Efficient Methods for Solving String Constraints with Large Repetition Bound

DengHang Hu
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
hudh@ios.ac.cn

Zhilin Wu
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
wuzl@ios.ac.cn

ABSTRACT

Regular expression with large repetition bounds frequently appears in the real world. However, state-of-art string solvers can not efficiently solve string constraints with large repetition bounds, which slows down string constraint applications like formal verification and symbolic execution. In this paper, we propose a new systematic algorithm based on the decision procedure in our previous work to solve the problem efficiently. We use heuristic ways like over-approximation and under-approximation to accelerate the search. Further, we extract amounts of regex expressions with repetition times from real-world programs and generate a significant benchmark. We evaluate the algorithm's performance on our generated and developed benchmarks. The result shows that our solver outperforms the state-of-art string solvers.

CCS CONCEPTS

• **Theory of computation → Regular languages**.

KEYWORDS

string constraints, cost-enriched automata, regex expression, bounded repetition

ACM Reference Format:
DengHang Hu and Zhilin Wu. 2022. Efficient Methods for Solving String Constraints with Large Repetition Bound. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Regular expressions are widely used in programming languages such as Javascript and Python. About 30–40 % of Java, JavaScript, and Python software uses regex matching[2]. To find potential software bugs automatically, many researchers prefer to use one of the formal verification technology - symbolic execution [7]. Symbolic execution views symbolic paths as multi-theories constraints and uses a constraints solver to solve these constraints. Thus the efficiency and the ability of the constraints solver are vitally important. However, the string theory is one of the most challenging theories in multi-theory constraints because it is easy to be undecidable[3]. So the string constraints solver with the higher ability (support more operators) usually performs slower. Most solvers try very hard to improve efficiency. For example, Z3 and cvc5 append many heuristic derivation rules to speed up their search in the DPLL framework. Trau+ proposes a new model called flat automaton and uses the CEGAR framework to under-approximate and over-approximate the string constraints again and again. Unfortunately, the string solvers above perform poorly when the string constraints contain large repetition bound. The DPLL-framework string solvers prefer to search model strings with smaller lengths, but large repetition bound most often stands for considerable model string lengths. The automaton-based solvers unwind the automaton naively, so the search space of the unwound automaton is exponential of the repetition bound. To address this issue, we extend cost-enriched finite automaton[1] whose search space is linear to the repetition bound. Our main contributions are:

1. We extend cost-enriched finite automaton to model the extended regular expression with large repetition bound.
2. We devise an efficient algorithm to solve string constraints with large repetition bound with the new automaton model.
3. We generate many instances with large repetition bound from real-word regular expressions.
4. We implemented our efficient algorithm on ostrich and compared it with state-of-art string solvers. The result shows the superiority of our automaton model and algorithm.

The structure of this paper is as follows: We introduce some abbreviations and concepts in section 2 and introduce the syntax and semantics of the string theory logic in section 3. Then we develop cost-enriched finite automaton model and its operations at section 4. The efficient algorithm is discussed in section 5, and the implementation of the algorithm is mentioned in section 6. Finally, we conclude our work and look into future work in section 7.

2 PRELIMINARIES

Tokens. We consider a finite *alphabet* Σ, the set of all *letters*. A *string* is a finite sequence of letters from Σ. We use Σ^* to denote the set of strings over Σ, ϵ to denote the empty string, and $a, b, \cdots$ to denote constant letters in Σ. We use $u, v, \cdots$ to denote constant string and $x, y, \cdots$ to denote variable string. Moreover, we also consider a set of natural integer numbers $\mathbb{N} = \{0, 1, 2, 3, \cdots\}$ and a set of integer numbers $\mathbb{Z} = \{\cdots, -1, 0, 1, \cdots\}$. We use $m, n, \cdots$ to denote constant integer, and $i, j \cdots$ to denote variable integer. For vector, we use $\bar{v}$ to denote the vector of constant integer, 0_n to denote the zero vector with length n, $|\bar{v}|$ to denote the length of $|\bar{v}|$,

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/22/12...$15.00
https://doi.org/XXXXXXX.XXXXXXX

and $v(i)$ to denote the integer value of $\bar{v}$ at position i, and $\cdot$ to denote the concatenation of two vectors.

CNF, DNF, clause, cube. We assume that the reader is familiar with first-order logic. A *literal* is an atomic proposition or its negation. Briefly, A *clause* is a disjunction of literals. The empty clause is $false$. A formula is in *conjunctive normal norm* (CNF) if it is a conjunction of clauses. A *cube* is a conjunction of a consistent set of literals; The empty cube is $true$. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of cubes.

Basic Regular Language. A nondeterministic finite state 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; $\delta \in Q \times \Sigma \times Q$ is the transition relation; $I, F \subseteq Q$ are the set of initial states and finite states respectively. We write a transition $(q, a, q') \in \delta$ as $q \xrightarrow{a} q'$ for readability. A *run* of an NFA $\mathcal{A}$ on a string $w = a_1 \cdots a_n$ is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \cdots q_{n-1} \xrightarrow{a_n} q_n$ with $q_0 \in I$. The run is *accepting* if $q_n \in F$. A word w is *accepted* by an NFA $\mathcal{A}$ if there is an accepting run on w. The *language* of $\mathcal{A}$ is the set of all words accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$. From automata theory[4], we know that the languages generated by basic regular expression are also in the class of $\mathcal{L}(\mathcal{A})$. We call them *Basic Regular Language.*

3 STRING THEORY WITH EXTENDED REGEX

3.1 Syntax

In this paper, we extend string theory in our previous work [1] with extended regex. We call the new theory as *Extended String Logic (ESL).* ESL contains the following sorts: string sort `Str`, integer sort `Int`, the enumerable set of string sort $\texttt{Str}^m$, and the enumerable set of integer sort $\texttt{Int}^m$ for each $m \in \mathbb{N}$. Furthermore, ESL includes the following predicate and function constants: string concatenation $\texttt{con} : \texttt{Str} \times \texttt{Str} \rightarrow \texttt{Str}$; string length $\texttt{len} : \texttt{Str} \rightarrow \texttt{Int}$; regular membership $\in$: $\texttt{Str} \times \texttt{Str}^m$; the basic regex operators such as Kleene, concatenation, conjunction, and disjunction; the extended regex operator repetition and complement; and the usual constants of linear arithmetic.

The syntax of ESL is presented in Table1. φ is a first-order logic formula that can be a regular membership $x \in \mathcal{R}$, a word equation e, a linear arithmetic (in)equality β, a negation of a formula, or a disjunction of two formulas. A word equation e is an equality of two string terms. The (in)equality operator $\odot$ contains $=, \geq$. β is an (in)equality of two integer terms. A string term s is an empty string ϵ, letters $a \in \Sigma$, string variable x, or a concatenation of two string terms. An integer term α is an integer constant 0, integer constant 1, integer variable i, the length of the string term s, minus an integer term, or 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 $\times$, Kleene $*$, complement C, and repetition $\{m, n\}$.

We use function $FV(\varphi)$ to return all free variables of φ. We consider *straight-line* fragment[1][5] of this string theory logic. A word equation *cube* $\bigwedge_{i=1}^{n} e_i$ is said to be straight-line if it can be rewritten as the form $\bigwedge_{j=1}^{m} x_j = s_j$ such that: (i) $x_1, \cdots, x_m$ are different variables; and (ii) $FV(s_j) \cap \{x_1, x_2, \cdots, x_{j-1}\} = \emptyset$. A formula $\varphi = (\bigwedge_{i=1}^{n} e_i) \wedge \mathfrak{R} \wedge P$ is said to be straight-line if $\bigwedge_{i=1}^{n} e_i$ is straight-line, $\mathfrak{R}$ is a *tube* of regular membership $x \in \mathcal{R}$ or its negation $x \notin \mathcal{R}$, and P is a Presburger arithmetic formula.

$$
\begin{aligned}
&\mathcal{R} ::= \epsilon \mid a \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} + \mathcal{R} \mid \mathcal{R} \times \mathcal{R} \mid \mathcal{R}^* \mid \mathcal{R}^C \mid \mathcal{R}\{m, n\} \\
&\varphi ::= x \in \mathcal{R} \mid e \mid \beta \mid \neg\varphi \mid \varphi \vee \varphi \\
&e ::= s_1 = s_2 \\
&\beta ::= \alpha_1 \odot \alpha_2 \\
&s ::= \emptyset \mid \epsilon \mid a \mid x \mid \texttt{concat}(s_1, s_2) \\
&\alpha ::= 0 \mid 1 \mid i \mid \texttt{len}(s) \mid -\alpha \mid \alpha_1 + \alpha_2
\end{aligned}
$$

Table 1: Syntax of ESL

Note that we can convert an ESL formula to a DNF formula such that each term in the DNF formula is the form $(\bigwedge_{i=1}^{n} e_i) \wedge \mathfrak{R} \wedge P$ where P is a Presburger formula. We call the form $(\bigwedge_{i=1}^{n} e_i) \wedge \mathfrak{R} \wedge P$ as *normal form.* The straight-line fragment is similar to the SSA form of the program. Sometimes we use $s_1 s_2$ to stand for $\texttt{concat}(s_1, s_2)$ and $|x|$ to stand for $\texttt{len}(x)$.

Example 3.1 (Straight-line Formula).

$$\varphi_1 ::= x = \texttt{concat}(y, z) \wedge y = z \wedge x \in a\{1, 100\} \quad (1)$$

The word equation cube $c ::= (x = \texttt{concat}(y, z) \wedge y = z)$ is straight-line because it santisfy : (i) $FV(z) = \{z\}$, (ii)$\{z\} \cap \{x\} = \emptyset$. So the formula $\varphi_1 = c \wedge x \in a\{1, 100\}$ is also straight-line.

Example 3.2 (Non Straight-line Formula).

$$\varphi_2 ::= x = \texttt{concat}(y, z) \wedge y = \texttt{concat}(x, z) \wedge x \in a\{1, 100\} \quad (2)$$

φ_2 is not straight-line because the word equation cube $(x = \texttt{concat}(y, z) \wedge y = z)$ is not straight-line.

3.2 Semantics

Extended Regular Language We have introduced basic regular language at section ??. In our paper, we add a new operator *repetition* $\mathcal{R}\{m, n\}$, which means repeating regex $\mathcal{R}$ at least m times and at most n times. Repetition can be syntactically rewritten by concatenation and disjunction: $\mathcal{R}\{m, n\} \equiv \mathcal{R}^m \mid \cdots \mid \mathcal{R}^n$ where $\mathcal{R}^k$ defines concatenate k times($m \leq k \leq n$) to $\mathcal{R}$. However, if we naively rewrite the repetition operation, the automaton size will be huge (linear to repetition times), causing the search space to be exponential. To address the issue, we proposed a new automaton model called *Cost-Enriched Finite Automaton (CEFA),* whose size is constant to repetition times. CEFA can not handle nested repetition ($\mathcal{R}\{m, n\}$ or $\mathcal{R}*$, where $\mathcal{R}$ contains the repetition operator). So we have to rewrite it syntactically when the regex contains nested repetition (e.g., $a\{1, 100\}\{2, 2\}$ is rewritten to $a\{1, 100\}a\{1, 100\}$). Our regular expression is semantically equal to basic regular language but syntactically extended. We call the language of our regex *Extended Regular Language* and denote it to $\mathcal{L}^e(\mathcal{R})$.

Semantics We assume that S is the set of string variables over Σ^*, and I is the set of integer variables. $\eta : S \times \Sigma \rightarrow \Sigma^*$ is the interpretation on string where $\eta(c) = c$ for every letter $c \in \Sigma$ and $\eta(s_1 s_2) = \eta(s_1)\eta(s_2)$. $\theta : I \rightarrow \mathbb{Z}$ is the interpretation of arithmetic that is the same as that of Presburger arithmetic. Then the semantics is given by a satisfaction relation: $\eta, \theta \models \varphi$ defined in Table 2. We say a formula φ is *satisfiable* if a solution exists (η, θ) such as $\eta, \theta \models \varphi$. A formula φ is *unsatisfiable* if no solution exists.

$\eta, \theta \models \varphi_1 \vee \varphi_2$	iff	$\eta, \theta \models \varphi_1$ or $\eta, \theta \models \varphi_2$
$\eta, \theta \models \neg\varphi$	iff	$\eta, \theta \not\models \varphi$
$\eta, \theta \models x \in \mathcal{R}$	iff	$\exists w \in \mathcal{L}^e(\mathcal{R}), \eta, \theta \models x = w$
$\eta, \theta \models s_1 = s_2$	iff	$\eta(s_1) = \eta(s_2)$
$\eta, \theta \models \alpha_1 \odot \alpha_2$	iff	$\theta(\alpha_1) \odot \theta(\alpha_2)$ where $\odot = \{=, \geq\}$

Table 2: Semantics

3.3 Problem

In this paper, we consider the following problem:

Problem:	Is a ESL formula φ satisfiable?
Input:	A straight-line ESL formula φ in the normal form.
Output:	*sat* or *unsat.*

4 COST-ENRICHED FINITE AUTOMATON AND BASIC OPERATION

In this section, we will define the cost-enriched finite automaton and operations on it. The operations are intersection, union, complement, concatenation, and repetition.

4.1 Cost-Enriched Finite Automaton

Definition 4.1 (Cost-Enriched Finite Automaton). A cost-enriched finite automaton(CEFA) $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, I, F, R, \theta)$ where

- Q, Σ, I, F is defined as NFA,
- $R = (r_1 \cdots r_n)$ is a vector of mutually distinct cost registers. We use $\mathcal{V}(r_i)$ to denote the value of r_1. Note that the initial value of each register is 0,
- δ is the transition set whose elements are tuple $(q, a, q', \bar{v})$ where q, q' are states in Q, a is a letter in alphabet Σ and $\bar{v}$ is vector of natural number. We write the transition $(q, a, q', \bar{v})$ as $q \xrightarrow[\bar{v}]{a} q'$ for readability.
- θ is the linear integer arithmetic on registers.

A *run* of $\mathcal{A}$ on string $a_1 \cdots a_m$ is a transition sequence $q_0 \xrightarrow[\bar{v_1}]{a_1} q_1 \cdots q_{m-1} \xrightarrow[\bar{v_m}]{a_m} q_m$ such that $q_0 \in I$ and $r_i = \sum_{j=1}^{m} v_j(i)$ for $r_i \in R$. The run is *accepting* if $q_m \in F$ and $\theta[r_i/\mathcal{V}(r_i)]$ is satisfiable. Note that $\theta[r_i/\mathcal{V}(r_i)]$ means a substitution of all occurrences of r_i in θ to its current value. A string word w is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. The language of $\mathcal{A}$ is the set of string words accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$.

Example 4.2 (The CEFA $\mathcal{A}_{ab\{1,100\}}$). We build a CEFA $\mathcal{A}_{ab\{1,100\}}$ (table 1) whose language is the language of the extended regular expression $ab\{1, 100\}$. The transition set of $\mathcal{A}_{ab\{1,100\}}$ are $\{q_0 \xrightarrow[1]{a} q_1, q_1 \xrightarrow[0]{b} q_2, q_2 \xrightarrow[0]{\epsilon} q_0\}$. The register is r_1. And the linear arithmetic is $1 \leq r_1 \leq 100$. Intuitively, the accepted run of $\mathcal{A}_{ab\{1,100\}}$ will repeatedly run word ab. The transition $q_0 \xrightarrow[1]{a} q_1$ should appear at least once and at most 100 times because of the linear constraint $1 \leq r_1 \leq 100$.

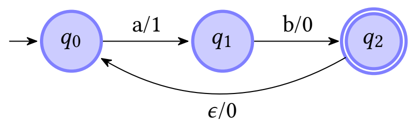


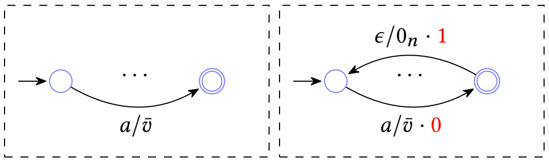
Figure 1: $\mathcal{A}_{ab\{1,100\}}$ **where** $R = (r_1)$ **and** $\theta = 1 \leq r_1 \leq 100$

The example 4.2 gives a general picture of constructing a CEFA for a specific extended regex. Firstly, we build an NFA from the extended regex $\mathcal{R}$ (e.g, To construct CEFA in example 4.2, we firstly construct an NFA with transition to be $q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{\epsilon} q_0$ and $I = q_0, F = q_2$). Secondly, we extend NFA to CEFA by adding registers to R and vectors $\bar{v}$ to each transition (e.g., we secondly add a register r_1 to store repetition times and add vector (1) to transition $q_0 \xrightarrow{a} q_1$ to update the repetition time). Finally, we set the linear arithmetic θ carefully to restrict the accepted word to be included in $\mathcal{L}(\mathcal{R})$ (e.g., we finally set $\theta = 1 \leq r_1 \geq 100$ to restrict the repetition times to $[1, 100]$). Besides the special operation repetition, other basic automaton operations are non-trivial. We individually discuss operations repetition, concatenation, union, intersection, complement, and Kleene.

4.2 Repetition

Given a CEFA $\mathcal{A} = (Q, \Sigma, \delta, I, F, R, \theta)$, its repetition with lower bound l and upper bound k is defined as $\mathcal{A}_{rep} = (Q, \Sigma, \delta', I, F, R \cup \{r'\}, \theta \wedge l \leq r' \leq k)$ where

- r' is the new register,
- and δ' is composed by transitions $q \xrightarrow[(v_1,\cdots,v_n,0)]{a} q'$ for all transitions $q \xrightarrow[(v_1,\cdots,v_n)]{a} q' \in \delta$, and transitions $q_m \xrightarrow[0_n \cdot 1]{\epsilon} q_0$ for all $q_m \in F$ and $q_0 \in I$,



(a) The CEFA $\mathcal{A}$ **(b) The repetition of $\mathcal{A}$**

Figure 2: Repetition

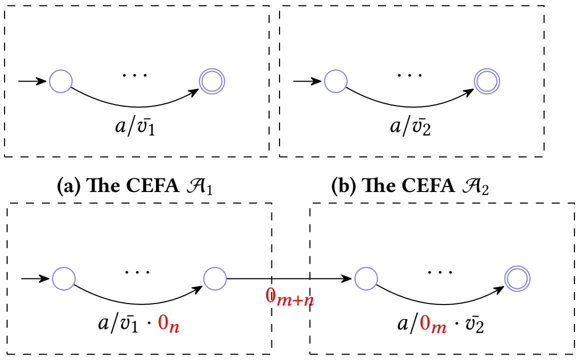
Figure 2 illustrates the transition relation between $\mathcal{A}$ and $\mathcal{A}_{rep}$. Informally, $\mathcal{A}_{rep}$ add a register r_{n+1} to store the repetition time and update the repetition time on the transitions starting from the initial state. $\mathcal{A}_{rep}$ add formula $l \leq r_{n+1} \leq k$ to ensure the repetition time is greater or equal to l and less or equal to k.

4.3 Concatenation

Given two CEFA $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, I_1, F_1, R_1, \theta_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, I_2, F_2, R_2, \theta_2)$ where $R_1 \cap R_2 = \emptyset$ and $|R_1| = m, |R_2| = n$, the concatenation is defined as $\mathcal{A}_{con} = (Q_1 \cup Q_2, \Sigma, \delta', I_1, F_2, R_1 \cup R_2, \theta_1 \wedge \theta_2)$ where δ' is composed by

- $q_1 \xrightarrow[\bar{v_1} \cdot 0_n]{a} q_1'$ for each transition $q_1 \xrightarrow[\bar{v_1}]{a} q_1' \in \delta_1$,

- $q_2 \xrightarrow[0_m \cdot \bar{v_2}]{a} q_2'$ for each transition $q_2 \xrightarrow[\bar{v_2}]{a} q_2' \in \delta_2$,
- and $q_1 \xrightarrow[0_{m+n}]{\epsilon} q_2$ for each $q_1 \in F_1$ and $q_2 \in I_2$.



(a) The CEFA $\mathcal{A}_1$ (b) The CEFA $\mathcal{A}_2$

(c) The concatenation of $\mathcal{A}_1$ and $\mathcal{A}_2$

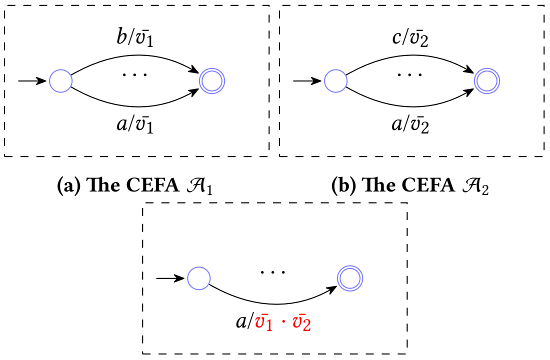
Figure 3: Concatenation

Figure 3 outlines the vector change on transitions when concatenating two CEFA. Without losing information, $\mathcal{A}_{con}$ contains all registers in $\mathcal{A}_1$ and $\mathcal{A}_2$. Furthermore, $\mathcal{A}_{con}$ update registers' value of $\mathcal{A}_1$ and $\mathcal{A}_2$ separately: $\mathcal{A}_{con}$ only update registers of R_1 on the transitions of $\mathcal{A}_1$ and update registers of R_2 on the transition of $\mathcal{A}_2$. θ' is the conjunction of θ_1 and θ_2 to make sure all linear constraints in $\mathcal{A}_1$ and $\mathcal{A}_2$ are satisfiable.

4.4 Intersection

Given two CEFA $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, I_1, F_1, R_1, \theta_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, I_2, F_2, R_2, \theta_2)$ with $R_1 \cap R_2 = \emptyset$, the intersection is defined as $\mathcal{A}_{inter} = (Q_1 \times Q_2, \Sigma, \delta', I_1 \times I_2, F_1 \times F_2, R_1 \cup R_2, \theta_1 \wedge \theta_2)$ where δ' is composed by

- transitions $(q_1, q_2) \xrightarrow[\bar{v_1} \cdot \bar{v_2}]{a} (q_1', q_2')$ when both transition $q_1 \xrightarrow[\bar{v_1}]{a} q_1'$ and transition $q_2 \xrightarrow[\bar{v_2}]{a} q_2'$ exist in δ.



(a) The CEFA $\mathcal{A}_1$ (b) The CEFA $\mathcal{A}_2$

(c) The intersection of $\mathcal{A}_1$ and $\mathcal{A}_2$

Figure 4: Intersection

The transition relation of $\mathcal{A}_1, \mathcal{A}_2$ and their intersection is displayed by Figure 4. The intersection of CEFA is similar to the intersection of NFA, except that the intersection of CEFA intersects registers updates and linear arithmetic.

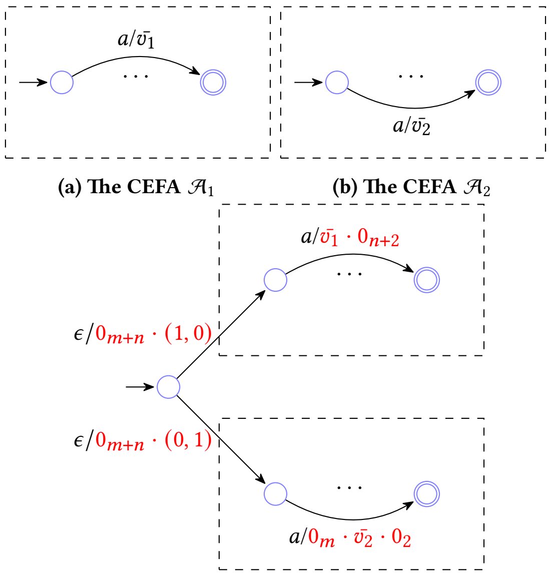
4.5 Union

Given two CEFA $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, I_1, F_1, R_1, \theta_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, I_2, F_2, R_2, \theta_2)$ with $R_1 \cap R_2 = \emptyset$ and $|R_1| = m, |R_2| = n$, the union is defined as $\mathcal{A}_{union} = (Q_1 \cup Q_2, \Sigma, \delta', \{q_0\}, F_1 \cup F_2, R_1 \cup R_2 \cup (r', r''), \theta')$ where δ' is composed by

- transitions $q_0 \xrightarrow[0_{m+n} \cdot (1,0)]{\epsilon} q_1$ for all $q_1 \in I_1$,
- transitions $q_0 \xrightarrow[0_{m+n} \cdot (0,1)]{\epsilon} q_2$ for all $q_2 \in I_2$,
- transitions $q_1 \xrightarrow[\bar{v_1} 0_{n+2}]{a} q_1'$ for all $q_1 \xrightarrow[\bar{v_1}]{a} q_1' \in \delta_1$,
- transitions $q_2 \xrightarrow[0_m \bar{v_2} 0_2]{a} q_2'$ for all $q_2 \xrightarrow[\bar{v_2}]{a} q_2' \in \delta_2$,

and

- r' and r'' are new registers,
- q_0 is a new state where $q_0 \notin Q_1$ and $q_0 \notin Q_2$,
- $\theta_{tmp} = (r' > 0 \wedge \theta_1) \vee (r'' > 0 \wedge \theta_2)$, θ' is constructed differently in four cases:
  - $I_1 \cap F_1 = \emptyset$ and $I_2 \cap F_2 = \emptyset$: $\theta' = \theta_{tmp}$;
  - $I_1 \cap F_1 \neq \emptyset$ and $I_2 \cap F_2 = \emptyset$: $\theta' = \theta_{tmp} \vee (r' == 0 \wedge r'' == 0 \wedge \theta_1)$;
  - $I_1 \cap F_1 = \emptyset$ and $I_2 \cap F_2 \neq \emptyset$: $\theta' = \theta_{tmp} \vee (r' == 0 \wedge r'' == 0 \wedge \theta_2)$;
  - $I_1 \cap F_1 \neq \emptyset$ and $I_2 \cap F_2 \neq \emptyset$: $\theta' = \theta_{tmp} \vee (r' == 0 \wedge r'' == 0 \wedge (\theta_1 \vee \theta_2))$;


$a/\bar{v_1}$
$a/\bar{v_2}$
$a/\bar{v_1} \cdot 0_{n+2}$
$\epsilon/0_{m+n} \cdot (1,0)$
$\epsilon/0_{m+n} \cdot (0,1)$
$a/0_m \cdot \bar{v_2} \cdot 0_2$

(a) The CEFA $\mathcal{A}_1$ (b) The CEFA $\mathcal{A}_2$

(c) The union of $\mathcal{A}_1$ and $\mathcal{A}_2$

Figure 5: Union

The Union of two CEFA is shown in Figure 5. To simulate the semantics of the union, we add an initial state q_0 and transitions from q_0 to initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$ to randomly choose one automaton. The values of r' and r'' distinguish which automaton we run.

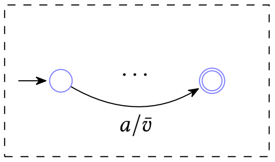
In the construction above, $r' > 0$ means that we choose $\mathcal{A}_1$ to run a word, $r'' > 0$ implies that we select r'' to run a word, and $r' == 0 \wedge r'' = 0$ means that we decide to run word ϵ.

4.6 Complement

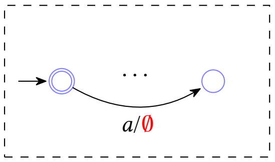
We use the function nfa to return the NFA form of a CEFA. More precisely, given a CEFA $\mathcal{A} = (Q, \Sigma, I, F, \delta, R, \theta)$, $nfa(\mathcal{A})$ is an NFA tuple $(Q, \Sigma, \delta', I, F)$ where δ' comprises transition (q, a, q') for $(q, a, q', \bar{v}) \in \delta$. Similarly, the function $cefa(\mathcal{A})$ is used to return the CEFA form of an NFA: $cefa(\mathcal{A}_{nfa}) = (Q, \Sigma, I, F, \delta, \emptyset, true)$ for $\mathcal{A}_{nfa} = (Q, \Sigma, I, F, \delta')$ where δ comprises transition $(q, a, q', \emptyset)$ for $(q, a, q') \in \delta'$. uion denotes the CEFA operation union.

Given a CEFA $\mathcal{A} = (Q, \Sigma, \delta, I, F, R, \theta)$, the complement of $\mathcal{A}$ is defined as $\mathcal{A}_{comp} = \texttt{union}(\mathcal{A}^C_{NFA}, \mathcal{A}_{negf})$ where $\mathcal{A}_{negf} = (Q, \Sigma, \delta, I, F, R, \neg\theta)$ and $\mathcal{A}^C_{NFA}$ is obtained by three steps:

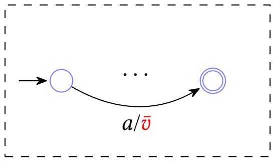
- compute the NFA form $\mathcal{A}_n = nfa(\mathcal{A}) = (Q, \sigma, \delta', I, F)$,
- the complement of $\mathcal{A}_n$ is $\mathcal{A}^C_n = (Q, \sigma, \delta', F, I)$,
- extend $\mathcal{A}^C_n$ to CEFA form $\mathcal{A}^C_{NFA} = cefa(\mathcal{A}^C_n)$.



(a) The CEFA $\mathcal{A}$



(b) The CEFA $\mathcal{A}^C_{NFA}$



(c) The CEFA $\mathcal{A}_{negf}$

Figure 6: Complement

As Figure 6 shows, the complement of CEFA is the union of $\mathcal{A}^C_{NFA}$ and $\mathcal{A}_{negf}$. $\mathcal{A}^C_{NFA}$ accepts words that $nfa(\mathcal{A})$ does not accept and $\mathcal{A}_{negf}$ accepted words might be accepted by the $nfa(\mathcal{A})$ but not satisfy linear constraints θ.

4.7 Kleene

The construction of Kleene is trivial. Given a CEFA $\mathcal{A} = (Q, \Sigma, \delta, I, F, R, \theta)$ with $|R| = n$, the Kleene of $\mathcal{A}_{Kleene}$ is defined as $(Q, \Sigma, \delta', I, F, R, \theta)$ where δ' comprises all transitions in δ and new transitions $q_f \xrightarrow[0_n]{\epsilon} q_0$ for each $q_0 \in I$ and $q_f \in F$.

4.8 From Extended Regex to CEFA

Basis: The basis has three parts with empty registers and valid linear arithmetic, shown in Fig. 7. In part 7a, we see how to handle the expression ϵ. The language of the automaton is easily seen to be $\{\epsilon\}$ since the only path from the start state to an accepting state is labeled ϵ. Part 7b gives the CEFA for a regular expression **a**. The language of this CEFA consists of the single string a, which is also $\mathcal{L}(\mathbf{a})$. Finally, part 7c shows the construction for $\emptyset$. There are no paths from the initial state to the accepting state.

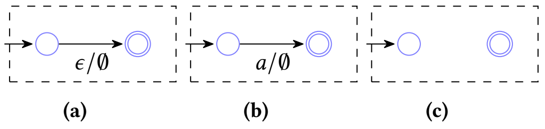


Figure 7: The basis of the construction of a CEFA from extended regex

Induction: Given the CEFA of the subexpression, we construct the CEFA in five cases:

(1) The expression is $\mathcal{R}\{m, n\}$ for some smaller expression $\mathcal{R}$. We can repeat the CEFA of $\mathcal{R}$ with lower bound m and upper bound n as shown in section 4.2.
(2) The expression is $\mathcal{R}_1 \cdot \mathcal{R}_2$ for some smaller expression $\mathcal{R}_1$ and $\mathcal{R}_2$. We can concatenate the CEFA of $\mathcal{R}_1$ and the CEFA of $\mathcal{R}_2$ as shown in section 4.3.
(3) The expression is $\mathcal{R}_1 \times \mathcal{R}_2$ for some smaller expression $\mathcal{R}_1$ and $\mathcal{R}_2$. We can intersect the CEFA of $\mathcal{R}_1$ and the CEFA of $\mathcal{R}_2$ as shown in section 4.4.
(4) The expression is $\mathcal{R}_1 + \mathcal{R}_2$ for some smaller expression $\mathcal{R}_1$ and $\mathcal{R}_2$. We can union the CEFA of $\mathcal{R}_1$ and the CEFA of $\mathcal{R}_2$ as shown in section 4.5.
(5) The expression is $\mathcal{R}*$ for some smaller expression $\mathcal{R}$. We can construct the Kleene Star of the CEFA of $\mathcal{R}$ as shown in section 4.7.
(6) The expression is $\mathcal{R}^C$ for some smaller expression $\mathcal{R}$. We can construct the complement of the CEFA of $\mathcal{R}$ as shown in section 4.6

Special Case: When the expression $\mathcal{R}$ contains nested repetition, we firstly write $\mathcal{R}$ to a new regex $\mathcal{R}'$ without nested repetition. Then we build the CEFA of the new expression $\mathcal{R}'$ by basis and induction steps.

5 SOLVE STRING CONSTRAINTS WITH LARGE REPETITION BOUND

Following the decision procedure defined in the paper [1], we propose an efficient algorithm to solve string constraints by CEFA. Firstly, we replace all length operations $i = \texttt{len}(x)$ to pre-images, and intersect these pre-images to $lenAut_x$. Then for each term x, we genearte CEFAs from all regular expressiones $\mathcal{R}$ in all terms $x \in \mathcal{R}$, and intersect the CEFAs with $lenAut_x$ to get the final CEFAs. Finally, we check whether the final CEFAs is empty under the linear integer arithmetic P. If one of the final CEFAs is empty, the string constraints are unsatisfiable. Otherwise, the string constraints are satisfiable. The high-level algorithm is shown in the Algorithm 1.

The emptiness checking problem of CEFAs under linear integer arithmetic P is theoretically pspace-complete[1]. To solve it efficiently for practical example, we add some heuristics. The details of the heuristics are shown in Algorithm 2.

5.1 High-level algorithm

The pseudocode presented in Algorithm 1 outlines the framework of our solving process. The implemented details are eliminated for

Algorithm 1 High-level algorithm

```
Input: Conjunction $\varphi$ of the form $x \in \mathcal{R}$, and conjunction of linear integer arithmetic $P$ over string lengths
Output: $sat$ or $unsat$

1:  for all length operation $i = \mathtt{len}(x)$ do
2:      $lenAut_x \leftarrow \mathcal{A}_{allstring}$
3:      Let $\mathcal{A}_i$ be the pre-image of length operation with length $i$
4:      $lenAut_x \leftarrow lenAut_x \times \mathcal{A}_i$
5:  end for
6:  $atomAuts \leftarrow \emptyset$
7:  for all string variables $x$ occurring in $\varphi$ do
8:      Let $S$ be the set of all regexes $\mathcal{R}$ in all terms $x \in \mathcal{R}$
9:      CEFA $\mathcal{A}_x \leftarrow$ intersection of $lenAut_x$ and all CEFAs corresponding to regexes in $S$
10:     if $nfa(\mathcal{A}_x)$ is empty then
11:         return $unsat$
12:     end if
13:     $atomAuts \leftarrow atomAuts \cup \{\mathcal{A}_x\}$
14: end for
15: if $\mathtt{isEmpty}(atomAuts, P)$ then
16:     return $unsat$
17: else
18:     return $sat$
19: end if
```

clarity. From line 1 to line 10, we construct the CEFAs of all length operations $i = \mathtt{len}(x)$ occurring in φ. The CEFA $lenAut_x$ is the automaton reserving all length information on x. $lenAut_x$ is initiated to accept all strings at line 2. And for each length operation $i = \mathtt{len}(x)$, we construct the pre-image of the length operation at line 3. Then we intersect $lenAut_x$ with the pre-image at line 4. Note that the final $lenAut_x$ is the intersection of many pre-images.

In the following steps, we call a CEFA to be *atomic* if we will not operate it anymore. The set *atomAuts* is the set of all atomic CEFAs for all terms x occurring in φ, and it is initiated to empty set at line 6. From line 6 to line 14, we construct the atomic CEFAs of all string variables x occurring in φ. For each string variable x, we construct the CEFA $\mathcal{A}_x$ by intersecting $lenAut_x$ and all CEFAs corresponding to regexes in S. If the NFA form of $\mathcal{A}_x$ is empty (i.e., the NFA accepts no word), we return *unsat* directly because the NFA form is an over-approximation of CEFA. Otherwise, we add $\mathcal{A}_x$ to *atomAuts* at line 9.

After obtaining the atomic CEFAs of all string terms, we check whether the atomic CEFAs are empty under the linear integer arithmetic P at line 15. If the atomic CEFAs are empty, we return *unsat*. Otherwise, we return *sat*. The emptiness checking is complex, and we will discuss its details in the following subsection.

5.2 Emptyness Checking

As mentioned, the emptiness checking problem of CEFA under linear integer arithmetic P is theoretically pspace-complete [1]. In our previous research, we rewrote CEFA to an infinite system and used a model-checking tool *NuSMV* to solve it. In this paper, we use a more efficient method to solve the emptiness checking problem. The details are shown in Algorithm 2.

The input CEFAs may have many transitions and registers, so the emptiness checking problem is hard to solve. We simplify the CEFAs at line 1. The simplification is to remove all duplicated transitions and registers in the CEFAs. The simplification is implemented by Algorithm 3. Then we try to find a solution by under-approximation at line 3. The under-approximation is implemented by Algorithm 4. If we find a solution, the CEFAs under linear integer arithmetic P are not empty. So we return $false$ at line 5. We will try many times until we find a solution or reach *MaxBound*. Otherwise, we try to find an unsat core by over-approximation at line 8. The over-approximation is implemented by Algorithm 5. If we discover an unsat core, we know that the CEFAs under P are empty, and we return $true$ at line 10. Suppose both under- and over-approximation do not make sense. In that case, we compute the Parikh images[8] of the simplified CEFAs and check if the Parikh images in conjunction with P are satisfiable at line 12. The unsatisfiability implies emptiness directly.

Algorithm 2 $\mathtt{isEmpty}(auts, P)$

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

1:  $simpliAuts \leftarrow \mathtt{simplify}(auts)$
2:  for $bound \leftarrow 1, MaxBound$ do
3:      $\varphi_{under} \leftarrow \mathtt{underApprox}(simpliAuts, bound)$
4:      if $\varphi_{under} \wedge P$ is sat then
5:          return $false$
6:      end if
7:  end for
8:  $\varphi_{over} \leftarrow \mathtt{overApprox}(simpliAuts)$
9:  if $\varphi_{over} \wedge P$ is unsat then
10:     return $true$
11: end if
12: $\varphi \leftarrow \mathtt{parikhImage}(simpliAuts)$
13: if $\varphi \wedge P$ is unsat then
14:     return $true$
15: else
16:     return $false$
17: end if
```

5.3 Simplification of CEFA

The main idea of simplification is to remove duplicated transitions and registers. In emptiness checking Algorithm 2, the vectors on the transitions are meaningful, but the letters on the transitions are not. So we see the alphabet of the CEFA as unary (i.e., the alphabet is $\{a\}$). Compared with NFA, the vectors in the CEFA are the letters in the NFA, and the vector $\bar{0}_n$ is seen as ϵ. Based on the similarity between NFA and CEFA, firstly, determinization and minimization [4] are applied to simplify the CEFA. Then we merge the registers having the same updating at all transitions and get the final CEFA. The main simplification framework is shown in Algorithm 3, and some details are eliminated for simplicity.

Algorithm 3 `simplify`($auts$)

```
Input: CEFAs $auts$
Output: Simplified CEFAs $simpliAuts$

1: $simpliAuts \leftarrow \emptyset$
2: for $aut \in auts$ do
3:     $aut \leftarrow$ determinizeByVec($aut$)
4:     $aut \leftarrow$ minimizeByVec($aut$)
5:     $aut \leftarrow$ mergeRegisters($aut$)
6:     $simpliAuts \leftarrow simpliAuts \cup \{aut\}$
7: end for
8: return $simpliAuts$
```

5.4 Under-Approximation

To solve the emptiness checking problem, we translate the CEFAs to linear integer arithmetic and check if the linear integer arithmetic in conjunction with P is satisfiable. The main idea of under-approximation is to translate the CEFAs step by step. A string length bound controls the translation, and linear integer arithmetic is generated from low to high length bound. This section only discusses the situation where the bound is fixed. The details are shown in the Algorithm 4. Firstly, the linear integer arithmetic φ_{under} is assigned to be false at line 1. Then for each CEFA, we enumerate all the accepted paths with lengths less than the bound and translate the paths to linear integer arithmetic. The linear integer arithmetic is in conjunction with φ_{under} at line 4.

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

```
Input: CEFAs $auts$ and length bound $bound$
Output: Linear integer arithmetic $\varphi_{under}$

1: $\varphi_{under} \leftarrow false$
2: for $aut \in auts$ do
3:     Let $\varphi_{aut}$ to be the linear integer arithmetic generated by
       enumerating all the paths with length less than $bound$ in $aut$
4:     $\varphi_{under} \leftarrow \varphi_{under} \wedge \varphi_{aut}$
5: end for
6: return $\varphi_{under}$
```

5.5 Over-Approximation

Sometimes when the bound reaches the maximum, the linear integer arithmetic φ_{under} is still unsatisfiable. We must use over-approximation to solve the emptiness checking problem in this case. The main idea of over-approximation is to split each CEFA into sub-CEFAs, translate each sub-CEFA respectively, and finally in conjunction with the LIA of all sub-CEFAs. The details are shown in the Algorithm 5. Firstly, each CEFA is split into n sub-CEFAs, where n is the number of registers in the CEFA. The sub-CEFA has only one register. After simplifying each sub-CEFA by Algorithm 3, the vector of each transition must be $\bar{1}$. So the register value of each sub-CEFA equals the length of the accepted path. We apply the efficient construction of semilinear representations[6] to get the linear integer arithmetic φ_{over}.

Algorithm 5 `overApprox`($auts$)

```
Input: CEFAs $auts$
Output: Linear integer arithmetic $\varphi_{over}$

1: $\varphi_{over} \leftarrow true$
2: for $aut \in auts$ do
3:     Let $subAut$ be the sub-CEFA of $aut$ with only one register
4:     Let $subAut \leftarrow$ simplify($subAut$)
5:     Let $\varphi_{subAut}$ be the linear integer arithmetic generated by
       efficient construction of semilinear representations
6:     $\varphi_{over} \leftarrow \varphi_{over} \wedge \varphi_{subAut}$
7: end for
8: return $\varphi_{over}$
```

6 IMPLEMENTATION AND EXPERIMENT

TODO

7 CONCLUSION AND FUTURE WORK

In this paper, we aim to solve the string constraints with large repetition time efficiently. A new automaton model CEFA is proposed to reduce the search space. Many basic operations on CEFA are non-trivial and we should formal them carefully. Moreover, we extend the algorithm in the paper [1] with heuristics such as under- and over-approximation. The implementation of the algorithm is done on string solver ostrich. The extensive empirical comparison against z3 over a large and diverse benchmark shows the power of our model and algorithm. In the future, we plan to explore the way to solve nested repetition and use CEFA to solve more string operations.

REFERENCES

[1] Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 325–342.

[2] James C. Davis. 2019. Rethinking Regex Engines to Address ReDoS *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1256–1258. https://doi.org/10.1145/3338906.3342509

[3] Vijay Ganesh and Murphy Berzish. 2016. Undecidability of a Theory of Strings, Linear Arithmetic over Length, and String-Number Conversion. (05 2016).

[4] John E. Hopcroft and Jeff D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company.

[5] Quang Loc Le and Mengda He. 2018. A Decision Procedure for String Logic with Quadratic Equations, Regular Expressions and Length Constraints. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 350–372.

[6] Zdeněk Sawa. 2010. Efficient Construction of Semilinear Representations of Languages Accepted by Unary NFA. In *Reachability Problems*, Antonín Kučera and Igor Potapov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–182.

[7] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. 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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 620–635. https://doi.org/10.1145/3453483.3454066

[8] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. 2005. On the Complexity of Equational Horn Clauses. In *Proceedings of the 20th International Conference on Automated Deduction* (Tallinn, Estonia) *(CADE' 20)*. Springer-Verlag, Berlin, Heidelberg, 337–352. https://doi.org/10.1007/11532231_25