A Symbolic Model Checking Approach to the Analysis of String and Length Constraints

Hung-En Wang
Graduate Institute of Electronics Engineering
National Taiwan University
Taiwan

Shih-Yu Chen
Graduate Institute of Electronics Engineering
National Taiwan University
Taiwan

Fang Yu
Department of Management Information Systems
National Chengchi University
Taiwan

Jie-Hong R. Jiang
Graduate Institute of Electronics Engineering, Department of Electrical Engineering
National Taiwan University
Taiwan

ABSTRACT

Strings with length constraints are prominent in software security analysis. Recent endeavors have made significant progress in developing constraint solvers for strings and integers. Most prior methods are based on deduction with inference rules or analysis using automata. The former may be inefficient when the constraints involve complex string manipulations such as language replacement; the latter may not be easily extended to handle length constraints and may be inadequate for counterexample generation due to approximation. Inspired by recent work on string analysis with logic circuit representation, we propose a new method for solving string with length constraints by an implicit representation of automata with length encoding. The length-encoded automata are of infinite states and can represent languages beyond regular expressions. By converting string and length constraints into a dependency graph of manipulations over length-encoded automata, a symbolic model checker for infinite state systems can be leveraged as an engine for the analysis of string and length constraints. Experiments show that our method has its unique capability of handling complex string and length constraints not solvable by existing methods.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Security and privacy** → *Logic and verification*;

KEYWORDS

length constraint, string analysis, symbolic model checking

ACM Reference Format:
Hung-En Wang, Shih-Yu Chen, Fang Yu, and Jie-Hong R. Jiang. 2018. A Symbolic Model Checking Approach to the Analysis of String and Length Constraints. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238189

1 INTRODUCTION

Formal analysis of string manipulating programs are crucial to detect and rectify service applications. Many web service vulnerabilities are due to improper input validation and sanitization. Any error in validation and sanitization of user input can lead to a significant vulnerability for a web application, and can be exploited by malicious users all around the world. Input validation and sanitization in web applications is crucial and challenging since communication between the different layers of a web application occurs through directives (commands) that often embed user input and are written in many languages, such as XML, SQL, and HTML. Programs that propagate and use malicious user inputs without validation and sanitization, or with improper validation and sanitization, are vulnerable to attacks such as XSS and SQL injection. In fact, to date these two types of vulnerabilities in web applications are notorious, and they are widely publicized [1]. This fact indicates the intrinsic difficulty of input validation and sanitization problems and demonstrates the need for automated tool support that can help programmers in preventing vulnerabilities due to errors in string manipulating programs. Even though many string constraint solving techniques, e.g., [2–9, 11, 12], have been proposed, analyzing complex string manipulation programs remains challenging.

String constraint solving can be particularly hard when the constraints involve complex string operations and involve both string and integer variables [13]. Specifically, it has been shown that solving string constraints with *replace all* and length constraints is undecidable [12, 14]. The *replace all* operation is widely used in input sanitization functions in Web applications. It defines the replace of a match pattern with a replacement pattern for the sentence within a given set of language. Such an operation can be implemented with finite state transducers [15, 16] or automata [17], and hence can be leveraged to solve string constraints with replacement [8]. While an automata-based string constraint solver takes advantage of automata construction for string manipulations of unbounded strings, it often adopts regular abstraction to approximate solutions that cannot be represented with finite-state automata. This regular approximation limits the ability of presentations on relations among string variables [18]. For example, to deal with length constraints, one approach is to derive and refine arithmetic automata, a.k.a. length automata, in [8, 19] from string operations to track length relations during the computation. The constraint on length of each string variable can then be derived from the arithmetic automata to refine the string automata. This method over approximates the relation of lengths of string variables in arithmetic automata. On the other hand, the SMT-based approaches, e.g., [2–7], for string constraint solving are native to deal with length constraints. While these DPLL(T)-based solvers handle a variety of string constraints, including word equations, regular expression membership, length constraints, and (more rarely) regular/rational relations; the solvers can not handle some string manipulations such as replace all. A recent work that extends S3 [2] to S3P [20] addresses this issue with recurrence to reason such operations. Later we show that the recursive operation may cause non-termination, and lead to non-robust results of constraint solving. Trau [11] integrates SMT solvers with the CEGAR framework using flat automata to over and under approximation the solution set.

In this work, we propose a new symbolic model checking approach to mixed string and integer constraint solving. A class of infinite state automata is introduced to encode length information within string automata. A length-encoded automaton keeps the length information of strings as integer values in its states. Through manipulations over length-encoded automata, we can not only model complex string operations, but also deal with length constraints. The constructed length-encoded automaton at the sink node of a dependency graph precisely characterize the corresponding mixed string and integer constraints. Symbolic model checking algorithms for infinite state systems can then be exploited for language emptiness checking of the length-encoded automata.

Building upon the SLOG framework [9], the proposed method is implemented as a tool, named Slent [10], to handle both string and length constraints. We extend the circuit representation of finite-state automata [9] to infinite-state automata by adding integer variables for tracking length information. We exploit software model checking algorithms [21, 22] as the backend computation engine. Experiments show the unique advantages of Slent compared to modern string constraint solvers, such as S3P [20], Z3STR3 [3], CVC4 [4], Norn [7], ABC [8], and TRAU [11]. Our method uniquely solves a number of complex benchmarks and can support counterexample generation of input strings as witness when the set of constraints under analysis is satisfiable.

2 OVERVIEW

To motivate, consider the following example of constraints

$$\begin{aligned} X_1 &\in a^*, \\ X_2 &\in b^*, \\ X_3 &= X_1.X_2, \\ X_4 &= \textsc{replace}(X_3, a^+b, ba), \\ \textsc{Len}(X_1) &= \textsc{Len}(X_2), \\ \textsc{Len}(X_1) &> \textsc{Len}(X_4), \end{aligned}$$

where the replace operator $Y = \textsc{replace}(X, R_1, R_2)$ is a language-to-language *replace-all* operation [17][1], which can be alternatively defined by

$$\begin{aligned} &((Y = X) \wedge X \notin (\Sigma^*.R_1.\Sigma^*)) \vee \\ &((X = X_1.X_2.X_3) \wedge (X_1 \notin (\Sigma^*.R_1.\Sigma^*)) \wedge \\ &(X_2 \in R_1) \wedge (Y = X_1.Y_1.Y_2) \wedge (Y_1 \in R_2) \wedge \\ &(Y_2 = \textsc{replace}(X_3, R_1, R_2)), \end{aligned}$$

and the Len operation constrains the length of a string variable. Note that the above constraints ($X_3 = X_1.X_2$) and ($\textsc{Len}(X_1) = \textsc{Len}(X_2)$) ensure that X_3 is in the language $a^n b^n$, for $n \geq 0$ being the lengths of X_1 and X_2. Then X_4 is obtained by performing *language to language replacement* on X_3. For $X_4 = \textsc{replace}(X_3, a^+b, ba)$, a substring of the form $a^m b$, for some $1 \leq m \leq n$, in the middle of $a^n b^n$ will be replaced with ba. In this case, we have $\textsc{Len}(X_4) = 2n-(m+1)+2 > n = \textsc{Len}(X_1)$, which contradicts the last constraint $\textsc{Len}(X_1) > \textsc{Len}(X_4)$. Hence the set of constraints is unsatisfiable.

For rule-based solvers, solving constraints involving both replace-all (in contrast to *replace-first* [23]) and length operations is difficult for three reasons. First, the recursive definition of replace-all operation may induce nontermination [20]. Second, in the recursive definition of replace-all operation querying non-membership of the form ($X \notin \Sigma^*.R.\Sigma^*$), where R is a regular expression, is fundamentally difficult for rule-based solvers. In fact, even the simple example ($X \notin a^*$) $\wedge$ ($X \in a^*a$) can be difficult for solvers such as CVC4 [4], Z3STR3 [3], and S3P [20] to solve. Third, the length relation between strings before and after several string operations is hard to be captured. For instance, let X_1 = "*aa*," X_2 = "*bb*," X_3 = "*aabb*," and X_4 = "*abab*" (obtained by replacing the middle *ab* in X_3 with *ba*). Then the first symbol "*a*" in X_1 is also remained in X_4, and contribute to both the lengths of X_1 and X_4. Such relations on string lengths may be even more complicated and harder to capture when more operations are involved between two strings.

For automata-based solvers, the replacement operation can be naturally achieved by automata-based construction [17]. However, the satisfying values of variables X_1, X_2, X_3, X_4 in the above example are not regular due to the condition imposed by the length constraints. They cannot be represented precisely with finite-state automata. To cope with length constraints, prior work [8, 24] constructed automata for string constraints and for length constraints separately. The arithmetic automata derived from length constraints were used to refine the string automata derived from string constraints. Nevertheless, the relations among variables cannot be kept precisely. This impreciseness may over-approximate the solutions to the above set of constraints and mistakenly conclude its satisfiability. In prior work [11], a CEGAR-based framework was proposed using flat automata for constraint refinement. Without capturing the length relation between variables before and after replacement, the method might not have proper refinement to guarantee termination on the above example. Recent work [25] exploited alternating automata to represent string formulas consisting of word equations,

[1]The recursive *replace-all* operation defined in [20] specifies each to-be-replaced substring being the leftmost possible candidate. In contrast, the replace-all operation considered in this work follows [17]. It allows a to-be-replaced substring being at any position, and thus may yield more replacement solutions than [20]. We refer to the two replace-all definitions as *replace-all-first* of S3P and *replace-all-any* of Slent.

regular constraints, and rational constraints. Although it may handle length constraints and relation among variables, its scope is limited to some decidable fragment of formulas. The above example is unfortunately beyond the decidable fragment as variable X_4 by the method would appear in two rational constraints.

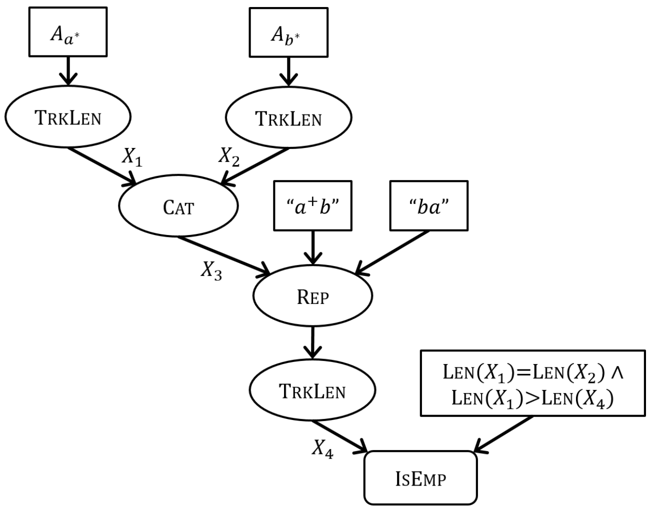

Figure 1: Dependency graph for constraint solving.

In this work, we propose length-encoded automata to characterize string and integer constraints. We assume the string and length constraints can be represented with a *dependency graph*, which is a tree that specifies how the length-encoded automata are constructed with respect to the given constraints. Figure 1 shows the dependency graph of automata construction corresponding to the above example of constraints. In the graph, each *source node* (without incoming edges) corresponds to a finite automaton accepting a pre-specified regular language, and each internal node corresponds to an automata operation such as concatenation, replacement, etc. For each *internal node* with operation Op, an automaton can be constructed by performing the operation Op on its immediate upstream automata. We introduce a new TrkLen operation to attach an integer variable globally seen by all states of an automaton to track the length information. A non-epsilon transition of an automaton should incur a length increment by one, and thus the integer indicates the length of the string currently taken by the automaton. Such automata with length encoded integers are referred to as *length-encoded automata*. By setting the initial value of an integer to zero, after taking an input sequence, the final value of the integer will be the length of this sequence. Note that the length constraints and initialization of the length integers are only imposed on the automaton, called *sink automaton*, at the *sink node* (without outgoing edges) of the dependency graph before the final language emptiness checking IsEmp. The set of length constraints is imposed as part of the accepting condition of the sink automaton. In an automata operation, an integer carried by an automaton A_i, which is involved in the operation, increases its value only when a transition takes place among the states of A_i, and retains its value when a transition takes place among the states of other automaton A_j for $j \neq i$. Thereby, the proposed method can precisely capture

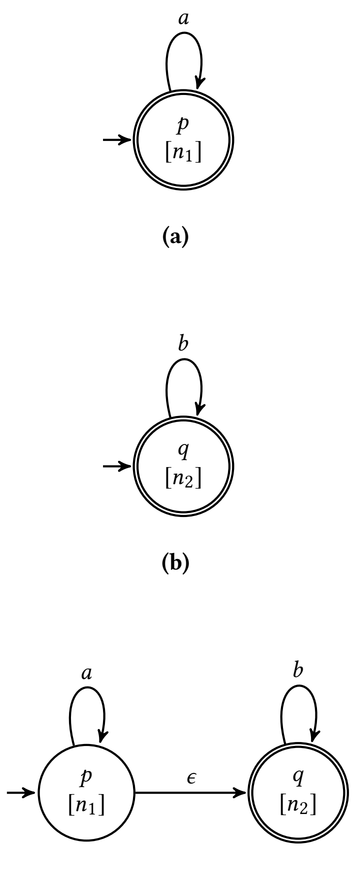

Figure 2: Length-encoded automata for (a) A_1, (b) A_2, and (c) A_3.

complex length relations among different strings with replacement involved.

To illustrate how integers are carried through automata operations, we consider in Figure 1 the construction of automaton A_3 for string variable X_3, which is the result of concatenating string variables X_1 and X_2. Figure 2 (a) and (b) show the two length-encoded automata A_1 and A_2 for variables X_1 and X_2, respectively. Automaton A_1 has one state p and recognizes the language a^*, and A_2 has one state q and recognizes the language b^*. By length tracking, A_1 is attached with an integer n_1, whose value increments whenever a non-ϵ-transition takes place among states sharing n_1. It records the length of the input sequence taken by A_1 so far. Similarly, A_2 is attached with an integer n_2. In general, a state of an automaton can possibly be attached with one or multiple integers, denoted $[n_{i_1}, \ldots, n_{i_k}]$, through a sequence of automata operations.

In Figure 2 (c), automaton A_3 is constructed by concatenating A_1 and A_2. In A_3, there are two states, state p attached with n_1 is from A_1, and state q attached with n_2 is from A_2. The integer n_1 (respectively n_2) increases its value only when the self-transition on state p (respectively q) takes place. Essentially, n_1 counts the number of a's taken so far on state p, and n_2 counts the number of b's taken so far on state q. If n_1 and n_2 are initialized to 0 and the constraint $n_1 = n_2$ is added to the accepting condition, then automaton A_3 accepts exactly the context free language $\{a^n b^n \mid n \in \mathbb{N}\}$. In fact, even the context-sensitive language $\{a^n b^m c^n d^m \mid n, m \in \mathbb{N}\}$ can be represented with the length-encoded automata.

3 PRELIMINARIES

An *automaton* is a 5-tuple (Q, Σ, I, O, T), where Q is a set of states, Σ is a finite set of input alphabet, $I \subseteq Q$ is the set of initial states, $O \subseteq Q$ is the set of accepting states, and $T \subseteq \Sigma \times Q \times Q$ is the transition relation. In this work, Q can be an infinite state set. In the sequel, we shall represent the initial states, accepting states, and transition relation by their characteristic functions $I : Q \to \mathbb{B}$, $O : Q \to \mathbb{B}$, and $T : \Sigma \times Q \times Q \to \mathbb{B}$, respectively, for $\mathbb{B}$ being the Boolean domain $\{\text{FALSE}, \text{TRUE}\}$. (A characteristic function χ is a propositional function representing a set S by having $\chi(e) = 1$ (True) if $e \in S$ and $\chi(e) = 0$ (False) if $e \notin S$.) We refer $\vec{x}$, $\vec{s}$ and $\vec{s}'$ to the input, current-state and next-state variables in the Boolean domain, and relate the valuations of variables $\vec{x}$, denoted $[\![\vec{x}]\!]$, and the valuations of variables $\vec{s}$, denoted $[\![\vec{s}]\!]$, to the input alphabet and state set, respectively.

An automaton can be either *deterministic* or *nondeterministic*, depending on whether or not there is a unique next state $q' \in Q$ that satisfies $T(\sigma, q, q')$ for every input $\sigma \in \Sigma$ and current state $q \in Q$. We assume an automaton can be nondeterministic and may have ϵ-transitions. For the sake of convenience, we reserve a symbol not in Σ to represent "ϵ", i.e., the empty input. For a given transition relation T, we write T^{ϵ} to denote the transition relation derived from T with an additional ϵ self-transition added to each state.

A *string* $\vec{\sigma}$ of length j over alphabet Σ is a finite sequence $\sigma_1, \ldots, \sigma_j$ of input symbols, where $\sigma_i \in \Sigma$ for $i = 1, \ldots, j$. The length of sequence $\vec{\sigma}$ is denoted as $\text{Len}(\vec{\sigma})$. A *run* of an automaton is a state-input alternating sequence $q_0, \sigma_1, q_1, \sigma_2, \ldots, \sigma_j, q_j$, which satisfies $T(\sigma_i, q_{i-1}, q_i) = \text{TRUE}$ for $i = 1, \ldots, j$. A string $\sigma_1, \ldots, \sigma_j$ is accepted by an automaton if there exists a run $q_0, \sigma_1, q_1, \sigma_2, \ldots, \sigma_j, q_j$ with $I(q_0) = \text{TRUE}$ and $O(q_j) = \text{TRUE}$. Given an automaton A, the *language* of A, denote as $\mathcal{L}(A)$, is the set of strings accepted by A.

We define a *length-encoded automaton* to be an infinite state automaton with its state set of the form $Q = Q_s \times Q_n$, where Q_s is a finite set and $Q_n = \mathbb{N}^k$ for some non-negative integer k and $\mathbb{N}$ being the set of non-negative integers. Note that when $k = 0$, the 5-tuple degenerates to a finite automaton. Since both Σ and Q_s are finite, their constituent elements are Boolean encoded in the sequel. Hence, the characteristic functions I, T, O can be represented with first order logic formulas over Boolean and integer variables.

The string and integer constraints considered in this work can be formally defined through the following syntactic construction rules.

$$\begin{array}{rrcl}
\textbf{Regular expression} & R & ::= & \epsilon \mid \Sigma \mid R.R \mid R \cup R \mid R^* \\
 & & \mid & \sim R \\
\textbf{Automata} & A & ::= & \text{Reg2Aut}(R) \mid \text{TrkLen}(A) \\
 & & \mid & \text{Int}(A, A) \mid \text{Uni}(A, A) \\
 & & \mid & \text{Cat}(A, A) \mid \text{Del}(A, \xi) \\
 & & \mid & \text{Rep}(A, A_R, A_R) \mid \text{Rev}(A) \\
 & & \mid & \text{Pfx}(A, n) \mid \text{Sfx}(A, n) \\
 & & \mid & \text{Sub}(A, n_1, n_2) \\
 & & \mid & \text{TrkIdx}(A, \vec{\sigma}, n) \\
\textbf{Length constraint} & C & ::= & c_1 n_1 + \ldots + c_m n_m = c \\
 & & \mid & c_1 n_1 + \ldots + c_m n_m > c \\
 & & \mid & \vec{n} = \text{GetIntVars}(A) \\
 & & \mid & C \wedge C \mid C \vee C \mid \neg C \\
\textbf{Language query} & P & ::= & \text{IsEmp}(A, C)
\end{array}$$

In the above expressions, R represents a regular expression, A represents an automaton (possibly with infinite states), A_R represents a finite state automaton accepting some regular language R, and C represents an Boolean combination of linear arithmetic constraints. Moreover, $\xi \in \Sigma$ is an input symbol, constant coefficients are expressed with letter c and c_i, and variable parameters are expressed with letter n and n_i. The regular expressions are constructed with operators: concatenation ".", union "$\cup$", Kleene star "$*$", and language complement "$\sim$". The automata operations include regular expression to automata conversion Reg2Aut, length tracking TrkLen, intersection Int, union Uni, concatenation Cat, deletion Del, replacement Rep, reverse Rev, prefix Pfx, suffix Sfx, substring Sub, and index tracking TrkIdx, which are to be detailed in Section 4. The automata operation TrkLen introduces an integer variable to track the length of each string accepted by A. The automata operation TrkIdx takes an automaton A, a string $\vec{\sigma}$, and an integer parameter n as input, and introduces an integer variable to track the position of the first occurrence of substring $\vec{\sigma}$ after the n^{th} position of each string accepted by A. The length constraint operation GetIntVars retrieves the integer variables introduced by automata operations TrkLen and TrkIdx. The language query operation IsEmpty checks whether the language $\mathcal{L}(A)$ of the given automaton A is empty subject to the given length constraint C.

Given a set of string and integer constraints, we construct a dependency graph whose sink length-encoded automaton corresponds to the solution set of the constraints. In this work, we focus on the set of string and integer constraints whose dependency graph representation exists.

4 AUTOMATA MANIPULATION

We extend the prior method [9] using circuit representation for finite state automata manipulation to length-encoded infinite state automata manipulation. The new representation differs from [9] in that it includes the integer length variables in the inputs of the circuit, which is therefore no longer Boolean. We show how automata operations, including *length tracking*, *intersection*, *union*, *concatenation*, *deletion*, *replacement*, *reversion*, *prefix*, *suffix*, *substring*, and *index tracking*, can be achieved under our representation of infinite state automata.

In the following discussions, we shall assume an automaton A (or A_i) (before length encoding) is represented as a Boolean circuit of its characteristic functions $T(\vec{x}, \vec{s}, \vec{n}, \vec{s}', \vec{n}')$, $I(\vec{s})$, and $O(\vec{s})$ (or $T_i(\vec{x}, \vec{s}_i, \vec{n}_i, \vec{s}_i', \vec{n}_i')$, $I_i(\vec{s}_i)$, and $O_i(\vec{s}_i)$ for $i = 1, 2, 3$). Also without loss of generality we assume that $|\vec{s}_1| = m$ and $|\vec{s}_2| = l$ for automata A_1 and A_2, respectively, with $m \leq l$ in our following discussions unless otherwise said.

4.1 Length Tracking

Given a finite state automaton A with its three characteristic functions $T(\vec{x}, \vec{s}, \vec{s}')$, $I(\vec{s})$, and $O(\vec{s})$, where $\vec{x}$, $\vec{s}$, and $\vec{s}'$ are the input, current-state, and next-state variables, respectively, the length tracked version, denoted A^L in the sequel, of A can be constructed by operation $\textsc{TrkLen}(A)$, which embeds an integer variable n to count the number of transitions in T. So the state set of A^L is augmented to $[\![\vec{s}]\!] \times \mathbb{N}$, and A^L becomes an infinite state automaton due to the unbounded length value of n. The characteristic functions of $A^L = \textsc{TrkLen}(A)$ can be obtained as follows.

$$\begin{aligned} T^L(\vec{x}, \vec{s}, n, \vec{s}', n') &= T(\vec{x}, \vec{s}, \vec{s}') \wedge (((\vec{x} \neq \epsilon) \wedge (n' = n + 1)) \\ &\quad \vee ((\vec{x} = \epsilon) \wedge (n' = n))) \\ I^L(\vec{s}) &= I(\vec{s}) \\ O^L(\vec{s}) &= O(\vec{s}) \end{aligned}$$

Therefore, after taking a non-ϵ transition from a current state $(\vec{s}, n)$ to a next state $(\vec{s}', n')$, the value of n' equals $n + 1$. Otherwise, under an ϵ transition, the value of n' stays the same as n.

Notice that the length tracking does not affect characteristic functions I and O. Rather, we will only initialize the length tracking integer variables on the initial states and impose the length constraints over the length tracking integer variables on the accepting states of the sink automaton of a dependency graph for the final language emptiness checking, to be detailed in Section 4.12.

Note also that during the iterative automata construction according to a dependency graph, an automaton may have multiple length tracking variables $\vec{n}$ inherited from its upstream automata in the dependency graph. If the length of the automaton needs to be considered, then a new tracking variable needs to be introduced in addition to the inherited tracking variables. That is, if an automaton A with transition relation $T(\vec{x}, \vec{s}, \vec{n}, \vec{s}', \vec{n}')$ for $\vec{n} = (n_1, \ldots, n_i)$ and $\vec{n}' = (n_1', \ldots, n_i')$, then the transition relation $T^L(\vec{x}, \vec{s}, (n_1, \ldots, n_{i+1}), \vec{s}', (n_1', \ldots, n_{i+1}'))$ of A^L equals $T(\vec{x}, \vec{s}, \vec{n}, \vec{s}', \vec{n}') \wedge (((\vec{x} \neq \epsilon) \wedge (n_{i+1}' = n_{i+1} + 1)) \vee ((\vec{x} = \epsilon) \wedge (n_{i+1}' = n_{i+1})))$.

4.2 Intersection

Given two automata A_1 and A_2, the three characteristic functions of automaton $A_{\textsc{Int}} = \textsc{Int}(A_1, A_2)$ that accepts language $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ can be constructed as follows.

$$\begin{aligned} T_{\textsc{Int}}(\vec{x}, \vec{s}, \vec{n}, \vec{s}', \vec{n}') &= T_1^\epsilon(\vec{x}, \vec{s}_1, \vec{n}_1, \vec{s}_1', \vec{n}_1') \wedge \\ &\quad T_2^\epsilon(\vec{x}, \vec{s}_2, \vec{n}_2, \vec{s}_2', \vec{n}_2'), \\ I_{\textsc{Int}}(\vec{s}) &= I_1(\vec{s}_1) \wedge I_2(\vec{s}_2), \\ O_{\textsc{Int}}(\vec{s}) &= O_1(\vec{s}_1) \wedge O_2(\vec{s}_2), \end{aligned}$$

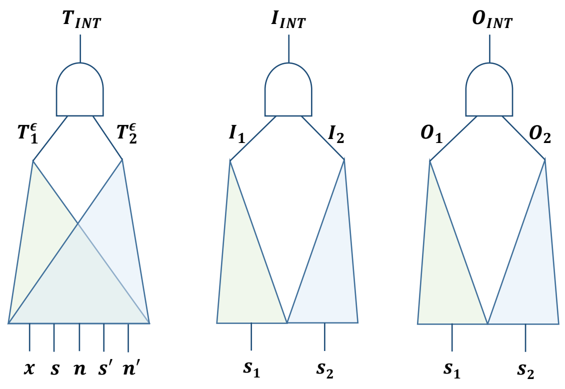

Figure 3: Circuit construction of $\textsc{Int}$ operation.

where $\vec{s} = (\vec{s}_1, \vec{s}_2)$ and $\vec{n} = (\vec{n}_1, \vec{n}_2)$. The corresponding circuit construction is shown in Figure 3.

4.3 Union

Given two automata A_1 and A_2, the three characteristic functions of automaton $A_{\textsc{Uni}} = \textsc{Uni}(A_1, A_2)$ that accepts language $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ can be constructed by disjointly unioning the two with state variables being merged and states being distinguished by a fresh new auxiliary Boolean variable α, similar to the multiplexed machine in [26], as follows.

$$\begin{aligned} T_{\textsc{Uni}}(\vec{x}, \vec{s}, \vec{n}, \vec{s}', \vec{n}') &= (\neg\alpha \wedge \neg\alpha' \wedge \\ &\quad T_1(\vec{x}, \langle\vec{s}_2\rangle_m, \vec{n}_1, \langle\vec{s}_2'\rangle_m, \vec{n}_1')) \vee \\ &\quad (\alpha \wedge \alpha' \wedge T_2(\vec{x}, \vec{s}_2, \vec{n}_2, \vec{s}_2', \vec{n}_2')), \\ I_{\textsc{Uni}}(\vec{s}) &= (\neg\alpha \wedge I_1(\langle\vec{s}_2\rangle_m)) \vee (\alpha \wedge I_2(\vec{s}_2)), \\ O_{\textsc{Uni}}(\vec{s}) &= (\neg\alpha \wedge O_1(\langle\vec{s}_2\rangle_m)) \vee (\alpha \wedge O_2(\vec{s}_2)), \end{aligned}$$

where $\vec{s} = (\vec{s}_2, \alpha)$, $\vec{n} = (\vec{n}_1, \vec{n}_2)$, and the bracket "$\langle\vec{s}_2\rangle_m$" indicates taking a subset of the first m variables of $\vec{s}_2$. Essentially the state variables $\vec{s}_1$ of A_1 are merged into $\vec{s}_2$ so that the first m variables of $\vec{s}_2$ are shared by both A_1 and A_2. (Recall the assumption that $|\vec{s}_1| \leq |\vec{s}_2|$.) Moreover, the α bit of the state variables $\vec{s}$ distinguishes the states of A_1 by $\alpha = 0$ and the states of A_2 by $\alpha = 1$. That is, a state $q \in [\![\vec{s}]\!]$ belongs to A_1 if its variable α valuates to 0, and to A_2 if α valuates to 1. The corresponding circuit construction is shown in Figure 4.

4.4 Concatenation

Given two automata A_1 and A_2, the three characteristic functions of the automaton $A_{\textsc{Cat}} = \textsc{Cat}(A_1, A_2)$ that accepts the concatenated

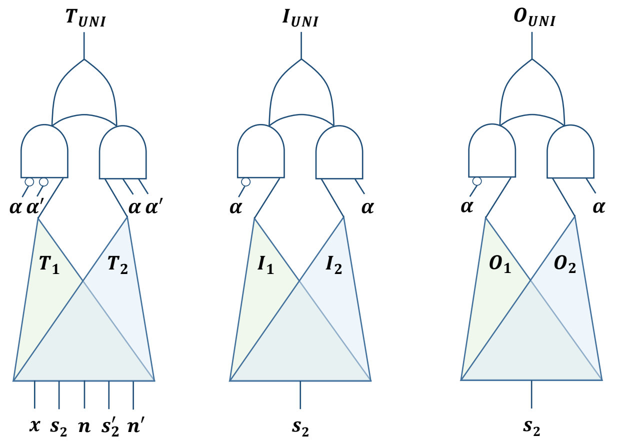

Figure 4: Circuit construction of Uni operation.

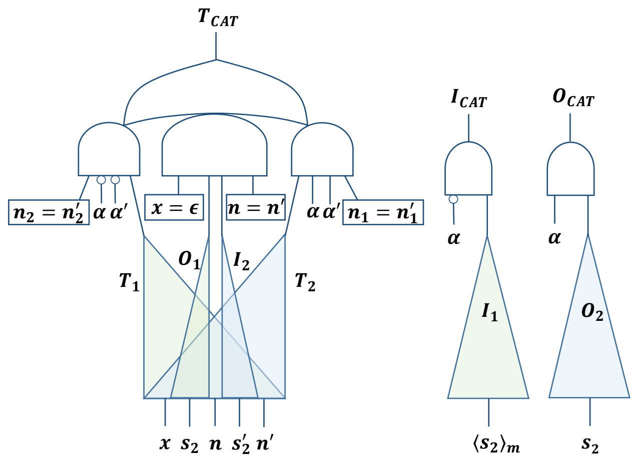

Figure 5: Circuit construction of Cat operation.

language $\mathcal{L}(A) = \mathcal{L}(A_1).\mathcal{L}(A_2)$ can be constructed as follows.

$$\begin{aligned}
T_{\text{CAT}}(\vec{x},\vec{s},\vec{n},\vec{s}\,',\vec{n}') &= (\neg\alpha \wedge \neg\alpha' \wedge \\
&\quad T_1(\vec{x},\langle\vec{s}_2\rangle_m,\vec{n}_1,\langle\vec{s}_2'\rangle_m,\vec{n}_1') \wedge \\
&\quad (\vec{n}_2 = \vec{n}_2')) \vee \\
&\quad (\alpha \wedge \alpha' \wedge T_2(\vec{x},\vec{s}_2,\vec{n}_2,\vec{s}_2',\vec{n}_2') \wedge \\
&\quad (\vec{n}_1 = \vec{n}_1')) \vee \\
&\quad ((\vec{x} = \epsilon) \wedge \neg\alpha \wedge \alpha' \wedge O_1(\langle\vec{s}_2\rangle_m) \wedge \\
&\quad I_2(\vec{s}_2') \wedge (\vec{n} = \vec{n}')), \\
I_{\text{CAT}}(\vec{s}) &= \neg\alpha \wedge I_1(\langle\vec{s}_2\rangle_m), \\
O_{\text{CAT}}(\vec{s}) &= \alpha \wedge O_2(\vec{s}_2),
\end{aligned}$$

where $\vec{s} = (\vec{s}_2, \alpha)$ and $\vec{n} = (\vec{n}_1, \vec{n}_2)$. The corresponding circuit construction is shown in Figure 5.

4.5 Deletion

Given an automaton A and a symbol ξ, the three characteristic functions of the automaton $A_{\text{DEL}} = \text{DEL}(A, \xi)$ that accepts the strings in $\mathcal{L}(A)$ with each occurrence of ξ being eliminated from them can be constructed as follows.

$$\begin{aligned}
T_{\text{DEL}_\xi}(\vec{x},\vec{s},\vec{n},\vec{s}\,',\vec{n}') &= (T(\vec{x},\vec{s},\vec{n},\vec{s}\,',\vec{n}') \vee ((\vec{x} = \epsilon) \wedge \\
&\quad T(\xi,\vec{s},\vec{n},\vec{s}\,',\vec{n}'))) \wedge (\vec{x} \neq \xi), \\
I_{\text{DEL}_\xi}(\vec{s}) &= I(\vec{s}), \\
O_{\text{DEL}_\xi}(\vec{s}) &= O(\vec{s}).
\end{aligned}$$

4.6 Replacement

Given three automata A_1, A_2, and A_3, for A_2 and A_3 being finite state automata without length tracking, with $\epsilon \notin \mathcal{L}(A_2)$, we construct the automaton $A_{\text{REP}} = \text{REP}(A_1, A_2, A_3)$ that accepts the language $\{(\vec{\sigma}_1.\vec{\tau}_1.\vec{\sigma}_2.\vec{\tau}_2\ldots) \in \Sigma^* \mid (\vec{\sigma}_1.\vec{\rho}_1.\vec{\sigma}_2.\vec{\rho}_2\ldots) \in \mathcal{L}(A_1),\ \vec{\sigma}_i \notin (\Sigma^*.\mathcal{L}(A_2).\Sigma^*),\ \vec{\rho}_i \in \mathcal{L}(A_2)$ and $\vec{\tau}_i \in \mathcal{L}(A_3)$ for all $i\}$, that is, replacing $\mathcal{L}(A_2)$ with $\mathcal{L}(A_3)$ in $\mathcal{L}(A_1)$. We extend the construction in [9] based on [17].

We first construct $A_1^{\lhd\rhd}$ to accept strings in $\mathcal{L}(A_1)$ with any substring being flanked by two fresh new symbols $\lhd$ and $\rhd$ as follows.

$$\begin{aligned}
T_1^{\lhd\rhd} &= ((\alpha = \alpha') \wedge (\vec{x} \neq \lhd) \wedge (\vec{x} \neq \rhd) \wedge \\
&\quad T_1(\vec{x},\vec{s}_1,\vec{n}_1,\vec{s}_1',\vec{n}_1')) \vee \\
&\quad ((\vec{s}_1 = \vec{s}_1') \wedge (\vec{n}_1 = \vec{n}_1') \wedge ((\neg\alpha \wedge \alpha' \wedge (\vec{x} = \lhd)) \vee \\
&\quad (\alpha \wedge \neg\alpha' \wedge (\vec{x} = \rhd)))), \\
I_1^{\lhd\rhd} &= \neg\alpha \wedge I_1(\vec{s}_1), \\
O_1^{\lhd\rhd} &= \neg\alpha \wedge O_1(\vec{s}_1).
\end{aligned}$$

Then we construct A_4, a finite automaton without length tracking, to accept the language $\{\vec{\sigma}_1 \lhd \vec{\tau}_1 \rhd \vec{\sigma}_2 \lhd \vec{\tau}_2 \rhd \ldots \vec{\sigma}_\ell \mid \vec{\tau}_i \in \mathcal{L}(A_2), \vec{\sigma}_i \in\sim (\Sigma^*.\mathcal{L}(A_2).\Sigma^*)\}$.

$$\begin{aligned}
T_4 &= (\neg\beta \wedge \neg\beta' \wedge (\vec{x} \neq \lhd) \wedge (\vec{x} \neq \rhd) \wedge T_h(\vec{x},\vec{s}_h,\vec{s}_h')) \vee \\
&\quad (\beta \wedge \beta' \wedge (\vec{x} \neq \lhd) \wedge (\vec{x} \neq \rhd) \wedge \\
&\quad T_2(\vec{x},\langle\vec{s}_h\rangle_m,\langle\vec{s}_h'\rangle_m)) \vee \\
&\quad (\neg\beta \wedge \beta' \wedge (\vec{x} = \lhd) \wedge O_h(\vec{s}_h) \wedge I_2(\langle\vec{s}_h'\rangle_m)) \vee \\
&\quad (\beta \wedge \neg\beta' \wedge (\vec{x} = \rhd) \wedge O_2(\langle\vec{s}_h\rangle_m) \wedge I_h(\vec{s}_h')), \\
I_4 &= \neg\beta \wedge I_h(\vec{s}_h), \\
O_4 &= \neg\beta \wedge O_h(\vec{s}_h),
\end{aligned}$$

where A_h is an automaton that accepts the language $\sim (\Sigma^*.\mathcal{L}(A_2).\Sigma^*)$. Often A_2 is known *a priori*, and the complement of $\Sigma^*.\mathcal{L}(A_2).\Sigma^*$ can be done in a preprocessing step.

After building $A_5 = \textsc{Int}(A_1^{\lhd\rhd}, A_4)$, we can finish the construction of the replacement automaton $A_{\textsc{Rep}}$ as follows.

$$\begin{aligned}
T_{\textsc{Rep}}(\vec{x},\vec{s},\vec{n},\vec{s}',\vec{n}') &= (\neg\alpha \wedge \neg\alpha' \wedge T_5(\vec{x},\vec{s}_5,\vec{n}_5,\vec{s}_5',\vec{n}_5') \wedge \\
&\quad \neg\gamma \wedge \neg\gamma' \wedge I_3(\vec{s}_3) \wedge I_3(\vec{s}_3')) \vee \\
&\quad (\neg\alpha \wedge \neg\alpha' \wedge (\vec{x} = \epsilon) \wedge (\vec{s}_5 = \vec{s}_5') \wedge \\
&\quad (\vec{n}_5 = \vec{n}_5') \wedge \\
&\quad \neg\gamma \wedge \gamma' \wedge I_3(\vec{s}_3) \wedge I_3(\vec{s}_3')) \vee \\
&\quad (\neg\alpha \wedge \neg\alpha' \wedge (\vec{s}_5 = \vec{s}_5') \wedge \\
&\quad (\vec{n}_5 = \vec{n}_5') \wedge \\
&\quad \gamma \wedge \gamma' \wedge T_3(\vec{x},\vec{s}_3,\vec{s}_3')) \vee \\
&\quad (\neg\alpha \wedge \alpha' \wedge (\vec{x} = \epsilon) \wedge \\
&\quad T_5(\lhd,\vec{s}_5,\vec{n}_5,\vec{s}_5',\vec{n}_5') \wedge \\
&\quad \gamma \wedge \neg\gamma' \wedge I_3(\vec{s}_3') \wedge O_3(\vec{s}_3)) \vee \\
&\quad (\alpha \wedge \alpha' \wedge (\vec{x} = \epsilon) \wedge \\
&\quad \exists\vec{y}.[T_5(\vec{y},\vec{s}_5,\vec{n}_5,\vec{s}_5',\vec{n}_5')] \wedge \\
&\quad \neg\gamma \wedge \neg\gamma' \wedge I_3(\vec{s}_3) \wedge I_3(\vec{s}_3')) \vee \\
&\quad (\alpha \wedge \neg\alpha' \wedge (\vec{x} = \epsilon) \wedge \\
&\quad T_5(\rhd,\vec{s}_5,\vec{n}_5,\vec{s}_5',\vec{n}_5') \wedge \\
&\quad \neg\gamma \wedge \neg\gamma' \wedge I_3(\vec{s}_3) \wedge I_3(\vec{s}_3')), \\
I_{\textsc{Rep}}(\vec{s}) &= \neg\gamma \wedge I_5(\vec{s}_5) \wedge I_3(\vec{s}_3), \\
O_{\textsc{Rep}}(\vec{s}) &= \neg\gamma \wedge O_5(\vec{s}_5) \wedge I_3(\vec{s}_3),
\end{aligned}$$

where $\vec{s} = (\vec{s}_5, \vec{s}_3, \gamma)$, and $\vec{n} = \vec{n}_5 = \vec{n}_1$.

4.7 Reverse

Given an automaton A, the three characteristic functions of automaton $A_{\textsc{Rev}} = \textsc{Rev}(A)$ that accepts the set of reversed string of $\vec{\sigma} \in \mathcal{L}(A)$ can be constructed as follows.

$$\begin{aligned}
T_{\textsc{Rev}}(\vec{x},\vec{s},\vec{n},\vec{s}',\vec{n}') &= T(\vec{x},\vec{s}',\vec{n},\vec{s},\vec{n}'), \\
I_{\textsc{Rev}}(\vec{s}) &= O(\vec{s}), \\
O_{\textsc{Rev}}(\vec{s}) &= I(\vec{s}).
\end{aligned}$$

4.8 Prefix

Given an automaton A_1, the three characteristic functions of automaton $A_{\textsc{Pfx}_k} = \textsc{Pfx}(A_1, k)$ that accepts the language $\mathcal{L}(A) = \{\vec{\sigma} | \exists\vec{\rho}.[\vec{\sigma}\vec{\rho} \in \mathcal{L}(A_1)] \wedge len(\vec{\sigma}) = k\}$, can be constructed as follows.

$$\begin{aligned}
T_{\textsc{Pfx}_k}(\vec{x},\vec{s},\vec{n},\vec{s}',\vec{n}') &= (\neg\alpha \wedge \neg\alpha' \wedge T_1^{\epsilon}(\vec{x},\vec{s}_1,\vec{n}_1,\vec{s}_1',\vec{n}_1') \\
&\quad \wedge((\vec{x} \neq \epsilon \wedge (k' = k+1)) \vee \\
&\quad (\vec{x} = \epsilon \wedge (k' = k)))) \vee \\
&\quad (\neg\alpha \wedge \alpha' \wedge (\vec{x} = \epsilon) \wedge (\vec{s}_1 = \vec{s}_1') \wedge \\
&\quad (\vec{n}_1 = \vec{n}_1') \wedge (k' = k)) \vee \\
&\quad (\alpha \wedge \alpha' \wedge (\vec{x} = \epsilon) \wedge \\
&\quad \exists\vec{y}.T_1(\vec{y},\vec{s}_1,\vec{n}_1,\vec{s}_1',\vec{n}_1') \wedge \\
&\quad (k' = k)), \\
I_{\textsc{Pfx}_k}(\vec{s}) &= \neg\alpha \wedge I_1(\vec{s}_1), \\
O_{\textsc{Pfx}_k}(\vec{s}) &= \alpha \wedge O_1(\vec{s}_1),
\end{aligned}$$

where $\vec{s} = (\vec{s}_1, \alpha)$ and $\vec{n} = (\vec{n}_1, k)$. Here k is used to track $len(\vec{\sigma})$. If there is any constraint on k, it should be in the arithmetic constraint C imposed on the language emptiness checking for the sink node of the dependency graph.

4.9 Suffix

Given an automaton A_1, the three characteristic functions of automaton $A_{\textsc{Sfx}_k} = \textsc{Sfx}(A_1, k)$ that accepts the language $\mathcal{L}(A) = \{\vec{\sigma} | \exists\vec{\rho}.(\vec{\rho}\vec{\sigma} \in \mathcal{L}(A_1)) \wedge len(\vec{\rho}) = k\}$ can be constructed as follows.

$$\begin{aligned}
T_{\textsc{Sfx}_k}(\vec{x},\vec{s},\vec{n},\vec{s}',\vec{n}') &= (\neg\alpha \wedge \neg\alpha' \wedge (\vec{x} = \epsilon) \wedge \\
&\quad \exists\vec{y}.[T_1^{\epsilon}(\vec{y},\vec{s}_1,\vec{n}_1,\vec{s}_1',\vec{n}_1') \wedge \\
&\quad ((\vec{y} \neq \epsilon \wedge (k' = k+1)) \vee \\
&\quad (\vec{y} = \epsilon \wedge (k' = k)))]) \vee \\
&\quad (\neg\alpha \wedge \alpha' \wedge (\vec{x} = \epsilon) \wedge (\vec{s}_1 = \vec{s}_1') \wedge \\
&\quad (\vec{n}_1 = \vec{n}_1') \wedge (k' = k)) \vee \\
&\quad (\alpha \wedge \alpha' \wedge T_1(\vec{x},\vec{s}_1,\vec{n}_1,\vec{s}_1',\vec{n}_1') \wedge \\
&\quad (k' = k)), \\
I_{\textsc{Sfx}_k}(\vec{s}) &= \neg\alpha \wedge I_1(\vec{s}_1), \\
O_{\textsc{Sfx}_k}(\vec{s}) &= \alpha \wedge O_1(\vec{s}_1),
\end{aligned}$$

where $\vec{s} = (\vec{s}_1, \alpha)$ and $\vec{n} = (\vec{n}_1, k)$. Similar to the construction of $\textsc{Pfx}$ operation, if there is any constraint on k, it should be in the arithmetic constraint C imposed on the language emptiness checking for the sink node of the dependency graph.

4.10 Substring

Given an automaton A, the automaton $A_{\textsc{Sub}_{i,k}} = \textsc{Sub}(A, i, k)$ for $i \leq k$ that accepts the language containing the set of substrings $\{\vec{\sigma}_2 \in \Sigma^* \mid \vec{\sigma}_1.\vec{\sigma}_2.\vec{\sigma}_3 \in \mathcal{L}(A), |\vec{\sigma}_1| = i-1, \text{ and } |\vec{\sigma}_2| \leq k-i\}$ can be derived by $\textsc{Sfx}(\textsc{Pfx}(A, k-1), i-1)$.

4.11 Index Tracking

Given an automaton A, a non-empty string $\vec{\sigma}$, and an integer parameter n_1, let $A_{\textsc{TrkIdx}_{\vec{\sigma},n_1}} = \textsc{TrkIdx}(A, \vec{\sigma}, n_1)$ be an automaton that accepts the language $\mathcal{L}(A) \cap \Sigma^*.\vec{\sigma}.\Sigma^*$ with an integer being added to track the position of the first occurrence of substring $\vec{\sigma}$ after the k^{th} position of each string accepted by $A_{\textsc{TrkIdx}_{\vec{\sigma},n_1}}$. Automaton $A_{\textsc{TrkIdx}_{\vec{\sigma},n_1}}$ can be constructed as follows.

We first construct $A_1 = \textsc{Cat}(\textsc{TrkLen}(A_{\Sigma^*}), A_{\vec{\sigma}})$, where A_{Σ^*} and $A_{\vec{\sigma}}$ are the automata accepting languages Σ^* and $\vec{\sigma}$, respectively, and let the added integer in $\textsc{TrkLen}(A_{\Sigma^*})$ be n_2. Next, we construct $A_3 = \textsc{Int}(A_1, A_2)$, where A_2 accepts the language $(\sim(\Sigma^*.\vec{\sigma}.\Sigma^*)).\Sigma$. This construction ensures that no other $\vec{\sigma}$ pattern can appear before this $\vec{\sigma}$ pattern. We then build $A_4 = \textsc{Cat}(A_3, A_{\Sigma^*})$, and $A_5 = \textsc{Cat}(\textsc{TrkLen}(A_{\Sigma^*}), A_4)$, where the integer added in $\textsc{TrkLen}(A_{\Sigma^*})$ is n_1. To this end, we know that $n_1 + n_2 + 1$ is the asked index, since n_2 counts the number of symbols between the n_1^{th} position and the head of $\vec{\sigma}$. Finally, we have $A_{\textsc{TrkIdx}_{\vec{\sigma},x}} = \textsc{Int}(A, A_5)$.

4.12 Language Emptiness Checking

Given an automaton A and a length constraint C over variables $\vec{n}$, the language emptiness checking procedure $\textsc{IsEmp}(A, C)$ verifies whether $\mathcal{L}(A)$ is empty subject to the length constraint C. Essentially, the procedure performs reachability analysis over the following infinite state transition system

$$\begin{aligned} T_{\text{IsEmp}}(\vec{x}, \vec{s}, \vec{n}, \vec{s}', \vec{n}') &= T(\vec{x}, \vec{s}, \vec{n}, \vec{s}', \vec{n}'), \\ I_{\text{IsEmp}}(\vec{s}, \vec{n}) &= I(\vec{s}) \wedge (\vec{n} = \vec{0}), \\ O_{\text{IsEmp}}(\vec{s}, \vec{n}) &= O(\vec{s}) \wedge C(\vec{n}). \end{aligned}$$

We exploit state-of-the-art software model checking algorithms [21, 22] for the computation.

5 COUNTEREXAMPLE GENERATION

When the answer to the IsEmp query at the sink node of a dependency graph is negative, it often corresponds to some program error or vulnerability. Tracking the counterexample backward from the sink node to the input nodes of the dependency graph is crucial to validate the system property violation with concrete inputs.

Apart from the length encoding, our automata construction closely follows [9]. The counterexample generation process is almost the same as [9], except that there are additional satisfying assignments to the length variables. Observe that once a length variable is added to an automaton by operation TrkLen or TrkIdx in the dependency graph, it will be carried all the way through to the sink automaton. This transparency makes counterexample generation for the length variables immediate from the satisfying solution to the final length constraint.

Taking automata concatenation for example, let $A_{\text{CAT}} = \text{Cat}(A_1, A_2)$, and (q_1, c_1, u_1, v_1), σ_1, (q_2, c_2, u_2, v_2), σ_2, …, $(q_\ell, c_\ell, u_\ell, v_\ell)$ be the counterexample run of automaton A_{CAT}, where $q_i \in [\![\vec{s}_2]\!]$, $c_i \in [\![\alpha]\!]$, $u_i \in [\![\vec{n}_1]\!]$, $v_i \in [\![\vec{n}_2]\!]$, $\sigma_i \in \Sigma$ and $(\vec{s}_2, \alpha, \vec{n}_1, \vec{n}_2)$ being the state variables of A_{CAT} as construction in Section 4.4. Then the counterexample runs of A_1 and A_2 can be generated by the following rule.

$$\frac{A_1\colon (p^1_1, \sigma_1, \ldots, p^1_i) \qquad A_2\colon (p^2_{i+1}, \sigma_{i+1}, \ldots, p^2_\ell)}{A_{\text{CAT}}\colon (p_1, \sigma_1 \ldots, p_i, \sigma_i, p_{i+1}, \sigma_{i+1}, \ldots, p_\ell)}\ \text{CatCex}$$

where each $p_j = (q_j, 0, u_j, v_j)$ for all $j \leq i$, $p_j = (q_j, 1, u_j, v_j)$ for all $j \geq i+1$, $p^1_j = (q_j, u_j)$ and $p^2_j = (q_j, v_j)$ for all j, and $\sigma_i = \epsilon$. Similarly, a run of an automaton built for some other operation can be decomposed into runs of its parent automata for counterexample backtracking.

6 EXPERIMENTAL EVALUATION

The proposed method was implemented as a tool, called Slent, using IC3ia [21] as the backend symbolic model checker for emptiness checking on string and integer constraints. To evaluate the effectiveness of our tool, Slent is compared against state-of-the-art mixed string and integer constraint solvers, including ABC [8, 24], CVC4 [4], Norn [6], S3P [20], Trau [11], and Z3STR3 [3].[2]

Note that since Sloth [25] does not support length constraint solving in the current released version, it is excluded from the comparison. On the other hand, two sets of test cases were obtained for evaluation, including those modified from the Kaluza [27] and Stranger [28] benchmarks. The experiments were conducted on a Linux machine with Intel 2.8 GHz CPU and 4 GB RAM.

[2] In the experiments, ABC was downloaded on 30-Jan-2018, CVC4 is version 1.5, Norn is version 1.0.1, S3P was downloaded on 09-Aug-2017, Trau was downloaded on 11-Dec-2017, and Z3 is version 4.6.1 for Z3STR3.

Table 1: Solver statistics on Kaluza benchmarks.

solver	time (s)	#SAT	#UNSAT	#TO
ABC	255.76	1013	983	4
CVC4	88.89	1017	983	0
Norn	2025.30	1013	983	4
S3P	137.90	1015	983	2
Slent	1397.82	1013	983	4
Trau	123.85	1017	983	0
Z3STR3	56.46	1017	983	0

The first experiment was conducted on 2000 test cases randomly sampled from the Kaluza benchmarks, which involve string concatenation operations and length constraints. We excluded benchmarks that contain any cyclic dependency among string variables as not all solvers handle the cyclicity equally. The test cases are modified and translated to dependency graphs as input to Slent. We ensured the solvers work on the same set of string and integer constraints. A timeout limit of 200 seconds was imposed on each test case.

Table 1 summarizes the solver performance on the first experiment. Columns "solver," "time(s)," "#SAT," "#UNSAT," and "#TO," indicate the solver name, overall runtime (excluding timeout cases), number of reported SAT instances, number of reported UNSAT instances, and number of timeout instances, respectively. The results suggest that SMT-based solvers, including CVC4, S3P, and Z3STR3, outperform other solvers, including automata-based solvers ABC and Slent, and SMT and automata-mixed solvers Norn and Trau, on the Kaluza benchmarks, which involve only simple string manipulations. Slent timed out on 4 cases due to the expensive reachability analysis on infinite state automata.

The second experiment was conducted on 337 test cases from the Stranger benchmarks that involve at least one *replace-all* operation. The timeout limit is set to 600 seconds for each test case. The benchmarks were obtained from real web applications such as Moodle and Php-Fusion; each case originally corresponds to a dependency graph for a reported vulnerability by Stranger [28], specifying the flow of string operations on user inputs to a sensitive function. Among them, 236 cases involve string-to-string (but no language to language) replacement, and 101 cases involve language-to-language replacement. Because the Stranger benchmarks contain no integer constraints, to evaluate mixed string and integer constraint solving, we added length constraints of the form $\text{Len}(X) + \text{Len}(Y) \,\square\, c$, where predicate $\square \in \{=, <, >\}$, constant $c \in \{10, 20, 30, 40, 50\}$, Y corresponds to the sink automaton, and X corresponds to a node in the dependency graph that undergoes the largest number of *replace-all* operations to reach the sink node. The constraint solving hence involves relations between variables before and after replacement operations as specified in the previous motivating example. Notice that the original web applications of Stranger benchmarks may contain length constraints, but the length conditions are not captured due to the string only capability of Stranger. In this case, adding length constraints back could reflect even more the real constraint complexity. We use the following code snippet (taken from Example 1 of [29]) to justify the practicality of our manually created length constraints.

```
var y = goog.string.escapeString(x);
```

Table 2: Solver statistics on Stranger benchmarks with string-to-string replacement.

solver	time(s)	#SAT	#UNSAT	#TO	#abort
ABC	2282.84	109(31)	111(0)	0	16
S3P	605.79	30(0)	114(3)	22	70
Slent	26692.55	88(0)	141(0)	7	0
Trau	687.49	54(2)	139(22)	5	38

Table 3: Solver statistics on Stranger benchmarks with language-to-language replacement.

solver	time (s)	#SAT	#UNSAT	#TO	#abort
ABC	977.80	46(2)	41(0)	1	13
Slent	4413.25	44(0)	38(0)	19	0

```
catElem.innerHTML = '<button onclick= "createCatList(\''
+ y + '\')">' + x + '</button>';
```

where `catElem.innerHTML` is the concatenation of some constants with two string variables x and y that have a replace relation. To restrict the size of the hyperlink, the length constraint in the form $len(x) + len(y) < c$ (as we created in experiments) needs to be enforced besides string constraints. In practice, it is common to restrict the size of output string variables in web applications.

For the 236 cases involving string-to-string, but no language-to-language, replace-all operations, we compare Slent against other solvers, including ABC, S3P and Trau, capable of solving such constraints. Equivalent constraint formulas in the SMT format are generated from dependency graphs as test cases for these solvers. As noted in Section 2, there is a difference of the replace-all operation definition among the solvers. Both S3P and Trau follow the replace-all-first definition, whereas both ABC and Slent follow the replace-all-any definition. The difference occurs when the replacement candidates overlap. Taking the string "*aaa*" for example, suppose the pattern "*aa*" is to be replaced with "*b*". There are two replacement candidates, underlined in "$\underline{aa}a$" and "$a\underline{aa}$," which overlap in the second position. As a result, S3P and Trau produce "*ba*" as a solution, whereas ABC and Slent produce "*ba*" and "*ab*" as a solution. Nevertheless, in the Stranger benchmarks, all the patterns to be replaced are of the form, such as "`http://`" and "`.gif`," that excludes overlapped replacement candidates. Under such circumstances, all the solvers should have the same interpretation of the replace-all operation. Therefore the experiment is fair for comparison.

Table 2 shows the statistics of solver performance on the cases involving string-to-string replace-all operation and length constraints. The numbers of false alarms of each solver are shown in the parentheses in the entries of Columns "#SAT" and "#UNSAT." The inconsistent answers among the solvers were carefully examined and verified. Solvers ABC, S3P, and Trau aborted on 16, 70, and 38 cases, produced 31, 0, and 2 false positives among the SAT cases, and produced 0, 3, and 22 false negatives among the UNSAT cases, respectively. In contrast, Slent took longer time, but managed to solve more instances than the other solvers and raised no false alarms. The experimental results suggest that Slent is reliable and robust in solving complex mixed string and integer constraints.

For the 101 cases involving language-to-language replace-all operations, we compare Slent and ABC, the only two solvers capable of solving such constraints. Table 3 shows the results on cases involving language-to-language replacement with length constraints. Slent managed to solve 82 out of 101 instances without any false alarms, whereas ABC aborted on 13 cases and produced 2 false positives. Among the 19 timeout cases, Slent was able to solve 10 of them under an extended time limit of 1800 seconds. The experimental results reconfirm the reliability and robustness of Slent in solving complex mixed string and integer constraints.

7 RELATED WORK

String analysis has been studied extensively in recent years due to its importance to detect string manipulation errors in modern software, such as SQL/XSS vulnerabilities in web applications. We review prior work that supports mixed string constraint solving with length constraints involved.

There have been significant efforts on developing inference-rule based solvers, such as Norn [6, 7], CVC4 [4, 5], Z3str3 [3], and S3P [20]. These SMT-based solvers use predefined inference rules and search on the derivation tree for a solution. CVC4 [4, 5] is an SMT solver that supports solving constraints with different theories. It is equipped with derivation rules that combine both string and integers to facilitate the ability of solving mixed string constraints. Norn [6, 7] solves mixed string constraints by first branching search in the derivation tree, and then solving related integer constraints before guessing string solutions for each branch. Z3str3 [3], the successor of Z3str [23] and Z3str2 [30], solves mixed string constraints by leveraging interactions of different theory solvers in Z3 with effective heuristics. One restriction of the above inference-rule based solvers is to reason theories of complex string manipulations such as the operation of replace-all. Among these rule-based solvers, Z3str3 and CVC4 support only the replacement of the first occurrence. Norn does not support replacement. S3P, the successor of S3 [2], is currently the only one that supports all string replacement with recurrence of string operations. It utilizes evaluation on node solution size in the derivation tree to find a minimal solution for a satisfiable formula, which avoids unnecessary derivations on branches. A recent work [31] targets on extending existing rule-based solvers to support more complicated string constraints and avoids the difficulties in modifying the search strategy to cope with new rules. The proposed search-driven method first constructs automata to restrict solution space of each variable by applying automata-based solvers on part of the supported constraints, then searches for complete solutions with ant-colony optimization.

Different from rule based solvers, automata-based string constraint solvers use automata to characterize possible solutions of a string constraint. String operations that can be handled with automata constructions, such as replace-all [17], can be natively handled by automata-based string constraint solvers. However, using finite-state automata, the solvers suffer from approximation/inexactness in a general setting. Since the solution set of a set of constraints may not be a regular language, under/over approximations have to be applied to have the solution set represented with finite automata. In the tool ABC [8, 24], finite-state automata are constructed to characterize the solution set of each string variable and an iterative refine process is applied to strengthen approximation. In the recent update of ABC [24], multi-track automata are used to model relations among string variables. In contrast, we overcome the limitation of automata-based string constraint solving using infinite state automata. Essentially, we encode (unbounded) length within string automata, and leverage symbolic model checking techniques to resolve emptiness checking of these infinite-state automata. Our method can precisely represent a large variety of constraint sets using length-encoded automata. For example, the context sensitive language $\{a^n b^m c^n d^m\}$ can be represented under our framework.

In the work [11], a new tool Trau is introduced, which is based on a counter-example guided abstract refinement (CEGAR) framework, and using flat automata for constraint refinement. The framework alternates between solving under- and over-approximation of constraints, which are refined during the process in order to find a non-spurious solution. Since the framework adopts under and over approximations, a wide variety of constraint set can be supported including transducers and constraints about membership in context-free grammar.

In recent work [25], the tool Sloth solves string constraints by constructing alternating finite automata (AFA) and introduce synchronize parameters to split automata through concatenation. Compared to their work, our length-encoded automata precisely represent the constraint set beyond regular approximations with length information being well kept in the automata structure. For example, a simple instance that is solvable by Slent but not Sloth is $X_2 = \textsc{replace}(X_1, R_1, R_2) \wedge \textsc{Len}(X_2) + \textsc{Len}(X_1) = 10$ for some regular expressions R_1 and R_2.

We note that the supported string logic fragments of [29] and Slent are incomparable. That is, no one strictly contains the other. For example, our method supports the reverse operation, which is clearly not a rational relation and has not be discussed in [25, 29]. Moreover, beyond the SL^e class of [29], we allow a string variable to appear on the left hand sides of string equations multiple times. On the other hand, we limit a string variable to appear on the right hand sides of string equations at most once. Although characterizing the string logic fragment that Slent corresponds to is out of the main focus of this paper, it is an interesting question we planned to address in the future.

There are different ways to count string lengths, e.g., in [29] an explicit counter is used for proving complexity arguments. In this work, we implicitly represent a counter by embedding it into a transition relation, and utilize it for formula construction targeting practical applications.

For the replace-all operation, we adopt the notion in [17]. While [32] and [17] are similar in using markers for constructions, the replace-all definitions are different. Unlike the ones defined in [32] requiring each match to be leftmost, the one in [17] does not have the restriction.

In [33], emptiness checking of intersections of finite automata is considered. However, it does not support string operations.

8 CONCLUSIONS

We present a novel technique based on length-encoded automata for solving string and integer constraints. Our solver Slent is particularly suitable for solving complex string and integer constraints. As Slent precisely maintains the relation among string and length variables, no approximation is required for constraint solving unlike other existing automata-based methods. The experiment shows the unique benefit of the proposed method on solving constraints with replace-all operation over string variables and with complex length relation. As Slent relies on off-the-shelf model checkers, it benefits from model checker advancements. Its performance and practicality may be improved over time. For future work, we plan to extend the proposed method to solve more complex constraints involving cyclic dependencies among string variables.

ACKNOWLEDGMENTS

The authors are grateful to William Eiers for helping the experiments with ABC of the University of California, Santa Barbara. This work was supported in part by the Ministry of Science and Technology of Taiwan under grants 104-2628-E-002-013-MY3, 105-2221-E-002-196-MY3, 105-2923-E-002-016-MY3, 106-2912-E-002-002-MY3, and MOST 107-2221-E-004-002-MY2.

REFERENCES

[1] OWASP top 10 project. https://www.owasp.org/index.php/Top_10-2017_Top_10 (2017)

[2] Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Proc. ACM SIGSAC Conference on Computer and Communications Security. (2014) 1232–1243

[3] Berzish, M., Zheng, Y., Ganesh, V.: Z3str3: A string solver with theory-aware branching. Computing Research Repository **abs/1704.07935** (2017)

[4] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. International Conference on Computer Aided Verification (CAV). (2011) 171–177

[5] Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(t) theory solver for a theory of strings and regular expressions. In: Proc. International Conference on Computer Aided Verification (CAV). (2014) 646–662

[6] Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Proc. International Conference on Computer Aided Verification (CAV). (2014) 150–166

[7] Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Proc. International Conference on Computer Aided Verification (CAV). (2015) 462–469

[8] Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Proc. International Conference on Computer Aided Verification (CAV). (2015) 255–272

[9] Wang, H.E., Tsai, T.L., Lin, C.H., Yu, F., Jiang, J.H.R.: String analysis via automata manipulation with logic circuit representation. In: Proc. International Conference on Computer Aided Verification (CAV). (2016) 241–260

[10] Slent: https://github.com/NTU-ALComLab/SLENT

[11] Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: A framework for efficient analysis of string constraints. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). (2017) 602–617

[12] Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proc. ACM on Programming Language **2**(POPL) (2018) 3:1–3:29

[13] Bultan, T., Yu, F., Alkhalaf, M., Aydin, A.: String Analysis for Software Verification and Security. Springer (2018)

[14] Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In Kowalewski, S., Philippou, A., eds.: Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2009) 307–321

[15] Minamide, Y.: Static approximation of dynamically generated web pages. In: Proc. International World Wide Web Conference (WWW). (2005) 432–441

[16] Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjorner, N.: Symbolic finite state transducers: Algorithms and applications. In: Proceedings of the 39th

Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '12, New York, NY, USA, ACM (2012) 137–150

[17] Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods in System Design **44**(1) (2014) 44–70

[18] Yu, F., Bultan, T., Ibarra, O.H.: Relational string verification using multi-track automata. International Journal of Foundations of Computer Science **22**(8) (2011) 1909–1924

[19] Yu, F., Bultan, T., Ibarra, O.H.: Symbolic string verification: Combining string analysis and size analysis. In: Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2009) 322–336

[20] Trinh, M.T., Chu, D.H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Proc. International Conference on Computer Aided Verification (CAV). (2016) 218–240

[21] Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2014) 46–61

[22] Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Proc. International Conference Theory and Applications of Satisfiability Testing (SAT). (2012) 157–171

[23] Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: Proc. Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). (2013) 114–124

[24] Aydin, A., Bang, L., Eiers, W., Brennan, T., Gavrilov, M., Bultan, T., Yu, F.: Parameterized model counting for string and numeric constraints. Technical report, University of California, Santa Barbara (2016)

[25] Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. In: Proc. ACM Symposium on Programming Language **2**(POPL) (2018) 4:1–4:32

[26] Jiang, J.H.R., Brayton, R.K.: On the verification of sequential equivalence. IEEE Trans. on Comp.-Aid Des. of Int. Cir. and Sys. 22(6), 686–697 (2003)

[27] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: Proc. IEEE Symposium on Security and Privacy (S&P) . (2010) 513–528

[28] Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for PHP. In: Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2010) 154–157

[29] Lin, A.W., Barceló, P.: String solving with word equations and transducers: Towards a logic for analysing mutation XSS. In: Proc. ACM Symposium on Programming Language (POPL). (2016) 123–136

[30] Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In: Proc. International Conference on Computer Aided Verification (CAV). (2015) 235–254

[31] Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: Search-driven string constraint solving for vulnerability detection. In: Proc. International Conference on Software Engineering (ICSE). (2017) 198–208

[32] Fu, X., Li, C.C.: Modeling regular replacement for string constraint solving. In: Proc. NASA Formal Methods (NFM). (2010) 67–76

[33] Gange, G., Navas, J.A., Stuckey, P.J., Søndergaard, H., Schachte, P.: Unbounded model-checking with interpolation for regular language constraints. In: Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2013) 277–291