is, whether there exist $w \in \Sigma^*$ and $(v_1, v_2, v_3, v_4) \in \mathbb{Z}^4$ such that w is accepted by $\mathcal{A}$, so that the resulting registers values (v_1, v_2, v_3, v_4) satisfy both $1 \le v_1 \le 60 \wedge 1 \le v_2 \le 60 \wedge 0 \le v_3 \le 60$ and φ. It is not hard to observe that the nonemptiness of $\mathcal{A}$ with respect to φ is independent of the characters of $\mathcal{A}$. Therefore, the characters in $\mathcal{A}$ can be ignored, resulting into an NFA $\mathcal{B}$ over the alphabet $\mathbb{C}$, where $\mathbb{C}$ is the set of vectors from $\mathbb{Z}^4$ occurring in the transitions of $\mathcal{A}$ (see Figure 3(a)). Then the original problem is reduced to the problem of deciding whether there exists a string $w' \in \mathbb{C}^*$ that is accepted by $\mathcal{B}$ and its Parikh image (i.e., numbers of occurrences of characters), say $\eta_{w'} : \mathbb{C} \to \mathbb{N}$, satisfies $1 \le v_1' \le 60 \wedge 1 \le v_2' \le 60 \wedge 0 \le v_3' \le 60 \wedge v_4' > 120$, where $(v_1', v_2', v_3', v_4') = \sum_{\vec{v} \in \mathbb{C}} \eta_{w'}(\vec{v})\vec{v}$ for each $\vec{v} \in \mathbb{C}$. Intuitively, (v_1', v_2', v_3', v_4') is a weighted sum of vectors $\vec{v} \in \mathbb{C}$, where the weight is the number of occurrences of $\vec{v}$ in w'. (See Section 6.2 for more detailed arguments.)

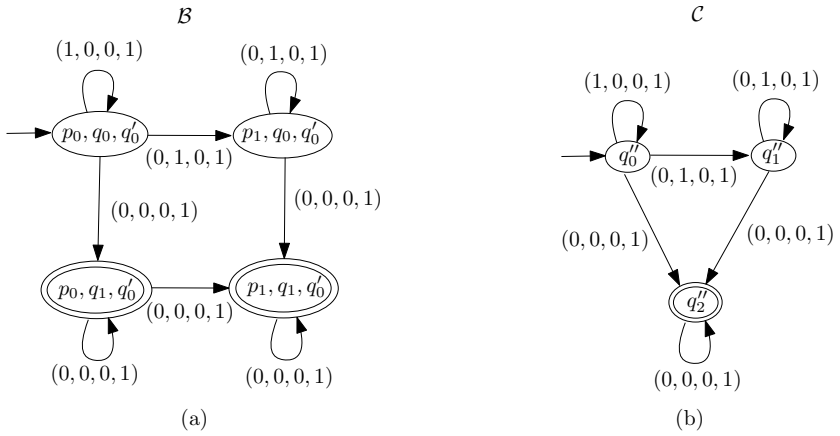

Figure 3: Reduced automaton $\mathcal{B}$ and $\mathcal{C}$

Let $\mathbb{C} = \{\vec{v_1}, \cdots, \vec{v_m}\}$. From the results in [29, 30], an existential LIA formula $\psi_{\mathcal{B}}(\mathfrak{x}_1, \cdots, \mathfrak{x}_m)$ can be computed to define the Parikh image of strings that are accepted by $\mathcal{B}$, where $\mathfrak{x}_1, \cdots, \mathfrak{x}_m$ are the integer variables to denote the number of occurrences of $\vec{v_1}, \cdots, \vec{v_m}$. Therefore, the satisfiability of the string constraint in (1) is reduced to the satisfiability of the following existential LIA formula,

$$\begin{aligned} &\psi_{\mathcal{B}}(\mathfrak{x}_1, \cdots, \mathfrak{x}_m) \wedge \bigwedge_{1 \le j \le 4} r_j = \sum_{1 \le k \le m} \mathfrak{x}_k v_{k,j} \wedge \\ &1 \le r_1 \le 60 \wedge 1 \le r_2 \le 60 \wedge 0 \le r_3 \le 60 \wedge r_4 > 120. \end{aligned} \tag{2}$$

which can be solved by the off-the-shelf SMT solvers.

Nevertheless, when the original regexes are complicated (e.g. contain occurrences of negation or intersection operators), the sizes of the NFA $\mathcal{B}$ can still be big and the sizes of the LIA formulas defining the Parikh image of $\mathcal{B}$ are also big. Since the satisfiability of LIA formulas is an NP-complete problem [31], big sizes of LIA formulas would be a bottleneck of the performance. To tackle this issue, we propose techniques to reduce the sizes of the NFA $\mathcal{B}$.

Specifically, to reduce the sizes of $\mathcal{B}$, we determinize $\mathcal{B}$, and apply the minimization algorithm to the resulting deterministic finite automaton (DFA), resulting in a DFA $\mathcal{C}$, as illustrated in Figure 3(b). Note that $\mathcal{C}$ contains only three states q_0'', q_1'', q_2'' and six transitions, while $\mathcal{B}$ contains four states and eight transitions. Furthermore, if $\mathcal{B}$ contains $\vec{0}$-labeled transitions, then we can take these transitions as ϵ-transitions and potentially reduce the sizes of automata further.

We implement all the aforementioned techniques on the top of OSTRICH, resulting in a solver OSTRICH$^{\text{RECL}}$. It turns out that OSTRICH$^{\text{RECL}}$ is able

to solve the string constraint in (1) within one second, while the state-of-the-art string solvers are incapable of solving it within 120 seconds.

3. Preliminaries

We write $\mathbb{N}$ and $\mathbb{Z}$ for the sets of natural and integer numbers. For $n \in \mathbb{N}$ with $n \geq 1$, $[n]$ denotes $\{1, \ldots, n\}$; for $m, n \in \mathbb{N}$ with $m \leq n$, $[m, n]$ denotes $\{i \in \mathbb{N} \mid m \leq i \leq n\}$. Throughout the paper, Σ is a finite alphabet, ranged over $a, b, \ldots$.

For a function f from X to Y and $X' \subseteq X$, we use $\mathsf{prj}_{X'}(f)$ to denote the restriction (aka projection) of f to X', that is, $\mathsf{prj}_{X'}(f)$ is the function from X' to Y, and $\mathsf{prj}_{X'}(f)(x') = f(x')$ for each $x' \in X'$.

Strings and languages. A string over Σ is a (possibly empty) sequence of elements from Σ, denoted by $u, v, w, \ldots$. An empty string is denoted by ε. We use Σ_ε to denote $\Sigma \cup \{\varepsilon\}$. We write Σ^* (resp., Σ^+) for the set of all (resp. nonempty) strings over Σ. For a string u, we use $|u|$ to denote the number of letters in u. In particular, $|\varepsilon| = 0$. Moreover, for $a \in \Sigma$, let $|u|_a$ denote the number of occurrences of a in u. Assume that $u = a_1 \cdots a_n$ is nonempty and $1 \leq i < j \leq n$. We let $u[i]$ denote a_i and $u[i, j]$ for the substring $a_i \cdots a_j$. Let u, v be two strings. We use $u \cdot v$ to denote the *concatenation* of u and v. A language L over Σ is a subset of strings. Let L_1, L_2 be two languages. Then the concatenation of L_1 and L_2, denoted by $L_1 \cdot L_2$, is defined as $\{u \cdot v \mid u \in L_1, v \in L_2\}$. The union (resp. intersection) of L_1 and L_2, denoted by $L_1 \cup L_2$ (resp. $L_1 \cap L_2$), is defined as $\{u \mid u \in L_1 \text{ or } u \in L_2\}$ (resp. $\{u \mid u \in L_1 \text{ and } u \in L_2\}$). The complement of L_1, denoted by $\overline{L_1}$, is defined as $\{u \mid u \in \Sigma^*, u \notin L_1\}$. The difference of L_1 and L_2, denoted by $L_1 \setminus L_2$, is defined as $L_1 \cap \overline{L_2}$. For a language L and $n \in \mathbb{N}$, we define L^n inductively as follows: $L^0 = \{\varepsilon\}$, $L^{n+1} = L \cdot L^n$ for every $n \in \mathbb{N}$. Finally, define $L^* = \bigcup_{n \in \mathbb{N}} L^n$.

Finite automata. A *(nondeterministic) finite automaton* (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states. For readability, we write a transition $(q, a, q') \in \delta$ as $q \xrightarrow[\delta]{a} q'$ (or simply $q \xrightarrow{a} q'$). The *size* of an NFA $\mathcal{A}$, denoted by $|\mathcal{A}|$, is defined as the number of transitions of $\mathcal{A}$. A *run* of $\mathcal{A}$ on a string $w = a_1 \cdots a_n$ is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \cdots q_{n-1} \xrightarrow{a_n} q_n$ such that

$q_0 \in I$. The run is *accepting* if $q_n \in F$. A string w is accepted by an NFA $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. In particular, the empty string ε is accepted by $\mathcal{A}$ if $I \cap F \neq \emptyset$. The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of strings accepted by $\mathcal{A}$. An NFA $\mathcal{A}$ is said to be *deterministic* if I is a singleton and, for every $q \in Q$ and $a \in \Sigma$, there is at most one state $q' \in Q$ such that $(q, a, q') \in \delta$. We use DFA to denote deterministic finite automata.

It is well-known that finite automata capture regular languages. Moreover, the class of regular languages is closed under union, intersection, concatenation, Kleene star, complement, and language difference [32].

Let $w \in \Sigma^*$. The *Parikh image* of w, denoted by $\mathsf{parikh}(w)$, is defined as the function $\eta : \Sigma \to \mathbb{N}$ such that $\eta(a) = |w|_a$ for each $a \in \Sigma$. The *Parikh image* of an NFA $\mathcal{A}$, denoted by $\mathsf{parikh}(\mathcal{A})$, is defined as $\{\mathsf{parikh}(w) \mid w \in \mathcal{L}(\mathcal{A})\}$.

Linear integer arithmetic and Parikh images. We use standard existential *linear integer arithmetic* (LIA) formulas, which typically range over $\psi, \varphi, \Phi, \alpha$. For a set $\mathfrak{X}$ of variables, we use $\psi/\varphi/\Phi/\alpha(\mathfrak{X})$ to denote the set of existential LIA formulas whose free variables are from $\mathfrak{X}$. For example, we use $\varphi(\vec{\mathfrak{x}})$ with $\vec{\mathfrak{x}} = (\mathfrak{x}_1, \cdots, \mathfrak{x}_k)$ to denote an LIA formula φ whose free variables are from $\{\mathfrak{x}_1, \cdots, \mathfrak{x}_k\}$. For an LIA formula $\varphi(\vec{\mathfrak{x}})$, we use $\varphi[\vec{t}/\vec{\mathfrak{x}}]$ to denote the formula obtained by replacing (simultaneously) $\mathfrak{x}_i$ with t_i for every $i \in [k]$ where $\vec{\mathfrak{x}} = (\mathfrak{x}_1, \cdots, \mathfrak{x}_k)$ and $\vec{t} = (t_1, \cdots, t_k)$ are tuples of integer terms.

At last, we recall the result about Parikh images of NFA. For each $a \in \Sigma$, let $\mathfrak{z}_a$ be an integer variable. Let $\mathfrak{Z}_\Sigma$ denote the set of integer variables $\mathfrak{z}_a$ for $a \in \Sigma$. Let $\mathcal{A}$ be an NFA over the alphabet Σ. Then we say that an LIA formula $\psi(\mathfrak{Z}_\Sigma)$ defines the Parikh image of $\mathcal{A}$, if $\{\eta : \Sigma \to \mathbb{N} \mid \psi[(\eta(a)/\mathfrak{z}_a)_{a\in\Sigma}] \text{ is true}\} = \mathsf{parikh}(\mathcal{A})$.

Theorem 1 ([29]). *Given an NFA $\mathcal{A}$, an existential LIA formula $\psi_\mathcal{A}(\mathfrak{Z}_\Sigma)$ can be computed in linear time that defines the Parikh image of $\mathcal{A}$.*

4. String constraints with regex-counting and string-length

In the sequel, we define the string constraints with regex-counting and string-length functions, i.e., **RE**gex-**C**ounting **L**ogic (abbreviated as RECL). The syntax of RECL is defined by the rules in Figure 4, where x is a string variable, $\mathfrak{x}$ is an integer variable, a is a character from an alphabet Σ, and m, n are integer constants. A RECL formula φ is a conjunction of atomic formulas of the form $x \in e$ or $t_1 \; o \; t_2$, where e is a regular expression, t_1 and t_2

are integer terms, and $o \in \{=, \neq, \leq, \geq, <, >\}$. Atomic formulas of the form $x \in e$ are called *regular membership* constraints, and atomic formulas of the form $t_1\ o\ t_2$ are called *length* constraints. A regular expression e is built from $\emptyset$, the empty string ϵ, and the character a by using concatenation $\cdot$, union $+$, Kleene star *, intersection $\cap$, complement $^-$, difference $\setminus$, counting $^{\{m,n\}}$ or $^{\{m,\infty\}}$. An integer term is built from constants n, variables $\mathfrak{x}$, and string lengths $|x|$ by operators $+$ and $-$.

$$
\begin{array}{lcll}
\varphi & ::= & x \in e \mid t_1\ o\ t_2 \mid \varphi \wedge \varphi & \text{formulas} \\
e & ::= & \emptyset \mid \epsilon \mid a \mid e \cdot e \mid e + e \mid e^* \mid e \cap e \mid \overline{e} \mid e \setminus e \mid e^{\{m,n\}} \mid e^{\{m,\infty\}} \mid (e) & \text{regexes} \\
t & ::= & n \mid \mathfrak{x} \mid |x| \mid t - t \mid t + t & \text{integer terms}
\end{array}
$$

Figure 4: Syntax of RECL

Moreover, for $S \subseteq \Sigma$ with $S = \{a_1, \cdots, a_k\}$, we use S as an abbreviation of $a_1 + \cdots + a_k$.

For each regular expression e, the language defined by e, denoted by $\mathcal{L}(e)$, is defined recursively. For instance, $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(a) = \{a\}$, and $\mathcal{L}(e_1 \cdot e_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$, $\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$, and so on. It is well-known that regular expressions define the same class of languages as finite state automata, that is, the class of regular languages [32].

Let φ be a RECL formula and $\mathsf{SVars}(\varphi)$ (resp. $\mathsf{IVars}(\varphi)$) denote the set of string (resp. integer) variables occurring in φ. Then the semantics of φ is defined with respect to a mapping $\theta : \mathsf{SVars}(\varphi) \to \Sigma^* \uplus \mathsf{IVars}(\varphi) \to \mathbb{Z}$ (where $\uplus$ denotes the disjoint union). Note that the mapping θ can naturally extend to the set of integer terms. For instance, $\theta(|x|) = |\theta(x)|$, $\theta(t_1 + t_2) = \theta(t_1) + \theta(t_2)$. A mapping θ is said to satisfy φ, denoted by $\theta \models \varphi$, if one of the following holds: $\varphi \equiv x \in e$ and $\theta(x) \in \mathcal{L}(e)$, $\varphi \equiv t_1\ o\ t_2$ and $\theta(t_1)\ o\ \theta(t_2)$, $\varphi \equiv \varphi_1 \wedge \varphi_2$ and $\theta \models \varphi_1$ and $\theta \models \varphi_2$. A RECL formula φ is satisfiable if there is a mapping θ such as $\theta \models \varphi$. The satisfiability problem for RECL (which is abbreviated as $\mathsf{SAT}_{\mathsf{RECL}}$) is deciding whether a given RECL formula φ is satisfiable.

5. Cost-enriched finite automata

In this section, we define cost-enriched finite state automata (CEFA), which was introduced in [18] and will be used to solve the satisfiability problem of RECL later on. Intuitively, CEFA adds write-only cost registers to finite state automata. "write-only" means that the cost registers can only be written/updated but cannot be read, i.e., they cannot be used to guard the transitions.

Definition 1 (Cost-Enriched Finite Automaton). *A cost-enriched finite automaton $\mathcal{A}$ is a tuple $(R, Q, \Sigma, \delta, I, F, \alpha)$ where*

- *$R = \{r_1, \cdots, r_k\}$ is a finite set of registers,*
- *Q, Σ, I, F are as in the definition of NFA,*
- *$\delta \subseteq Q \times \Sigma \times Q \times \mathbb{Z}^R$ is a transition relation, where $\mathbb{Z}^R$ denotes the updates on the values of registers.*
- *$\alpha \in \Phi(R)$ is an LIA formula specifying an accepting condition.*

For convenience, we use $R_{\mathcal{A}}$ to denote the set of registers of $\mathcal{A}$. We assume a linear order on R and write R as a vector $(r_1, \cdots, r_k)$. Accordingly, we write an element of $\mathbb{Z}^R$ as a vector $(v_1, \cdots, v_k)$, where v_i is the update of r_i for each $i \in [k]$. We also write a transition $(q, a, q', \vec{v}) \in \delta$ as $q \xrightarrow[\vec{v}]{a} q'$.

The semantics of CEFA is defined as follows. Let $\mathcal{A} = (R, Q, \Sigma, \delta, I, F, \alpha)$ be a CEFA. A *run* of $\mathcal{A}$ on a string $w = a_1 \cdots a_n$ is a sequence $q_0 \xrightarrow[\vec{v_1}]{a_1} q_1 \cdots q_{n-1} \xrightarrow[\vec{v_n}]{a_n} q_n$ such that $q_0 \in I$ and $q_{i-1} \xrightarrow[\vec{v_i}]{a_i} q_i$ for each $i \in [n]$. A run $q_0 \xrightarrow[\vec{v_1}]{a_1} q_1 \cdots q_{n-1} \xrightarrow[\vec{v_n}]{a_n} q_n$ is *accepting* if $q_n \in F$ and $\alpha(\vec{v'}/R)$ is true, where $\vec{v'} = \sum_{j \in [n]} \vec{v_j}$. The string w is *accepted* by the CEFA if and only if there is an accepting run of w. The vector $\vec{v'} = \sum_{j \in [n]} \vec{v_j}$ is called the *cost* of an accepting run $q_0 \xrightarrow[\vec{v_1}]{a_1} q_1 \cdots q_{n-1} \xrightarrow[\vec{v_n}]{a_n} q_n$.Note that the values of all registers are initiated to zero and updated to $\sum_{j \in [n]} \vec{v_j}$ after all the transitions in the run are executed. We use $\vec{v'} \in \mathcal{A}(w)$ to denote the fact that there is an accepting run of $\mathcal{A}$ on w whose cost is $\vec{v'}$. We define the semantics of a CEFA $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, as $\{(w; \vec{v'}) \mid \vec{v'} \in \mathcal{A}(w)\}$. In particular, if $I \cap F \neq \emptyset$ and $\alpha[\vec{0}/R]$ is true, then $(\varepsilon; \vec{0}) \in \mathcal{L}(\mathcal{A})$. Moreover, we define the *output* of a CEFA $\mathcal{A}$, denoted by $\mathcal{O}(\mathcal{A})$, as $\{\vec{v'} \mid \exists w.\ \vec{v'} \in \mathcal{A}(w)\}$.

We want to remark that the definition of CEFA above is slightly different from that of [18], where CEFA did not include accepting conditions. Moreover, the accepting conditions α in CEFA are defined in a *global* fashion because the accepting condition does not distinguish final states. This technical choice is made so that the determinization and minimization of NFA can be utilized to reduce the size of CEFA in the next section.

In the sequel, we define three CEFA operations: union, intersection, and concatenation. The following section will use these three operations to solve RECL constraints. Note that the union, intersection, and concatenation operations are defined in a slightly more involved manner than register automata [23] and counter automata [19], as a result of the (additional) accepting conditions.

Let $\mathcal{A}_1 = (R_1, Q_1, \Sigma, \delta_1, q_{1,0}, F_1, \alpha_1)$ and $\mathcal{A}_2 = (R_2, Q_2, \Sigma, \delta_2, q_{2,0}, F_2, \alpha_2)$ be two CEFA that share the alphabet. Moreover, suppose that $R_1 \cap R_2 = \emptyset$ and $Q_1 \cap Q_2 = \emptyset$.

The *union* of $\mathcal{A}_1$ and $\mathcal{A}_2$ is denoted by $\mathcal{A}_1 \cup \mathcal{A}_2$. Two fresh auxiliary registers say $r'_1, r'_2 \notin R_1 \cup R_2$, are introduced so that the accepting condition knows whether a run is from $\mathcal{A}_1$ (or $\mathcal{A}_2$). Specifically, $\mathcal{A}_1 \cup \mathcal{A}_2 = (R', Q', \Sigma, \delta', I', F', \alpha')$ where

- $R' = R_1 \cup R_2 \cup \{r'_1, r'_2\}$, $Q' = Q_1 \cup Q_2 \cup \{q'_0\}$ with $q'_0 \notin Q_1 \cup Q_2$, $I' = \{q'_0\}$,
- δ' is the union of four transitions sets:
  - $\{(q'_0, a, q'_1, (\vec{v_1}, \vec{0}, 1, 0)) \mid \exists q_1 \in I_1.\ (q_1, a, q'_1, \vec{v_1}) \in \delta_1\}$
  - $\{(q'_0, a, q'_2, (\vec{0}, \vec{v_2}, 0, 1)) \mid \exists q_2 \in I_2.\ (q_2, a, q'_2, \vec{v_2}) \in \delta_2\}$
  - $\{(q_1, a, q'_1, (\vec{v_1}, \vec{0}, 0, 0)) \mid q_1 \notin I_1.\ (q_1, a, q'_1, \vec{v_1}) \in \delta_1\}$
  - $\{(q_2, a, q'_2, (\vec{0}, \vec{v_2}, 0, 0)) \mid q_2 \notin I_2.\ (q_2, a, q'_2, \vec{v_2}) \in \delta_2\}$

  where $(\vec{v_1}, \vec{0}, 1, 0)$ is a vector that updates R_1 by $\vec{v_1}$, updates R_2 by $\vec{0}$, and updates r'_1, r'_2 by $1, 0$ respectively. Similarly for $(\vec{0}, \vec{v_2}, 0, 1)$, and so on.
- F' and α' are defined as follows,
  - if $(\epsilon; \vec{0})$ belongs to $\mathcal{L}(\mathcal{A}_1)$ or $\mathcal{L}(\mathcal{A}_2)$, i.e., one of the two automata accepts the empty string ϵ, then $F' = F_1 \cup F_2 \cup \{q'_0\}$ and $\alpha' = (r'_1 = 0 \wedge r'_2 = 0) \vee (r'_1 = 1 \wedge \alpha_1) \vee (r'_2 = 1 \wedge \alpha_2)$,
  - otherwise, $F' = F_1 \cup F_2$ and $\alpha' = (r'_1 = 1 \wedge \alpha_1) \vee (r'_2 = 1 \wedge \alpha_2)$.

From the construction, we know that

$$\mathcal{L}(\mathcal{A}_1 \cup \mathcal{A}_2) = \left\{ (w; \vec{v}) \;\middle|\; \begin{array}{l} (w; \mathsf{prj}_{R_1}(\vec{v})) \in \mathcal{L}(\mathcal{A}_1) \text{ and } \mathsf{prj}_{r'_1}(\vec{v}) = 1, \text{ or} \\ (w; \mathsf{prj}_{R_2}(\vec{v})) \in \mathcal{L}(\mathcal{A}_2) \text{ and } \mathsf{prj}_{r'_2}(\vec{v}) = 1, \text{ or} \\ (w; \vec{v}) = (\epsilon, \vec{0}) \text{ if } \mathcal{A}_1 \text{ or } \mathcal{A}_2 \text{ accepts } \epsilon \end{array} \right\}.$$

Intuitively, $\mathcal{A}_1 \cup \mathcal{A}_2$ accepts the words that are accepted by one of the $\mathcal{A}_1$ and $\mathcal{A}_2$ and outputs the costs of the corresponding automaton.

The *intersection* of $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \cap \mathcal{A}_2 = (R', Q', \Sigma, \delta', I', F', \alpha')$, is defined in the sequel.

- $R' = R_1 \cup R_2$, $Q' = Q_1 \times Q_2$, $I' = I_1 \times I_2$, $F' = F_1 \times F_2$, $\alpha' = \alpha_1 \wedge \alpha_2$,
- δ' comprises the tuples $((q_1, q_2), a, (q_1', q_2'), (\vec{v_1}, \vec{v_2}))$ such that $(q_1, a, q_1', \vec{v_1}) \in \delta_1$ and $(q_2, a, q_2', \vec{v_2}) \in \delta_2$.

From the construction,

$$\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2) = \{(w; \vec{v}) \mid (w; \mathsf{prj}_{R_1}(\vec{v})) \in \mathcal{L}(\mathcal{A}_1) \text{ and } (w; \mathsf{prj}_{R_2}(\vec{v})) \in \mathcal{L}(\mathcal{A}_2)\}.$$

Intuitively, $\mathcal{A}_1 \cap \mathcal{A}_2$ accepts the words that are accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$ and outputs the costs of $\mathcal{A}_1$ and $\mathcal{A}_2$.

The *concatenation* of $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \cdot \mathcal{A}_2$, is defined similarly as that of NFA, that is, a tuple $(Q', \Sigma, \delta', I', F', \alpha')$, where $Q' = Q_1 \cup Q_2$, $I' = I_1$, $\alpha' = \alpha_1 \wedge \alpha_2$, $\delta' = \{(q_1, a, q_1', (\vec{v_1}, \vec{0})) \mid (q_1, a, q_1', \vec{v_1}) \in \delta_1\} \cup \{(q_2, a, q_2', (\vec{0}, \vec{v_2})) \mid (q_2, a, q_2', \vec{v_2}) \in \delta_2\} \cup \{(q_1, a, q_2, (\vec{0}, \vec{v_2})) \mid q_1 \in F_1, \exists q' \in I_2.\ (q', a, q_2, \vec{v_2}) \in \delta_2\}$, moreover, if $I_2 \cap F_2 \neq \emptyset$, then $F' = F_1 \cup F_2$, otherwise, $F' = F_2$. From the construction,

$$\mathcal{L}(\mathcal{A}_1 \cdot \mathcal{A}_2) = \{(w_1 w_2; \vec{v}) \mid (w_1; \mathsf{prj}_{R_1}(\vec{v})) \in \mathcal{L}(\mathcal{A}_1) \text{ and } (w_2; \mathsf{prj}_{R_2}(\vec{v})) \in \mathcal{L}(\mathcal{A}_2)\}.$$

Furthermore, the union, intersection, and concatenation operations can be extended naturally to multiple CEFA, that is, $\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$, $\mathcal{A}_1 \cap \cdots \cap \mathcal{A}_n$, $\mathcal{A}_1 \cdot \cdots \cdot \mathcal{A}_n$. For instance, $\mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 = (\mathcal{A}_1 \cup \mathcal{A}_2) \cup \mathcal{A}_3$, $\mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3 = (\mathcal{A}_1 \cap \mathcal{A}_2) \cap \mathcal{A}_3$, and $\mathcal{A}_1 \cdot \mathcal{A}_2 \cdot \mathcal{A}_3 = (\mathcal{A}_1 \cdot \mathcal{A}_2) \cdot \mathcal{A}_3$.

6. Solving RECL constraints

The goal of this section is to show how to solve RECL constraints by utilizing CEFA. At first, we reduce the satisfiability of RECL constraints to a decision problem defined in the sequel. Then we propose a decision procedure for this problem.

Definition 2 ($\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$)**.** *Let $x_1, \cdots, x_n$ be string variables, $\Lambda_{x_1}, \cdots, \Lambda_{x_n}$ be nonempty sets of CEFA over the alphabet Σ with $\Lambda_{x_i} = \{\mathcal{A}_{i,1}, \cdots, \mathcal{A}_{i,l_i}\}$ for every $i \in [n]$ where the sets of registers $R_{\mathcal{A}_{1,1}}, \cdots, R_{\mathcal{A}_{1,l_1}}, \cdots, R_{\mathcal{A}_{n,1}}, \cdots, R_{\mathcal{A}_{n,l_n}}$*

are mutually disjoint, moreover, let φ *be an LIA formula whose free variables are from* $\bigcup_{i\in[n],j\in[l_i]} R_{\mathcal{A}_{i,j}}$. *Then the CEFA in* $\Lambda_{x_1},\cdots,\Lambda_{x_n}$ *are said to be nonempty w.r.t.* φ *if there are assignments* $\theta : \{x_1,\cdots,x_n\} \to \Sigma^*$ *and vectors* $\vec{v_{i,j}}$ *such that* $(\theta(x_i);\vec{v_{i,j}}) \in \mathcal{L}(\mathcal{A}_{i,j})$ *and* $\varphi[(\vec{v_{i,j}}/R_{\mathcal{A}_{i,j}})]$ *is true, for every* $i \in [n], j \in [l_i]$.

Proposition 1 ([18]). $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ *is PSPACE-complete.*

Note that the decision procedure in [18] was only used to prove the upper bound in Proposition 1 and not implemented as a matter of fact. Instead, the symbolic model checker nuXmv [33] was used to solve $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$. We do not rely on nuXmv in this work and shall propose a new algorithm for solving $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ in Section 6.2.

6.1. From $\mathsf{SAT}_{\mathsf{RECL}}$ *to* $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$

Let φ be a RECL constraint and $x_1,\cdots,x_n$ be an enumeration of the string variables occurring in φ. Moreover, let $\varphi \equiv \varphi_1 \wedge \varphi_2$ such that φ_1 is a conjunction of regular membership constraints of φ, and φ_2 is a conjunction of length constraints of φ. We shall reduce the satisfiability of φ to an instance of $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$.

At first, we show how to construct a CEFA from a regex where counting operators may occur. Our approach is based on Thompson's construction [34] for regexes without counting operators. Let us start with register-representable regexes defined in the sequel.

Let us fix an alphabet Σ.

Let e be a regex over Σ. Then an occurrence of counting operators in e, say $(e')^{\{m,n\}}$ (or $(e')^{\{m,\infty\}}$), is said to be *register-representable* if $(e')^{\{m,n\}}$ (or $(e')^{\{m,\infty\}}$) is not in the scope of a Kleene star, another counting operator, complement, or language difference in e. We say that e is *register-representable* if all occurrences of counting operators in e are register-representable. For instance, $a^{\{2,6\}} \cap a^{\{4,\infty\}}$ is register-representable, while $\overline{a^{\{2,6\}}}$ and $(a^{\{2,6\}})^{\{4,\infty\}}$ are not since $a^{\{2,6\}}$ is in the scope of complement and the counter operator $\{2,\infty\}$ respectively.

Let e be a register-representable regex over Σ. By the following procedure, we will construct a CEFA out of e, denoted by $\mathcal{A}_e$.

1. For each sub-expression $(e')^{\{m,n\}}$ with $m \leq n$ (resp. $(e')^{\{m,\infty\}}$) of e, we construct a CEFA $\mathcal{A}_{(e')^{\{m,n\}}}$ (resp. $\mathcal{A}_{(e')^{\{m,\infty\}}}$). Let $\mathcal{A}_{e'} = (Q,\Sigma,\delta,I,F)$.

Then $\mathcal{A}_{(e')\{m,n\}} = ((r'), Q', \Sigma, \delta'', I', F', \alpha')$, where r' is a new register, $Q' = Q \cup \{q_0\}$ with $q_0 \notin Q$, $I' = \{q_0\}$, $F' = F \cup \{q_0\}$, and

$$\begin{aligned}\delta'' \;=\; & \{(q, a, q', (0)) \mid (q, a, q') \in \delta\} \cup \\ & \{(q_0, a, q', (1)) \mid \exists q_0' \in I.\ (q_0', a, q') \in \delta\} \cup \\ & \{(q, a, q', (1)) \mid q \in F, \exists q_0' \in I.\ (q_0', a, q') \in \delta\},\end{aligned}$$

moreover, $\alpha' = m \leq r' \leq n$ if $I \cap F = \emptyset$, otherwise $\alpha' = r' \leq n$. (Intuitively, if ε is accepted by $\mathcal{A}_{e'}$, then the value of r' can be less than m.) Moreover, $\mathcal{A}_{(e')\{m,\infty\}}$ is constructed by adapting α' in $\mathcal{A}_{(e')\{m,n\}}$ as follows: $\alpha' = m \leq r'$ if $I \cap F = \emptyset$ and $\alpha' = \texttt{true}$ otherwise.

2. For each sub-expression e' of e such that e' contains occurrences of counting operators but e' itself is not of the form $(e_1')^{\{m,n\}}$ or $(e_1')^{\{m,\infty\}}$, from the assumption that e is register-representable, we know that e' is of the form $e_1' \cdot e_2'$, $e_1' + e_2'$, $e_1' \cap e_2'$, or (e_1'). For $e' = (e_1')$, we have $\mathcal{A}_{e'} = \mathcal{A}_{e_1'}$. For $e' = e_1' \cdot e_2'$, $e' = e_1' + e_2'$, or $e' = e_1' \cap e_2'$, suppose that CEFA $\mathcal{A}_{e_1'}$ and $\mathcal{A}_{e_2'}$ have been constructed.
3. For each maximal sub-expression e' of e such that e' contains no occurrences of counting operators, an NFA $\mathcal{A}_{e'}$ can be constructed by structural induction on the syntax of e'. Then we have $\mathcal{A}_{e'} = \mathcal{A}_{e_1'} \cdot \mathcal{A}_{e_2'}$, $\mathcal{A}_{e'} = \mathcal{A}_{e_1'} \cup \mathcal{A}_{e_2'}$, or $\mathcal{A}_{e'} = \mathcal{A}_{e_1'} \cap \mathcal{A}_{e_2'}$.

For non-register-representable regexes, we first transform them into register-representable regexes by unfolding all the non-register-representable occurrences of counting operators. After that, we utilize the aforementioned procedure to construct CEFA. For instance, $(a^{\{2,6\}} \cdot b^*)^{\{2,\infty\}}$ is transformed into $(aa(\varepsilon + a + aa + aaa + aaaa) \cdot b^*)^{\{2,\infty\}}$. The unfoldings of the inner counting operators of non-register-representable regexes incur an exponential blowup in the worst case. Nevertheless, those regexes occupy only 5% of the 48,843 regexes that are collected from the practice (see Section 8.1). Moreover, the unfoldings are partial in the sense that the outmost counting operators are not unfolded. It turns out that our approach can solve almost all the RECL constraints involving these 48,843 regexes, except 181 of them (See Figure 1).

For each $i \in [n]$, let $x_i \in e_{i,1}, \cdots, x_i \in e_{i,l_i}$ be an enumeration of the regular membership constraints for x_i in φ_1. Then we can construct CEFA $\mathcal{A}_{i,j}$ from $e_{i,j}$ for each $i \in [n]$ and $j \in [l_i]$. Moreover, we construct another CEFA $\mathcal{A}_{i,0}$ for each $i \in [n]$ to model the length of x_i. Specifically, $\mathcal{A}_{i,0}$ is constructed as $((r_{i,0}), \{q_{i,0}\}, \Sigma, \delta_{i,0}, \{q_{i,0}\}, \{q_{i,0}\}, \texttt{true})$ where $r_{i,0}$ is a fresh register and $\delta_{i,0} = \{(q_{i,0}, a, q_{i,0}, (1)) \mid a \in \Sigma\}$. Let $\Lambda_{x_i} = \{\mathcal{A}_{i,0}, \mathcal{A}_{i,1}, \cdots, \mathcal{A}_{i,l_i}\}$

for each $i \in [n]$, and $\varphi_2' \equiv \varphi_2[r_{1,0}/|x_1|, \cdots, r_{n,0}/|x_n|]$. Then the satisfiability of φ is reduced to the nonemptiness of CEFAs in $\Lambda_{x_1}, \cdots, \Lambda_{x_n}$ w.r.t. φ_2'.

6.2. Solving $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$

In this section, we present a procedure to solve the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem: Suppose that $x_1, \cdots, x_n$ are mutually distinct string variables, $\Lambda_{x_1}, \cdots, \Lambda_{x_n}$ are nonempty sets of CEFA over the same alphabet Σ where $\Lambda_{x_i} = \{\mathcal{A}_{i,1}, \cdots, \mathcal{A}_{i,l_i}\}$ for every $i \in [n]$. Moreover, the sets of registers $R_{\mathcal{A}_{1,1}}, \cdots, R_{\mathcal{A}_{n,l_n}}$ are mutually disjoint, and φ is a LIA formula whose free variables are from $\bigcup_{i\in[n],j\in[l_i]} R_{\mathcal{A}_{i,j}}$.

The procedure comprises three steps.

Step 1 (Computing intersection automata). For each $i \in [n]$, compute a CEFA $\mathcal{B}_i = \mathcal{A}_{i,1} \cap \cdots \cap \mathcal{A}_{i,l_i}$, and let $\Lambda'_{x_i} := \{\mathcal{B}_i\}$. □

After Step 1, the nonemptiness of CEFAs in $\Lambda_{x_1}, \cdots, \Lambda_{x_n}$ w.r.t. φ is reduced to the nonemptiness of CEFAs in $\Lambda'_{x_1}, \cdots, \Lambda'_{x_n}$ w.r.t. φ.

In the following steps, we reduce the non-emptiness of CEFAs in $\Lambda'_{x_1}, \cdots, \Lambda'_{x_n}$ w.r.t. φ to the satisfiability of an LIA formula. The reduction relies on the following two observations.

Observation 1. *CEFA in $\Lambda'_{x_1} = \{\mathcal{B}_1\}, \cdots, \Lambda'_{x_n} = \{\mathcal{B}_n\}$ are nonempty w.r.t. φ iff there are $\vec{v_1} \in \mathcal{O}(\mathcal{B}_1), \cdots, \vec{v_n} \in \mathcal{O}(\mathcal{B}_n)$ such that $\varphi[\vec{v_1}/R_{\mathcal{B}_1}, \cdots, \vec{v_n}/R_{\mathcal{B}_n}]$ is true.*

Let $\mathcal{A} = (R, Q, \Sigma, \delta, I, F, \alpha)$ be a CEFA and $\mathbb{C}_{\mathcal{A}} = \{\vec{v} \mid \exists q, a, q'.\ (q, a, q', \vec{v}) \in \delta\}$. Moreover, let $\mathcal{U}_{\mathcal{A}} = (Q, \mathbb{C}_{\mathcal{A}}, \delta', I, F)$ be an NFA over the alphabet $\mathbb{C}_{\mathcal{A}}$ that is obtained from $\mathcal{A}$ by dropping the accepting condition and ignoring the characters, that is, δ' comprises tuples $(q, \vec{v}, q')$ such that $(q, a, q', \vec{v}) \in \delta$ for $a \in \Sigma$.

Observation 2. *For each CEFA $\mathcal{A} = (R, Q, \Sigma, \delta, I, F, \alpha)$,*

$$\mathcal{O}(\mathcal{A}) = \left\{ \sum_{\vec{v}\in\mathbb{C}_{\mathcal{A}}} \eta(\vec{v})\vec{v} \;\middle|\; \eta \in \mathsf{parikh}(\mathcal{U}_{\mathcal{A}}) \text{ and } \alpha\left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{A}}} \eta(\vec{v})\vec{v}/R\right] \text{ is true} \right\}.$$

For $i \in [n]$, let α_i be the accepting condition of $\mathcal{B}_i$. Then from Observation 2, we know that the following two conditions are equivalent,

- there are $\vec{v_1} \in \mathcal{O}(\mathcal{B}_1), \cdots, \vec{v_n} \in \mathcal{O}(\mathcal{B}_n)$ such that $\varphi[\vec{v_1}/R_{\mathcal{B}_1}, \cdots, \vec{v_n}/R_{\mathcal{B}_n}]$ is true,

- there are $\eta_1 \in \mathsf{parikh}(\mathcal{U}_{\mathcal{B}_1}), \cdots, \eta_n \in \mathsf{parikh}(\mathcal{U}_{\mathcal{B}_n})$ such that

$$\bigwedge_{i\in[n]} \alpha_i \left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_i}} \eta_i(\vec{v})\vec{v}/R_{\mathcal{B}_i}\right] \wedge \varphi \left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_1}} \eta_1(\vec{v})\vec{v}/R_{\mathcal{B}_1}, \cdots, \sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_n}} \eta_n(\vec{v})\vec{v}/R_{\mathcal{B}_n}\right]$$

  is true.

Therefore, to solve the nonemptiness of CEFA in $\Lambda'_{x_1}, \cdots, \Lambda'_{x_n}$ w.r.t. φ, it is sufficient to compute the existential LIA formulas $\psi_{\mathcal{U}_{\mathcal{B}_1}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_1}}), \cdots, \psi_{\mathcal{U}_{\mathcal{B}_n}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_n}})$ to represent the Parikh images of $\mathcal{U}_{\mathcal{B}_1}$, $\cdots$, $\mathcal{U}_{\mathcal{B}_n}$ respectively, where $\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_i}} = \{\mathfrak{z}_{i,\vec{v}} \mid \vec{v} \in \mathbb{C}_{\mathcal{B}_i}\}$ for $i \in [n]$, and solve the satisfiability of the following existential LIA formula

$$\bigwedge_{i\in[n]} \left(\psi_{\mathcal{U}_{\mathcal{B}_i}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_i}}) \wedge \alpha_i \left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_i}} \mathfrak{z}_{i,\vec{v}}\vec{v}/R_{\mathcal{B}_i}\right]\right) \wedge \varphi \left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_1}} \mathfrak{z}_{1,\vec{v}}\vec{v}/R_{\mathcal{B}_1}, \cdots, \sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_n}} \mathfrak{z}_{n,\vec{v}}\vec{v}/R_{\mathcal{B}_n}\right].$$

Intuitively, the integer variables $\mathfrak{z}_{i,\vec{v}}$ represent the number of occurrences of $\vec{v}$ in the strings accepted by $\mathcal{U}_{\mathcal{B}_i}$.

Because the sizes of the LIA formulas $\psi_{\mathcal{U}_{\mathcal{B}_1}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_1}}), \cdots, \psi_{\mathcal{U}_{\mathcal{B}_n}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_n}})$ are proportional to the sizes (more precisely, the alphabet size, the number of states and transitions) of NFA $\mathcal{U}_{\mathcal{B}_1}, \cdots, \mathcal{U}_{\mathcal{B}_n}$, and the satisfiability of existential LIA formulas is NP-complete, it is vital to reduce the sizes of these NFAs to improve the performance.

Since $\sum_{\vec{v}\in\mathbb{C}(\mathcal{B}_i)} \eta(\vec{v})\vec{v} = \sum_{\vec{v}\in\mathbb{C}(\mathcal{B}_i)\setminus\{\vec{0}\}} \eta(\vec{v})\vec{v}$ for each $i \in [n]$ and $\eta \in \mathsf{parikh}(\mathcal{U}_{\mathcal{B}_i})$, it turns out that the $\vec{0}$-labeled transitions in $\mathcal{U}_{\mathcal{B}_i}$ do not contribute to the final output $\sum_{\vec{v}\in\mathbb{C}(\mathcal{B}_i)} \eta(\vec{v})\vec{v}$. Therefore, we can apply the following size-reduction technique for $\mathcal{U}_{\mathcal{B}_i}$'s.

Step 2 (Reducing automata sizes). For each $i \in [n]$, we view the transitions $(q, \vec{0}, q')$ in $\mathcal{U}_{\mathcal{B}_i}$ as ε-transitions (q, ε, q'), and remove the ε-transitions from $\mathcal{U}_{\mathcal{B}_i}$. Then we determinize and minimize the resulting NFA. □

For $i \in [n]$, let $\mathcal{C}_i$ denote the DFA obtained from $\mathcal{U}_{\mathcal{B}_i}$ by executing Step 2 and $\mathbb{C}_{\mathcal{C}_i} := \mathbb{C}_{\mathcal{B}_i} \setminus \{\vec{0}\}$. From the construction, we know that $\mathsf{parikh}(\mathcal{C}_i) = \mathsf{prj}_{\mathbb{C}_{\mathcal{C}_i}}(\mathsf{parikh}(\mathcal{U}_{\mathcal{B}_i}))$ for each $i \in [n]$. Therefore, we compute LIA formulas from $\mathcal{C}_i$'s, instead of $\mathcal{U}_{\mathcal{B}_i}$'s, to represent the Parikh images.

Step 3 (Computing Parikh images). For each $i \in [n]$, we compute an existential LIA formula $\psi_{\mathcal{C}_i}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{C}_i}})$ from $\mathcal{C}_i$ to represent $\mathsf{parikh}(\mathcal{C}_i)$. Then we solve the satisfiability of the following formula,

$$\bigwedge_{i\in[n]} \left(\psi_{\mathcal{C}_i}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{C}_i}}) \wedge \alpha_i \left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{C}_i}} \mathfrak{z}_{i,\vec{v}}\vec{v}/R_{\mathcal{B}_i} \right] \right) \wedge \varphi \left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{C}_1}} \mathfrak{z}_{1,\vec{v}}\vec{v}/R_{\mathcal{B}_1}, \cdots, \sum_{\vec{v}\in\mathbb{C}_{\mathcal{C}_n}} \mathfrak{z}_{n,\vec{v}}\vec{v}/R_{\mathcal{B}_n} \right].$$

7. Heuristics for Complex RECL Constraints

In the previous paper [35], we naively unwind the counting operators when they are in other counting or complement operator. However, this approach is not efficient when the counting bounds are large. In this section, we present some heuristics to solve the RECL constraints more efficiently.

7.1. Heuristic for RECL Constraints with Nested Counting

The naive approach for nested counting in the paper [35] is to unwind all inner counting operators. This approach may generate a large number of states, which is linear to the product of the counting bounds in regex and exponential to the length of the regex. For example, naive unwinding regex $(a^{\{1,1000\}})^{\{1,2\}}$ generate a CEFA containing 1000 states. But if we only unwind the outer counting operator of the regex, not inner, we obtain a CEFA containing only 2 states. The side effect is that we need two registers to store the information of conuting in the inner regex (unwinding of outer counting operator repeat the inner regex, which also repeat the registers of the inner regex).We present a heuristic to reduce the number of states generated by the unwinding process. The idea is to unwind the counting operator with less bound and less side effect firstly. We present the heuristic in the following algorithm.

Step 1. (Compute the number of states generated by each counting operator if we unwind it).

For a regex e, we can approximate the number of states $size(e)$ generated by the following formula:

- If $e = \emptyset$ or $e = \epsilon$, then $size(e) = 0$.
- If $e = e_1 \cdot e_2$ or $e = e_1 + e_2$, then $size(e) = size(e_1) + size(e_2)$.

- If $e = e_1^*$, then $size(e) = size(e_1)$.
- If $e = e_1 \cap e_2$, then $size(e) = min(size(e_1), size(e_2))$.
- If $e = \overline{e_1}$, then $size(e) = 2^{size(e_1)} + 1$.
- $e = e_1 \backslash e_2$ can be transfered to $e = e_1 \cap \overline{e_2}$ and then the size can be computed by the formula of $\cap$ and $\bar{}$.
- For $e = e_1^{\{n,m\}}$ or $e = e_1^{\{m,\infty\}}$, $size(e)$ is equal to $m * size(e_1)$ if it is unwinded, and $size(e_1)$ otherwise.

For each counting operator $^{\{n,m\}}$ or $^{\{m,\infty\}}$ with e as its subregex, the number of states generated by it is $m * size(e)$ if we unwind it.

The formula above is an approximation of the states of CEFA generated by Thompson's construction. Note that the number of states is linear to the bound for the counting operator.

Step 2. (Compute the number of registers generated by each counting operator if we unwind it).

We unwind the counting operator with less bound first. The reason is that the number of states generated by the counting operator is linear to its bound. If we unwind the counting operator with less bound first, the number of states generated by the unwinding process is less. However, the hardness of solving the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem is not only relevant to the number of states, but also the number of registers. For a regex e, we can approximate the number of registers $reg(e)$ generated by the following formula:

- If $e = \emptyset$, $e = \epsilon$, $e = e_1^*$ or $e = \overline{e_1}$, then $reg(e) = 0$.
- If $e = e_1 \cdot e_2$ or $e = e_1 \cap e_2$, then $reg(e) = reg(e_1) + reg(e_2)$.
- If $e = e_1 + e_2$, then $reg(e) = reg(e_1) + reg(e_2) + 1$.
- $e = e_1 \backslash e_2$ can be transfered to $e = e_1 \cap \overline{e_2}$ and then the number of registers can be computed by the formula of $\cap$ and $\bar{}$.
- For $e = e_1^{\{n,m\}}$ or $e = e^{\{m,\infty\}}$, $reg(e)$ is equal to $m * reg(e_1)$ if it is unwinded, and 1 otherwise.

For each counting operator $^{\{n,m\}}$ or $^{\{m,\infty\}}$ with e as its subregex, the number of registers generated by it is $m * reg(e)$ if we unwind it.

Step 3. (Unwind the counting operator with less bound and less registers first).

In the above two steps, we compute the number of states and registers generated by each counting operator. In some cases, we can unwind the counting operator with less bound and less registers directly. But in most cases, there are multiple counting operators where one counting operator has less bound and another counting operator has less registers. To consider the number of states and registers at the same time, we can compute the score of each counting operator by the following formula:

- $score(e) = size(e) * (reg(e) + 1)$.

The reason why we use product to combine the two factors is that the hardness of the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem is highly relevant to the number of variables in the final existential LIA formula, which is linear to the product of the number of states and the number of registers. Use $reg(e)+1$ not $reg(e)$ is because the registers number may be 0.We unwind the counting operator with the smallest score first.

Example 1. *Consider the regex* $(a^{\{1,1000\}})^{\{1,2\}}$. *The score of* $^{\{1,1000\}}$ *is* $1000 * (0 + 1) = 1000$ *and the score of* $^{\{1,2\}}$ *is* $2 * (2 + 1) = 6$. *We unwind the outer counting operator first and obtain a CEFA containing only 2 states.*

7.2. Heuristic for RECL Constraints with Complement on Counting

As we mentioned in the above section 7.1, the unwinding process of the complement operator is exponential to the counting bounds in the worst case. We present a heuristic to reduce the number of states generated by the unwinding process. The idea is to under-approximate and over-approximate the language of the regex first, so that we can avoid unwinding counting operators in complement. We present the heuristic in the following algorithm.

Step 1. (Under-approximate the language of the regex with complement operators).

The under-approximation is replacing each counting operator in the subregex of complement with a Kleene star operator. The language of the subregex is a superset of the language of it after under-approximation. In sequential, the complement of the subregex is a subset of origin. Note that

this approach only works for non-nested complement, which means there is no other complement operators in the complement. For nested complement, we naively handle the inner complement operators as paper [35] and then under-approximate the outer complement operator.

Step 2. (Over-approximate the language of the regex with complement operators).

The over-approximation is to "complement" the CEFA of the subregex directly, without unwinding the counting operators in the subregex. The over-approximated complementary algorithm for CEFA is as follows:

Given a CEFA $\mathcal{A} = (R, Q, \Sigma, \delta, I, F, \alpha)$, the over-approximated complement of it is a tuple $(R \cup \{r\}, Q \cup \{s, f\}, \Sigma, \delta', I, (Q \cup \{s, f\}) \backslash F, \alpha')$, where :

- r is a new regesiter not in R.
- s and f are two new states not in Q.
- δ' is a transitions set containing all transitions in δ and transitions from each state q in Q to s with label a if there is no transitions from q to any state with label a in $\mathcal{A}$. To be more detailed, δ' is the union of four sets:
  - $\{(q, a, q', (\vec{v}, \vec{0})) \mid (q, a, q', \vec{v}) \in \delta\}$
  - $\{(q, a, s, (\vec{0}, \vec{0})) \mid \forall q' \in Q, \vec{v}.(q, a, q', \vec{v}) \notin \delta\}$
  - $\{(q, a, f, (\vec{v}, \vec{1})) \mid \exists q' \in F, (q, a, q', \vec{v}) \in \delta\}$
  - $\{(s, \Sigma, s, (\vec{0}, \vec{0}))\}$
- α' is $r \neq 1 \vee \neg\alpha$

r is the register used to check whether the accepted state is also accepted in the original CEFA. s is the self loop state accepting all string that is not accepted by the original CEFA. f is the merged accepted state of original CEFA. If there is a transition $(q, a, q', \vec{v})$ teminates at the accepted state q' in the original CEFA, we add a corresponding transition $(q, a, f, (\vec{v}, \vec{1}))$ in the over-approximated complement. From the definition of δ', we can see that only the transitions goto f increase the value of regesiter r. The value of r is 1 if and only if the accepted state is also accepted in the original CEFA. With above construction, the meaning of the accepting condition α' is obvious: If

the string teminates at the accepting state in the original CEFA, then the accepting condition α is false.

Theorem 2. *The language of over-approximated complement of CEFA is a superset of the language of exact complement.*

Proof 1. *Suppose that the original CEFA is* $\mathcal{A} = (R, Q, \Sigma, \delta, I, F, \alpha)$*, the over-approximated complement of it is* $\overline{\mathcal{A}}_{over} = (R \cup \{r\}, Q \cup \{s, f\}, \Sigma, \delta', I, (Q \cup \{s, f\}) \backslash F, \alpha')$*, and the exact complement of it is* $\overline{\mathcal{A}}$*. To prove the theorem, we only need to show that if a string is accepted by* $\overline{\mathcal{A}}$*, then it is accepted by* $\overline{\mathcal{A}}_{over}$*. In other words, we need to show that if a string is not accepted by* $\overline{\mathcal{A}}_{over}$*, then it is not accepted by* $\overline{\mathcal{A}}$*, which means it is accepted by* $\mathcal{A}$*. We prove it by construction.*

In the definition of CEFA in section 5, a run $q_0 \xrightarrow[\vec{v_1}]{a_1} q_1 \cdots q_{n-1} \xrightarrow[\vec{v_n}]{a_n} q_n$ *is* accepting *by* $\overline{\mathcal{A}}_{over}$ *if* $q_n \in (Q \cup \{s, f\}) \backslash F$ *and* $\alpha'(\vec{v}'/(R \cup \{r\}))$ *is true, where* $\vec{v}' = \sum_{j \in [n]} \vec{v_j}$*. If a string is not accepted by* $\overline{\mathcal{A}}_{over}$*, then each run of the string is not accepting run, which means the run teminates at state* $q \in F$ *or the accepting condition* $\alpha'(\vec{v}'/(R \cup \{r\}))$ *is* $false$*.*

If each run teminates at state $q_f \in F$*, suppose that each is of form* $q_0 \xrightarrow[(\vec{v_1}, \vec{0})]{a_1} q_1 \cdots q_{n-1} \xrightarrow[(\vec{v_n}, \vec{0})]{a_n} q_f$*, where* $\vec{v_i}$ *is* $(\vec{v_i}, \vec{0})$ *or* $(\vec{v_i}, \vec{1})$ *for* $1 \leq i \leq n$*. Then there is a transition* $(q_{n-1}, a_n, q_f, \vec{v_n})$ *in* δ *because there is a transition* $(q_{n-1}, a_n, q_f, (\vec{v_n}, \vec{0}))$ *in* δ'*. So there is a transition* $(q_{n-1}, a_n, f, (\vec{v_n}, \vec{1}))$ *in* δ' *based on our construction for* δ'*. The run* $q_0 \xrightarrow[(\vec{v_1}, \vec{0})]{a_1} q_1 \cdots q_{n-1} \xrightarrow[(\vec{v_n}, \vec{1})]{a_n} f$ *teminates at the accepting state* f*, contridicting the assumption.*

If each run makes the accepting condition $\alpha'(\vec{v}'/(R \cup \{r\}))$ *false, then the negation of the accepting condition* $\neg\alpha'(\vec{v}'/(R \cup \{r\}))$ *is true. The negation can be rewriten to* $(r = 1 \wedge \alpha)(\vec{v}'/(R \cup \{r\})) = (r = 1)(\vec{v}'/(R \cup \{r\})) \wedge \alpha(\vec{v}'/(R \cup \{r\}))$*. In our construction, only the transitions teminate at* f *increase one to the value of* r*, so that the run must end at* f *to make* $r = 1$ *be true.* $\alpha(\vec{v}'/(R \cup \{r\})) = \alpha(\vec{v'}/R)$ *is also need to be true, implying that the run is an accepting run of* $\mathcal{A}$*.*

7.3. Heuristic for Finding an Accepted Word

In the section 6, we present an algorithm to solve the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem. The algorithm computes existential LIA formulas and then uses SMT

solver to solve it. However, this procedure only anwser the question whether the language of the regex is empty or not. If the language is not empty, we need to find a word in the language in practice. In our preivous paper [35], we do not explicitly present the algorithm to find a word in the language because of the space limitation. We have used a naive approach which randomly searching the CEFAs by the depth-first search, based on the regesiters values obtained from solving the existential LIA formula. However, this approach is not efficient. Suppose that there are n registers whose values are $r_1, \cdots, r_n$ respectively in the CEFA, we may visit each transition $O(r_1 * r_2 \cdots * r_n)$ times during the search. To relieve this problem, we present a heuristic to find a word in the language as follows.

Step 1. (Compute the transitions times visited by the accepted string of CEFAs).

For each CEFA $\mathcal{A} = (R, Q, \Sigma, \delta, I, F, \alpha)$, we already know the values of its registers after solving the existential LIA formula. The values can be seen as a constant vector of length $|R|$. We use $\vec{R}$ to denote the vector and use $\eta((q, a, q', \vec{v}))$ to denote the parikh image of $\mathcal{A}$. Then the transitions times visited by the accepted stirng is obtained from the following formula:

$$\sum_{(q,a,q',\vec{v})\in\delta} \eta((q, a, q', \vec{v}))\vec{v} = \vec{R}$$

Step 2. (Prune the search tree early by the transitions times during the search process).

After computing the transitions times visited by the accepted string, we prune the search tree of the CEFA based on the transitions times. The idea is that we only visit transitions whose visited times are greater than 0. Other transitions are deleted from the CEFA so that we can reduce the search space. Note that the transitions times is updated during the search process so the search space is reduced dynamically. The presudo code is presented in Algorithm 1.

The procedure FINDMODEL takes a CEFA $\mathcal{A}$ and a map *tmap* as input. The map *tmap* maps each transition to its visited times. Stack *todo* is used to store the state need to be visited and its corresponding map and word. The procedure uses a set *visited* to store the state and its corresponding map that have been visited. In the while loop from line 4 to line 23, the procedure depth-first search the CEFA. If the state is an accepting state and all transitions are visited with correct times, the procedure returns the word

(line 7). During the search process, the procedure prunes the search space by the transitions times. If the visited times of a transition is less than 0, the procedure does not visit it (line 11). If the state and its corresponding map have been visited, the procedure does not visit it (line 14). If there is a transition with visited times greater than 0 not reachable from the current state, the procedure does not visit it(line 18). In practice, the pruning in line 18 make a great contribution to the efficiency of the search process.

8. Experiments

We implemented the algorithm in Section 6 and Section 7 on top of OSTRICH, resulted to a string solver called OSTRICH$^{\text{RECL}}$. In this section, we evaluate the performance of OSTRICH$^{\text{RECL}}$ on three benchmark suites, that is, RegCoL, AutomatArk, and ComplexCounting. In the sequel, we first describe the three benchmark suites as well as the experiment setup. Then we present the experiment results. We evaluate the performance and correctness of our solver against the state-of-the-art string constraint solvers, including CVC5 [3], Z3seq [1], Z3str3 [6], Z3str3RE [8], and OSTRICH [11]. We also compare OSTRICH$^{\text{RECL}}$ with two variants of it, that is, OSTRICH$^{\text{RECL}}_{-\text{ASR}}$ and OSTRICH$^{\text{RECL}}_{\text{NUXMV}}$, to justify the technical choices made in the decision procedure of Section 6.2.

8.1. Benchmark suites and experiment setup

Our experiments utilize two benchmark suites, namely, *RegCoL* and *AutomatArk*. Other industrial benchmark suites are not utilized because they contain no counting operators. There are **DH:** todo **:HD** instances in total, and all benchmark instances are in the SMTLIB2 format.

RegCoL benchmark suite. There are 40,628 RECL instances in the RegCoL suite. These instances are generated by extracting regexes with counting operators from the open source regex library [36, 26] and manually constructing a RECL constraint $x \in e \wedge x \in e_{sani} \wedge |x| > 10$ for each regex e, where $e_{sani} \equiv \overline{\Sigma^*(< + > +' +'' +\&)\Sigma^*}$ is a regular expression that sanitizes all occurrence of special characters $<$, $>$, $'$, $''$, or &. The expression e_{sani} is introduced in view of the fact that these characters are usually sanitized in Web browsers to alleviate the XSS attacks [16, 37].

AutomatArk benchmark suite. This benchmark suite is adapted from the AutomatArk suite [38] by picking out the string constraints containing counting

operators. We also add the length constraint $|x| > 10$ for each string variable x. There are 8,215 instances in the AutomatArk suite. Note that the original AutomatArk benchmark suite [38] includes 19,979 instances, which are conjunctions of regular membership queries generated out of regular expressions in [39].

ComplexCounting benchmark suite. This benchmark suite is generated by the extension of StringFuzz [**?**]. We extend the StringFuzz tool with counting operators and generate 1,000 instances in which half of them contain complement on the counting operators and the other half contain nested counting. The length bound is set to $|x| > 50$ for each string variable x.

Distribution of problem instances w.r.t. counting bounds. The distribution of problem instances w.r.t. the counting bounds in RegCoL and AutomatArk suites is shown in Fig 5, where the x-axis represents the counting bound and the y-axis represents the number of problem instances whose maximum counting bound is equal to the value of the x-axis. From Fig 5, we can see that while most problem instances contain only small bounds, there are still around 3,000 (about 6%) of them using large counting bounds (i.e. greater than or equal to 50).

Experiment setup. All experiments are conducted on CentOS Stream release 8 with 4 Intel(R) Xeon(R) Platinum 8269CY 3.10GHz CPU cores and 190 GB memory. We use the ZAligVinder framework [40] to execute the experiments, with a timeout of 60s for each instance.

8.2. Performance evaluation

We evaluate the performance of OSTRICH$^{\text{RECL}}$ against the state-of-the-art string constraint solvers, including CVC5 [3], Z3seq [1], Z3str3 [6], Z3str3RE [8], and OSTRICH [11], on RegCoL, AutomatArk and ComplexCounting benchmark suites. The experiment results can be found in Figure 1. Note that we take the results of CVC5 as the ground truth[2], and the results different from the ground truth are classified as *soundness error*. We can see that OSTRICH$^{\text{RECL}}$ solves almost all **DH:** todo **:HD** instances, except 182 of them, that is, it solves 48,662 instances correctly. The number is

[2]Initially, we used the majority vote of the results of the solvers as the ground truth. Nevertheless, on some problem instances, all the results of the three solvers in the Z3 family are wrong (after manual inspection), thus failing this approach on these instances.

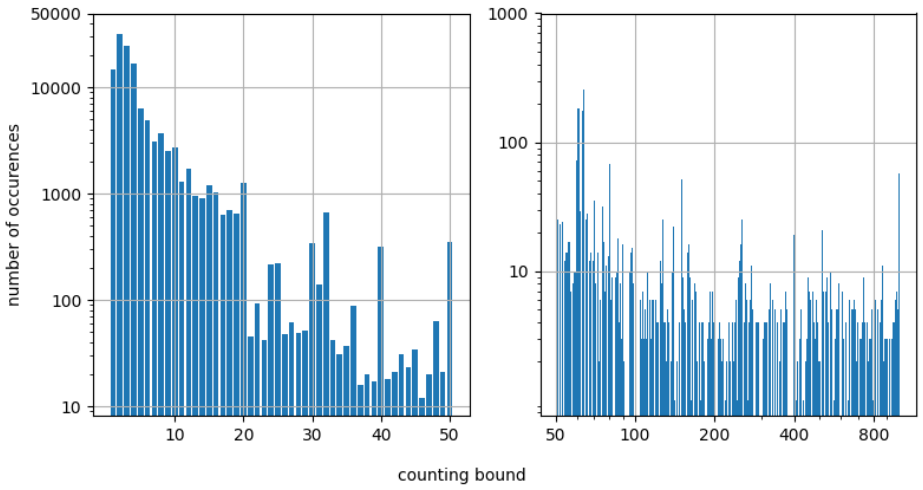

Figure 5: Distribution of problem instances w.r.t. counting bounds

3,908/1,111/12,838/2,306/2,396 more than the number of instances solved by CVC5/Z3str3RE/Z3str3/Z3seq/OSTRICH respectively. Moreover, $\text{OSTRICH}^{\text{RECL}}$ is the second fastest solver, whose average time on each instance is close to the fastest solver Z3str3RE (1.93s versus 1.62s).

8.3. Empirical justification of the technical choices and heuristics

We also do experiments to evaluate the effectiveness of the automata size-reduction techniques, that is, Step 2 in Section 6.2. Then we compare the performance of several heuristics in Section 7.

8.3.1. Empirical justification of the technical choices

We compare $\text{OSTRICH}^{\text{RECL}}$ with $\text{OSTRICH}^{\text{RECL}}_{-\text{ASR}}$ and $\text{OSTRICH}^{\text{RECL}}_{\text{NUXMV}}$, where $\text{OSTRICH}^{\text{RECL}}_{-\text{ASR}}$ is obtained from $\text{OSTRICH}^{\text{RECL}}$ by removing the automata size-reduction technique (i.e. Step 2 in Section 6.2), and $\text{OSTRICH}^{\text{RECL}}_{\text{NUXMV}}$ is obtained from $\text{OSTRICH}^{\text{RECL}}$ by using the nuXmv model checker to solve the nonemptiness of $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$.

The experiment results can be found in Table 2. We can see that $\text{OSTRICH}^{\text{RECL}}$ solves 1,503 more instances and is 2.21 times faster than $\text{OSTRICH}^{\text{RECL}}_{-\text{ASR}}$. Therefore, the automata size-reduction technique indeed plays an essential role in the performance improvement. Moreover, $\text{OSTRICH}^{\text{RECL}}$ solves 1,798 more instances and is 3.13 times faster than $\text{OSTRICH}^{\text{RECL}}_{\text{NUXMV}}$. Therefore, the

	CVC5	Z3str3RE	Z3str3	Z3seq	OSTRICH	OSTRICH$^{\text{RECL}}$
sat	27813	28283	23126	27761	25975	**28360**
unsat	16941	19312	12742	18651	20291	**20302**
unknown	**8**	99	6990	98	160	28
timeout	4081	1149	5985	2333	2417	**153**
soundness error	**0**	44	44	56	**0**	**0**
solved correctly	44754	47551	35824	46356	46266	**48662**
average time (s)	5.64	**1.62**	7.63	3.59	5.94	1.93

Table 1: Overall performance evaluation

decision procedure in Section 6.2 is more efficient to solve the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem than nuXmv.

	OSTRICH$^{\text{RECL}}_{-\text{ASR}}$	OSTRICH$^{\text{RECL}}_{\text{NUXMV}}$	OSTRICH$^{\text{RECL}}$
sat	26884	26603	**28360**
unsat	20275	20261	**20302**
unknown	48	45	**28**
timeout	1637	1935	**153**
soundness error	**0**	**0**	**0**
solved correctly	47159	46864	**48662**
average time (s)	4.27	6.05	**1.93**

Table 2: Empirical justification of the technical choices in the decision procedure

8.3.2. Empirical justification of the heuristics

We compare the performance of OSTRICH$^{\text{RECL}}$ with OSTRICH$^{\text{RECL}}_{-\text{NEST}}$, OSTRICH$^{\text{RECL}}_{-\text{COMP}}$, OSTRICH$^{\text{RECL}}_{-\text{FIND}}$, and OSTRICH$^{\text{RECL}}_{-\text{ALL}}$ where OSTRICH$^{\text{RECL}}_{-\text{NEST}}$

	CVC5	OSTRICH$^{\text{RECL}}_{-\text{ALL}}$	OSTRICH$^{\text{RECL}}_{-\text{COMP}}$	OSTRICH$^{\text{RECL}}_{-\text{NEST}}$	OSTRICH$^{\text{RECL}}_{-\text{FIND}}$	OSTRICH$^{\text{RECL}}$
sat	28282	28488	28714	29007	28735	29084
unsat	16809	20536	20536	20554	20554	20554
unknown	8	86	13	32	13	14
timeout	4744	733	580	250	541	191
soundness error	0	0	0	0	0	0
program crashes	0	0	0	0	0	0
Total correct	45091	49024	49250	49561	49289	49638
Time (s)	6.43	3.19	3.39	3.25	3.57	2.88
Time w/o timeouts (s)	0.79	2.34	2.72	2.96	2.95	2.66

Table 3: Empirical justification of the heuristics

is obtained from OSTRICH$^{\text{RECL}}$ by removing the heuristic for nested counting, OSTRICH$^{\text{RECL}}_{-\text{COMP}}$ is obtained from OSTRICH$^{\text{RECL}}$ by removing the heuristic for complement on counting operators, OSTRICH$^{\text{RECL}}_{-\text{FIND}}$ is obtained from OSTRICH$^{\text{RECL}}$ by removing the heuristic for finding the accepted word of the automaton, and OSTRICH$^{\text{RECL}}_{-\text{ALL}}$ is obtained from OSTRICH$^{\text{RECL}}$ by removing all the heuristics. The experiment results can be found in Table 3.

9. Conclusion

This work proposed an efficient automata-theoretical approach for solving string constraints with regex-counting and string-length. The approach is based on encoding counting operators in regular expressions by cost registers symbolically instead of unfolding them explicitly. Moreover, this work proposed automata-size reduction techniques to improve performance further. Finally, we used two benchmark suites comprising 48,843 instances in total to evaluate the performance of our approach. The experimental results show that our approach can solve more instances than the state-of-the-art best solvers, at a comparable or faster speed, especially when the counting and length bounds are large. For the future work, we plan to investigate how the symbolic approach can be extended to reason about nested counting operators.

References

[1] L. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 337–340.

[2] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, M. Deters, A DPLL(T) theory solver for a theory of strings and regular expressions, in: CAV, 2014, pp. 646–662.
URL https://doi.org/10.1007/978-3-319-08867-9_43

[3] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, CVC5: A versatile and industrial-strength smt solver, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, 2022, pp. 415–442.

[4] Y. Zheng, X. Zhang, V. Ganesh, Z3-str: a Z3-based string solver for web application analysis, in: ESEC/SIGSOFT FSE, 2013, pp. 114–124.
URL https://doi.org/10.1145/2491411.2491456

[5] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, X. Zhang, Effective search-space pruning for solvers of string equations, regular expressions and length constraints, in: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, Springer, 2015, pp. 235–254.
URL https://doi.org/10.1007/978-3-319-21690-4_14

[6] M. Berzish, V. Ganesh, Y. Zheng, Z3str3: A string solver with theory-aware heuristics, in: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017, 2017, pp. 55–59.
URL https://doi.org/10.23919/FMCAD.2017.8102241

[7] M. Berzish, Z3str4: A solver for theories over strings, Ph.D. thesis, University of Waterloo, Ontario, Canada (2021).
URL https://hdl.handle.net/10012/17102

[8] M. Berzish, J. D. Day, V. Ganesh, M. Kulczynski, F. Manea, F. Mora, D. Nowotka, Towards more efficient methods for solving regular-expression heavy string constraints, Theor. Comput. Sci. 943 (2023) 50–72.

[9] D. Bui, contributors, Z3-trau, https://github.com/diepbp/z3-trau (2019).

[10] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, J. Dolby, P. Janků, H.-H. Lin, L. Holík, W.-C. Wu, Efficient handling of string-number conversion, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 943957.
URL https://doi.org/10.1145/3385412.3386034

[11] T. Chen, M. Hague, A. W. Lin, P. Rümmer, Z. Wu, Decision procedures for path feasibility of string-manipulating programs with complex operations, PACMPL 3 (POPL) (Jan. 2019).
URL https://doi.org/10.1145/3290362

[12] H.-E. Wang, S.-Y. Chen, F. Yu, J.-H. R. Jiang, A symbolic model checking approach to the analysis of string and length constraints, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, ACM, 2018, p. 623633.
URL https://doi.org/10.1145/3238147.3238189

[13] C. Chapman, K. T. Stolee, Exploring regular expression usage and context in Python, in: A. Zeller, A. Roychoudhury (Eds.), Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, ACM, 2016, pp. 282–293.
URL https://doi.org/10.1145/2931037.2931073

[14] J. C. Davis, C. A. Coghlan, F. Servant, D. Lee, The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 246256.

[15] P. Wang, K. T. Stolee, How well are regular expressions tested in the wild?, in: G. T. Leavens, A. Garcia, C. S. Pasareanu (Eds.), Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, ACM, 2018, pp. 668–678.

[16] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song, A symbolic execution framework for JavaScript, in: 2010 IEEE Symposium on Security and Privacy, 2010, pp. 513–528.
URL https://doi.org/10.1109/SP.2010.38

[17] T. Liang, N. Tsiskaridze, A. Reynolds, C. Tinelli, C. W. Barrett, A decision procedure for regular membership and length constraints over unbounded strings, in: C. Lutz, S. Ranise (Eds.), Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings, Vol. 9322 of Lecture Notes in Computer Science, Springer, 2015, pp. 135–150.

[18] T. Chen, M. Hague, J. He, D. Hu, A. W. Lin, P. Rümmer, Z. Wu, A decision procedure for path feasibility of string manipulating programs with integer data type, in: D. V. Hung, O. Sokolsky (Eds.), Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings, Vol. 12302 of Lecture Notes in Computer Science, Springer, 2020, pp. 325–342.
URL https://doi.org/10.1007/978-3-030-59152-6_18

[19] W. Gelade, M. Gyssens, W. Martens, Regular expressions with counting: Weak versus strong determinism, SIAM J. Comput. 41 (1) (2012) 160–190.
URL https://doi.org/10.1137/100814196

[20] H. Chen, P. Lu, Checking determinism of regular expressions with counting, Inf. Comput. 241 (2015) 302–320.
URL https://doi.org/10.1016/j.ic.2014.12.001

[21] B. Loring, D. Mitchell, J. 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, ACM, 2019, pp. 425–438.
URL https://doi.org/10.1145/3314221.3314645

[22] T. Chen, A. Flores-Lamas, M. Hague, Z. Han, D. Hu, S. Kan, A. W. Lin, P. Rümmer, Z. Wu, Solving string constraints with regex-dependent functions through transducers with priorities and variables, Proc. ACM Program. Lang. 6 (POPL) (2022) 1–31.
URL https://doi.org/10.1145/3498707

[23] M. Kaminski, N. Francez, Finite-memory automata, in: Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science, 1990, pp. 683–688 vol.2. doi:10.1109/FSCS.1990.89590.

[24] L. D'Antoni, T. Ferreira, M. Sammartino, A. Silva, Symbolic register automata, in: I. Dillig, S. Tasiran (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2019, pp. 3–21.

[25] M. L. Minsky, Computation: Finite and Infinite Machines, Prentice-Hall Series in Automatic Computation, Prentice-Hall, 1967.

[26] L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, T. Vojnar, Regex matching with counting-set automata, Proc. ACM Program. Lang. 4 (OOPSLA) (nov 2020).
URL https://doi.org/10.1145/3428286

[27] L. Holík, J. Síc, L. Turonová, T. Vojnar, Fast matching of regular patterns with synchronizing counting, in: O. Kupferman, P. Sobocinski (Eds.), FoSSaCS 2023, Vol. 13992 of Lecture Notes in Computer Science, Springer, 2023, pp. 392–412.

[28] A. Le Glaunec, L. Kong, K. Mamouras, Regular expression matching using bit vector automata, Proc. ACM Program. Lang. 7 (OOPSLA1) (apr 2023).
URL https://doi.org/10.1145/3586044

[29] H. Seidl, T. Schwentick, A. Muscholl, P. Habermehl, Counting in trees for free, in: Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings, 2004, pp. 1136–1149.
URL https://doi.org/10.1007/978-3-540-27836-8_94

[30] K. N. Verma, H. Seidl, T. Schwentick, On the complexity of equational Horn clauses, in: CADE, 2005, pp. 337–352.

[31] C. Haase, A survival guide to Presburger arithmetic, ACM SIGLOG News 5 (3) (2018) 6782.
URL https://doi.org/10.1145/3242953.3242964

[32] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[33] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuXmv symbolic model checker, in: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, 2014, pp. 334–342.

[34] K. Thompson, Programming techniques: Regular expression search algorithm, Commun. ACM 11 (6) (1968) 419422. doi:10.1145/363347.363387.
URL https://doi.org/10.1145/363347.363387

[35] D. Hu, Z. Wu, String constraints withăregex-counting andăstring-length solved more efficiently, in: H. Hermanns, J. Sun, L. Bu (Eds.), Dependable Software Engineering. Theories, Tools, and Applications, Springer Nature Singapore, Singapore, 2024, pp. 1–20.

[36] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, D. Lee, Why arent 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, Association for Computing Machinery, New York, NY, USA, 2019, p. 443454.
URL https://doi.org/10.1145/3338906.3338909

[37] T. Chen, Y. Chen, M. Hague, A. W. Lin, Z. Wu, What is decidable about string constraints with the replaceall function, PACMPL 2 (POPL) (2018) 3:1–3:29. doi:10.1145/3158091.

[38] M. Berzish, M. Kulczynski, F. Mora, F. Manea, J. D. Day, D. Nowotka, V. Ganesh, An SMT solver for regular expressions and linear arithmetic

over string length, in: A. Silva, K. R. M. Leino (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2021, pp. 289–312.

[39] L. D'Antoni, Automatark: Automata benchmark (2018).
URL https://github.com/lorisdanto/automatark

[40] M. Kulczynski, F. Manea, D. Nowotka, D. B. Poulsen, Zaligvinder: A generic test framework for string solvers, Journal of Software: Evolution and Process 35 (4) (2023) e2400. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2400.
URL https://doi.org/10.1002/smr.2400

Algorithm 1 Find a word in the language of a CEFA

1: **procedure** FINDMODEL($\mathcal{A}, tmap$) ▷ $tmap$ maps transition to its visited times
2: $todo \leftarrow Stack(\mathcal{A}_{init}, tmap, "")$ ▷ todo is a stack of 3-tuple (state, tmap, word)
3: $visited \leftarrow \emptyset$ ▷ visited is a set of 2-tuple (state, tmap)
4: **while** $todo$ is not empty **do**
5: $(q, tmap, w) \leftarrow todo.pop()$
6: **if** $q \in \mathcal{A}_{final}$ and all times of transitions in $tmap$ are 0 **then**
7: **return** w
8: **end if**
9: **for** $(q, a, q', \vec{v}) \in \delta$ **do**
10: $tmap'[(q, a, q', \vec{v})] \leftarrow tmap[(q, a, q', \vec{v})] - 1$
11: **if** $tmap'[(q, a, q', \vec{v})] < 0$ **then**
12: **continue** ▷ Prune
13: **end if**
14: **if** $(q', tmap') \in visited$ **then**
15: **continue** ▷ Prune
16: **end if**
17: **if** there exisits t with $tmap[t] > 0$ not reachable from q' **then**
18: **continue** ▷ Prune
19: **end if**
20: $todo.push((q', tmap', w + a))$
21: $visited.add((q', tmap'))$
22: **end for**
23: **end while**
24: **return** $\emptyset$
25: **end procedure**