A Symbolic Algorithm for the Case-Split Rule in String Constraint Solving

Yu-Fang Chen[1], Vojtěch Havlena[2], Ondřej Lengál[2][0000−0002−3038−5875], and Andrea Turrini[3,4][0000−0003−4343−9323]

[1] Academia Sinica, Taiwan
[2] FIT, IT4I Centre of Excellence, Brno University of Technology, Czech Republic
[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[4] Institute of Intelligent Software, Guangzhou, China

Abstract. Case split is a core proof rule in current decision procedures for the theory of string constraints. Its use is the primary cause of the state space explosion in string constraint solving, since it is the only rule that creates branches in the proof tree. Moreover, explicit handling of the case split rule may cause recomputation of the same tasks in multiple branches of the proof tree. In this paper, we propose a symbolic algorithm that significantly reduces such a redundancy. In particular, we encode a string constraint as a regular language and proof rules as rational transducers. This allows to perform similar steps in the proof tree only once, alleviating the state space explosion. In our preliminary experimental results, we validated that our technique (implemented in a Python prototype) works in many practical cases where other state-of-the-art solvers, such as CVC4 or Z3, fail to provide an answer.

1 Introduction

Constraint solving is a technique used as an enabling technology in many areas of formal verification and analysis, such as symbolic execution [21,27], static analysis [23,48], or synthesis [22,38]. For instance, in symbolic execution, feasibility of a path in a program is tested by creating a constraint that encodes the evolution of values of variables on the given path and checking if it is satisfiable. Due to the features used in the analysed programs, checking satisfiability of the constraint can be a complex task. For instance, the solver has to deal with different data types, such as Boolean, Integer, Real, or String. Theories for the first three data types are well known, widely developed, and implemented in tools, while the theory for the String data type has started to be investigated only recently [2, 4, 5, 11, 15, 16, 24, 26, 31–33, 47, 50, 52], despite having been considered already by A. A. Markov in the late 1960s in connection with Hilbert's 10th problem [18, 28, 36].

Most current decision procedures for string constraints involve the so-called *case-split* rule. This rule performs a case split w.r.t. the possible alignment of the variables. The case-split rule is used in most, if not all, (semi-)decision procedures for string constraints, including Makanin's algorithm [34], Nielsen transformation [37] (a.k.a. Levi's lemma [30]), and the procedures implemented in most state-of-the-art solvers such as Z3 [11], CVC4 [31], Z3Str3 [52], Norn [4], and many more. In this paper, we will explain the general idea of our symbolic approach using Nielsen transformation, which is the simplest of the approaches; nonetheless, we believe that the approach is applicable also to other procedures.

Consider the *word equation* $xz = yw$, the primary type of *atomic string constraints* considered in this paper, where x, z, y, and w are *string variables*. When establishing satisfiability of the word equation, Nielsen transformation [37] proceeds by first performing a case split based on the possible alignments of the variables x and y, the first symbol of the left and right-hand sides of the equation, respectively. More precisely, it reduces the satisfiability problem for $xz = yw$ into satisfiability of (at least) one of the following four (non-disjoint) cases (1) y is a prefix of x, (2) x is a prefix of y, (3) x is an empty string, and (4) y is an empty string. For these cases, the Nielsen transformation generates new equations that we describe in the following paragraphs.

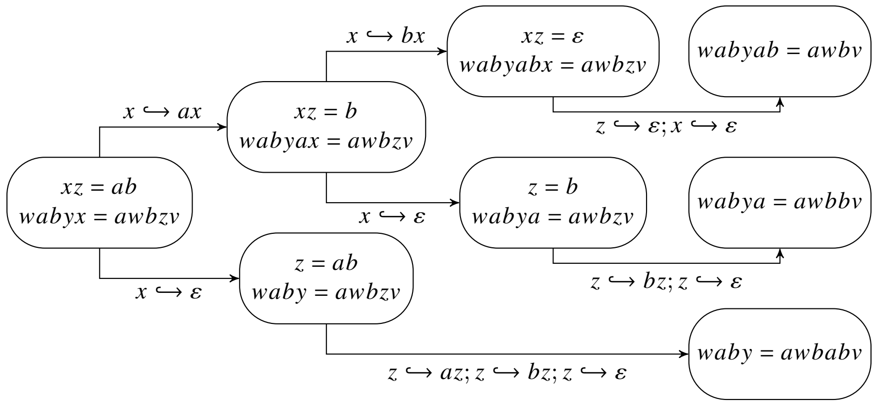
For the case (1), all occurrences of x in $xz = yw$ are substituted to yx', where x' is a fresh string variable (we denote this case as $x \hookrightarrow yx'$), i.e., we obtain the equation $yx'z = yw$, which can be simplified to $x'z = w$. In fact, since the transformation $x \hookrightarrow yx'$ removes all occurrences of the variable x, we can just reuse the variable x and perform the transformation $x \hookrightarrow yx$ instead (and take this into account when constructing a model). The case (2) of the Nielsen transformation is just a symmetric counterpart of case (1) discussed above. For cases (3) and (4), the variables x and y, respectively, are replaced by empty strings. Taking into account all four possible transformations of the equation $xz = yw$, we obtain the following four equations:

$$(1)\ xz = w, \qquad (2)\ z = yw, \qquad (3)\ z = yw, \qquad (4)\ xz = w.$$

If $xz = yw$ has a solution, then at least one of the above equations has a solution, too. Nielsen transformation keeps applying the transformation rules on the obtained equations, building a proof tree and searching for a tautology of the form $\varepsilon = \varepsilon$.

Treating each of the obtained equations separately can cause some redundancy. Let us consider the example in Fig. 1, where we apply Nielsen transformation to solve the string constraint $xz = ab \land wabyx = awbzv$, where v, w, x, y, and z are string variables and a and b are constant symbols. After processing the first word equation $xz = ab$, we obtain a proof tree with three similar leaf nodes $wabyab = awbv$, $wabya = awbbv$, and $waby = awbabv$, which share the prefixes $waby$ and awb on the left and right-hand side of the equations, respectively. If we continue applying Nielsen transformation on the three leaf nodes, we will create three similar subtrees, with almost identical operations. In particular, the nodes near the root of such subtrees, which transform $waby \ldots = awb \ldots$, are going to be essentially the same. The resulting proof trees will therefore start to differ only after processing such a common part. Therefore, handling those equations separately will cause some operations to be performed multiple times. In the case the proof tree of each word equation has n leaves and the string constraint is a conjunction of k word equations, we might need to create n^k similar subtrees.

The case split can be performed more efficiently if we process the common part of the said leaves together using a symbolic encoding. In this paper, we use an encoding of a set of equations as a regular language, which is represented by a *finite automaton*. An example is given in Fig. 2, which shows a finite automaton over a 2-track alphabet,



$xz = ab$ $wabyx = awbzv$
$x \hookrightarrow ax$
$xz = b$ $wabyax = awbzv$
$x \hookrightarrow bx$
$xz = \varepsilon$ $wabyabx = awbzv$
$wabyab = awbv$
$z \hookrightarrow \varepsilon; x \hookrightarrow \varepsilon$
$x \hookrightarrow \varepsilon$
$z = b$ $wabya = awbzv$
$wabya = awbbv$
$z \hookrightarrow bz; z \hookrightarrow \varepsilon$
$x \hookrightarrow \varepsilon$
$z = ab$ $waby = awbzv$
$waby = awbabv$
$z \hookrightarrow az; z \hookrightarrow bz; z \hookrightarrow \varepsilon$

Fig. 1: A partial proof tree of applying Nielsen transformation on $xz = ab \wedge wabyx = awbzv$. The leaves are the outcome of processing the first word equation $xz = ab$. Branches leading to contradictions are omitted.

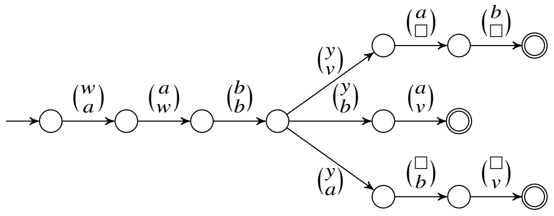


Fig. 2: A finite automaton encoding the three equations $wabyab = awbv$, $wabya = awbbv$, and $waby = awbabv$.

where each of the two tracks represents one side of the equation. For instance, the equation $wabyab = awbv$ is represented by the word $\binom{w}{a}\binom{a}{w}\binom{b}{b}\binom{y}{v}\binom{a}{\square}\binom{b}{\square}$ accepted by the automaton, where the $\square$ symbol is a padding used to make sure that both tracks are of the same length.

Given our regular language-based symbolic encoding, we need a mechanism to perform the Nielsen transformation steps on a set of equations encoded as a regular language. We show that the transformations can be encoded as *rational relations*, represented using *finite transducers*, and the whole satisfiability checking problem can be encoded within the framework of *regular model checking* (RMC). In the past, RMC has already been considered for solving string constraints (cf. [7, 49–51]). In those approaches, the languages of the automata are, however, the "models of the formula", so the approaches can be considered "model-theoretic". In our approach, the automata languages are the derived constraints. Hence the approach is closer to "proof-theoretic". We believe this novel aspect has a great potential for further investigation and can bring new ideas to the field of string solving.

We will provide more details on how this is done in Sections 3 to 5 stepwise. In Section 3, we describe the approach for a simpler case where the input is a *quadratic* *word equation*, i.e., a word equation with at most two occurrences of every variable. In this case, Nielsen transformation is sound and complete. In Section 4, we extend the technique to support *conjunctions* of *non-quadratic* word equations. In Section 5, we extend our approach to support arbitrary Boolean combinations of string constraints.

We implemented our approach in a prototype Python tool called RETRO and evaluated its performance on two benchmark sets: `Kepler`$_{22}$ obtained from [29] and PYEX-HARD obtained by running the PyEx symbolic execution engine on Python programs [42] and collecting examples on which CVC4 or Z3 fail. RETRO solved most of the problems in `Kepler`$_{22}$ (on which CVC4 and Z3 do not perform well). Moreover, it solved over 50 % of the benchmarks in PYEX-HARD that could be solved by neither CVC4 nor Z3.

2 Preliminaries

An *alphabet* Σ is a finite set of *symbols* and a *word* over Σ is a sequence $w = a_1 \dots a_n$ of symbols from Σ, with ε denoting the *empty word*. We use $w_1.w_2$ (and often just $w_1 w_2$) to denote the *concatenation* of words w_1 and w_2. Σ^* is the set of all words over Σ, $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$, and $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. A *language* over Σ is a subset L of Σ^*. Given a word $w = a_1 \dots a_n$, we use $|w|$ to denote the length n of w and $|w|_a$ to denote the number of occurrences of the character $a \in \Sigma$ in w. Further, we use $w[i]$ to denote a_i, the i-th character of w, and $w[i:]$ to denote the word $a_i \dots a_n$. When $i > n$, the value of $w[i]$ and $w[i:]$ is in both cases $\bot$, a special *undefined* value, which is different from all other values and also from itself (i.e., $\bot \neq \bot$). Given an alphabet Σ, we use Σ^k to denote the k-tape alphabet $\underbrace{\Sigma \times \cdots \times \Sigma}_{k}$.

Automata and transducers. A *(finite) k-tape transducer* is a tuple $\mathcal{T} = (Q, \Delta, \Sigma, Q_i, Q_f)$ where Q is a finite set of *states*, $\Delta \subseteq Q \times \Sigma_\varepsilon^k \times Q$ is a set of *transitions*, Σ is an alphabet, $Q_i \subseteq Q$ is a set of *initial states*, and $Q_f \subseteq Q$ is a set of *final states*. A run π of $\mathcal{T}$ over a k-tuple of words $(w_1, \dots, w_k)$ is a sequence of transitions $(q_0, a_1^1, \dots, a_1^k, q_1)$, $(q_1, a_2^1, \dots, a_2^k, q_2)$, $\dots$, $(q_{n-1}, a_n^1, \dots, a_n^k, q_n) \in \Delta$ such that $\forall i : w_i = a_1^i a_2^i \dots a_n^i$ (note that a_m^i can be ε, so w_i and w_j may be of a different length, for $i \neq j$). The run π is *accepting* if $q_0 \in Q_i$ and $q_n \in Q_f$, and a k-tuple $(w_1, \dots, w_k)$ is *accepted* by $\mathcal{T}$ if there exists an accepting run of $\mathcal{T}$ over $(w_1, \dots, w_k)$. The *language* $L(\mathcal{T})$ of $\mathcal{T}$ is defined as the k-ary relation $L(\mathcal{T}) = \{ (w_1, \dots, w_k) \in (\Sigma^*)^k \mid (w_1, \dots, w_k) \text{ is accepted by } \mathcal{T} \}$. We call the class of relations accepted by transducers *rational relations*. $\mathcal{T}$ is *length-preserving* if no transition in Δ contains ε. We call the class of relations accepted by length-preserving transducers *regular relations*. A *finite automaton* (FA) is a 1-tape finite transducer. We call the class of languages accepted by finite automata *regular languages*. Given two k-ary relations R_1, R_2, we define their *concatenation* $R_1.R_2 = \{ (u_1 v_1, \dots, u_k v_k) \mid (u_1, \dots, u_k) \in R_1 \wedge (v_1, \dots, v_k) \in R_2 \}$ and given two binary relations R_1, R_2, we define their *composition* $R_1 \circ R_2 = \{ (x, z) \mid \exists y : (x, y) \in R_2 \wedge (y, z) \in R_1 \}$. Given a k-ary relation R we define $R^0 = \{\varepsilon\}^k$, $R^{i+1} = R.R^i$ for $i \geq 0$. *Iteration* of R is then defined as $R^* = \bigcup_{i \geq 0} R^i$. Given a language L and a binary relation R, we define the *R-image* of L as $R(L) = \{ y \mid \exists x \in L : (x, y) \in R \}$.

$$\frac{\alpha u = \alpha v}{u = v}\ (\text{trim}) \qquad \frac{xu = v}{u[x \mapsto \varepsilon] = v[x \mapsto \varepsilon]}\ (x \hookrightarrow \varepsilon) \qquad \frac{xu = \alpha v}{x(u[x \mapsto \alpha x]) = v[x \mapsto \alpha x]}\ (x \hookrightarrow \alpha x)$$

Fig. 3: Rules of Nielsen transformation, for $x \in \mathbb{X}$, $\alpha \in \Sigma_{\mathbb{X}}$, and $u, v \in \Sigma_{\mathbb{X}}^*$. Symmetric rules are omitted.

Proposition 1 ([10]). *(i) The class of binary rational relations is closed under union, composition, concatenation, and iteration and is not closed under intersection and complement. (ii) For a binary rational relation R and a regular language L, the language $R(L)$ is also effectively regular (i.e., it can be computed). (iii) The class of regular relations is closed under Boolean operations.*

String constraints. Let Σ be an alphabet and $\mathbb{X}$ be a set of *string variables* ranging over Σ^* s.t. $\mathbb{X} \cap \Sigma = \emptyset$. We use $\Sigma_{\mathbb{X}}$ to denote the extended alphabet $\Sigma \cup \mathbb{X}$. An *assignment of* $\mathbb{X}$ is a mapping $I : \mathbb{X} \to \Sigma^*$. A *word term* is a string over the alphabet $\Sigma_{\mathbb{X}}$. We lift an assignment I to word terms by defining $I(\varepsilon) = \varepsilon$, $I(a) = a$, and $I(x.w) = I(x).I(w)$, for $a \in \Sigma$, $x \in \Sigma_{\mathbb{X}}$, and $w \in \Sigma_{\mathbb{X}}^*$. A *word equation* φ_e is of the form $t_1 = t_2$ where t_1 and t_2 are word terms. I is a *model* of φ_e if $I(t_1) = I(t_2)$. We call a word equation an *atomic string constraint*. A *string constraint* is obtained from atomic string constraints using Boolean connectives ($\land, \lor, \neg$), with the semantics defined in the standard manner. A string constraint is *satisfiable* if it has a model. Given a word term $t \in \Sigma_{\mathbb{X}}^*$, a variable $x \in \mathbb{X}$, and a word term $u \in \Sigma_{\mathbb{X}}^*$, we use $t[x \mapsto u]$ to denote the word term obtained from t by replacing all occurrences of x by u, e.g. $(abxcxy)[x \mapsto cy] = abcyccyy$. We call a string constraint *quadratic* if each variable has at most two occurrences, and *cubic* if each variable has at most three occurrences.

2.1 Nielsen Transformation

As already briefly mentioned in the introduction, Nielsen transformation can be used to check satisfiability of a conjunction of word equations. We use the three rules shown in Fig. 3; besides the rules $x \hookrightarrow \alpha x$ and $x \hookrightarrow \varepsilon$ that we have seen in the introduction, there is also the (trim) rule, used to remove a shared prefix from both sides of the equation.

Given a system of word equations, multiple Nielsen transformations might be applicable to it, resulting in different transformed equations on which other Nielsen transformations can be performed, as shown in Fig. 1. Trying all possible transformations generates a tree (or a graph in general) whose nodes contain conjunctions of word equations and whose edges are labelled with the applied transformation. The conjunction of word equations in the root of the tree is satisfiable if and only if at least one of the leaves in the graph is a tautology, i.e., it contains a conjunction $\varepsilon = \varepsilon \land \cdots \land \varepsilon = \varepsilon$.

Lemma 1 (cf. [17,34]). *Nielsen transformation is sound. Moreover, it is complete when the systems of word equations is quadratic.*

Lemma 1 is correct even if we construct the proof tree using the following strategy: every application of $x \hookrightarrow \alpha x$ or $x \hookrightarrow \varepsilon$ is followed by as many applications of the (trim) rule as possible. We use $x \rightarrowtail \alpha x$ to denote the application of one $x \hookrightarrow \alpha x$ rule followed by as many applications of (trim) as possible, and $x \rightarrowtail \varepsilon$ for the application of $x \hookrightarrow \varepsilon$ repeatedly followed by (trim).

2.2 Regular Model Checking

Regular model checking (RMC) [1, 12, 13] is a framework for verifying infinite state systems. In RMC, each *system configuration* is represented as a word over an alphabet Σ. The set of *initial configurations* $\mathcal{I}$ and *destination configurations* $\mathcal{D}$ are captured as regular languages over Σ. The *transition relation* $\mathcal{T}$ is captured as a binary rational relation over Σ^*. A regular model checking *reachability problem* is represented by the triple $(\mathcal{I}, \mathcal{T}, \mathcal{D})$ and asks whether $\mathcal{T}^{rt}(\mathcal{I}) \cap \mathcal{D} \neq \emptyset$, where $\mathcal{T}^{rt}$ represents the reflexive and transitive closure of $\mathcal{T}$. One way how to solve the problem is to start computing the sequence $\mathcal{T}^{(0)}(\mathcal{I}), \mathcal{T}^{(1)}(\mathcal{I}), \mathcal{T}^{(2)}(\mathcal{I}), \ldots$ where $\mathcal{T}^{(0)}(\mathcal{I}) = \mathcal{I}$ and $\mathcal{T}^{(n+1)}(\mathcal{I}) = \mathcal{T}(\mathcal{T}^{(n)}(\mathcal{I}))$. During computation of the sequence, we can check if we find $\mathcal{T}^{(i)}(\mathcal{I})$ that overlaps with $\mathcal{D}$, and if yes, we can deduce that $\mathcal{D}$ is reachable. On the other hand, if we obtain a sequence such that $\bigcup_{0 \leq i < n} \mathcal{T}^i(\mathcal{I}) \supseteq \mathcal{T}^n(\mathcal{I})$, we know that we have explored all possible system configurations without reaching $\mathcal{D}$, so $\mathcal{D}$ is unreachable.

3 Solving Word Equations using RMC

In this section, we describe a symbolic RMC-based framework for solving string constraints. The framework is based on encoding a string constraint into a regular language and encoding steps of Nielsen transformation as a rational relation. Satisfiability of a string constraint is then reduced to a reachability problem of RMC.

3.1 Nielsen Transformation as Word Operations

In the following, we describe how Nielsen transformation of a single word equation can be expressed as operations on words. We view a word equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$ as a pair of word terms $e_{eq} = (\mathsf{t}_\ell, \mathsf{t}_r)$ corresponding to the two sides of the equation; therefore $e_{eq} \in \Sigma_{\mathbb{X}}^* \times \Sigma_{\mathbb{X}}^*$. Without loss of generality we assume that $\mathsf{t}_\ell[1] \neq \mathsf{t}_r[1]$; if this is not the case, we pre-process the equation by applying the (trim) Nielsen transformation (cf. Fig. 3) to trim the common prefix of t_ℓ and t_r.

Example 1. The word equation $eq_1 : xay = yx$ is represented by the pair of word terms $e_1 = (xay, yx)$. □

A rule of Nielsen transformation (cf. Section 2.1) is represented using a (partial) function $\tau \colon (\Sigma_{\mathbb{X}}^* \times \Sigma_{\mathbb{X}}^*) \to (\Sigma_{\mathbb{X}}^* \times \Sigma_{\mathbb{X}}^*)$. Given a pair of word terms $(\mathsf{t}_\ell, \mathsf{t}_r)$ of a word equation eq, the function τ transforms it into a pair of word terms of a word equation eq' that would be obtained by performing the corresponding step of Nielsen transformation on eq. Before we express the rules of Nielsen transformation, we define functions performing the corresponding substitution. For $x \in \mathbb{X}$ and $\alpha \in \Sigma_{\mathbb{X}}$ we define

$$\begin{aligned} \tau_{x \mapsto \alpha x} &= \{\, (\mathsf{t}_\ell, \mathsf{t}_r) \mapsto (\mathsf{t}'_\ell, \mathsf{t}'_r) \mid \mathsf{t}'_\ell = \mathsf{t}_\ell[x \mapsto \alpha x] \land \mathsf{t}'_r = \mathsf{t}_r[x \mapsto \alpha x] \,\} \text{ and} \\ \tau_{x \mapsto \varepsilon} &= \{\, (\mathsf{t}_\ell, \mathsf{t}_r) \mapsto (\mathsf{t}'_\ell, \mathsf{t}'_r) \mid \mathsf{t}'_\ell = \mathsf{t}_\ell[x \mapsto \varepsilon] \land \mathsf{t}'_r = \mathsf{t}_r[x \mapsto \varepsilon] \,\}. \end{aligned} \tag{1}$$

The function $\tau_{x \mapsto \alpha x}$ performs a substitution $x \mapsto \alpha x$ while the function $\tau_{x \mapsto \varepsilon}$ performs a substitution $x \mapsto \varepsilon$.

Example 2. Consider the pair of word terms e_1 from Example 1. The application $\tau_{x \mapsto yx}(e_1)$ would produce the pair $e_2 = (yxay, yyx)$ while the application $\tau_{x \mapsto \varepsilon}(e_1)$ would produce the pair $e_3 = (ay, y)$. □

The functions introduced above do not take into account the first symbols of each side and do not remove a common prefix of the two sides of the equation, which is a necessary operation for Nielsen transformation to terminate. Let us, therefore, define the following function, which trims (the longest) matching prefix of word terms of the two sides of an equation:

$$\tau_{trim} = \{ (\mathsf{t}_\ell, \mathsf{t}_r) \mapsto (\mathsf{t}'_\ell, \mathsf{t}'_r) \mid \exists i (\mathsf{t}_\ell[i] \neq \mathsf{t}_r[i] \land \forall j (j < i \rightarrow \mathsf{t}_\ell[j] = \mathsf{t}_r[j]) \land \mathsf{t}'_\ell = \mathsf{t}_\ell[i{:}] \land \mathsf{t}'_r = \mathsf{t}_r[i{:}]) \}. \tag{2}$$

Example 3. Continuing in our running example, the application $\tau_{trim}(e_2)$ produces the pair $e'_2 = (xay, yx)$ while $\tau_{trim}(e_3)$ produces the pair $e'_3 = (ay, y)$. □

Now we are ready to define functions corresponding to the rules of Nielsen transformation. In particular, the rule $x \rightarrowtail \alpha x$ for $x \in \mathbb{X}$ and $\alpha \in \Sigma_{\mathbb{X}}$ (cf. Section 2.1) can be expressed using the function

$$\tau_{x \rightarrowtail \alpha x} = \tau_{trim} \circ \{ (\mathsf{t}_\ell, \mathsf{t}_r) \mapsto \tau_{x \mapsto \alpha x}(\mathsf{t}_\ell, \mathsf{t}_r) \mid (\mathsf{t}_\ell[1] = \alpha \land \mathsf{t}_r[1] = x) \lor (\mathsf{t}_r[1] = \alpha \land \mathsf{t}_\ell[1] = x) \} \tag{3}$$

while the rule $x \rightarrowtail \varepsilon$ for $x \in \mathbb{X}$ can be expressed as the function

$$\tau_{x \rightarrowtail \varepsilon} = \tau_{trim} \circ \{ (\mathsf{t}_\ell, \mathsf{t}_r) \mapsto \tau_{x \mapsto \varepsilon}(\mathsf{t}_\ell, \mathsf{t}_r) \mid \mathsf{t}_\ell[1] = x \lor \mathsf{t}_r[1] = x \}. \tag{4}$$

If we keep applying the functions defined above on individual pairs of word terms, while searching for the pair $(\varepsilon, \varepsilon)$—which represented the case when a solution to the original equation *eq* exists—, we would obtain the Nielsen transformation graph (cf. Section 2.1). In the following, we show how to perform the steps *symbolically* on a representation of a *whole set of word equations* at once.

3.2 Symbolic Algorithm for Word Equations

In this section, we describe the main idea of our symbolic algorithm for solving word equations. We first focus on the case of a single word equation and in subsequent sections extend the algorithm to a richer class.

Our algorithm is based on applying the transformation rules not on a single equation, but on a whole *set of equations* at once. Given a set of equations, the transformation rules are applied atomically, i.e., a single transformation rule is applied on the whole set of equations without interleaving with other transformation rules. For this, we define the relations $\mathcal{T}_{x \rightarrowtail \alpha x}$ and

$$\mathcal{T}_{x \rightarrowtail \alpha x} = \bigcup_{x \in \mathbb{X}, \alpha \in \Sigma_{\mathbb{X}}} \tau_{x \rightarrowtail \alpha x}$$

$$\mathcal{T}_{x \rightarrowtail \varepsilon} = \bigcup_{x \in \mathbb{X}} \tau_{x \rightarrowtail \varepsilon}$$

Fig. 4: Transformation relations

$\mathcal{T}_{x \rightarrowtail \varepsilon}$ that aggregate the versions of $\tau_{x \rightarrowtail \alpha x}$ and $\tau_{x \rightarrowtail \varepsilon}$ for all possible $x \in \mathbb{X}$ and $\alpha \in \Sigma_{\mathbb{X}}$. The signature of these relations is $(\Sigma_{\mathbb{X}}^* \times \Sigma_{\mathbb{X}}^*) \times (\Sigma_{\mathbb{X}}^* \times \Sigma_{\mathbb{X}}^*)$ and they are defined in Fig. 4. Note the following two properties of the relations: (i) they produce outputs of all possible Nielsen transformation steps applicable with the first symbols on the two sides of the equations and (ii) they include the *trimming* operation.

We compose the introduced relations into a single one, denoted as $\mathcal{T}_{step}$ and defined as $\mathcal{T}_{step} = \mathcal{T}_{x \rightarrowtail \alpha x} \cup \mathcal{T}_{x \rightarrowtail \varepsilon}$. The relation $\mathcal{T}_{step}$ can then be used to compute *all successors* of a set of word terms of equations in one step. For a set of word terms S we can compute the $\mathcal{T}_{step}$-image of S to obtain all successors of pairs of word terms in S. The initial configuration, given a word equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$, is the set $E_{eq} = \{(\mathsf{t}_\ell, \mathsf{t}_r)\}$.

Example 4. Lifting our running example to the introduced notions over sets, we start with the set $E_{eq} = \{e_1 = (xay, yx)\}$. After applying $\mathcal{T}_{step}$ on E_{eq}, we obtain the set $S_1 = \{e_2' = (xay, yx), e_3' = (ay, y), (axy, yx), (a, \varepsilon)\}$. The pairs e_2' and e_3' were described earlier, the pair (axy, yx) is obtained by the transformation $\tau_{y \rightarrowtail xy}$, and the pair (a, ε) is obtained by the transformation $\tau_{y \rightarrowtail \varepsilon}$. If we continue by computing $\mathcal{T}_{step}(S_1)$, we obtain the set $S_2 = S_1 \cup \{(ax, x)\}$, with the pair (ax, x) obtained from (axy, yx) by using the transformation $\tau_{y \rightarrowtail \varepsilon}$. □

Using the symbolic representation, we can formulate the problem of checking satisfiability of a word equation eq as the task of

- either testing whether $(\varepsilon, \varepsilon) \in \mathcal{T}_{step}^{rt}(E_{eq})$; this means that eq is satisfiable, or
- finding a set (called *unsat-invariant*) E_{inv} such that $E_{eq} \subseteq E_{inv}$, $(\varepsilon, \varepsilon) \notin E_{inv}$, and $\mathcal{T}_{step}(E_{inv}) \subseteq E_{inv}$, implying that eq is unsatisfiable.

In the following sections, we show how to encode the problem into the RMC framework.

Example 5. To proceed in our running example, when we apply $\mathcal{T}_{step}$ on S_2, we get $\mathcal{T}_{step}(S_2) \subseteq S_2$. Since $e_1 \in S_2$ and $(\varepsilon, \varepsilon) \notin S_2$, the set S_2 is our unsat-invariant, which means eq_1 is unsatisfiable. □

3.3 Towards Symbolic Encoding

Let us now discuss some possible encodings of the word equations satisfiability problem into RMC. Recall that our task is to find an encoding such that the encoded equation (corresponding to initial configurations in RMC) and satisfiability condition (corresponding to destination configurations) are regular languages and transformation (transition) relation is a rational relation. We start by describing two possible methods of encodings that do not work and then describe the one that we use.

The first idea about how to encode a set of word equations as a regular language is to encode a pair $e_{eq} = (\mathsf{t}_\ell, \mathsf{t}_r)$ as a word $\mathsf{t}_\ell \cdot \circleddash \cdot \mathsf{t}_r$, where $\circleddash \notin \Sigma_{\mathbb{X}}$. One immediately finds out that although the transformations $\tau_{x \rightarrowtail \alpha x}$ and $\tau_{x \rightarrowtail \varepsilon}$ are rational (i.e., expressible using a transducer), the transformation τ_{trim}, which removes the longest matching prefix from both sides, is not (a transducer with an unbounded memory to remember the prefix would be required).

Another attempt of an encoding may be encoding $e_{eq} = (\mathsf{t}_\ell, \mathsf{t}_r)$ as a rational binary relation, represented, e.g., by a (non-length-preserving) 2-tape transducer (with a tape for t_ℓ and a tape for t_r) and use 4-tape transducers to represent the transformations (with two input tapes for t_ℓ, t_r and two output tapes for t'_ℓ, t'_r). The transducers implementing $\tau_{x \rightarrowtail yx}$ and $\tau_{x \rightarrowtail \varepsilon}$ can be constructed easily and so can be the transducer implementing τ_{trim}, so this solution looks appealing. One, however, quickly realizes an issue with computing $\mathcal{T}_{step}(E_{eq})$. In particular, since E_{eq} and $\mathcal{T}_{step}$ are both represented as rational relations, the intersection $(E_{eq} \times \Sigma^*_{\mathbb{X}} \times \Sigma^*_{\mathbb{X}}) \cap \mathcal{T}_{step}$, which needs to be computed first, may not be rational. Why? Consider $E_{eq} = \{ (a^m b^n, c^m) \mid m, n \geq 0 \}$ and $\mathcal{T}_{step} = \{ (a^m b^n, c^n, \varepsilon, \varepsilon) \mid m, n \geq 0 \}$. The intersection $(E_{eq} \times \Sigma^*_{\mathbb{X}} \times \Sigma^*_{\mathbb{X}}) \cap \mathcal{T}_{step} = \{ (a^n b^n, c^n, \varepsilon, \varepsilon) \mid n \geq 0 \}$ is clearly not rational.

3.4 Symbolic Encoding of Quadratic Equations into RMC

We therefore converge on the following method of representing word equations by a regular language. A set of pairs of word terms is represented as a regular language over a 2-track alphabet with padding $\Sigma^2_{\mathbb{X},\square}$, where $\Sigma_{\mathbb{X},\square} = \Sigma_{\mathbb{X}} \cup \{\square\}$, using an FA. For instance, $e_1 = (xay, yx)$ would be represented by the regular language $\binom{x}{y}\binom{a}{x}\binom{y}{\square}\binom{\square}{\square}^*$. Formally, we first define the *equation encoding function* $\mathsf{eqencode}\colon (\Sigma^*_{\mathbb{X}})^2 \to (\Sigma^2_{\mathbb{X},\square})^*$ such that for $\mathsf{t}_\ell = a_1 \ldots a_n$ and $\mathsf{t}_r = b_1 \ldots b_m$ (without loss of generality we assume that $|\mathsf{t}_\ell| \geq |\mathsf{t}_r|$), we have $\mathsf{eqencode}(\mathsf{t}_\ell, \mathsf{t}_r) = \binom{a_1}{b_1}\binom{a_2}{b_2} \ldots \binom{a_m}{b_m}\binom{a_{m+1}}{\square} \ldots \binom{a_n}{\square}$. We lift $\mathsf{eqencode}$ to sets in the usual way and to relations on pairs of word terms τ as $\mathsf{eqencode}(\tau) = \{ (\mathsf{eqencode}(\mathsf{t}_\ell, \mathsf{t}_r), \mathsf{eqencode}(\mathsf{t}'_\ell, \mathsf{t}'_r)) \mid ((\mathsf{t}_\ell, \mathsf{t}_r), (\mathsf{t}'_\ell, \mathsf{t}'_r)) \in \tau \}$.

Let σ be a symbol. We define the *padding* of a k-tuple of words $(w_1, \ldots, w_k)$ with respect to σ as the set $\mathsf{pad}_\sigma(w_1, \ldots, w_k) = \{(w'_1, \ldots, w'_k) \mid w'_i \in w_i.\{\sigma\}^*\}\}$, i.e., it is a set of k-tuples obtained from $(w_1, \ldots, w_k)$ by extending some of the words by an arbitrary number of σ's. We lift pad_σ to a k-ary relation R as $\mathsf{pad}_\sigma(R) = \bigcup_{x \in R} \mathsf{pad}_\sigma(x)$. Finally, we define the function encode, which we use for encoding word equations into regular languages and word operations into rational relations, as $\mathsf{encode} = \mathsf{pad}_{\binom{\square}{\square}} \circ \mathsf{eqencode}$. Properties of encode are given by the following lemmas.

Lemma 2. *If T is a binary regular relation on pairs of word terms, then* $\mathsf{encode}(T)$ *is rational. If L is a regular language, then* $\mathsf{encode}(L)$ *is regular.*

Lemma 3. *Given a word equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$ for $\mathsf{t}_\ell, \mathsf{t}_\ell \in \Sigma^*_{\mathbb{X}}$, the set* $\mathsf{encode}(eq)$ *is regular.*

Observe that because of the padding part, which introduces unbounded number of padding symbols at the end of an encoded relation, even if T is finite, $\mathsf{encode}(T)$ is infinite. Using the presented encoding, when trying to express the $\tau_{x \rightarrowtail \alpha x}$ and $\tau_{x \rightarrowtail \varepsilon}$ transformations, we, however, encounter an issue with the need of an unbounded memory. For instance, for the language $L = \binom{x}{y}^*$, the transducer implementing $\tau_{x \rightarrowtail yx}$ would need to remember how many times it has seen x on the first track of its input (indeed, the image $\{ \mathsf{encode}(u, v) \mid \exists n : u = (yx)^n \land v = y^n \square^n \}$ is no longer regular).

We address this issue in several steps: first, we give a rational relation that correctly represents the transformation rules for cases when the equation eq is quadratic, and extend our algorithm to equations with more occurrences of variables in Section 4. Let us define the following, more general, restriction of $\tau_{x \rightarrowtail \alpha x}$ to equations with at most

```
Input: Encoding $\mathcal{I}$ of a formula $\varphi$ (the initial set), transformers $\mathcal{T}_{x \rightarrowtail \alpha x}$, $\mathcal{T}_{x \rightarrowtail \varepsilon}$, and
       the destination set $\mathcal{D}$
Output: A model of $\varphi$ if $\varphi$ is satisfiable, false otherwise
1  $reach_0 := \emptyset$;
2  $reach_1 := \mathcal{I}$;
3  $processed := reach_0$;
4  $\mathcal{T} := \mathcal{T}_{x \rightarrowtail \alpha x} \cup \mathcal{T}_{x \rightarrowtail \varepsilon}$;
5  $i := 1$;
6  while $reach_i \not\subseteq processed$ do
7      if $\mathcal{D} \cap reach_i \neq \emptyset$ then
8          return $ExtractModel(reach_1, \ldots, reach_i)$;
9      $processed := processed \cup reach_i$;
10     $reach_{i+1} := \mathcal{T}(reach_i)$;
11     $i$++;
12 return false;
```

Algorithm 1: Solving a string constraint φ using RMC

$i \in \mathbb{N}$ occurrences of variable x as $\tau^{\leq i}_{x \rightarrowtail \alpha x} = \tau_{x \rightarrowtail \alpha x} \cap \{ ((\mathsf{t}_\ell, \mathsf{t}_r), (w, w')) \mid w, w' \in \Sigma^*_{\mathbb{X}}, |\mathsf{t}_\ell.\mathsf{t}_r|_x \leq i \}$. We define $\tau^{\leq i}_{x \rightarrowtail \varepsilon}$, $\tau^{\leq i}_{x \mapsto \alpha x}$, and $\tau^{\leq i}_{x \mapsto \varepsilon}$ similarly.

Lemma 4. *Given $i \in \mathbb{N}$, the relations* $\mathsf{encode}(\tau^{\leq i}_{x \rightarrowtail \alpha x})$ *and* $\mathsf{encode}(\tau^{\leq i}_{x \rightarrowtail \varepsilon})$ *are rational.*

In Algorithm 1, we give a high-level algorithm for solving string constraints using RMC. The algorithm is parameterized by the following: a regular language $\mathcal{I}$ encoding a formula φ (the initial set), rational relations $\mathcal{T}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}_{x \rightarrowtail \varepsilon}$, and the destination set $\mathcal{D}$ (also given as a regular language). The algorithm tries to solve the RMC problem $(\mathcal{I}, \mathcal{T}_{x \rightarrowtail \alpha x} \cup \mathcal{T}_{x \rightarrowtail \varepsilon}, \mathcal{D})$ by an iterative unfolding of the transition relation $\mathcal{T}$ computed in Line 4, looking for an element w_i from $\mathcal{D}$. If such an element is found in $reach_i$, we extract a model of the original word equation by starting a backward run from w_i, computing pre-images $w_{i-1}, \ldots, w_1$ over transformers $\mathcal{T}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}_{x \rightarrowtail \varepsilon}$ (restricting them to $reach_j$ for every w_j), while updating values of the variables according to the transformation that was performed.

$$\mathcal{I}^{eq} = \mathsf{encode}(\mathsf{t}_\ell, \mathsf{t}_r)$$
$$\mathcal{D}^{eq} = \left\{ \binom{\square}{\square} \right\}^*$$
$$\mathcal{T}^{eq}_{x \rightarrowtail \alpha x} = \bigcup_{x \in \mathbb{X}, \alpha \in \Sigma_{\mathbb{X}}} \mathsf{encode}(\tau^{\leq 2}_{x \rightarrowtail \alpha x})$$
$$\mathcal{T}^{eq}_{x \rightarrowtail \varepsilon} = \bigcup_{x \in \mathbb{X}} \mathsf{encode}(\tau^{\leq 2}_{x \rightarrowtail \varepsilon})$$

Fig. 5: RMC instantiation for a quadratic equation

Our first instantiation of the algorithm is for checking satisfiability of a single quadratic word equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$. We instantiate the RMC problem with $(\mathcal{I}^{eq}, \mathcal{T}^{eq}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{eq}_{x \rightarrowtail \varepsilon}, \mathcal{D}^{eq})$ defined in Fig. 5.

Lemma 5. *The relations $\mathcal{T}^{eq}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{eq}_{x \rightarrowtail \varepsilon}$ are rational.*

Lemma 6. *If $eq : \mathsf{t}_\ell = \mathsf{t}_r$ is quadratic then Algorithm 1 instantiated with $(\mathcal{I}^{eq}, \mathcal{T}^{eq}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{eq}_{x \rightarrowtail \varepsilon}, \mathcal{D}^{eq})$ is sound and complete.*

4 Solving a System of Word Equations using RMC

In the previous section we described how to solve a single quadratic word equation in the RMC framework. In this section we focus on an extension of this approach to handle a system of word equations $\Phi : \mathsf{t}^1_\ell = \mathsf{t}^1_r \wedge \mathsf{t}^2_\ell = \mathsf{t}^2_r \wedge \ldots \wedge \mathsf{t}^n_\ell = \mathsf{t}^n_r$. In the first step we need to encode the system Φ as a regular language. For this we extend the encode function to a system of word equations by defining

$$\mathsf{encode}(\Phi) = \mathsf{encode}(\mathsf{t}^1_\ell, \mathsf{t}^1_r).\left\{\binom{\#}{\#}\right\}.\ldots.\left\{\binom{\#}{\#}\right\}.\mathsf{encode}(\mathsf{t}^n_\ell, \mathsf{t}^n_r), \tag{5}$$

where $\#$ is a delimiter symbol, $\# \notin \Sigma_{\mathbb{X}}$. From Lemma 3 we know that $\mathsf{encode}(\mathsf{t}^i_\ell, \mathsf{t}^i_r)$ is regular for all $1 \le i \le n$. Moreover, since regular languages are closed under concatenation (Proposition 1), the set $\mathsf{encode}(\Phi)$ is also regular. Because each equation is now separated by a delimiter, we need to extend the destination set to $\left\{\binom{\square}{\square}, \binom{\#}{\#}\right\}^*$.

For the transition relation, we need to extend $\tau^{\le i}_{x \rightarrowtail \alpha x}$ and $\tau^{\le i}_{x \rightarrowtail \varepsilon}$ from Section 3 to support delimiters. An application of a rule $x \rightarrowtail \alpha x$ on a system of equations can be described as follows: the rule $x \rightarrowtail \alpha x$ is applied on the first non-empty equation and the rest of the equations are modified according to the substitution $x \mapsto \alpha x$. The substitution on the other equations is performed regardless of their first symbols. The procedure is analogous for the rule $x \rightarrowtail \varepsilon$. A series of applications of the rules can reduce the number of equations, which then leads to a string in our encoding with a prefix from $\left\{\binom{\square}{\square}, \binom{\#}{\#}\right\}^*$. The relation implementing $x \rightarrowtail \alpha x$ or $x \rightarrowtail \varepsilon$ on an encoded system of equations skips this prefix. Formally, the rule $x \rightarrowtail \alpha x$ for a system of equations where every equation has at most i occurrences of every variable is given by the following relation:

$$T^{eqs,i}_{x \rightarrowtail \alpha x} = T_{skip}.\mathsf{encode}(\tau^{\le i}_{x \rightarrowtail \alpha x}).\left(\left\{\binom{\#}{\#} \mapsto \binom{\#}{\#}\right\}.\mathsf{encode}(\tau_{trim} \circ \tau^{\le i}_{x \mapsto \alpha x})\right)^*, \tag{6}$$

where $T_{skip} = \left\{\binom{\square}{\square} \mapsto \binom{\square}{\square}, \binom{\#}{\#} \mapsto \binom{\#}{\#}\right\}^*$. The relation $T^{eqs,i}_{x \rightarrowtail \varepsilon}$ is defined similarly.

Lemma 7. *The relations $T^{eqs,i}_{x \rightarrowtail \alpha x}$ and $T^{eqs,i}_{x \rightarrowtail \varepsilon}$ are rational.*

4.1 Quadratic Case

When Φ is quadratic, its satisfiability problem can be reduced to an RMC problem $(\mathcal{I}^{q\text{-}eqs}_\Phi, \mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \varepsilon}, \mathcal{D}^{q\text{-}eqs})$ where the items are defined in Fig. 6.

Rationality of $\mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \varepsilon}$ follows directly from Proposition 1. The soundness and completeness of our procedure for a system of quadratic word equations is summarized by the following lemma.

$$\begin{aligned}
\mathcal{I}^{q\text{-}eqs}_\Phi &= \mathsf{encode}(\Phi) \\
\mathcal{D}^{q\text{-}eqs} &= \left\{\binom{\square}{\square}, \binom{\#}{\#}\right\}^* \\
\mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \alpha x} &= \bigcup_{x \in \mathbb{X}, \alpha \in \Sigma_{\mathbb{X}}} T^{eqs,2}_{x \rightarrowtail \alpha x} \\
\mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \varepsilon} &= \bigcup_{x \in \mathbb{X}} T^{eqs,2}_{x \rightarrowtail \varepsilon}
\end{aligned}$$

Fig. 6: RMC instantiation for a system of quadratic equations

Lemma 8. *If Φ is quadratic then Algorithm 1 instantiated with $(\mathcal{I}^{q\text{-}eqs}_\Phi, \mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \varepsilon}, \mathcal{D}^{q\text{-}eqs})$ is sound and complete.*

```
Input: System of word equations $\Phi$
Output: Equisatisfiable cubic system of word equations $\Psi$
$\Psi := \Phi$;
while There is a word variable $x$ that occurs more than three times in $\Psi$ do
    Replace two occurrences of $x$ in $\Phi$ by a fresh string variable $x'$ to obtain a new
      system $\Psi'$;
    $\Psi := \Psi' \land x = x'$;
return $\Psi$;
```

Algorithm 2: Transformation to a cubic system of equations

4.2 General Case

Let us now consider the general case when the system Φ is not quadratic. In this section, we show that this general case is also reducible to an extended version of RMC.

We first apply Algorithm 2 to a general system of string constraints Φ to get an equisatisfiable cubic system of word equations Φ'. Then we can use the transition relations $T^{eqs,3}_{x \rightarrowtail \alpha x}$ and $T^{eqs,3}_{x \rightarrowtail \varepsilon}$ to construct transformations of the encoded system Φ'.

Lemma 9. *Any system of word equations can be transformed by Algorithm 2 to an equisatisfiable cubic system of word equations.*

One more issue we need to solve is to make sure that we work with a cubic system of word equations in every step of our algorithm. It may happen that a transformation of the type $x \rightarrowtail yx$ increases the number of occurrences of the variable y by one, so if there had already been three occurrence of y before the transformation, the result will not be cubic any more.

More specifically, assume a cubic system of word equations $x.\mathtt{t}_\ell = y.\mathtt{t}_r \land \Phi$, where x and y are string variables and $\mathtt{t}_\ell$ and $\mathtt{t}_r$ are word terms. If we apply the transformation $x \rightarrowtail yx$, we will obtain $x(\mathtt{t}_\ell[x \mapsto yx]) = \mathtt{t}_r[x \mapsto yx] \land \Phi[x \mapsto yx]$. Observe that (1) the number of occurrences of y is first *reduced by one* because the first y on the right-hand side of $x.\mathtt{t}_\ell = y.\mathtt{t}_r$ is removed and (2) then the number of occurrences of y can be at most *increased by two* because there exist at most two occurrences of x in $\mathtt{t}_\ell$, $\mathtt{t}_r$, and Φ. Therefore, after the transformation $x \rightarrowtail yx$, a cubic system of word equations might become *(y-)quartic system of word equations* (at most four occurrences of the variable y and at most three occurrences of any other variable).

$$\mathcal{I}^{eqs}_{\Phi} = \mathsf{encode}(\Phi')$$
$$\mathcal{D}^{eqs} = \left\{\binom{\square}{\square}, \binom{\#}{\#}\right\}^*$$
$$\mathcal{T}^{v_i,eqs}_{x \rightarrowtail \alpha x} = T_{C_{v_i}} \circ \bigcup_{x \in \mathbb{X}, \alpha \in \Sigma_{\mathbb{X}}} T^{eqs,3}_{x \rightarrowtail \alpha x}$$
$$\mathcal{T}^{v_i,eqs}_{x \rightarrowtail \varepsilon} = T_{C_{v_i}} \circ \bigcup_{x \in \mathbb{X}} T^{eqs,3}_{x \rightarrowtail \varepsilon}$$

Fig. 7: RMC instantiation for a system of cubic equations

Given a fresh variable v, we use C_v to denote the transformation from a single-quartic system of word equations to a cubic system of equations.

Lemma 10. *The relation T_{C_v} performing the transformation C_v on an encoded single-quartic system of equations is rational.*

To express solving a system of string constraints Φ in the terms of a (modified) RMC, we first convert Φ (using Algorithm 2) to an equisatisfiable cubic system Φ'. The satisfiability of a system of word equations Φ can be reduced to a modified RMC problem $(\mathcal{I}_{\Phi}^{eqs}, \mathcal{T}_{x \rightarrowtail \alpha x}^{v_i, eqs} \cup \mathcal{T}_{x \rightarrowtail \varepsilon}^{v_i, eqs}, \mathcal{D}^{eqs})$ instantiating Algorithm 1 with components given in Fig. 7.

For the modified RMC algorithm, we need to assume $v_i \notin \Sigma_{\mathbb{X}}$. We also need to update Line 4 of Algorithm 1 to $\mathcal{T}^{v_i} := \mathcal{T}_{x \rightarrowtail \alpha x}^{v_i} \cup \mathcal{T}_{x \rightarrowtail \varepsilon}^{v_i}$ and Line 10 to $reach_{i+1} := \mathcal{T}^{v_i}(reach_i)$; $\mathbb{X} := \mathbb{X} \cup \{v_i\}$; to allow using a new variable v_i in every iteration. Rationality of $\mathcal{T}_{x \rightarrowtail \alpha x}^{v_i, eqs}$ and $\mathcal{T}_{x \rightarrowtail \varepsilon}^{v_i, eqs}$ follows directly from Proposition 1.

Lemma 11. *The modified Algorithm 1 instantiated with* $(\mathcal{I}_{\Phi}^{eqs}, \mathcal{T}_{x \rightarrowtail \alpha x}^{v_i, eqs} \cup \mathcal{T}_{x \rightarrowtail \varepsilon}^{v_i, eqs}, \mathcal{D}^{eqs})$ *is sound if* Φ *is cubic.*

Completeness. Since Nielsen transformation does not guarantee termination for the general case, neither does our algorithm. Investigation of possible symbolic encodings of complete algorithms, e.g. Makanin's algorithm [34], is our future work.

5 Handling a Boolean Combination of String Constraints

In this section, we will extend the procedure from handling a *conjunction* of word equations into a procedure that handles their arbitrary Boolean combination. The negation of word equations can be handled in the standard way. For instance, we can use the approach in [4] to convert a negated word equation $\mathsf{t}_\ell \neq \mathsf{t}_r$ to the string constraint

$$\bigvee_{c \in \Sigma} (\mathsf{t}_\ell = \mathsf{t}_r \cdot cx \vee \mathsf{t}_\ell \cdot cx = \mathsf{t}_r) \qquad \vee \bigvee_{c_1, c_2 \in \Sigma, c_1 \neq c_2} (\mathsf{t}_\ell = x_3 c_1 x_1 \wedge \mathsf{t}_r = x_3 c_2 x_2). \qquad (7)$$

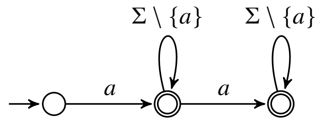
The first part of the constraint says that either t_ℓ is a strict prefix of t_r or the other way around. The second part says that t_ℓ and t_r have a common prefix x_3 and start to differ in the next symbols c_1 and c_2. For word equations connected using $\wedge$ and $\vee$, we apply distributive laws to obtain an equivalent formula in the conjunctive normal form (CNF) whose size is at worst exponential to the size of the original formula.

Let us now focus on how to express solving a string constraint Φ composed of arbitrary Boolean combination of word equations using a (modified) RMC. We start by removing inequalities in Φ using Eq. (7), then we convert the system without inequalities into CNF, and, finally, apply the procedure in Lemma 9 to convert the CNF formula to an equisatisfiable and cubic CNF Φ'. For deciding satisfiability of Φ' in the terms of RMC, both the transition relations and the destination set remain the same as in Section 4.2. The only difference is the initial configuration because the system is not a conjunction of terms any more but rather a general formula in CNF. For this, we extend the definition of encode to a clause $c = (\mathsf{t}_\ell^1 = \mathsf{t}_r^1 \vee \ldots \vee \mathsf{t}_\ell^n = \mathsf{t}_r^n)$ as $\mathsf{encode}(c) = \bigcup_{1 \leq j \leq n} \mathsf{encode}(\mathsf{t}_\ell^j, \mathsf{t}_r^j)$. Then the initial configuration for Φ' is given as

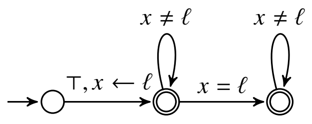
$$\mathcal{I}_{\Phi'}^{sc} = \mathsf{encode}(c_1).\left\{\binom{\#}{\#}\right\}.\ldots.\left\{\binom{\#}{\#}\right\}.\mathsf{encode}(c_m), \qquad (8)$$

where Φ' is of the form $\Phi' : c_1 \wedge \ldots \wedge c_m$ and each clause c_i is of the form $c_i = (\mathsf{t}_\ell^1 = \mathsf{t}_r^1 \vee \ldots \vee \mathsf{t}_\ell^{n_i} = \mathsf{t}_r^{n_i})$. We obtain the following lemma directly from Proposition 1.

Lemma 12. *The initial set* $\mathcal{I}_{\Phi'}^{sc}$ *is regular.*



(a) An FA $\mathcal{A}_a$ for $a \in \Sigma$.



(b) The FRT $\mathcal{R}$; ℓ denotes the input symbol.

Fig. 8: Automata accepting L

The transition relation and the destination set are the same as the ones in the previous section, i.e., $\mathcal{T}^{v_i,sc}_{x \rightarrowtail \alpha x} = \mathcal{T}^{v_i,eqs}_{x \rightarrowtail \alpha x}$, $\mathcal{T}^{v_i,sc}_{x \rightarrowtail \varepsilon} = \mathcal{T}^{v_i,eqs}_{x \rightarrowtail \varepsilon}$, and $\mathcal{D}^{sc} = \mathcal{D}^{eqs}$. The soundness of our procedure for a Boolean combination of word equations is summarized by the following lemma.

Lemma 13. *Given a Boolean combination of word equations* Φ*, Algorithm 1 instantiated with* $(\mathcal{I}^{sc}_{\Phi'}, \mathcal{T}^{v_i,sc}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{v_i,sc}_{x \rightarrowtail \varepsilon}, \mathcal{D}^{sc})$ *is sound.*

6 Implementation

We created a prototype Python tool called Retro, where we implemented the symbolic procedure for solving systems of word equations. Retro implements a modification of the RMC loop from Algorithm 1. In particular, instead of standard transducers defined in Section 2, it uses the so-called *finite-alphabet register transducers* (FRTs), which allow a more concise representation of a rational relation.

Informally, an FRT is a register automaton (in the sense of [25]) where the alphabet is finite. The finiteness of the alphabet implies that the expressive power of FRTs coincides with the class of regular languages, but the advantage of using FRTs is that they allow a more concise representation than FAs.

In particular, transducers (without registers) corresponding to the transformers $\mathcal{T}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}_{x \rightarrowtail \varepsilon}$ contain branching at the beginning for each choice of x and α. Especially in the case of huge alphabets, this yields huge transducers (consider for instance the Unicode alphabet with over 1 million symbols). The use of FRTs yields much smaller automata because the choice of x and α is stored into registers and then processed symbolically. To illustrate the effect of using registers, consider the following example.

Example 6. Consider the language $L = \{ w \in \Sigma^* \mid |w| \geq 1 \land |w|_{w[1]} \leq 2 \}$. Fig. 8a shows an FA $\mathcal{A}_a$ accepting words starting with a and having at most two occurrences of a (it corresponds to a single choice of the first symbol in L). We obtain the FA $\mathcal{A}$ for L as the union of all choices, i.e., $\mathcal{A} = \bigcup_{a \in \Sigma} \mathcal{A}_a$ ($\mathcal{A}$ has $1 + 2|\Sigma|$ states). On the other hand, Fig. 8b shows an FRT $\mathcal{R}$ accepting L with just 3 states (for any alphabet size). □

As another feature, Retro uses deterministic FAs (i.e., FAs having for each state and each symbol at most one successor and having a single initial state) to represent configurations in Algorithm 1. It also uses eager automata minimization, since it has a big impact on the performance, especially on checking the termination condition of the RMC algorithm, which is done by testing language inclusion between the current configuration and all so-far processed configurations.

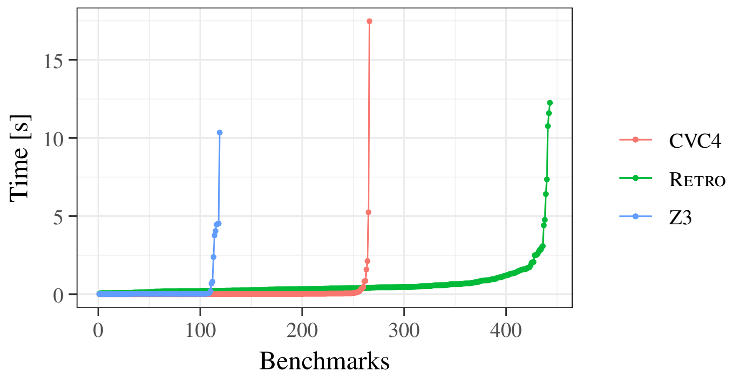


Fig. 9: A cactus plot comparing Retro, CVC4, and Z3 on the `Kepler`$_{22}$ benchmark

7 Experimental Evaluation

We compared the performance of our approach (implemented in Retro) with two current state-of-the-art SMT solvers that support the string theory: Z3 4.8.7 and CVC4 1.7.

The first set of benchmarks is `Kepler`$_{22}$, obtained from [29]. `Kepler`$_{22}$ contains 600 hand-crafted string constraints composed of quadratic word equations with length constraints. In Fig. 9, we give a cactus plot of the results of the solvers on the `Kepler`$_{22}$ benchmark set with the timeout of 20 s. The total numbers of the solved benchmarks within the timeout were: 119 for Z3, 266 for CVC4, and 443 for Retro (out of which 179 could not be solved by CVC4). On this benchmark set, Retro can solve significantly more benchmarks than both Z3 and CVC4.

The other set of benchmarks that we tried is PyEx-Hard. Here we want to see the potential of integrating Retro with DPLL(T)-based string solvers, like Z3 or CVC4, as a specific string theory solver. The input of this component is a conjunction of atomic string formulae (e.g., $xy = zb \land z = ax$) that is a model of the Boolean structure of the top-level formula. The conjunction of atomic string formulae is then, in several layers, processed by various string theory solvers, which either add more conflict clauses or return a model. To evaluate whether Retro is suitable to be used as "one of the layers" of Z3 or CVC4's string solver, we analyzed the PyEx benchmarks [42] and extracted from it 967 difficult instances that neither CVC4 nor Z3 could solve in 10 seconds. From those instances, we obtained 20,020 conjunctions of word equations that Z3's DPLL(T) algorithm sent to its string theory solver when trying to solve them. We call those 20,020 conjunctions of word equations PyEx-Hard. We then evaluated the three solvers on PyEx-Hard with the timeout of 20 s. Out of these, Z3 could not solve 3,232, CVC4 could not solve 188, and Retro could not solve 3,099 instances.

Let us now closely look at the hard instances in the PyEx-Hard benchmark set, in particular on the instances that either CVC4 or Z3 could not solve. These benchmarks cannot be handled by the (several layers of) fast heuristics implemented in CVC4 and Z3,

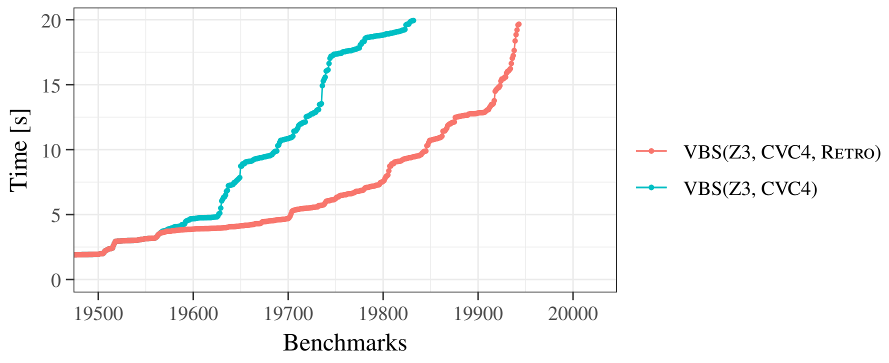


Fig. 10: A cactus plot comparing the Virtual Best Solver with and without RETRO on the PYEX-HARD benchmark. We show ~500 most difficult benchmarks (from 20,020).

which are sufficient to solve many benchmarks without the need to start applying the case-split rule.[5] The set contains the 3,232 benchmarks that Z3 could not solve within 20 seconds. Out of these, CVC4 could not solve 188 benchmarks (CVC4 could solve every constraint that Z3 could solve), and RETRO could not solve 568 benchmarks. When we compared the solvers on the examples that Z3 and CVC4 failed to solve, RETRO could solve 2,664 examples (82.4 %) out of those where Z3 failed and 111 examples (59.04 %) of those where CVC4 failed. In Fig. 10, we give a cactus plot of the *Virtual Best Solver* on the benchmarks with and without RETRO. Given a set of solvers S, we use $VBS(S)$ to denote the solver that would be obtained by taking, for each benchmark, the solver that is the fastest on the given benchmark. The graph shows that our approach can significantly help solvers deal with hard equations.

Discussion. From the obtained results, we see that our approach works well in *hard cases*, where the fast heuristics implemented in state-of-the-art solvers are not sufficient to quickly discharge a formula, in particular when the (un)satisfiability proof is complex. Our approach can exploit the symbolic representation of the proof tree and use it to reduce the redundancy of performing transformations. Note that we can still beat the heavily optimized Z3 and CVC4 written in C++ by a Python tool in those cases. We believe that implementing our symbolic algorithm as a part of a state-of-the-art SMT solver would push the applicability of string solving even further, especially for cases of string constraints with a complex structure, which need to solve multiple DPLL(T) queries in order to establish the (un)satisfiability of a string formula.

[5] For instance, when Z3 receives the word equation $xy = yax$, it infers the length constraint $|x| + |y| = |y| + 1 + |x|$, which implies unsatisfiability of the word equation without the need to start applying the case-split rule at all.

8 Related Work

The study of solving string constraint traces back to 1946, when Quine [41] showed that the first-order theory of word equations is undecidable. Makanin achieved a milestone result in [34], where he showed that the class of quantifier-free word equation is decidable. Since then, several works, e.g., [4,6,8,15,16,19,20,32,35,39,40,43,44], consider the decidability and complexity of different classes of string constraints. Efficient solving of satisfiability of string constraints is a challenging problem. Moreover, decidability of the problem of satisfiability of word equations combined with length constraints of the form $|x| = |y|$ has already been open for over 20 years [14].

The strong practical motivation led to the rise of several string constraint solvers that concentrate on solving practical problem instances. The typical procedure implemented within *DPLL(T)-based* string solvers [3,5,9,16,24,45,46,52,52] is to split the constraints into simpler sub-cases based on how the solutions are aligned, combining with powerful techniques for Boolean reasoning to efficiently explore the resulting exponentially-sized search space. The case-split rule is usually performed explicitly. In contrast, our approach performs case-splits symbolically.

A related topic is about *automata-based* string solvers for analyzing string-manipulating programs. ABC [7] and Stranger [49] soundly over-approximates string constraints using transducers [51]. The main difference of these approaches to ours is that they use transducers to encode possible models (solutions) to the string constraints, while we use automata and transducers to encode the string constraint transformations.

Acknowledgment. We thank the anonymous reviewers for helpful comments on how to improve the paper and Mohamed Faouzi Atig for discussing the topic. This work has been partially supported by the Guangdong Science and Technology Department (grant no. 2018B010107004), by the National Natural Science Foundation of China (grant nos. 61761136011, 61532019, 61836005), the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, the Czech Science Foundation project 20-07487S, the FIT BUT internal project FIT-S-20-6427, and the project of Ministry of Science and Technology, Taiwan (grant nos. 109-2628-E-001-001-MY3 and 106-2221-E-001-009-MY3).

References

1. Abdulla, P.A.: Regular model checking. STTT **14**(2), 109–118 (2012)
2. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In: PLDI. pp. 602–617 (2017)
3. Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: FMCAD. pp. 1–5 (2018)
4. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: CAV. pp. 150–166 (2014)
5. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: CAV. pp. 462–469 (2015)
6. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P.: Chain-free string constraints. In: ATVA. pp. 277–293 (2019)

7. Aydin, A., Eiers, W., Bang, L., Brennan, T., Gavrilov, M., Bultan, T., Yu, F.: Parameterized model counting for string and numeric constraints. In: SIGSOFT. pp. 400–410 (2018)
8. Barceló, P., Figueira, D., Libkin, L.: Graph logics with rational relations. arXiv preprint arXiv:1304.4150 (2013)
9. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: CAV. pp. 171–177 (2011)
10. Berstel, J.: Transductions and context-free languages. Vieweg+Teubner Verlag (1979)
11. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: TACAS. pp. 307–321 (2009)
12. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. STTT **14**(2), 167–191 (2012)
13. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: CAV. pp. 403–418 (2000)
14. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. In: The Collected Works of J. Richard Büchi, pp. 671–683 (1990)
15. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the ReplaceAll function. PACMPL **2**(POPL), 3:1–3:29 (2018)
16. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. PACMPL **3**(POPL), 49 (2019)
17. Diekert, V.: Makanin's Algorithm, pp. 387–442 (2002)
18. Durnev, V.G., Zetkina, O.V.: On equations in free semigroups with certain constraints on their solutions. J. Math. Sci. **158**(5), 671–676 (2009)
19. Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. arXiv preprint arXiv:1605.09442 (2016)
20. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: HVC. pp. 209–226 (2012)
21. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI. pp. 213–223 (2005)
22. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI. pp. 62–73 (2011)
23. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI (2008)
24. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 4 (2018)
25. Kaminski, M., Francez, N.: Finite-memory automata. TCS **134**(2), 329–363 (1994)
26. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. TOSEM **21**(4), 25:1–25:28 (2012)
27. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
28. Kosovskii, N.K.: Properties of the solutions of equations in a free semigroup. J. Math. Sci. **6**(4) (1976)
29. Le, Q.L., He, M.: A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: APLAS. pp. 350–372 (2018)
30. Levi, F.W.: On semigroups. Bulletin of the Calcutta Math. Soc. **36**, 141–146 (1944)
31. Liang, T., Reynolds, A., Tinelli, C., Barrett, C.W., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: CAV. pp. 646–662 (2014)
32. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: POPL. pp. 123–136 (2016)
33. Lin, A.W., Majumdar, R.: Quadratic word equations with length constraints, counter systems, and Presburger arithmetic with divisibility. In: ATVA. pp. 352–369 (2018)

34. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **145**(2), 147–236 (1977)
35. Matiyasevich, Y.: Computation paradigms in light of Hilbert's tenth problem. In: New computational paradigms, pp. 59–85 (2008)
36. Matiyasevich, Y.V.: A connection between systems of word and length equations and Hilbert's tenth problem. Zap. Nauchnykh Semin. POMI **8**, 132–144 (1968)
37. Nielsen, J.: Die isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. Mathematische Annalen **78**(1), 385–397 (1917)
38. Osera, P.M.: Constraint-based type-directed program synthesis. In: TyDe. pp. 64–76 (2019)
39. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. In: FOCS. pp. 495–500 (1999)
40. Plandowski, W.: An efficient algorithm for solving word equations. In: STOC. pp. 467–476 (2006)
41. Quine, W.V.: Concatenation as a basis for arithmetic. JSYML **11**(4), 105–114 (1946)
42. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: CAV. pp. 453–474 (2017)
43. Robson, J.M., Diekert, V.: On quadratic word equations. In: STACS. pp. 217–226 (1999)
44. Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: IWWERT. pp. 85–150 (1990)
45. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: CCS. pp. 1232–1243 (2014)
46. Trinh, M.T., Chu, D.H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: CAV. pp. 218–240 (2016)
47. Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R.: String analysis via automata manipulation with logic circuit representation. In: CAV. pp. 241–260 (2016)
48. Wang, Y., Zhou, M., Jiang, Y., Song, X., Gu, M., Sun, J.: A static analysis tool with optimizations for reachability determination. In: ASE. pp. 925–930 (2017)
49. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for php. In: TACAS. pp. 154–157 (2010)
50. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. FMSD **44**(1), 44–70 (2014)
51. Yu, F., Shueh, C.Y., Lin, C.H., Chen, Y.F., Wang, B.Y., Bultan, T.: Optimal sanitization synthesis for web application vulnerability repair. In: ISSTA. pp. 189–200 (2016)
52. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. FMSD **50**(2-3), 249–288 (2017)