Journal of Systems Architecture 160 (2025) 103340

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

An efficient string solver for string constraints with regex-counting and string-length

Denghang Hu *, Zhilin Wu

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

ARTICLE INFO

Keywords:
String constraints
String solver
Regular expression
String length
Counting operator

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 efficiently, 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 various optimization techniques. We implement the algorithms and validate our approach on 49,843 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 or when the counting operators are nested with some other counting operators or complement operators.

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*. It analyzes the feasibility of execution paths in a program by encoding the paths as logical formulas and utilizing the constraint solvers (e.g. Z3 [1]) to solve the satisfiability of the formulas. 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 [4–7], 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 [13–15], regexes are used in about 30–40% 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. [8,17]).

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 although the Kleene star and the Kleene plus operator are special cases of counting, i.e., e^* is equivalent to $e^{\{0,\infty\}}$ and e^+ is equivalent to $e^{\{1,\infty\}}$, 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^+. Counting is a frequently used feature of regexes. According to the statistics from [13], counting occurs in more than 20% of 1204 Python projects. Therefore, an efficient analysis of string manipulating programs requires efficient solvers for string constraints containing regexes with counting and string-length function at least.

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 s,

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

* Corresponding author.
E-mail address: hudh@ios.ac.cn (D. Hu).

https://doi.org/10.1016/j.sysarc.2025.103340
Received 2 June 2024; Received in revised form 7 January 2025; Accepted 7 January 2025
Available online 17 January 2025

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 non-emptiness 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 non-emptiness 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. Moreover, we propose techniques to reduce the sizes (i.e., the number of states and transitions) of CEFA in order to achieve better performance.
- For complex RECL constraints where the counting operators may be nested with the other counting operators or complement operators, we propose optimization techniques to improve the performance. Furthermore, for the satisfiable RECL constraints where the solutions of the LIA formulas can be obtained, but the models of the original RECL constraint cannot be generated, we propose optimization techniques to accelerate the model generation process.
- We implement the algorithm on top of OSTRICH, resulting in a string solver[1] called OSTRICH$^{\text{RECL}}$. Finally, we utilize a collection of benchmark suites[2] comprising 49,843 problem 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 (see Tables 1–5 for details). For instance, OSTRICH$^{\text{RECL}}$ solves 49,638 out of 49,843 instances, while the best state-of-the-art solver solve only 47,721 instances. We also do experiments to justify some of our technical choices and evaluate the performance of the optimizations

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 non-emptiness 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. CEFAs can also be considered as a variation of Parikh automata [26] in which the numbers in the alphabet can be integers, instead of natural numbers. Counting-set automata were proposed in [27,28] to quickly match a subclass of regexes with counting. Moreover, a variant of nondeterministic counter automata, called bit vector automata, was proposed recently in [29] to enable fast matching of a more expressive class of regexes with counting. Nevertheless, the non-emptiness 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.

Structure. The rest of this paper is structured as follows: Section 2 introduces the preliminaries. Section 3 gives an overview of the approach in this paper. Section 4 presents the syntax and semantics of RECL. Section 5 defines CEFA. Section 6 introduces the algorithm to solve RECL constraints. Section 7 presents some heuristics for improving the performance of solving RECL constraints where the counting operators are nested with some other counting operators or complement operators. Section 9 shows the experiment results. Section 10 concludes this paper.

A preliminary version of this paper has appeared in the proceedings of SETTA 2023 [30]. This paper extends the SETTA paper in the following three aspects.

- To improve the performance of complex RECL constraints where the counting operators are nested with some other counting operators or complement operators, various optimizations are proposed.
- To deal with the issue that the satisfiability of LIA formulas can be solved, but models of the original RECL constraints cannot be generated, we propose techniques to optimize the model generation process.
- The benchmarks are extended to cover the situations where the counting operators are nested with some other counting operators or complement operators, and the experiments are extended accordingly.

2. 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 $x' \in X'$.

[1] The solver is open source and available at https://github.com/SimpleXiaohu/ostrich/tree/jsa2024

[2] The benchmarks are available at https://github.com/SimpleXiaohu/ostrich/tree/jsa2024/zaligvinder/models/jsa2024.

Table 1
Overall performance evaluation.

	CVC5	Z3str3RE	Z3str3	Z3seq	OSTRICH	OSTRICH$^{\text{RECL}}$
sat	28,282	28,449	23,322	28,212	25,975	**29,084**
unsat	16,809	19,316	12,742	18718	20,529	**20,554**
unknown	**8**	134	7021	203	162	14
timeout	4744	1944	6758	2710	3177	**191**
error	**0**	44	44	56	**0**	**0**
solved	45,091	47,721	36,020	46,874	46,504	**49,638**
average time (s)	6.43	**1.99**	8.48	4.04	6.75	2.88

Table 2
Performance evaluation on the NestC benchmark suite.

	CVC5	Z3str3RE	Z3str3	Z3seq	OSTRICH	OSTRICH$^{\text{RECL}}$
sat	508	166	196	451	0	**635**
unsat	0	4	0	67	**238**	**238**
unknown	**0**	35	31	105	2	6
timeout	492	795	773	377	760	**121**
error	**0**	**0**	**0**	**0**	**0**	**0**
solved	508	170	196	518	238	**873**
average time (s)	31.57	20.09	50.06	26.13	46.11	**19.49**

Table 3
Evaluation of the technical choices and optimizations in Section 6.

	OSTRICH$^{\text{RECL}}_{\text{NUXMV}}$	OSTRICH$^{\text{RECL}}_{-\text{ASR}}$	OSTRICH$^{\text{RECL}}$
sat	26,607	27,295	**29,084**
unsat	20,499	20,513	**20,554**
unknown	45	68	14
timeout	2,693	1,968	**191**
error	**0**	**0**	**0**
solved	47,106	47,808	**49,638**
average time (s)	6.86	4.76	**2.88**

Table 4
Evaluation of the optimizations in Section 7.

	OSTRICH$^{\text{RECL}}_{-\text{NEST}}$	OSTRICH$^{\text{RECL}}_{-\text{COMP}}$	OSTRICH$^{\text{RECL}}$
sat	553	296	**635**
unsat	**238**	**238**	**238**
unknown	27	8	6
timeout	182	458	**121**
error	0	0	0
solved	791	534	**873**
average time (s)	19.95	37.49	**19.49**

Table 5
Evaluation of the optimizations for the model generation in Section 8.

	OSTRICH$^{\text{RECL}}_{-\text{MODEL}}$	OSTRICH$^{\text{RECL}}$
sat	28,735	**29,084**
unsat	**20,554**	**20,554**
unknown	**13**	14
timeout	541	**191**
error	**0**	**0**
solved	49,289	**49,638**
average time (s)	3.57	**2.88**

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 $1 \le i < j \le n$. 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 [31].

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 $\vec{\mathfrak{x}} = (\mathfrak{x}_1, \ldots, \mathfrak{x}_k)$ to denote an LIA formula φ whose free variables are from $\{\mathfrak{x}_1, \ldots, \mathfrak{x}_k\}$. 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 $\vec{\mathfrak{x}} = (\mathfrak{x}_1, \ldots, \mathfrak{x}_k)$ and $\vec{t} = (t_1, \ldots, t_k)$ 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 $\{\eta : \Sigma \to \mathbb{N} \mid \psi[(\eta(a)/\mathfrak{z}_a)_{a \in \Sigma}] \text{ is true}\} = \mathsf{parikh}(\mathcal{A})$.

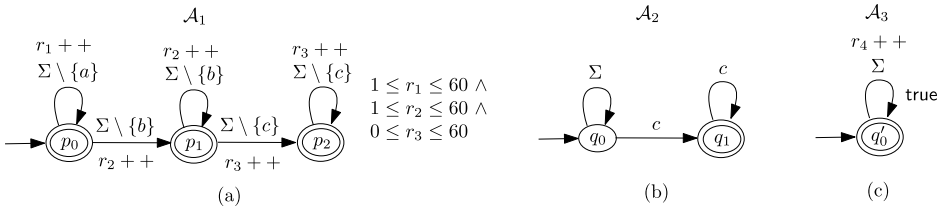

Fig. 1. CEFA for $(\Sigma\setminus a)^{\{1,60\}}(\Sigma\setminus b)^{\{1,60\}}(\Sigma\setminus c)^{\{0,60\}}$, Σ^*c^+, and $|x|$.

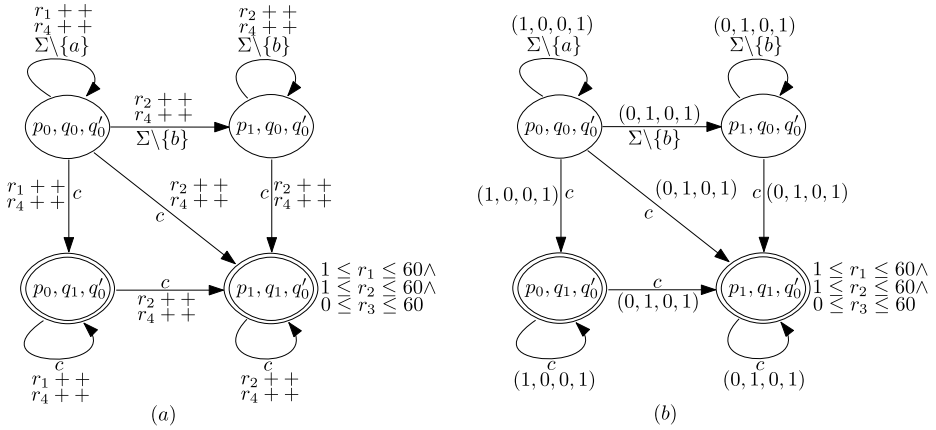

Fig. 2. $\mathcal{A}_1\cap\mathcal{A}_2\cap\mathcal{A}_3$: Intersection of $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$.

Theorem 1 ([32]). *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 $\mathcal{A}$.*

3. Overview

In this section, we utilize the string constraint in Eq. (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 $1\le r_1\le 60\wedge 1\le r_2\le 60\wedge 0\le r_3\le 60$, resulting in a CEFA $\mathcal{A}_1$ illustrated in Fig. 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 Σ^*c^+ (see Fig. 1(b)) and $\mathcal{A}_3$ for string length function (see Fig. 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.

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 Fig. 2(a), where the states that cannot 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 Fig. 2(b).

Finally, the satisfiability of the original string constraint is reduced to the non-emptiness 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 non-emptiness 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 Fig. 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.)

Let $\mathbb{C}=\{\vec{v}_1,\dots,\vec{v}_m\}$. From the results in [32,33], an existential LIA formula $\psi_{\mathcal{B}}(\mathfrak{x}_1,\dots,\mathfrak{x}_m)$ can be computed to define the Parikh image of strings that are accepted by $\mathcal{B}$, where $\mathfrak{x}_1,\dots,\mathfrak{x}_m$ are the integer variables to denote the number of occurrences of $\vec{v}_1,\dots,\vec{v}_m$. Therefore, the satisfiability of the string constraint in (1) is reduced to the satisfiability of the following existential LIA formula,

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

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 [34], 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 Fig. 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

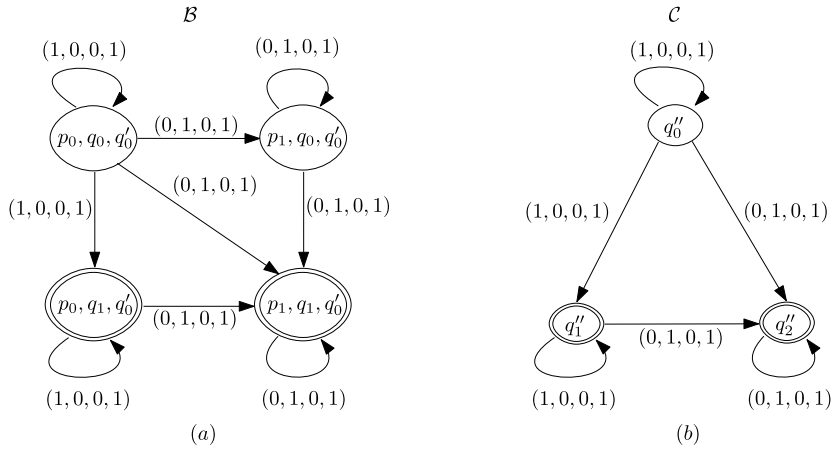

Fig. 3. Reduced automaton $\mathcal{B}$ and $\mathcal{C}$.

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

Fig. 4. Syntax of RECL.

$\vec{0}$-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 s.

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 Fig. 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 $-$.

Moreover, for $S \subseteq \Sigma$ with $S = \{a_1, \ldots, a_k\}$, 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 [31].

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 $\theta \vDash \varphi$, if one of the following holds: (1) $\varphi \equiv x \in e$ and $\theta(x) \in \mathcal{L}(e)$, (2) $\varphi \equiv t_1 \ o \ t_2$ and $\theta(t_1) \ o \ \theta(t_2)$, (3) $\varphi \equiv \varphi_1 \wedge \varphi_2$ and $\theta \vDash \varphi_1$ and $\theta \vDash \varphi_2$. A RECL formula φ is satisfiable if there is a mapping θ such as $\theta \vDash \varphi$. 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

- $R = \{r_1, \ldots, r_k\}$ 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 $(r_1, \ldots, r_k)$. Accordingly, we write an element of $\mathbb{Z}^R$ as a vector $(v_1, \ldots, v_k)$, 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 $(\epsilon;\vec{0}) \in \mathcal{L}(\mathcal{A})$. 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 $\mathcal{A}_1 = (R_1, Q_1, \Sigma, \delta_1, I_1, F_1, \alpha_1)$ and $\mathcal{A}_2 = (R_2, Q_2, \Sigma, \delta_2, I_2, F_2, \alpha_2)$ 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 $\mathcal{A}_2$). Note that the fresh auxiliary registers are introduced to distinguish whether $\mathcal{A}_1$ or $\mathcal{A}_2$ accepts and weave the two accepting conditions of $\mathcal{A}_1$ and $\mathcal{A}_2$ into one accepting condition. 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. Note that an integer vector $\vec{v}_1$ in $\mathcal{A}_1$, is copied at most twice in $\mathcal{A}_1 \cup \mathcal{A}_2$, namely, $(\vec{v}_1, \vec{0}, 0, 0)$ and $(\vec{v}_1, \vec{0}, 1, 0)$. Similarly for the integer vectors in $\mathcal{A}_2$.

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 $(R', Q', \Sigma, \delta', I', F', \alpha')$, where $R' = R_1 \cup R_2$, $Q' = Q_1 \cup Q_2$, $I' = I_1$, $\alpha' = \alpha_1 \wedge \alpha_2$, $\delta' = \{(q_1, a, q'_1, (\vec{v}_1, \vec{0})) \mid (q_1, a, q'_1, \vec{v}_1) \in \delta_1\} \cup \{(q_2, a, q'_2, (\vec{0}, \vec{v}_2)) \mid (q_2, a, q'_2, \vec{v}_2) \in \delta_2\} \cup \{(q_1, a, q_2, (\vec{0}, \vec{v}_2)) \mid q_1 \in F_1, \exists q' \in I_2.\ (q', a, q_2, \vec{v}_2) \in \delta_2\}$, 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 $x_1, \ldots, x_n$ be string variables, $\Lambda_{x_1}, \ldots, \Lambda_{x_n}$ be nonempty sets of CEFA over the alphabet Σ with $\Lambda_{x_i} = \{\mathcal{A}_{i,1}, \ldots, \mathcal{A}_{i,l_i}\}$ for every $i \in [n]$ where the sets of registers $R_{\mathcal{A}_{1,1}}, \ldots, R_{\mathcal{A}_{1,l_1}}, \ldots, R_{\mathcal{A}_{n,1}}, \ldots, R_{\mathcal{A}_{n,l_n}}$ 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 $\Lambda_{x_1}, \ldots, \Lambda_{x_n}$ are said to be nonempty w.r.t. φ if there are assignments $\theta : \{x_1, \ldots, x_n\} \to \Sigma^*$ 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 [35] 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 $x_1, \ldots, x_n$ 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 largely follows Thompson's construction [36] but we avoid using the ϵ-transitions. Similar ideas of representing counting operators as counters in automata have appeared in e.g. [27].

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 and only 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 $\{4,\infty\}$ 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
$$\begin{aligned}\delta'' \quad = \quad & \{(q, a, q', (0)) \mid (q, a, q') \in \delta\} \cup \\ & \{(q_0, a, q', (1)) \mid \exists q_0' \in I.\ (q_0', a, q') \in \delta\} \cup \\ & \{(q, a, q', (1)) \mid q \in F, \exists q_0' \in I.\ (q_0', a, q') \in \delta\},\end{aligned}$$
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 convert 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 $(aa(\varepsilon + a + aa + aaa + aaaa) \cdot b^*)^{\{2,\infty\}}$.

For each $i \in [n]$, let $x_i \in e_{i,1}, \ldots, x_i \in e_{i,l_i}$ 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 $\Lambda_{x_i} = \{\mathcal{A}_{i,0}, \mathcal{A}_{i,1}, \ldots, \mathcal{A}_{i,l_i}\}$ for each $i \in [n]$, and $\varphi_2' \equiv \varphi_2[r_{1,0}/|x_1|, \ldots, r_{n,0}/|x_n|]$. Then the satisfiability of φ is reduced to the non-emptiness of CEFAs in $\Lambda_{x_1}, \ldots, \Lambda_{x_n}$ w.r.t. φ_2'.

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

In this section, we present a procedure to solve the $\mathsf{NE}_{\mathsf{LIA}}$(CEFA) problem: Suppose that $x_1, \ldots, x_n$ are mutually distinct string variables, $\Lambda_{x_1}, \ldots, \Lambda_{x_n}$ are nonempty sets of CEFA over the same alphabet Σ where $\Lambda_{x_i} = \{\mathcal{A}_{i,1}, \ldots, \mathcal{A}_{i,l_i}\}$ for every $i \in [n]$. Moreover, the sets of registers $R_{\mathcal{A}_{1,1}}, \ldots, R_{\mathcal{A}_{n,l_n}}$ 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 non-emptiness of CEFAs in $\Lambda_{x_1}, \ldots, \Lambda_{x_n}$ w.r.t. φ is reduced to the non-emptiness of CEFAs in $\Lambda'_{x_1}, \ldots, \Lambda'_{x_n}$ w.r.t. φ.

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

Observation 1. *CEFA in $\Lambda'_{x_1} = \{\mathcal{B}_1\}, \ldots, \Lambda'_{x_n} = \{\mathcal{B}_n\}$ are nonempty w.r.t. φ iff there are outputs $\vec{v}_1 \in \mathcal{O}(\mathcal{B}_1), \ldots, \vec{v}_n \in \mathcal{O}(\mathcal{B}_n)$ such that $\varphi[\vec{v}_1/R_{\mathcal{B}_1}, \ldots, \vec{v}_n/R_{\mathcal{B}_n}]$ 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 outputs $\vec{v}_1 \in \mathcal{O}(\mathcal{B}_1), \ldots, \vec{v}_n \in \mathcal{O}(\mathcal{B}_n)$ such that $\varphi[\vec{v}_1/R_{\mathcal{B}_1}, \ldots, \vec{v}_n/R_{\mathcal{B}_n}]$ is true,
- there are $\eta_1 \in \mathsf{parikh}(\mathcal{U}_{\mathcal{B}_1}), \ldots, \eta_n \in \mathsf{parikh}(\mathcal{U}_{\mathcal{B}_n})$ such that
$$\bigwedge_{i\in[n]} \alpha_i\left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_i}} \eta_i(\vec{v})\vec{v}/R_{\mathcal{B}_i}\right] \wedge \varphi\left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_1}} \eta_1(\vec{v})\vec{v}/R_{\mathcal{B}_1}, \ldots, \sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_n}} \eta_n(\vec{v})\vec{v}/R_{\mathcal{B}_n}\right]$$
is true.

Therefore, to solve the non-emptiness of CEFA in $\Lambda'_{x_1}, \ldots, \Lambda'_{x_n}$ w.r.t. φ, it is sufficient to compute the existential LIA formulas $\psi_{\mathcal{U}_{\mathcal{B}_1}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_1}}), \ldots, \psi_{\mathcal{U}_{\mathcal{B}_n}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_n}})$ 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
$$\bigwedge_{i\in[n]}\left(\psi_{\mathcal{U}_{\mathcal{B}_i}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_i}}) \wedge \alpha_i\left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_i}} \mathfrak{z}_{i,\vec{v}}\vec{v}/R_{\mathcal{B}_i}\right]\right) \wedge \varphi\left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_1}} \mathfrak{z}_{1,\vec{v}}\vec{v}/R_{\mathcal{B}_1}, \ldots, \sum_{\vec{v}\in\mathbb{C}_{\mathcal{B}_n}} \mathfrak{z}_{n,\vec{v}}\vec{v}/R_{\mathcal{B}_n}\right].$$
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 $\psi_{\mathcal{U}_{\mathcal{B}_1}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_1}}), \ldots, \psi_{\mathcal{U}_{\mathcal{B}_n}}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{B}_n}})$ are proportional to the sizes (more precisely, the alphabet size, the number of states and transitions) of NFA $\mathcal{U}_{\mathcal{B}_1}, \ldots, \mathcal{U}_{\mathcal{B}_n}$, 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 an LIA formula). 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 the $\mathsf{NE}_{\mathsf{LIA}}$(CEFA) problem is reduced to the satisfiability of the following LIA formula, □
$$\bigwedge_{i\in[n]}\left(\psi_{\mathcal{C}_i}(\mathfrak{Z}_{\mathbb{C}_{\mathcal{C}_i}}) \wedge \alpha_i\left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{C}_i}} \mathfrak{z}_{i,\vec{v}}\vec{v}/R_{\mathcal{B}_i}\right]\right) \wedge \varphi\left[\sum_{\vec{v}\in\mathbb{C}_{\mathcal{C}_1}} \mathfrak{z}_{1,\vec{v}}\vec{v}/R_{\mathcal{B}_1}, \ldots, \sum_{\vec{v}\in\mathbb{C}_{\mathcal{C}_n}} \mathfrak{z}_{n,\vec{v}}\vec{v}/R_{\mathcal{B}_n}\right]. \tag{3}$$

Step 4 (Solving the LIA formula). We utilize some off-the-shelf SMT solver (e.g. Z3) to solve the satisfiability of the LIA formula in Eq. (3). If the solver reports "sat" and returns a solution $\mathfrak{m}$, then we go to Step 5. Otherwise, the $\mathsf{NE}_{\mathsf{LIA}}$(CEFA) problem has no solution. □

Step 5 (Constructing a solution). Let $\mathfrak{m}$ be a solution of the LIA formula in Eq. (3). We want to construct from $\mathfrak{m}$ and the CEFAs $\mathcal{B}_i$ for $i \in [n]$,

a solution $\mathfrak{m}'$ of the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem. The construction of $\mathfrak{m}'$ is obtained by constructing $\mathfrak{m}'(x_i)$ for each $i \in [n]$. Let $i \in [n]$. We describe how to construct $\mathfrak{m}'(x_i)$. The free variables $\mathfrak{z}_{\mathbb{C}_{\mathcal{C}_i}}$ correspond to the number of occurrences of characters in $\mathbb{C}_{\mathcal{C}_i}$, i.e. the alphabet of the NFA $\mathcal{C}_i$. Moreover, in $\mathcal{C}_i$, the characters are just the integer vectors and the characters in the original CEFAs $\mathcal{B}_i$ used to construct $\mathcal{C}_i$ are ignored. The construction of $\mathfrak{m}'(x_i)$ is actually to solve the reachability problem in a finite transition system where

- the nodes are of the form $(q, (\mathfrak{n}_{i,\vec{v}})_{\vec{v}\in\mathbb{C}_{\mathcal{C}_i}})$, where $\mathfrak{n}_{i,\vec{v}}$ denotes the number of the remaining occurrences of the integer vector $\vec{v}$ in $\mathcal{B}_i$,
- the transitions are of the form

  $((q, (\mathfrak{n}_{i,\vec{v}})_{\vec{v}\in\mathbb{C}_{\mathcal{C}_i}}), (q, a, q', \vec{u}), (q', (\mathfrak{n}'_{i,\vec{v}})_{\vec{v}\in\mathbb{C}_{\mathcal{C}_i}}))$

  such that $(q, a, q', \vec{u})$ is a transition in $\mathcal{B}_i$, $\mathfrak{n}'_{i,\vec{u}} = \mathfrak{n}_{i,\vec{u}} - 1$, moreover, $\mathfrak{n}'_{i,\vec{v}} = \mathfrak{n}_{i,\vec{v}}$ for every $\vec{v} \neq \vec{u}$.

Moreover, the initial and final state of the reachability problem are set to be $(q_0, (\mathfrak{m}(\mathfrak{z}_{i,\vec{v}}))_{\vec{v}\in\mathbb{C}_{\mathcal{C}_i}})$ and $(q_f, (0, \ldots, 0))$ for some initial state q_0 and final state q_f in $\mathcal{B}_i$. If a path from the initial state to the final state is found, then the path gives us a desired string $\mathfrak{m}'(x_i)$. □

7. Optimizations for non-register-representable RECL constraints

In Section 6, to deal with the non-register-representable RECL constraints, we unfold all the non-register-representable occurrences of counting operators. Nevertheless, the naive unfolding incurs an exponential blow-up over the formula sizes, especially when the counting bounds are large. Recall that an occurrence of counting operators is said to be non-register-representable if it is in the scope of some Kleene star, another counting operator, complement, or language difference. Since Kleene star can be seen as a special case of the counting operators, and language difference $e_1 \setminus e_2$ can be rewritten as $e_1 \cap \overline{e_2}$, without loss of generality, we can assume that a non-register representable occurrence of a counting operator is in the scope of another counting operator or some complement operator. In the sequel, to alleviate the formula-size blow-up problem resulted from the naive unfolding, we present various optimizations, for the nesting of counting operators, and the nesting of counting operators and a complement operator respectively.

7.1. Optimizations for the nesting of counting operators

In the sequel, we propose an optimization for the nesting of counting operators. The optimization is designed for the special case that the nesting depth of counting operators is one. For a RECL constraint whose nesting depth of counting operators is greater than two, we can apply the optimization for the inner-most nesting of counting operators and decrement the nesting depth, then apply the optimization again to the resulting constraint, and continue this process, until obtaining a register-representable RECL constraint.

The main idea of the optimization is illustrated by the constraint $x \in (a^{\{1,1000\}})^{\{1,2\}}$. If we apply the naive unfolding in Section 6, we will unfold $a^{\{1,1000\}}$ and obtain the constraint $x \in (a(\varepsilon + a + aa + \cdots + \underbrace{a \cdots a}_{999}))^{\{1,2\}}$. It is easy to observe that a smarter way is to unfold $^{\{1,2\}}$, instead of $^{\{1,1000\}}$. If we do this, then we get a constraint $x \in (a^{\{1,1000\}})(\varepsilon + a^{\{1,1000\}})$, whose size is much smaller, compared to $x \in (a(\varepsilon + a + aa + \cdots + \underbrace{a \cdots a}_{999}))^{\{1,2\}}$. (Recall that we assume a binary encoding of the integer constants in counting operators.)

Let us describe the optimization more precisely.

- For a regular expression $e = (e_1)^{\{m,n\}}$ where e_1 is a register-representable regular expression, we can unfold the counting operators in the following ways to obtain a register-representable RECL constraint: (i) unfold the counting operators in e_1, (ii) unfold the counting operator $^{\{m,n\}}$. Let $\mathsf{ufld}_1(e)$ and $\mathsf{ufld}_2(e)$ denote the resulting regular expressions. Note that in $\mathsf{ufld}_1(e)$, all the counting operators in e_1 (if there is any) disappear, while in $\mathsf{ufld}_2(e)$, all the counting operators in e_1 are preserved and copied for n times.
- In the sequel, we estimate the sizes (the number of transitions) of the NFAs $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_1(e)}}$ and $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_2(e)}}$, denoted by sz_1 and sz_2. If $sz_1 \leq sz_2$, then we choose to unfold the counting operators in e_1. Otherwise, we choose to unfold the counting operator $^{\{m,n\}}$.
- To estimate the sizes of $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_1(e)}}$ and $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_2(e)}}$, we first estimate the sizes of the CEFAs $\mathcal{A}_{\mathsf{ufld}_1(e)}$ and $\mathcal{A}_{\mathsf{ufld}_2(e)}$ as two pairs $(\mathsf{snum}_1(e), \mathsf{vnum}_1(e))$ and $(\mathsf{snum}_2(e), \mathsf{vnum}_2(e))$, where $\mathsf{snum}_1(e)$ and $\mathsf{vnum}_1(e)$ are the estimates of the number of states and integer vectors in $\mathcal{A}_{\mathsf{ufld}_1(e)}$ separately, similarly for $(\mathsf{snum}_2(e), \mathsf{vnum}_2(e))$. Since the NFAs $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_1(e)}}$ and $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_2(e)}}$ are constructed from $\mathcal{A}_{\mathsf{ufld}_1(e)}$ and $\mathcal{A}_{\mathsf{ufld}_2(e)}$ by dropping the accepting conditions and ignore characters, $\mathsf{snum}_1(e)$ and $\mathsf{vnum}_1(e)$ are the estimates of the number of states and characters of $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_1(e)}}$, similarly for $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_2(e)}}$. Then we estimate the sizes (number of transitions) of the NFAs $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_1(e)}}$ and $\mathcal{U}_{\mathcal{A}_{\mathsf{ufld}_2(e)}}$ as $sz_1 = (\mathsf{snum}_1(e))^2 \cdot \mathsf{vnum}_1(e)$ and $sz_2 = (\mathsf{snum}_2(e))^2 \cdot \mathsf{vnum}_2(e)$ respectively.

It remains to show how to compute $(\mathsf{snum}_1(e), \mathsf{vnum}_1(e))$ and $(\mathsf{snum}_2(e), \mathsf{vnum}_2(e))$ from $e = (e_1)^{\{m,n\}}$. Let $e = (e_1)^{\{m,n\}}$ be a regular expression where e_1 is a register-representable regular expression. Note that there is only one register in $\mathcal{A}_{\mathsf{ufld}_1(e)}$. As a result, we set $\mathsf{vnum}_1(e) = 2$. Moreover, in the sequel, we inductively compute $\mathsf{snum}'_1(e_1)$ and $(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1))$ from e_1, then let $\mathsf{snum}_1(e) = \mathsf{snum}'_1(e_1)$ and $(\mathsf{snum}_2(e), \mathsf{vnum}_2(e)) = (\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1))$.

- If $e_1 = \emptyset$ or $e_1 = \epsilon$, then $\mathsf{snum}'_1(e_1) = 0$ and

  $(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1)) = (0, 0)$.

  (Note that here the constant 0, instead of 1, is chosen, so that later on, when we consider $e_1 = e_2 \cap e_3$, $\mathsf{snum}'_2(e_2) \times \mathsf{snum}'_2(e_3)$ and $\mathsf{vnum}'_2(e_2) \times \mathsf{vnum}'_2(e_3)$ will become 0 since $\epsilon \cap e_3 = \epsilon$ and $\emptyset \cap e_3 = \emptyset$.)

- If $e_1 = a$, then $\mathsf{snum}'_1(e_1) = 2$ and

  $(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1)) = (2n, 1)$.

- If $e_1 = e_2 \cdot e_3$, then $\mathsf{snum}'_1(e_1) = \mathsf{snum}'_1(e_2) + \mathsf{snum}'_1(e_3)$ and

  $(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1)) = (n(\mathsf{snum}'_2(e_2) + \mathsf{snum}'_2(e_3)), n(\mathsf{vnum}'_2(e_2) + \mathsf{vnum}'_2(e_3)))$.

  (Note that because e_1 is concatenated for n times in $\mathsf{ufld}_2(e)$, we define $\mathsf{snum}'_2(e_1)$ as $n(\mathsf{snum}'_2(e_2) + \mathsf{snum}'_2(e_3))$. Moreover, since the integer vectors in $\mathcal{A}_1 \cdot \mathcal{A}_2$ are of the form $(\vec{v}_1, \vec{0})$ and $(\vec{0}, \vec{v}_2)$ where $\vec{v}_1$ and $\vec{v}_2$ are integer vectors in $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, we define $\mathsf{vnum}'_2(e_1)$ as $n(\mathsf{vnum}'_2(e_2) + \mathsf{vnum}'_2(e_3)))$.
- If $e_1 = e_2 + e_3$, then $\mathsf{snum}'_1(e_1) = \mathsf{snum}'_1(e_2) + \mathsf{snum}'_1(e_3)$ and

  $(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1)) = (n(\mathsf{snum}'_2(e_2) + \mathsf{snum}'_2(e_3) + 1), n(2\mathsf{vnum}'_2(e_2) + 2\mathsf{vnum}'_2(e_3)))$.

  (Recall that each integer vector in $\mathcal{A}_1$ resp. $\mathcal{A}_2$ is copied at most twice in the construction of $\mathcal{A}_1 \cup \mathcal{A}_2$.)
- If $e_1 = e_2 \cap e_3$, then $\mathsf{snum}'_1(e_1) = \mathsf{snum}'_1(e_2) \times \mathsf{snum}'_1(e_3)$ and

  $(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1)) = (n(\mathsf{snum}'_2(e_2) \times \mathsf{snum}'_2(e_3)), n(\mathsf{vnum}'_2(e_2) \times \mathsf{vnum}'_2(e_3)))$.

  (Recall that the integer vectors in $\mathcal{A}_1 \cap \mathcal{A}_2$ are of the form $(\vec{v}_1, \vec{v}_2)$, where $\vec{v}_1$ and $\vec{v}_2$ are the integer vectors in $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively.)
- If $e_1 = \overline{e_2}$, then $\mathsf{snum}'_1(e_1) = 2^{\mathsf{snum}'_1(e_2)}$ and

  $(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1)) = (n \cdot 2^{\mathsf{snum}'_2(e_2)}, 1)$.

(In this case, since e_1 is register-representable, we know that e_2 contains no counting operators. As a result, we first construct the NFA for e_2, then determinize and complement it to construct the automaton for e_1.)

- If $e_1 = (e_2)^*$, then $\mathsf{snum}'_1(e_1) = \mathsf{snum}'_1(e_2)$ and

$$(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1)) = (n\mathsf{snum}'_2(e_2), 1).$$

(In this case, since e_1 is register-representable, we know that e_2 contains no counting operators.)

- If $e_1 = (e_2)^{\{n',n'\}}$ or $e_1 = (e_2)^{\{n',\infty\}}$, then $\mathsf{snum}'_1(e_1) = n'\mathsf{snum}'_1(e_2)$ and

$$(\mathsf{snum}'_2(e_1), \mathsf{vnum}'_2(e_1)) = (n\mathsf{snum}'_2(e_2), n).$$

(In this case, since e_1 is register-representable, we know that e_2 contains no counting operators.)

7.2. Optimizations for the nesting of counting operators and a complement operator

As mentioned in Section 6, if some counting operators are in the scope of a complement operator, it is necessary to unfold these counting operators before the complement operation. The exponential blow-up in the unfolding combined with the blow-up in the complement operation will produce a huge number of states. To alleviate this problem, we propose two approximations of the complement operation, i.e., an under-approximation and an over-approximation, and combine them into a procedure that achieves a nice balance between efficiency and precision.

Let $e = \overline{e_1}$ be a regular expression, where e_1 is a register-representable regular expression.

Under-approximation. One way of doing the complement operation is to transform e_1 into a regular expression that does not contain counting operators. We compute an under-approximation of e by replacing each occurrence of the counting operators in e_1, say $(e_2)^{\{m,n\}}$ or $(e_2^{\{n,\infty\}})$, by e_2^*. Let us use e'_1 to denote the resulting regular expression. Note that e'_1 does not contain counting operators. Moreover, it is easy to observe that $\mathcal{L}(e_1) \subseteq \mathcal{L}(e'_1)$. As a result, we have $\mathcal{L}(\overline{e'_1}) \subseteq \mathcal{L}(\overline{e_1})$. That is, $\overline{e'_1}$ is an under-approximation of $\overline{e_1}$.

Over-approximation. For $e = \overline{e_1}$, we utilize the CEFA constructed from e_1, say $\mathcal{A}_{e_1}$, to construct a CEFA $\mathcal{B}$ of e so that $\mathcal{L}(\mathcal{B})$ is an over-approximation of $\mathcal{L}(e)$. Let $\mathcal{A}_{e_1} = (R, Q, \Sigma, \delta, I, F, \alpha)$. The main idea of the construction is explained as follows: If a string w is not accepted by $\mathcal{A}_{e_1}$, then either there are no runs of $\mathcal{A}_{e_1}$ on w, or every run of $\mathcal{A}_{e_1}$ on w stop at a non-final state or enter a final state but do not satisfy the accepting condition α. We shall construct a CEFA $\mathcal{B}$ that accepts when one of these situations occurs. Then $\overline{\mathcal{L}(\mathcal{A}_{e_1})} \subseteq \mathcal{L}(\mathcal{B})$, that is, $\mathcal{B}$ is an over-approximation of e.

Without loss of generality, we can assume that there are no transitions out of F in $\mathcal{A}_{e_1}$. If this is not the case, then a new state q_f can be introduced to transform $\mathcal{A}_{e_1}$ into a CEFA satisfying this constraint, where q_f is the unique accepting state. Then we define $\mathcal{B} = (R \cup \{r'\}, Q \cup \{q_s\}, \Sigma, \delta', I, Q \cup \{q_s\}, \alpha')$, where

- r' is a new register not in R,
- q_s is a new state not in Q,
- δ' comprises the following transitions,
  - $(q, a, q', (\vec{v}, 0))$ such that $(q, a, q', \vec{v}) \in \delta$ and $q' \notin F$,
  - $(q, a, q_s, (\vec{0}, 0))$ such that there do *not* exist a-labeled transitions out of q, that is, for every q' and $\vec{v}$, we have $(q, a, q', \vec{v}) \notin \delta$,
  - $(q, a, q', (\vec{v}, 1))$ such that $(q, a, q', \vec{v}) \in \delta$ and $q' \in F$,
  - $(q_s, a, q_s, (\vec{0}, 0))$ such that $a \in \Sigma$,
- α' is defined as $r' = 1 \to \neg\alpha$.

Intuitively, q_s is a sink state that deals with the situations where there are no runs, and r' is a register newly introduced so that the different cases of the non-accepting runs of $\mathcal{A}_{e_1}$ can be captured by a single accepting condition α'.

Note that $\mathcal{B}$ is a strict over-approximation of $e = \overline{e_1}$ since $\mathcal{A}_{e_1}$ may be nondeterministic. For instance, if there are two runs of $\mathcal{A}_{e_1}$ on a string w, say one run that ends at a non-final state and another run that ends at a final state and satisfies accepting condition α. Then $w \in \mathcal{L}(\mathcal{A}_{e_1})$, thus $w \notin \mathcal{L}(\overline{\mathcal{A}_{e_1}})$. Nevertheless, according to the construction, w is accepted by $\mathcal{B}$, witnessed by the run of $\mathcal{A}_{e_1}$ that ends at a non-final state.

Combination of under- and over-approximation. We combine the under- and over-approximation as well as the decision procedure into the following procedure.

1. We first apply the under-approximation.
2. If the under-approximation returns "sat", then we know that the original constraint is satisfiable. Otherwise, the under-approximation returns "unsat" and we apply the over-approximation.
3. If the over-approximation returns "unsat", then we know that the original constraint is unsatisfiable. Otherwise, the over-approximation returns "sat" and we resort to the decision procedure (without approximations) in Section 6 to solve the RECL constraint.

8. Optimizations for model generation

Recall that in Step 5 of Section 6, we construct a solution $\mathfrak{m}'$ from the model $\mathfrak{m}$ of the LIA formula in Eq. (3) by solving the reachability problems in a finite transition system. It is easy to see that the transition system contains exponentially many nodes in general and a naive search over the transition system is not efficient. In the sequel, we describe two optimizations so that the reachability problem can be solved more efficiently.

To ease the presentation, let us assume that n, i.e. the number of string variables, to be 1.

At first, we refine the states of the transition system by recording the number of transitions, instead of just the number of occurrences of the integer vectors.

Refining the states of the transition system by recording the number of transitions. Let θ denote the formula in Eq. (3). We shall construct the formula θ' so that any model of θ' necessarily encodes the number of occurrences of transitions within $\mathcal{B}_1$.

$$\theta' \equiv \psi_{\mathcal{B}_1}(\mathfrak{Z}_{\mathcal{B}_1}) \wedge \bigwedge_{\vec{v} \in \mathbb{C}_{\mathcal{C}_1}} \mathfrak{z}_{1,\vec{v}} = \mathfrak{m}(\mathfrak{z}_{1,\vec{v}}) \wedge \bigwedge_{\vec{v} \in \mathbb{C}_{\mathcal{C}_1}} \mathfrak{z}_{1,\vec{v}} = \sum_{(q,a,q') \text{ s.t. } (q,a,q',\vec{v}) \text{ is a transition in } \mathcal{B}_1} \mathfrak{z}_{1,(q,a,q',\vec{v})},$$

where $\mathfrak{Z}_{\mathcal{B}_1} = \{\mathfrak{z}_{1,(q,a,q',\vec{v})} \mid (q, a, q', \vec{v}) \text{ is a transition in } \mathcal{B}_1\}$ is the set of variables $\mathfrak{z}_{1,(q,a,q',\vec{v})}$ denoting the number of occurrences of $(q, a, q', \vec{v})$. Note that compared to θ, θ' uses the LIA formula $\psi_{\mathcal{B}_1}$, which is constructed from $\mathcal{B}_1$ by ignoring its accepting condition and taking it as an NFA where the alphabet is seen as the set of transitions. Furthermore, θ' relates the values of the variables $\mathfrak{z}_{1,\vec{v}}$ and $\mathfrak{z}_{1,(q,a,q',\vec{v})}$, while simultaneously constraining the values of $\mathfrak{z}_{1,\vec{v}}$ to be $\mathfrak{m}(\mathfrak{z}_{1,\vec{v}})$. We then submit θ' to SMT solver, which returns a model of θ', in particular, the values of the variables $\mathfrak{z}_{1,(q,a,q',\vec{v})}$. Equipped with the model, we refine the transition system by considering the nodes $(q, (\mathfrak{n}_{1,(q,a,q',\vec{v})})_{(q,a,q',\vec{v})}$ is a transition in $\mathcal{B}_1)$, where $\mathfrak{n}_{1,(q,a,q',\vec{v})}$ denotes the number of remaining occurrences of $(q, a, q', \vec{v})$. Note that this refinement can avoid searching useless transitions. For instance, if $\mathfrak{n}_{1,(q,a,q',\vec{v})} = 0$ for some $(q, a, q', \vec{v})$, then we can avoid searching the transition $(q, a, q', \vec{v})$ out of q. This stands in contrast to the original transition system where $\mathfrak{n}_{1,\vec{v}}$ is used, it may be the case that we have

$\mathfrak{n}_{1,\vec{v}} > 0$ (since there may be multiple transitions associated with $\vec{v}$) and the transition $(q, a, q', \vec{v})$ should still be searched.

Moreover, we propose a method to reduce the number of transitions in the transition system further.

Pruning the transitions in the transition system. Let the current state of the transition system be $(p, (\mathfrak{n}_{1,(q,a,q',\vec{v})})_{(q,a,q',\vec{v})}$ is a transition in $\mathcal{B}_1$). Let $\mathcal{T}_p$ denote the set of transitions that are reachable from the state p in $\mathcal{B}_1$. If $\{(q, a, q', \vec{v}) \mid \mathfrak{n}_{1,(q,a,q',\vec{v})} > 0\} \nsubseteq \mathcal{T}_p$, in other words, there exists some transitions $(q, a, q', \vec{v})$ not reachable from the state p in $\mathcal{B}_1$ while $\mathfrak{n}_{1,(q,a,q',\vec{v})} > 0$ is established in the current state $(p, (\mathfrak{n}_{1,(q,a,q',\vec{v})})_{(q,a,q',\vec{v})}$ is a transition in $\mathcal{B}_1$), then we know that it is impossible to reach the final state $(q_f, (0, \ldots, 0))$ from the current state so that we can remove all the transitions out of the current state in the transition system.

9. Experiments

We implemented the procedures and optimizations in Sections 6–8 on top of OSTRICH, resulting in 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 NestC. 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 several variants of itself, to evaluate the effectiveness of the technical choices and optimizations in Sections 6–8.

9.1. Benchmark suites and the experiment setup

Our experiments utilize three benchmark suites, namely, *RegCoL*, *AutomatArk*, and *NestC*. There are 49,843 instances in total. All benchmark instances are in the SMTLIB2 format. In the sequel, we give more details of these three benchmark suites.

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 [27,37] and manually constructing a RECL constraint $x \in e \wedge x \in e_{sani} \wedge |x| > 10$ for each regex e, where $e_{sani} \equiv \overline{\Sigma^*(\langle + \rangle + ' + '' + \&)\Sigma^*}$ 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 [16,38].

AutomatArk benchmark suite. This benchmark suite is adapted from the AutomatArk suite [39] by picking out the string constraints containing counting operators. We also add the length constraint $|x| > 10$ for each string variable x. There are 8215 instances in the AutomatArk suite. Note that the original AutomatArk benchmark suite [39] includes 19,979 instances, which are conjunctions of regular membership queries generated out of regular expressions in [40].

NestC benchmark suite. We generate this benchmark suite by utilizing the string fuzzing tool StringFuzz [41]. Note that StringFuzz is slightly extended by adding the counting operators. For the benchmarks that contain the nestings of counting operators, we use the following template

$$x \in e_{nest} \wedge x \in e_{sani} \wedge |x| > 50$$

where e_{nest} is of the form

$$(e_1^{\{m_1,n_1\}} e_2^{\{m_2,n_2\}} e_3^{\{m_3,n_3\}} e_4^{\{m_4,n_4\}})^{\{m_5,n_5\}},$$

such that $0 \le m_i \le n_i \le 20$ for each $i \in [5]$ and e_i contains no counting operators for each $i \in [4]$, moreover,

$$e_{sani} \equiv \overline{\Sigma^*(\langle + \rangle + ' + '' + \&)\Sigma^*}.$$

Based on this template, we generate 500 instances.

For the benchmarks that contain the nestings of counting operators and a complement operator, we utilize the following template

$$x \in e_{comp1} \wedge x \in e_{comp2} \wedge x \in e_{sani} \wedge |x| > 50,$$

where both e_{comp1} and e_{comp2} are of the form $\overline{e_1^{\{m_1,n_1\}} e_2^{\{m_2,n_2\}} e_3^{\{m_3,n_3\}} e_4^{\{m_4,n_4\}}}$, such that for each $i \in [4]$, $20 \le m_i \le n_i \le 1000$ and e_i contains no counting operators. Note that two regular expressions e_{comp1} and e_{comp2} are introduced to have a better balance between the numbers of satisfiable and unsatisfiable instances. Based on this template, we also generate 500 instances.

In total, the NestC benchmark suite contains 1000 instances.

Note that the two templates are chosen so that on the one hand, the nestings of counting operators or nestings of counting operators and complement operators are available, and on the other hand, some additional regular and length constraints are introduced, so that the constraints are not trivially satisfied, in particular, the constraints are not satisfied by very short strings.

Distribution of problem instances w.r.t. counting bounds. The distribution of problem instances w.r.t. the counting bounds in the three suites is shown in Fig. 5, where the x-axis represents the counting bound and the y-axis represents the number of problem instances containing the counting bound. In our statistics, while most problem instances contain only small bounds, there are still around 3000 (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 3.10 GHz CPU cores and 190 GB memory. We use the ZALIGVINDER framework [42] to execute the experiments, with a timeout of 60 s per instance.

9.2. Performance evaluation against the other solvers

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 NestC benchmark suites. The experiment results can be found in Table 1. Note that we take the results of CVC5 as the ground truth,[3] and the results different from the ground truth are classified as *error*. We can see that OSTRICH$^{\text{RECL}}$ solves almost all instances, except 182 of them, that is, it solves 49,638 instances correctly. The number is 4,547/1,917/13,618/2,764/3,134 more than the number of instances solved by CVC5/Z3str3RE/Z3str3/Z3seq/OSTRICH respectively. Moreover, OSTRICH$^{\text{RECL}}$ is the second fastest solver, whose average time on each instance is close to the fastest solver Z3str3RE (2.88 s versus 1.99 s).

Furthermore, in order to see how OSTRICH$^{\text{RECL}}$ performs on the RECL constraints where the counting and length bounds are large or when the counting operators are nested with some other counting operators or with complement operators, we also compare OSTRICH$^{\text{RECL}}$ with the other solvers solely on the NestC benchmark suite. The experiment results can be found in Table 2. From the results, we can see that OSTRICH$^{\text{RECL}}$ outperforms the other solvers in both the number of solved instances and the average time. Specifically, OSTRICH$^{\text{RECL}}$ solves 873 out of 1000 instances, while the best state-of-the-art solver (Z3seq) solves only 518 instances. Moreover, while the average time of OSTRICH$^{\text{RECL}}$ is only slightly better than Z3str3RE (19.49 s versus 20.09 s), OSTRICH$^{\text{RECL}}$ solves much more instances than it (873 versus 170). The results show the great advantage of OSTRICH$^{\text{RECL}}$ against the other solvers on the NestC benchmark suite.

[3] 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.

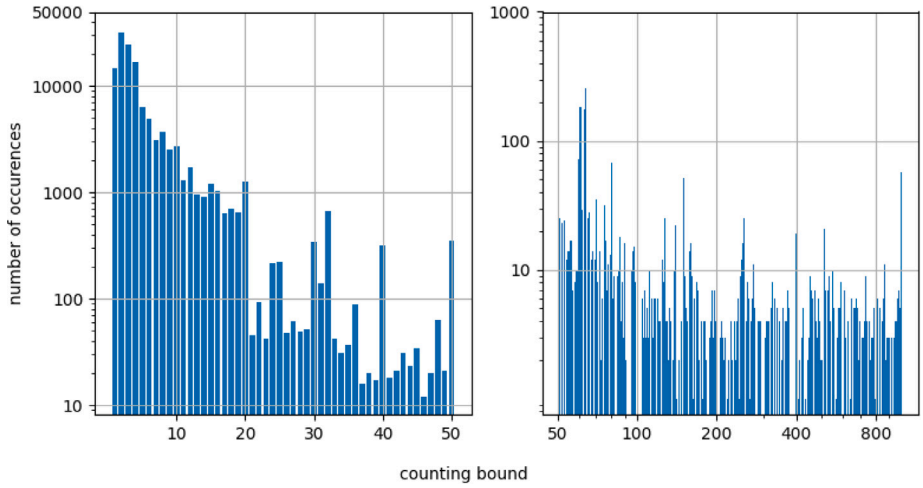

Fig. 5. Distribution of problem instances w.r.t. counting bounds.

9.3. Evaluation of the technical choices and optimizations in Sections 6 -8

At first, we do experiments to evaluate the effectiveness of the automata size-reduction techniques (i.e., Step 2) in Section 6. Then we justify our technical choices of reducing the satisfiability problem to that of LIA formulas and resort to the SMT solvers to solve them. Specifically, we compare OSTRICH$^{\text{RECL}}$ with a variant where the nuXmv model checker like [18] is used to solve the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem. We also evaluate the effectiveness of the optimizations in Section 7, i.e., the optimizations for the nesting of counting operators as well as the nesting of counting operators and a complement operator. Finally, we evaluate the effectiveness of the optimizations for the model generation in Section 8.

Evaluation of the technical choices and optimizations in Section 6. We compare OSTRICH$^{\text{RECL}}$ with OSTRICH$^{\text{RECL}}_{\text{NUXMV}}$, which is a variant of OSTRICH$^{\text{RECL}}$ where the nuXmv model checker is used to solve the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem. Moreover, we also compare OSTRICH$^{\text{RECL}}$ with OSTRICH$^{\text{RECL}}_{-\text{ASR}}$, which is obtained from OSTRICH$^{\text{RECL}}$ by removing the automata size-reduction technique (i.e. Step 2 in Section 6.2).

The experiment results can be found in Table 3. From the results, we can see that OSTRICH$^{\text{RECL}}$ solves 2532 more instances and is 2.38 times faster than OSTRICH$^{\text{RECL}}_{\text{NUXMV}}$. Therefore, the technical choice in Section 6 where LIA formulas are computed and then solved by the SMT solvers is reasonable, since OSTRICH$^{\text{RECL}}$ with this technical choice is more efficient than OSTRICH$^{\text{RECL}}_{\text{NUXMV}}$, where nuXmv is used to solve the $\mathsf{NE}_{\mathsf{LIA}}(\mathsf{CEFA})$ problem. Moreover, OSTRICH$^{\text{RECL}}$ solves 1083 more instances and is 1.65 times faster than OSTRICH$^{\text{RECL}}_{-\text{ASR}}$. From the results, we can see that the automata-size reduction technique indeed plays an important role for the performance improvement.

Evaluation of the optimizations in Section 7. We also compare the performance of OSTRICH$^{\text{RECL}}$ with its variants OSTRICH$^{\text{RECL}}_{-\text{NEST}}$ and OSTRICH$^{\text{RECL}}_{-\text{COMP}}$, where OSTRICH$^{\text{RECL}}_{-\text{NEST}}$ is obtained from OSTRICH$^{\text{RECL}}$ by removing the optimization for the nesting of counting operators, and OSTRICH$^{\text{RECL}}_{-\text{COMP}}$ is obtained by removing the optimization for the nesting of counting operators and a complement operator. Since these optimizations are not targeted for register-representable RECL constraints and there are very few non-register-representable RECL constraints in the RegCoL and AutomatArk benchmark suites, we restrict our attention to the NestC benchmark suite here.

The experiment results can be found in Table 4. From the results, we can see that on 1000 problem instances, OSTRICH$^{\text{RECL}}$ solves 82/339 more instances and is 1.02/1.92 times faster than OSTRICH$^{\text{RECL}}_{-\text{NEST}}$/ OSTRICH$^{\text{RECL}}_{-\text{COMP}}$ respectively. As a result, while both optimizations do play a role in the performance improvement in solving these complex RECL constraints, the optimization for the nesting of counting operators and a complement operator plays a greater role since it allows solving around 1/3 more instances.

Evaluation of the optimizations for the model generation in Section 8. We compare the performance of OSTRICH$^{\text{RECL}}$ with OSTRICH$^{\text{RECL}}_{-\text{MODEL}}$, where OSTRICH$^{\text{RECL}}_{-\text{MODEL}}$ is obtained from OSTRICH$^{\text{RECL}}$ by removing the optimizations for the model generation.

The experiment results can be found in Table 5. From the results, we can see that compared to OSTRICH$^{\text{RECL}}$, OSTRICH$^{\text{RECL}}_{-\text{MODEL}}$ solves 349 fewer instances and is 23% slower on average (3.57 s versus 2.88 s). As a result, the optimization for the model generation does improve the performance, especially when the sizes of the CEFAs are huge. Note that the optimization for model generation only affects the satisfiable instances, that is, those instances where the generated LIA formulas can be solved, but the generation of models for these RECL constraints is unsuccessful.

10. 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 various optimization techniques to improve the performance further. Finally, we used three benchmark suites comprising 49,843 instances in total to evaluate the performance of our approach. The experimental results show that our approach can solve more instances than the best state-of-the-art solver, at a comparable or faster speed, especially when the counting and length bounds are large or when counting operators are nested with some other counting operators or complement operators.

CRediT authorship contribution statement

Denghang Hu: Writing – original draft, Methodology. **Zhilin Wu:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] L.M. de Moura, N. 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, 2008, pp. 337–340, http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[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, http://dx.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, http://dx.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, http://dx.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, http://dx.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, Theoret. Comput. Sci. 943 (2023) 50–72.

[9] Diep Bui and contributors, Z3-trau, 2019, https://github.com/diepbp/z3-trau.

[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, in: PLDI 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 943–957, http://dx.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) (2019) http://dx.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, in: ASE 2018, ACM, 2018, pp. 623–633, http://dx.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, http://dx.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, in: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 246–256.

[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, http://dx.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, in: Lecture Notes in Computer Science, vol. 9322, 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, in: Lecture Notes in Computer Science, 12302, Springer, 2020, pp. 325–342, http://dx.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, http://dx.doi.org/10.1137/100814196.

[20] H. Chen, P. Lu, Checking determinism of regular expressions with counting, Inform. and Comput. 241 (2015) 302–320, http://dx.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, http://dx.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, http://dx.doi.org/10.1145/3498707.

[23] M. Kaminski, N. Francez, Finite-memory automata, in: Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science, Vol. 2, 1990, pp. 683–688, http://dx.doi.org/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, in: Prentice-Hall Series in Automatic Computation, Prentice-Hall, 1967.

[26] F. Klaedtke, H. Rueß, Monadic second-order logics with cardinalities, in: J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger (Eds.), Automata, Languages and Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 681–696.

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

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

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

[30] D. Hu, Z. Wu, String constraints with regex-counting and string-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.

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

[32] 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, http://dx.doi.org/10.1007/978-3-540-27836-8_94.

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

[34] C. Haase, A survival guide to Presburger arithmetic, ACM SIGLOG News 5 (3) (2018) 67–82, http://dx.doi.org/10.1145/3242953.3242964.

[35] 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.

[36] K. Thompson, Programming techniques: Regular expression search algorithm, Commun. ACM 11 (6) (1968) 419–422, http://dx.doi.org/10.1145/363347.363387.

[37] J.C. Davis, L.G. Michael IV, C.A. Coghlan, F. Servant, D. 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, in: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 443–454, http://dx.doi.org/10.1145/3338906.3338909.

[38] 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, http://dx.doi.org/10.1145/3158091.

[39] 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.

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

[41] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, V. Ganesh, StringFuzz: A fuzzer for string solvers, in: H. Chockler, G. Weissenbacher (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2018, pp. 45–51.

[42] M. Kulczynski, F. Manea, D. Nowotka, D.B. Poulsen, ZaligVinder: A generic test framework for string solvers, J. Software: Evol. Process. 35 (4) (2023) e2400, http://dx.doi.org/10.1002/smr.2400, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2400.