Efficient Methods for Solving String Constraints with Bounded Repetition

Denghang Hu[1] and Zhilin Wu[1]

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
{hudh,wuzl}@ios.ac.cn

Abstract. Regular expression with repetition frequently appears in the real world, such as $(ab)\{1, 100\}$. However, state-of-art string solvers can not efficiently solve string constraints with repetition bounds, especially when the repetition bounds are large. This paper focuses on string logic containing regular membership predicate with repetition and linear integer constraints on string length. Repetition is captured by cost-enriched finite automaton(CEFA), whose transitions have symbolic updates of integers limited by linear arithmetic. We propose a new systematic algorithm based on CEFA, and heuristic ways like under-approximation and symbolic-aware simplification are used to accelerate. We implement a powerful string solver OstrichCEA and evaluate it on instances from the real world and other typical benchmarks. The experiment result shows superiority.

Keywords: String Constraints, Automaton Theory, Regex Expression, Bounded Repetition, Linear Length Constraints

1 Introduction

String constraint solving is a subfield of computer science that deals with analyzing and manipulating strings, which are sequences of characters. String constraints solving aim to automatically infer properties of string variables, such as their length, content, and structure, to reason about the behavior of software systems that manipulate strings. It is crucial in security-critical applications, where strings represent sensitive data such as passwords, user input, and network addresses. String constraints solving combines techniques from formal methods, automata theory, and constraint solving and has many applications in areas such as software testing[5][39], program analysis[6][15][34], and malware detection[41][36][33]. In this context, efficient algorithms and tools for solving string constraints are crucial to improve the security and reliability of modern software systems.

Regular expressions are widely used in programming languages. About 30–40 % Java, JavaScript, and Python software use regex matching[23]. Bounded repetition is widely used in regex matching[18] and is dangerous because Regular expression Denial of Service (ReDoS) may derive from it[38]. We have analyzed

regular expressions sourced from the Internet[1][2], among which about 10% include bounded repetition. Operations like `length` that take string inputs and return an integer frequently appear in real-world JavaScript applications (78% of string operations in 18 applications[33]). These statistics show that a resultful string solver reasoning about regular membership predicate with bounded repetition and `length` operation is required. However, when the string constraints contain bounded repetition and lower bound on string variables, the state-of-art string solvers ([35][11][31][12][8][20][3][4]) lose efficiency. For example, CVC5[8] (which is the winner of QF_Strings track in SMT-COMP 2022[9]) failed to solve string constraints (1) with timeout 60s.

$$x \in \Sigma_{/a}\{1, 300\} \wedge x \in \Sigma^* a \Sigma^* \tag{1}$$

$\Sigma_{/a}\{1, 300\}$ repeats a letter excluding a from one time to 300 times. $\Sigma^* a \Sigma^*$ accepts a string containing a. It is easy to see that the string constraints are unsat. Nevertheless, CVC5 is hard to find a solution for it can not answer unsat until attempting all strings with lengths less than or equal to 300. The number of the searched string is exponential. Z3str3RE[12] can not solve string constraints (2) because its length abstraction is over-approximation.

$$x \in \Sigma_{/a}\{13, 13\} \wedge |x| > 13 \tag{2}$$

On account of the large automaton corresponding to the regular expression, Ostrich(+)[20][19] crashes on string constraints (3) where $\Sigma_{[0-9A-Za-z]}$ is a digital letter or an English letter.

$$x \in \Sigma_{[0-9A-Za-z]}\{0, 63\} \wedge |x| > 0 \tag{3}$$

To address these issues, we formalize bounded repetition using cost-enriched finite automaton (CEFA)[19] with linear arithmetical constraints. It limits the max(min) run times of some transitions on the accepting run of the automaton, which completely captures the semantics of bounded repetition. The satisfiability problem of string constraints with bounded repetition is reduced to the emptiness problem of cost-enriched finite automata under linear arithmetical constraints (abbreviated as SAT_{CL} problem), which is decidable and can be solved by our previous work [19]. However, the previous work could perform better practically, especially when regular expressions are complex. In this paper, we propose two heuristical methods to improve performance. The first one is under-approximation, which is inspired by Bounded Model Checking [14][21][13]. The second one is the symbolic-aware simplification, which is motivated by the observation that the emptiness problem is only related to the symbolic update functions. We implement a powerful string solver OstrichCEA and evaluate it on instances from the real world and other typical benchmarks. The experiment result shows superiority.

Related Work To handle the problems in software verification reasoning about strings and integers, string theories with regular membership predicate and linear length constraints are raised. String solvers such as Z3Str3[11], Z3Seq[35],

Z3Str3RE[12], CVC4[**?**], CVC5[8], (Z3-)Trau[4][3], Ostrich(+)[20][19], Sloth[28], S3[37], and ABC[7] are developed on these theories. Although string theory is easily undecidable[25][26], the satisfiability problems for string constraints of regular expressions, linear integer arithmetic, and the string length is decidable[10]. There are mainly two strategies for solving string constraints: (1) DPLL(T)-based[27] string solvers[11][8] apply heuristic derivation rules to unwind the regular expression gradually and identify several classes of simplification techniques for efficiency. (2) Automata-based string solvers[4][3][20][19][12] construct a finite automaton to model the regular expression and use the emptiness problem of the automaton (under linear integer arithmetic) to decide the satisfiability of string constraints.

Our Contributions Our main contributions are as follows:

1. Encode regular expression with bounded repetition into CEFA with linear arithmetical constraints.
2. Devise heuristic methods like under-approximation and symbolic-aware simplification to solve the SAT_{CL} problem.
3. Generate significant instances with bounded repetition from real-world regular expressions.
4. Implement our decision procedure on solver OstrichCEA and compare it with state-of-art string solvers. The result shows the superiority of our encoding and heuristics.

The rest of the paper is structured as follows: Section 2 introduces the preliminaries. Section 4 outlines the overall algorithm. Section **??** illustrates how to construct an automaton from a regular expression. Section 5 shows how to solve a string constraint with repetition. Section 6 presents the experimental results. Section 7 concludes the paper and discusses future work.

2 Preliminaries

Tokens A finite *alphabet* Σ is the set of all *letters*. A *string* (or *word*) is a finite sequence of letters from Σ. Σ^* is the set of strings over Σ. ϵ is the empty string. L is grammar. A language $\mathcal{L}(L)$ is a set of words generated by L. $\mathbb{N}$ is the set of natural numbers and $\mathbb{Z}$ is the set of integer numbers. We use $a, b, \cdots$ to denote the constant letters in Σ, $u, v, \cdots$ to denote constant string, $x, y, \cdots$ to denote variable string, $m, n, \cdots$ to denote integer constant, and $i, j \cdots$ to denote integer variable. For vector, we use $\vec{v}$ to denote the vector of integer constant, m_n to denote the vector $(m, \cdots, m)$ with length n, $\vec{v}[i]$ to denote the integer value of $\vec{v}$ at position i, $\vec{v_1} \cdot \vec{v_2}$ to denote the concatenation of $\vec{v_1}$ and $\vec{v_2}$, R to denote the vector of registers, $R_1 \cap R_2$ to denote the same registers in R_1 and R_2, and $|\vec{v}|$ or $|R|$ to denote the length of the vector.

Syntax This paper proposes a quantifier-free first-order logic called *Extended String Logic (ESL)*, whose syntax is presented in Table 1. φ is a quantifier-free formula that can be a regular membership $x \in \mathcal{R}$, a quantifier-free Presburger formula $\alpha_1 \leq \alpha_2$, a negation of a formula, or a disjunction of two formulas. α is an integer term which can be an integer constant m, integer variable i, the length of the string term $|s|$, the minus of an integer term, and the plus of two integer terms. The regular expression $\mathcal{R}$ is built on empty string ϵ, constant letter $a \in \Sigma$. The supported regex operations involve concatenation $\cdot$, disjunction $+$, intersection $\times$, closure $*$, complement C, and bounded repetition $\{m, n\}$. We use $term(\varphi)$ to denote the set of terms and $strvar(\varphi)$ to denote the string variables occurring in φ. The *literals* of φ are $x \in \mathcal{R}$ and $\alpha_1 \leq \alpha_2$. $x \in \mathcal{R}$ is called *regular literals* and $\alpha_1 \leq \alpha_2$ is called *linear literals*.

$$
\begin{array}{lr}
\varphi ::= x \in \mathcal{R} \mid \alpha_1 \leq \alpha_2 \mid \neg\varphi \mid \varphi \vee \varphi & \text{formulae} \\
\mathcal{R} ::= \epsilon \mid a \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} + \mathcal{R} \mid \mathcal{R} \times \mathcal{R} \mid \mathcal{R}^* \mid \mathcal{R}^C \mid \mathcal{R}\{m, n\} & \text{regular expressions} \\
\alpha ::= m \mid i \mid |s| \mid -\alpha \mid \alpha_1 + \alpha_2 & \text{integer terms}
\end{array}
$$

Table 1: Syntax

Semantic We assume that S is the set of string variables over Σ^*, and I is the set of integer variables. $\eta : S \times \Sigma \to \Sigma^*$ is the interpretation on string where $\eta(c) = c$ for every letter $c \in \Sigma$. $\pi : I \to \mathbb{Z}$ is the interpretation of Presburger arithmetic. Then the semantics is given by a satisfaction relation: $\eta, \pi \models \varphi$ defined in Table 2. We say a formula φ is *satisfiable* if a solution (η, π) exists such as $\eta, \pi \models \varphi$. A formula φ is *unsatisfiable* if no solution exists. In addition

$$
\begin{array}{ll}
\eta, \pi \models \varphi_1 \vee \varphi_2 & \textsf{iff } \eta, \pi \models \varphi_1 \text{ or } \eta, \pi \models \varphi_2 \\
\eta, \pi \models \neg\varphi & \textsf{iff } \eta, \pi \not\models \varphi \\
\eta, \pi \models x \in \mathcal{R} & \textsf{iff } \exists w \in \mathcal{L}(\mathcal{R}), \eta, \pi \models x = w \\
\eta, \pi \models \alpha_1 \leq \alpha_2 & \textsf{iff } \pi(\alpha_1) \leq \pi(\alpha_2)
\end{array}
$$

Table 2: Semantics

to the classic regex operators (union, concatenation, closure), we syntactically support intersection, complement, and repetition. As we all know, the classic regular language is closed under intersection and complement [29]. Furthermore, the operation *repetition* $\mathcal{R}\{m, n\}$ means repeating regex $\mathcal{R}$ at least m times and at most n times. It can be syntactically rewritten by concatenation and union: $\mathcal{R}\{m, n\} \equiv \mathcal{R}^m \mid \cdots \mid \mathcal{R}^n$ where $\mathcal{R}^k$ defines concatenate $\mathcal{R}$ k times ($m \leq k \leq n$). So the regular language $\mathcal{L}(\mathcal{R})$ defined in Table 1 is semantically equal to the classic regular language.

3 Overview

The main idea is to use CEFAs (see Subsection 4.1) to simulate the semantics of length operations and regular memberships with bounded repetitions. As mentioned, the ESL formula contains regular literals and linear literals. The regular literal $x \in \mathcal{R}$ directly results in one CEFA recognizing it (see Subsection 4.2). The linear literals $\alpha_1 \leq \alpha_2$ with no length operation remain unchanged. For each linear literal $\alpha_1 \leq \alpha_2$ with length operation $|x|$, we generate a fresh variable i to replace all occurrences of $|x|$ and propagate new formula $i = |x|$. Then we generate the pre-image $\mathcal{A}_i$ whose accepting words are strings with length i (see Example 3). After the process above, the satisfiability problem of string constraints becomes a SAT_{CL} problem, which has a decision procedure to check (see Subsection 5.2). Example 1 illustrate it.

Example 1.

$$\varphi \equiv x \in (ab)\{1, 100\} \wedge y \in ab \wedge |x| > |y|$$

ESL conjunction φ is made up of regular literals $x \in (ab)\{1, 100\}$ and $y \in ab$, linear literals $|x| > |y|$. The linear literal is translated to $i = |x| \wedge j = |y| \wedge i > j$. Our algorithm solves the formula in four steps. First, we construct CEFAs for regular memberships $x \in (ab)\{1, 100\}$ and $y \in ab$ (Fig.1a and Fig.1b). Second, we compute the pre-images of length operations $i = |x|$ and $j = |y|$ (Fig.1c and Fig.1d). Then we intersect pre-images to automata corresponding to the regular memberships for each string variable (Fig.1e and Fig.1f). We translate the satisfiability problem of φ to the emptiness checking problem of automata (Fig.1c and Fig.1d) under linear arithmetic constants $i > y$, which could be solved by the decision procedure illustrated in section 5.

4 Automaton Model

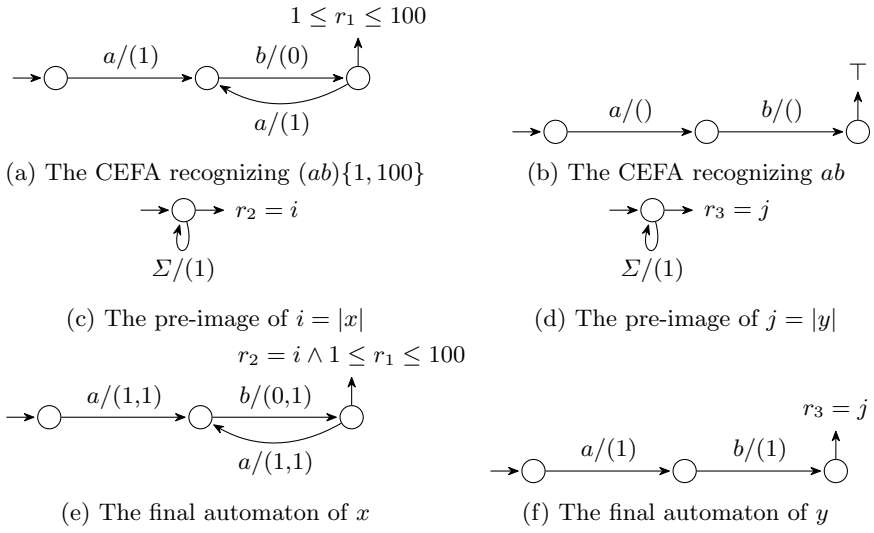
In this section, we recall the definition of CEFA and Parikh image.

4.1 Cost-Enriched Finite Automaton

The definition of CEFA in this paper is lightly different from the definition in [19]. In [19], the cost function is defined as a function $\eta : \Sigma \to \mathbb{N}$. In this paper, we define the cost function as an integer vector whose elements are the incremental value of registers. Furthermore, we add a new linear integer arithmetic constraint θ to each final state of the CEFA, which restricts the value of registers. Two types of definitions have the same expressive ability on the SAT_{CL} problem.

Definition 1 (Cost-Enriched Finite Automaton). *A cost-enriched finite automaton $\mathcal{A}$ is a 7-tuple $(Q, \Sigma, \delta, q_I, F, R, \theta)$ where*

- *Q, Σ, q_I, F is defined as NFA,*



(a) The CEFA recognizing $(ab)\{1, 100\}$ (b) The CEFA recognizing ab

(c) The pre-image of $i = |x|$ (d) The pre-image of $j = |y|$

(e) The final automaton of x (f) The final automaton of y

Fig. 1: All automata used in the example 1

- $R = (r_1 \cdots r_n)$ *is a vector of mutually distinct cost registers,*
- δ *is a transition set whose elements are tuples* $(q, c, q', \vec{v})$ *where* q, q' *are states of* Q*,* c *is a letter in alphabet* $\Sigma \cup \{\epsilon\}$ *and* $\vec{v}$ *is the cost update function for registers, which is an integer vector whose* i*-th element is the incremental value of register* r_i*.* $(q, a, q', \vec{v})$ *is written as* $q \xrightarrow[\vec{v}]{a} q'$ *for readability.*
- $\theta : F \to \varphi$ *is a linear integer arithmetic constraint function on final states.* θ *is called* accepting condition*.*

The value of each register r_i *in* R *is written as* $\mathcal{V}(r_i)$ *and is initiated to 0 at the initial state. A* run *of* $\mathcal{A}$ *on string* $a_1 \cdots a_m$ *is a transition sequence* $q_I \xrightarrow[\vec{v_1}]{a_1} q_1 \cdots q_{m-1} \xrightarrow[\vec{v_m}]{a_m} q_m$ *and* $\mathcal{V}(r_i) = \sum_{j=1}^{m} \vec{v_j}[i], i \in [1, n]$ *is the value of* r_i *after the run.* $\theta[R/\mathcal{V}(R)](q_m)$ *is obtained from* $\theta(q_m)$ *by replacing each register* r_i *to its value* $\mathcal{V}(r_i)$*. The run is* accepting *if* $q_m \in F$ *and* $\theta[R/\mathcal{V}(R)](q_m)$ *is satisfiable.* $\top$ *is the valid formula that is always satisfiable. A string* w *is accepted by* $\mathcal{A}$ *if it has an accepting run of* $\mathcal{A}$*. The language of* $\mathcal{A}$ *is the set of strings accepted by* $\mathcal{A}$*, denoted by* $\mathcal{L}(\mathcal{A})$*.*

Example 2. Fig.2 illustrates the CEFA recognizing $(ab)\{1, 100\}$. The register vector is (r_1), and the linear arithmetic constraint on q_2 is $1 \le r_1 \le 100$. Intuitively, the transition $q_0 \xrightarrow[(1)]{a} q_1$ accept the char a and increase 1 on the value of r_1. The transition $q_1 \xrightarrow[(0)]{b} q_2$ accepts the char b and does not change the value. The transition $q_2 \xrightarrow[(0)]{\epsilon} q_2$ is a nondeterministic choice back to initial state q_0.

Because of the linear arithmetic constraint $1 \leq r_1 \leq 100$, the value of r_1 in the accepting run must be $[1, 100]$. $\mathcal{V}(r_1)$ equal to the occurrence number of the transition $q_0 \xrightarrow[(1)]{a} q_1$, so that the accepted string is $ab \cdots ab$ in which a appears at least once and at most 100 times. That is, the language of the automaton is $(ab)\{1, 100\}$.

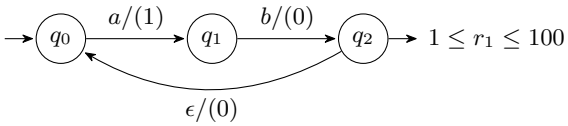


Fig. 2: An automaton recognizing the language of $(ab)\{1, 100\}$

Example 3 (The pre-image of length operation). The length operation can be captured by a CEFA base on our previous work [19]. Fig.3 shows a pre-image of linear literal $i = |x|$. It is easy to see that the pre-image accepts string words with length i.

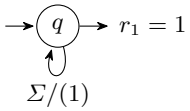


Fig. 3: The pre-image of $i = |x|$

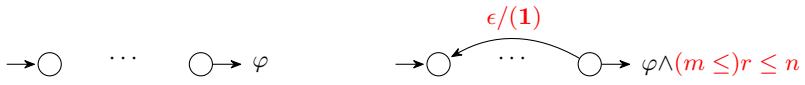
4.2 Encode Bounded Repetition to CEFA

In this section, we define the construction of CEFA from the regular expression with bounded repetition. We must syntactically rewrite bounded repetition $\mathcal{R}\{m, n\}$ if it is the sub-regex of complement (e.g., $(\mathcal{R}\{m, n\})^C$), closure(e.g., $(\mathcal{R}\{m, n\})^*$), and bounded repetition(e.g., $(\mathcal{R}\{m, n\})\{m', n'\}$), we unwind it to $\mathcal{R}^m \mid \cdots \mid \mathcal{R}^n$. After this syntactic rewriting, we call the resulting regex $\mathcal{R}'$ *non-nested*. The non-nested regex $\mathcal{R}'$ possibly contains operations such as intersection, union, concatenation, complement, closure, and bounded repetition. To construct the CEFA of $\mathcal{R}'$, we first create CEFA for each sub-regex and then combine them. Similar to the construction of NFA from regex[29], our construction is inductive on the size of $\mathcal{R}'$. The base case is for a single character, an empty string, and an empty language. The inductive step is for the concatenation, union, and intersection of two automata and one automaton's repetition, closure, and complement. We only discuss the inductive step for bounded repetition because the other operations are trivial.

For a non-nested regular expression $R\{m,n\}$, suppose we have an CEFA $\mathcal{A} = (Q, \Sigma, \delta, q_I, F, \emptyset, \top)$ recognizing R, the CEFA recognizing $R\{m,n\}$ is defined as $\mathcal{A}_{m,n} = (Q, \Sigma, \delta', q_I, F, r, \theta')$ where:

- r is a new register,
- δ' is composed by transitions $q \xrightarrow[(0)]{a} q'$ for all transitions $q \xrightarrow[()]{a} q' \in \delta$, and transitions $q_f \xrightarrow[(1)]{\epsilon} q_I$ for each $q_f \in F$,
- For each accepting states $q \in F$, $\theta'(q)$ is the linear arithmetic $m \leq r \leq n$ if $q_I \notin F$. Otherwise, $\theta'(q)$ is the linear arithmetic $r \leq n$.

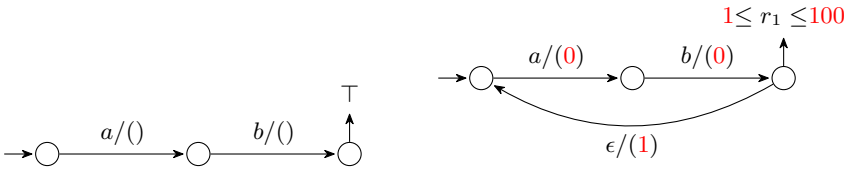
In the construction illustrated in Fig.4, a new register r is added to store the repetition times. The transition $q \xrightarrow[(1)]{\epsilon} q_I$ is set to update the repetition times for each $q \in F$. The accepting condition θ' is used to restrict the repetition times of the accepting word. Because arbitrary repetition times of an empty string are still empty, the value of r only needs to be less or equal to n when the initial state is accepted.



(a) The CEFA recognizing $\mathcal{R}$ (b) The CEFA recognizing $\mathcal{R}\{m,n\}$

Fig. 4: The construction of bounded repetition

Example 4. To construct the CEFA recognizing $(ab)\{1,100\}$, we first construct the CEFA recognizing ab as shown in Fig.5a. Then we add a new register r_1 and a transition $q_2 \xrightarrow[(1)]{\epsilon} q_0$ to update the repetition times. The accepting condition θ' map accepting state to $1 \leq r_1 \leq 100$ because the initial state is unacceptable. The CEFA recognizing $(ab)\{1,100\}$ is shown in Fig.5b.



(a) The CEFA recognizing ab (b) The CEFA recognizing $(ab)\{1,100\}$

Fig. 5: The example of bounded repetition

5 Solve ESL Formula with Large Repetition Bound

Based on the decision procedure defined in the paper [19], we propose an efficient algorithm to solve ESL formula $\varphi \wedge \psi$ where φ is the conjunction of literals and ψ is the conjunction of linear literals without length operation. For each string variable x, we first compute pre-images of all linear literal $i = |x|$. Then we generate CEFAs of all regular expressions $\mathcal{R}$ in all regular literals $x \in \mathcal{R}$ and intersect these CEFAs and pre-images to get the final automaton $\mathcal{A}_x$ for x. At present, there is a final CEFA for each string variable x. The linear arithmetic constraint ψ restricts the accepting word of final CEFAs. Thus the satisfiability problem is the emptiness checking problem of final CEFAs under ψ. If it is empty, the string constraints are unsatisfiable. Otherwise, the string constraints are satisfiable. The emptiness checking problem of CEFAs under linear integer arithmetic is theoretically pspace-complete[19]. To solve it efficiently for a practical example, we develop heuristic ways such as under-approximation (Section 5.2) and symbolic-aware simplification (Section 5.2).

5.1 High-level algorithm

Algorithm 1 High-level algorithm

Input: Conjunction of literals φ and conjunction of linear literals ψ
Output: *sat* or *unsat*

```
1: finalAuts ← ∅
2: for all string variables x occurring in φ do
3:     A_len ← intersection of all pre-images of i = |x| in φ
4:     A_regex ← intersection of all CEFAs of x ∈ R in φ
5:     A_x ← intersection of A_len and A_regex
6:     A_nfa ← NFA form of A_x
7:     if A_nfa is empty then
8:         return unsat
9:     end if
10:    finalAuts ← finalAuts ∪ {A_x}
11: end for
12: if isEmpty(finalAuts, ψ) then
13:    return unsat
14: else
15:    return sat
16: end if
```

The pseudocode presented in Algorithm 1 outlines the framework of our solving process. We construct the automata of all length operations occurring in the ESL conjunction φ at line 3 and the automata of all regular memberships at line 4. In the following steps, we call an automaton to be *final* if we will not

operate it anymore. The set $finalAuts$ contains all final automata for all string variables and is initially empty at line 1. The intersection of $\mathcal{A}_{len}$ and $\mathcal{A}_{regex}$ at line 5 ensures the final automaton of x reserves both length and regular information. At line 6, we compute an NFA form to throw unsat rapidly. An NFA form of CEFA is obtained by removing update functions and accepting conditions while maintaining graph structure. More exactly, given a CEFA $\mathcal{A} = (Q, \Sigma, \delta, q_I, F, R, \theta)$, the NFA form of $\mathcal{A}$ is $(Q, \Sigma, \delta', q_I, F)$ where δ' is composed of transition $q \xrightarrow{a} q'$ for $q \xrightarrow[\vec{v}]{a} q' \in \delta$. It is obvious that the NFA form is an over-approximation of the CEFA, so that unsat is directly thrown if we find the NFA form is already empty at line 7. Sometimes the over-approximation is useful. For example, to solve the string constraint $x \in \Sigma_{/a}\{1, 300\} \wedge x \in \Sigma^* a \Sigma^*$ which CVC5 has failed on, the NFA form is empty because there is no path to accepting states.

After obtaining all final CEFAs of all string variables and their NFA forms are not empty, we check whether the final CEFAs are empty under the linear integer arithmetic ψ at line 12. If they are empty under ψ, we return $unsat$. Otherwise, we return sat.

5.2 Emptiness Checking

As mentioned, the emptiness checking problem of CEFAs under linear integer arithmetic P is theoretically Pspace-complete. In our previous research [19], we rewrote CEFAs to an infinite system and used a model-checking tool $nuXmv$[16] to solve it. However, it needs to be more effective. So we put forward a new framework that brings in under-approximation and symbolic-aware simplification heuristics.

As shown in Algorithm 2, we simplify the input CEFAs at line 1 because they may have many transitions and registers. The purpose of the simplification is to deal with duplication. After simplification, we try to find a solution by under-approximation at line 3 and line 4. An off-the-shelf SMT solver gives the result of $\varphi_{under} \wedge P$. If we find a solution, we return $false$ at line 5 because it implies that the CEFAs under linear integer arithmetic P are not empty. Otherwise, we compute the Parikh images of the simplified CEFAs and check if the Parikh images are satisfiable in conjunction with P at line 12. The Parikh image checking is complete so that satisfiability implies non-emptiness directly.

The under-approximation program is executed many times until the $MaxBound$ is reached. $MaxBound$ is an empirical bound set to 15 in our experiment because it brings exponential growth of searching space. The details of simplification are shown in Algorithm 3 and under-approximation are shown in Algorithm 4.

Symbolic-Aware Simplification The purpose of simplification is to remove duplicated transitions and registers. In the emptiness checking Algorithm 2, the vectors on the transitions are meritorious, while the letters are not. So we see the alphabet of the CEFA as unary (i.e., the alphabet is $\{a\}$) and see the

Algorithm 2 `isEmpty`$(auts, P)$

Input: the CEFAs $auts$ and the linear integer arithmetic P
Output: $true$ or $false$

```
1: simpliAuts ← simplify(auts)
2: for bound ← 1, MaxBound do
3:     φ_under ← underApprox(simpliAuts, bound)
4:     if φ_under ∧ P is sat then
5:         return false
6:     end if
7: end for
8: φ ← parikhImage(simpliAuts)
9: if φ ∧ P is unsat then
10:    return true
11: else
12:    return false
13: end if
```

vectors in the CEFA as letters in the NFA. The vector $\vec{0}_n$ is seen as ϵ. Precisely, given an CEFA $\mathcal{A} = (Q, \Sigma, \delta, q_I, F, R, \theta)$, we obtain a symbolic form $\mathcal{A}_{sym} = (Q, \Sigma', \delta', q_I, F, R, \theta)$ where

- $\Sigma' = \{a\}$,
- the transition set δ' is composed of transition $q \xrightarrow[\vec{v}]{a} q'$ if there is a transition $q \xrightarrow[\vec{v}]{b} q'$ in δ for $b \in \Sigma$.

We directly apply algorithms of epsilon-closure, determination, and simplification in [29]. We see $q \xrightarrow[\vec{0}_n]{a} q'$ as epsilon transition and compute the epsilon-closure based on it. We see $q \xrightarrow[\vec{v}]{a} q'$ as a transition with label $(a, \vec{v})$ and determine the automaton based on the label. Two accepting states are equivalent if they have the same accepting conditions, and two states are equivalent if they reach equivalent states for each label. We simplify the automaton by merging equivalent states. The process above comprises line 3 and line 4 of Algorithm 3. From line 5 to line 13, we check whether some registers are duplicated. (i.e., the values of these registers are always the same). If two registers r_i and r_j are duplicated at line 6, we remove one at line 7 and the corresponding update of vectors at line 9 to ensure the vectors are consistent with the new registers. After deleting the duplicated registers, we add constraint $r_i = r_j$ to store the value of the deleted register at line 11.

Example 5. Considering the CEFA recognizing $(a|b)\{1, 100\}$ illustrated in Fig.6a. We first obtain a symbolic CEFA by using unary alphabet (Fig.6b) and then minimize it based on the symbolic label "$a/(1)$"(Fig.6c).

Example 6. Considering the string constraints $i = |x| \wedge j = |x|$. We compute the pre-images of $i = |x|$, $j = |x|$ and intersect these pre-images to get the

Algorithm 3 `simplify`($auts$)

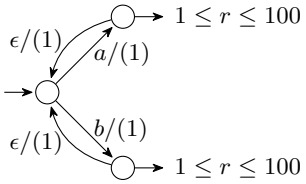
Input: A set of CEFAs
Output: The simplified CEFAs

1: **for** $\mathcal{A} \in auts$ **do**
2: Suppose that $\mathcal{A} = (Q, \Sigma, \delta, q_I, F, R, \theta)$ and $R = (r_1, \cdots, r_n)$
3: `determinizeByVec`($\mathcal{A}$)
4: `minimizeByVec`($\mathcal{A}$)
5: **for all** (i, j) where $1 \leq i \leq j \leq n$ **do**
6: **if** $\vec{v}[i] = \vec{v}[j]$ for every vector $\vec{v} \in \mathcal{A}$ **then**
7: $R = (\cdots r_{j-1} r_{j+1} \cdots)$ ▷ remove r_j from the register vector R
8: **for all** $\vec{v'} \in \mathcal{A}$ **do**
9: $\vec{v'} = (\cdots \vec{v'}[j-1]\vec{v'}[j+1] \cdots)$ ▷ remove $\vec{v'}[j]$ from each vector $\vec{v'}$
10: **end for**
11: $\theta = \theta \wedge r_i = r_j$
12: **end if**
13: **end for**
14: **end for**
15: **return** $auts$

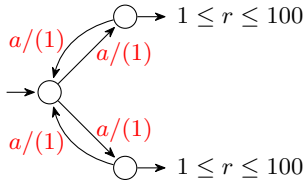
final CEFA represented in Fig.7a. However, the register r_1 and r_2 in the final CEFA are duplicated since their update functions are the same. We can simplify the CEFA by removing the duplicated register r_2 and corresponding update functions, as shown in Fig.7b. To maintain the value of r_2, we add a constraint $r_2 = r_1$ to restrict the value of r_2 to be the same as r_1.

Under-Approximation An under-approximation procedure solves the satisfiable instances of the emptiness checking problem. Inspired by Bounded Model Checking [14][21][13], we explore the states of CEFA in typological order. We use string length as the bound and probe the CEFA from low to high bound. This section discusses the situation in which the string length bound is fixed. The main idea is to enumerate all possible runs ending in accepting states with lengths less than the fixed bound. Although the number of possible runs is an exponential growth of the bound, we usually solve practical satisfiable instances within 1s. As shown in the Algorithm 4, for each CEFA, we enumerate all runs whose length is less than the bound and compute the value of registers by the sum of vectors on the run at line 4. If the run is ended with an accepting state, then we compute the updates of registers by the sum of vectors on the run at line 6. Together with the accepting condition of the accepting state, we add the updates of registers to the under-approximation formula at line 7.

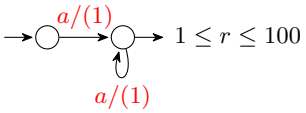
Example 7. Consider the automaton in Fig. 1e again. The solution of the Parikh image of it leads to a complete but inefficient result since the quantifier-free linear integer arithmetic formulas are solved in exponential time [40]. Under-approximation can accelerate the solving process. When we set the bound to 2, the under-approximation program will enumerate all runs whose length is



(a) The CEFA recognizing $(a|b)\{1, 100\}$

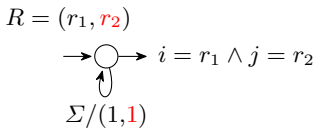


(b) The symbolic CEFA

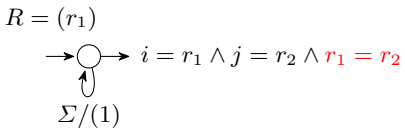


(c) Simplify the CEFA based on symbolic label

Fig. 6: Symbolic simplification can remove duplicated transitions and states.



(a) An automaton with duplicated registers



(b) Remove the duplicated registers and corresponding update functions

Fig. 7: Symbolic simplification can remove duplicated transitions and states.

2. One of them is $q_I \xrightarrow[(1,1)]{a} q_1 \xrightarrow[(0,1)]{b} q_f$. The corresponding registers' values are $r_1 = 1+0 = 1, r_2 = 1+1 = 2$. Checking for $r_1 = 1 \wedge r_2 = 2 \wedge r_2 = i \wedge 1 \le r_1 \le 100$ is a lightweight process because the values of r_1 and r_2 are fixed.

6 Implementation and Experimental Results

This section presents the empirical evaluation of OstrichCEA, which is our implementation of the decision procedure introduced in Section 5. Our objective is to validate the effectiveness of the proposed techniques by evaluating our tool's correctness and efficiency compared to other solvers. Furthermore, we assess the efficacy of our heuristics by testing different configurations of the tool. We have implemented our encoding for bounded repetition with two heuristic algorithms on Ostrich+ [19]. The pre-image computation for concatenation, `indexOf`, `substring`, `replaceAll`, `reverse` and finite transducer remain unchanged. OstrichCEA is written in Scala and based on the SMT solver Princess[32].

Algorithm 4 `underApprox`$(auts, bound)$

Input: The CEFAs $auts$ and string length $bound$
Output: The linear integer arithmetic φ_{under} representing under-approximation of the register values of $auts$

$\varphi_{under} \leftarrow false$
for $\mathcal{A} \in auts$ **do**
  Suppose $\mathcal{A} = (Q, \Sigma, \delta, q_I, F, (r_1, \cdots, r_m), \theta)$
  for all run $q_0 q_1 \cdots q_n$ whose length is less than $bound$ **do**
    if $q_n \in F$ **then**
      Let $\vec{V} \leftarrow$ the sum of the vectors on the run
      $\varphi_{under} \leftarrow \varphi_{under} \vee (\theta(q_n) \bigwedge_{i \in [1,m]} r_i = \vec{V}[i])$
    end if
  end for
end for
return φ_{under}

6.1 Benchmarks

We conducted a comparison on four sets of benchmarks based on regex with bounded repetition, consisting of a total of 49,379 instances. We analyze all 19 developed benchmarks listed in [30] and find that only **AutomatArk** benchmarks have regular membership with bounded repetition. In total, almost 18% of the instances we evaluated were sourced from published industrial benchmarks or other solver developers. All the other instances contain regular expressions from the real world. Each set of the benchmark is evenly divided into "large" and "small": the "large" set contains instances with large repetition upper bounds (the sum of upper bounds is greater than 50), and "small" contains the other instances. Less than 10% of the benchmarks are in a "large" set. More details about the benchmarks are shown below.

AutomatArk is the 8,751 instances generated by Berzish et al.[12]. It is based on real-world regular expression queries from Loris D'Antoni[22]. The origin set comprises two tracks, a simple and a hard track, with 19,979 instances. The simple track contains instances with a single regular expression membership constraint, whereas the hard track can hold up to five membership constraints for a single variable per instance. We extract 8,751 instances containing bounded repetition from 19,979 instances and partition them into "large" and "small" in terms of the size of the bounds.

ReDos is the set of 1,624 instances we generated. It is based on the ReDos-attacked regular expression collected by Lenka et al. For each regular expression, we generate an instance as the template (4) where $\mathcal{R}$ is the regular expression. The regular membership predicate $x \notin \Sigma^*(<|>|'|''| \ \&)\Sigma^*$ sanitizes the input

string x to avoid the attack. The length lower bound is set to 20 for the "small" set and 50 for the "large" set.

RegexLib is the set of 1,623 instances we generated similarly to the **ReDos** benchmark. It is based on the regular expressions collected by James C. Davis et al.[24] from regex lib website[1]. The website is the Internet's first Regular Expression Library. Currently, it has indexed 4149 expressions from 2818 contributors around the world since 2001. We extract 1,623 instances containing bounded repetition from 4149 instances and partition them into "large" and "small" in terms of the size of the bounds.

StackOverflow is the 37381 instances we generated similarly to the **ReDos** benchmark. As the Regexlib benchmark, the real-world regex expressions are collected by James C. Davis et al.[24] from StackOverflow website[2]. The website is a question-and-answer site for professional and enthusiast programmers. We extract 37,381 instances containing bounded repetition from almost 500,000 instances and partition them into "large" and "small" in terms of the size of the bounds.

$$x \in \mathcal{R} \wedge x \notin \Sigma^*(<|>|'|''| \,\&)\Sigma^* \wedge |x| > 50(20) \tag{4}$$

6.2 Experimental Setup and Compared Solvers

We have evaluated OstrichCEA compared to five other prominent string solvers currently available. We evaluate the solvers by directly comparing the number of cases correctly solved, the total time taken with and without timeouts, and the total count of soundness errors and program crashes. One of these solvers is CVC5[8], a general SMT solver that uses algebraic reasoning to handle strings and regular expressions and is the winner of SMT-COMP 2022[9]. Another solver, Z3str3[11], is the most recent addition to the Z3-str family and utilizes a word equation reduction approach to reason about regular expressions. Z3str3RE[12] is a variant of Z3str3 that incorporates length-aware algorithms and heuristics. Z3seq[35] is a sequence solver which uses a novel derivative theory for solving extended regular expressions. Z3-Trau[3] is the Z3 version of trau[4] that employs a flat automata-based approach, incorporating both under- and over-approximations. Finally, Ostrich[20] is the tool we extend which uses automaton to model the semantics of string functions and regular memberships. We used the 1.0.5 binary version of CVC5, commit 59e9c87 of Z3str3, last version of Z3str3RE, 4.8.9 binary version of Z3Seq, commit 1628747 of Z3-Trau and commit 8297d8d of Ostrich. Z3-Trau does not support `re.diff`. All other solvers support all syntax sugars listed in SMT-LIB standard[17]. We omitted Z3str4[31] because the provided reproduction package link is wrong. All experiments are conducted on CentOS Stream release 8 with 12 Intel(R) Xeon(R) Platinum 8269CY CPU T 3.10GHz processors and 190 GB memory. We used Zaligvinder[30] framework and set the timeout to 60 seconds.

6.3 Overall Evaluation

In Fig.8, the cactus plot illustrates the cumulative time each solver takes for all cases in ascending order of runtime. Solvers located towards the right and lower portion of the plot indicate better performance. OstrichCEA spent more time solving instances than other solvers. One reason was that OstrichCEA was written in Scala and ran on a Java virtual machine(JVM). The cold boot of JVM is cost. The other reason was that the implemented decision procedure was excessively cumbersome in resolving certain straightforward constraints.
Table 3 summarizes the results that demonstrate OstrichCEA's superior performance, solving the most significant number of instances and outperforming most competing solvers. Including timeouts, OstrichCEA is **1.52**× faster than Z3Seq, **2.23**× faster than Ostrich, **2.26**× faster than Z3-Trau, **3.08**× faster than Z3str3, **3.11**× faster than CVC5 and close to Z3str3RE with %2 speed loss . Note that CVC5[8] yielded 5370 timeouts(11% of all instances), and Z3str3[11] yielded 6139 timeouts(12% of all instances), which is much more than other solvers. Both CVC5 and Z3str3 are DPLL(T)-based solvers. It seems that almost 10% of the benchmarks we used seem unsuitable for them. Z3-trau[3] yielded 21,152 unknowns because it does not support `re.diff`. Z3-trau[3] yielded 6673 crashes (13% of the instances) and 1233 soundness errors (2% of the instances), which is a large portion. Z3-based solvers Z3str3 yielded 38 soundness errors, Z3seq[35] yielded 51 soundness errors and Z3str3RE[12] yielded 39 soundness error. Most of them are due to the mistake formalization of the backslash character. OstrichCEA resulted 5 unknowns because it reached the maximum threshold of limited memory 2GB.

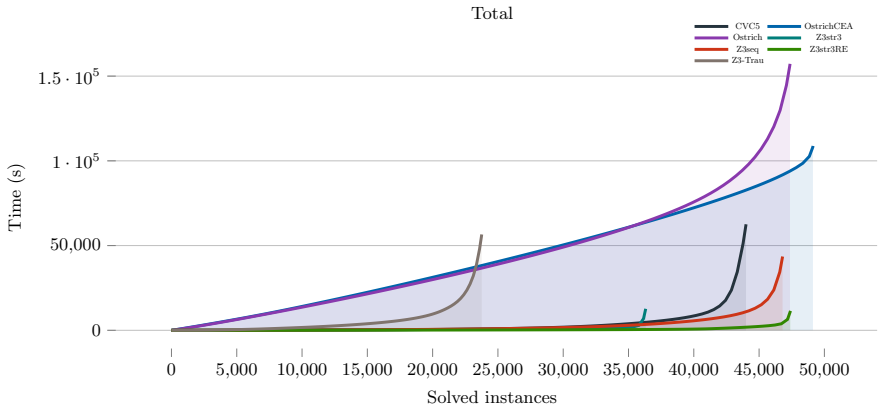


Fig. 8: Cactus plot summarizing performance on all benchmarks.

	CVC5	OstrichCEA	Ostrich	Z3str3	Z3seq	Z3str3RE	Z3-Trau
sat	21863	**23300**	21666	18413	22767	23284	6781
unsat	22138	**25828**	25706	17945	24097	24162	18211
unknown	8	**5**	76	6882	98	99	21152
timeout	5370	**246**	1931	6139	2417	1834	3235
soundness error	**0**	**0**	**0**	38	51	39	1233
program crashes	**0**	**0**	**0**	2	**0**	1	6673
Total correct	44001	**49128**	47372	36320	46813	47407	23759
Time (ms)	384770242	123773852	276633959	381744544	188675832	**121540521**	280818135
Time w/o timeouts (ms)	62569944	109013852	160773959	13404544	43655832	**11500521**	86718135

Table 3: Total results of string solvers on all benchmarks. OstrichCEA solved the most benchmarks in the second shortest time.

6.4 Detailed Evaluation

The detailed results for the **AutomatArk** benchmark are presented in Figure 9 and Table 4. OstrichCEA solved the most *unsat* instances. Z3str3RE solved the most *sat* instances. Ostrich solved the greatest number of instances, while Z3str3RE used the least time with timeouts.
The detailed results for the **Redos** benchmark are presented in Figure 10 and Table 5. OstrichCEA solved the most instances on both tracks of *sat* and *unsat*. Including timeouts, OstrichCEA is **4.4**× faster than CVC5, **3.74**× faster than Ostrich, **2.09**× faster than Z3str3, **5.40**× faster than Z3seq, **2.49**× faster than Z3-Trau and close to Z3str3RE with %20 speed loss.
The detailed results for the **RegexLib** benchmark are presented in Figure 11 and Table 6. Ostrich solved the most *unsat* instances. OstrichCEA solved the most *sat* instances. In total, OstrichCEA solved the greatest number of instances and was the solver with medium speed.
The detailed results for the **StackOverflow** benchmark are presented in Figure 12 and Table 7. OstrichCEA solved the most instances on both tracks of *sat* and *unsat* and was the faster solver.
Including timeouts, OstrichCEA is **3.98**× faster than CVC5, **2.63**× faster than Ostrich, **2.19**× faster than Z3str3, **1.59**× faster than Z3seq, **1.28**× faster than CVC5 and **1.47**× to Z3str3RE.

6.5 Analysis of Individual Heuristics

In order to demonstrate the efficacy of the individual heuristics outlined in Section 5 and incorporated into OstrichCEA, we assessed various tool configurations in which one or more heuristics were disabled. Figure 13 and Table 8 display the outcomes. The "OstrichCEA" plot line represents the tool's performance when all heuristics are enabled, while the "All heuristics off" line represents performance when all heuristics are disabled. The remaining plot lines exhibit performance with only the named heuristic disabled while all others are enabled. From the plots and table, it is evident that OstrichCEA functions most effectively when all heuristics are enabled. OstrichCEA performs 1.78 times faster

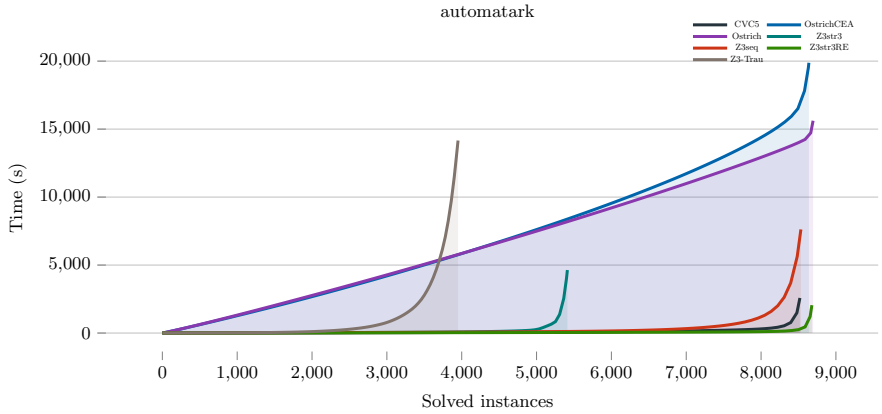


Fig. 9: The plot of a cactus graph depicting a comprehensive evaluation of the performance of the AutomatArk benchmark.

	CVC5	OstrichCEA	Ostrich	Z3str3	Z3seq	Z3str3RE	Z3-Trau
sat	4409	4371	4432	2109	4392	**4454**	1872
unsat	4110	**4268**	4263	3303	4140	4225	2485
unknown	**0**	**0**	4	357	**0**	1	2766
timeout	232	112	**52**	2982	219	71	1628
soundness error	**0**	**0**	**0**	**0**	**0**	**0**	406
program crashes	**0**	**0**	**0**	2	**0**	1	802
Total correct	8519	8639	**8695**	5412	8532	8679	3951
Time (ms)	16500129	26598963	18909762	183673696	20760921	**6341081**	122391663
Time w/o timeouts (ms)	2580115	19878963	15789762	4753696	**7620921**	2081081	24711663

Table 4: Detailed results for the AutomatArk benchmark.

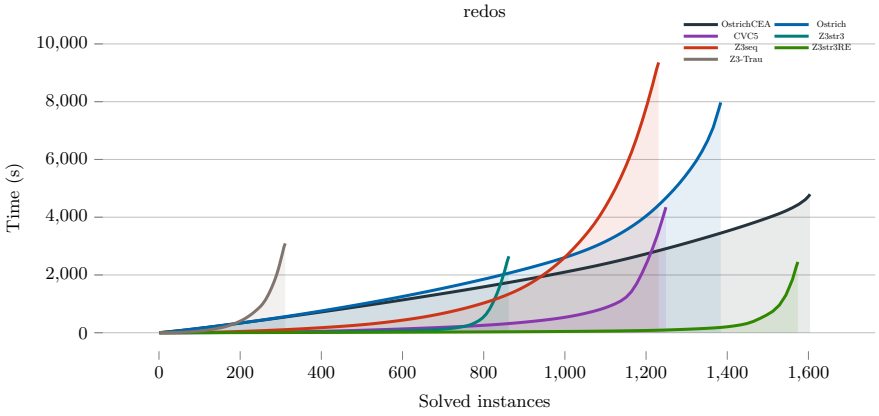


Fig. 10: The plot of a cactus graph depicting a comprehensive evaluation of the performance of the ReDos benchmark.

	CVC5	OstrichCEA	Ostrich	Z3str3	Z3seq	Z3str3RE	Z3-Trau
sat	1207	**1501**	1281	836	1157	1478	281
unsat	42	**103**	**103**	26	75	96	268
unknown	**0**	1	7	599	10	10	905
timeout	375	**19**	233	163	382	40	170
soundness error	**0**	**0**	**0**	**0**	**0**	**0**	238
program crashes	**0**	**0**	**0**	**0**	**0**	**0**	211
Total correct	1249	**1604**	1384	862	1232	1574	311
Time (ms)	26846266	5972831	22312081	12509222	32281077	**4851491**	17243714
Time w/o timeouts (ms)	4346266	4832831	8332081	2729222	9361077	**2451491**	7043714

Table 5: Detailed results for the ReDos benchmark.

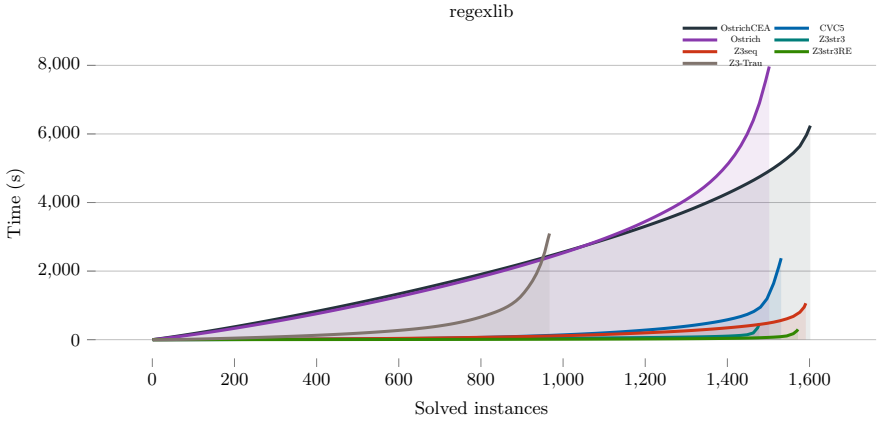


Fig. 11: The plot of a cactus graph depicting a comprehensive evaluation of the performance of the RegexLib benchmark.

	CVC5	OstrichCEA	Ostrich	Z3str3	Z3seq	Z3str3RE	Z3-Trau
sat	656	**706**	605	659	701	714	256
unsat	875	896	**898**	819	892	858	757
unknown	**0**	**0**	3	117	8	8	376
timeout	92	**21**	117	28	22	43	234
soundness error	**0**	**0**	**0**	**0**	**0**	**0**	46
program crashes	**0**	**0**	**0**	**0**	**0**	**0**	188
Total correct	1531	**1602**	1503	1478	1593	1572	967
Time (ms)	7896025	7501055	15128493	2162751	2390083	**2880392**	18031189
Time w/o timeouts (ms)	2376002	6241055	8108493	482751	1070083	**300392**	3991189

Table 6: Detailed results for the RegexLib benchmark.

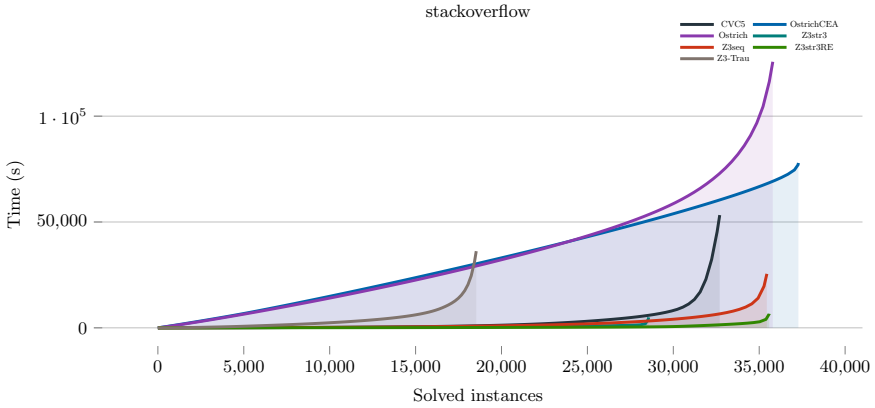


Fig. 12: The plot of a cactus graph depicting a comprehensive evaluation of the performance of the StackOverflow benchmark.

	CVC5	OstrichCEA	Ostrich	Z3str3	Z3seq	Z3str3RE	Z3-Trau
sat	15591	**16722**	15348	14809	16517	16638	4372
unsat	17111	**20561**	20442	13797	18990	18983	14701
unknown	8	**4**	62	5809	80	80	17105
timeout	4671	**94**	1529	2966	1794	1680	1203
soundness error	**0**	**0**	**0**	38	51	39	543
program crashes	**0**	**0**	**0**	**0**	**0**	**0**	5472
Total correct	32702	**37283**	35790	28568	35456	35582	18530
Time (ms)	333527822	**83701003**	220283623	183398875	133243751	107467557	123151569
Time w/o timeouts (ms)	53267561	78061003	128543623	**5438875**	25603751	6667557	50971569

Table 7: Detailed results for the StackOverflow benchmark.

when employing all of our heuristics than none. All other configurations of the tool's heuristics make sense in comparison.

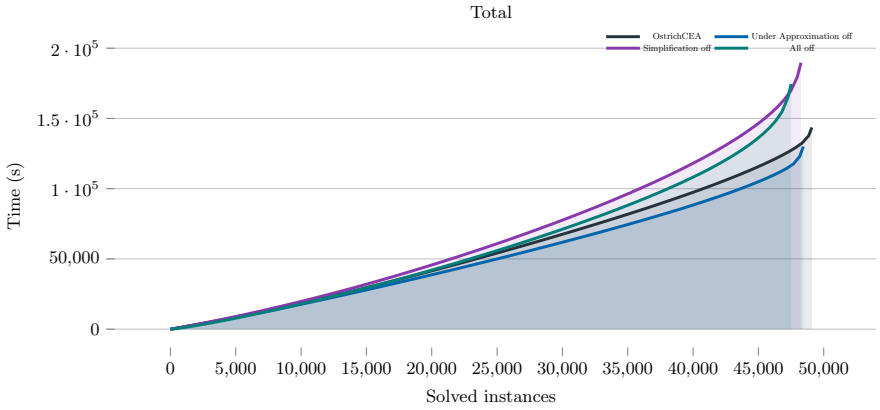


Fig. 13: A performance comparison was made on all benchmarks by disabling individual heuristics using a cactus plot.

	OstrichCEA	Under Approximation off	Simplification off	All off
sat	**23271**	22649	22609	21799
unsat	**25824**	25835	25730	25745
unknown	3	**2**	3	3
timeout	**281**	893	1037	1832
soundness error	**0**	**0**	**0**	**0**
program crashes	**0**	**0**	**0**	**0**
Total correct	**49095**	48484	48338	47542
Time (ms)	**160594317**	184660630	252806731	285102326
Time w/o timeouts (ms)	143734317	**131080630**	190586731	175182326

Table 8: A performance comparison was made on all benchmarks by disabling individual heuristics.

7 Conclusion and Future Work

This paper aims to efficiently solve the string constraints with bounded repetition and linear integer constraints on string length. The bounded repetition is encoded by an automaton model CEFA whose size is linear to the bound. Moreover, we extend the algorithm in the paper [19] with heuristics such as under-approximation and symbolic-aware simplification. The implementation of

the algorithm is done on string solver OstrichCEA. The extensive empirical comparison against state-of-art string solvers over a large and diverse benchmark shows the power of our encoding for bounded repetition and heuristic ways. In the future, we plan to explore how to solve nested repetition and use CEFA to solve more string operations.

References

1. regular expression library. `https://regexlib.com/` (May 2022)
2. Stackoverflow. `https://stackoverflow.com/` (May 2022)
3. Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Dolby, J., Janků, P., Lin, H.H., Holík, L., Wu, W.C.: Efficient handling of string-number conversion. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 943–957. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386034, `https://doi.org/10.1145/3385412.3386034`
4. Abdulla, P.A., Faouzi Atig, M., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: Smt solver for string constraints. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–5 (2018). https://doi.org/10.23919/FMCAD.2018.8602997
5. Amadini, R., Andrlon, M., Gange, G., Schachte, P., Søndergaard, H., Stuckey, P.J.: Constraint programming for dynamic symbolic execution of javascript. In: Rousseau, L.M., Stergiou, K. (eds.) Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 1–19. Springer International Publishing, Cham (2019)
6. Amadini, R., Jordan, A., Gange, G., Gauthier, F., Schachte, P., Søndergaard, H., Stuckey, P.J., Zhang, C.: Combining string abstract domains for javascript analysis: An evaluation. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 41–57. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
7. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 255–272. Springer International Publishing, Cham (2015)
8. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength smt solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer International Publishing, Cham (2022)
9. Barbosa, H., Bobot, F., Hoenicke, J.: The 17th international satisfiability modulo theories competition (smt-comp 2022) (2022), `https://smt-comp.github.io/2022/`
10. Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: String theories involving regular membership predicates: From practice to theory and back. In: Lecroq, T., Puzynina, S. (eds.) Combinatorics on Words. pp. 50–64. Springer International Publishing, Cham (2021)
11. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. p. 55–59. FMCAD '17, FMCAD Inc, Austin, Texas (2017)

12. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An smt solver for regular expressions and linear arithmetic over string length. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 289–312. Springer International Publishing, Cham (2021)
13. Biere, A.: Bounded model checking. Adv. Comput. **58**, 117–148 (2021)
14. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
15. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 307–321. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
16. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 334–342. Springer International Publishing, Cham (2014)
17. Cesare, T., Clark, B., Pascal, F.: Smt-lib: The satisfiability modulo theories library (2020), `http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml`
18. Chapman, C., Wang, P., Stolee, K.T.: Exploring regular expression comprehension. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. p. 405–416. ASE '17, IEEE Press (2017)
19. Chen, T., Hague, M., He, J., Hu, D., Lin, A.W., Rümmer, P., Wu, Z.: A decision procedure for path feasibility of string manipulating programs with integer data type. In: Hung, D.V., Sokolsky, O. (eds.) Automated Technology for Verification and Analysis. pp. 325–342. Springer International Publishing, Cham (2020)
20. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290362, `https://doi.org/10.1145/3290362`
21. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal methods in system design **19**, 7–34 (2001)
22. D'Antoni, L.: Automatark automata benchmark (2018). (2018), `https://github.com/lorisdanto/automatark`
23. Davis, J.C.: Rethinking regex engines to address redos. p. 1256–1258. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338906.3342509, `https://doi.org/10.1145/3338906.3342509`
24. Davis, J.C., Michael IV, L.G., Coghlan, C.A., Servant, F., Lee, D.: 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. p. 443–454. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338906.3338909, `https://doi.org/10.1145/3338906.3338909`
25. Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion (05 2016)
26. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: What's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) Hardware and Software: Verification and Testing. pp. 209–226. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

27. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Dpll(t): Fast decision procedures. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification. pp. 175–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
28. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proc. ACM Program. Lang. **2**(POPL) (dec 2017). https://doi.org/10.1145/3158092, https://doi.org/10.1145/3158092
29. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company (1979)
30. Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: Zaligvinder: A generic test framework for string solvers. Journal of Software: Evolution and Process **35**(4), e2400 (2023). https://doi.org/https://doi.org/10.1002/smr.2400, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2400
31. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A multi-armed string solver. In: Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings. p. 389–406. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-90870-6˙21, https://doi.org/10.1007/978-3-030-90870-6_21
32. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 274–289. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
33. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: 2010 IEEE Symposium on Security and Privacy. pp. 513–528 (2010). https://doi.org/10.1109/SP.2010.38
34. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: A selective record-replay and dynamic analysis framework for javascript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. p. 488–498. ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2491411.2491447, https://doi.org/10.1145/2491411.2491447
35. Stanford, C., Veanes, M., Bjørner, N.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 620–635. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454066, https://doi.org/10.1145/3453483.3454066
36. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. p. 1232–1243. CCS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2660267.2660372, https://doi.org/10.1145/2660267.2660372
37. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. p. 1232–1243. CCS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2660267.2660372, https://doi.org/10.1145/2660267.2660372

38. Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M., Vojnar, T.: Regex matching with counting-set automata. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428286, https://doi.org/10.1145/3428286
39. Veanes, M., Tillmann, N., de Halleux, J.: Qex: Symbolic sql query explorer. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 425–446. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
40. Verma, K.N., Seidl, H., Schwentick, T.: On the complexity of equational horn clauses. In: Proceedings of the 20th International Conference on Automated Deduction. p. 337–352. CADE' 20, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11532231˙25, https://doi.org/10.1007/11532231_25
41. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for php. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 154–157. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)