Z3-Noodler 1.3: Shepherding Decision Procedures for Strings with Model Generation

David Chocholatý (✉)[1], Vojtěch Havlena[1], Lukáš Holík[1,2], Jan Hranička[1], Ondřej Lengál[1], and Juraj Síč[1]

[1] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
ichocholaty@fit.vut.cz

[2] The Technical Faculty of IT and Design, Aalborg University, Aalborg, Denmark

Abstract. Z3-Noodler is a fork of the Z3 SMT solver replacing its string theory implementation with a portfolio of decision procedures and a selection mechanism for choosing among them based on the features of the input formula. In this paper, we give an overview of the used decision procedures, including a novel length-based procedure, and their integration into a robust solver with a good overall performance, as witnessed by Z3-Noodler winning the string division of SMT-COMP'24 by a large margin. We also extended the solver with a support for model generation, which is essential for the use of the solver in practice.

1 Introduction

In recent years, research in string solving gained a significant traction, motivated by problems such as finding security vulnerabilities in web applications [45,55,37,9] or analyzing user policies controlling access to cloud resources [31,5,44].

Currently, there is a lot of string solvers utilizing various string solving approaches, usually integrated in general SMT solvers, such as cvc5 [6,32,33,8,34,43,40,42] and Z3 [38,14,49,50,51,48,57,11,13,56,12,10], or more standalone string solvers, such as OSTRICH [36,20,23,21,22]. In order to achieve a high performance on various real-world problems, one specific decision procedure applied to all formulae is usually not sufficient enough. In practice, there often occur formulae with different characteristics that are usually not coverable by a single procedure. One way to overcome this problem lies in using a combination of dedicated procedures tailored for a particular fragment of string constraints. For example, given a formula containing only quadratic word equations, the Nielsen transformation [39] usually performs better than any other approach.

One of the currently best string solvers is Z3-Noodler[3] [25,24,29], a winner of the string division of SMT-COMP'24 [47]. In [24], version 1.0 of Z3-Noodler was introduced, which implements the *stabilization-based* procedure described in previous papers [15,25,29]. Here we introduce version 1.3, implementing a framework for selecting and running decision procedures and two novel decision procedures: (i) one implementing multiple heuristics for handling *pure regular constraints*, based on finite automata library Mata [26] and (ii) a procedure based on transforming *equational block*

[3] https://github.com/VeriFIT/z3-noodler

A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15697, pp. 23–44, 2025.
https://doi.org/10.1007/978-3-031-90653-4_2

string constraints into linear-integer arithmetic (LIA) constraints based on the lengths and alignments of string literals, which are then solved by the internal Z3 LIA solver. Furthermore, we show how Z3-Noodler implements and optimizes the Nielsen transformation, whose preliminary implementation was already present in version 1.0. On top of that, version 1.3 extends Z3-Noodler by *model generation*. In particular, we explain how a model can be constructed not only for each of the aforementioned procedures, but also for the stabilization-based procedure, for which model generation was also missing.

We evaluate the impact of the decision procedures and the model generation, and compare Z3-Noodler with other string solvers on standard SMT-LIB benchmarks. The results show that the implemented decision procedures have large impact on the number of solved instances and the solving time, while the model generation has just a minimal impact. The comparison with other tools exposes that Z3-Noodler outperforms other state-of-the-art tools on the SMT-LIB benchmarks.

2 Preliminaries

Sets, functions, and graphs. We use $\mathbb{N}$ to denote the set of natural numbers (including 0), $\mathbb{Z}$ to denote the set of integers, $\mathbb{B} = \{\top, \bot\}$ to denote the Boolean values, and $\mathbb{B}_3 = \mathbb{B} \cup \{\mathsf{undef}\}$ to denote $\mathbb{B}$ extended with an undefined value. We use $\uplus$ to denote the disjoin union. For a function $f\colon X \to Y$, we use $f \triangleleft \{x \mapsto y\}$ to denote the function $(f \setminus (\{x\} \times Y)) \cup \{x \mapsto y\}$. We use boldface $\boldsymbol{x}$ to denote vectors and $\boldsymbol{x}_i$ to denote the i-th item of $\boldsymbol{x}$. A *(directed) graph* is a pair $G = (V, E)$ where V is a set of nodes and $E \subseteq V \times V$ is a set of (directed) edges.

Strings and languages. We fix a finite alphabet Σ of symbols for the rest of the paper and we use letters from the start of the alphabet $(a, b, c, \dots)$ to denote symbols from Σ. A *string* (or *word*) over Σ is a finite sequence $u = a_1 \cdots a_n$ of symbols from Σ. We say that $|u| = n$ is the *length* of u. The length of the empty string ϵ is $|\epsilon| = 0$. The set of all strings over Σ is denoted by Σ^*. The *concatenation* of strings u and v is denoted $u \cdot v$ or uv for short (ϵ is the neutral element). Moreover, *iteration* of a word w is inductively defined as $w^0 = \epsilon$ and $w^{k+1} = w^k \cdot w$ for $k \in \mathbb{N}$. A *language* is a subset of Σ^*.

Automata. A *(nondeterministic) finite automaton* (NFA) over Σ is a tuple $A = (Q, \delta, I, F)$ where Q is a finite set of states, δ is a set of transitions of the form $q \text{-}\{a\}\!\!\rightarrow r$ with $q, r \in Q$ and $a \in \Sigma$, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A run of A over a word $w = w_1 \dots w_n \in \Sigma^*$ is a sequence of states $q_0 \dots q_n \in Q^{n+1}$ such that for all $1 \leq i \leq n$ it holds that $q_i \text{-}\{w_i\}\!\!\rightarrow q_{i+1} \in \delta$. The run is *accepting* if $q_0 \in I$ and $q_n \in F$, and the language $L(A)$ of A is the set of all words for which A has an accepting run. To intersect the languages of two automata, we construct their *product* $A \cap A' = (Q \times Q', \delta^\times, I \times I', F \times F')$ where $(q, q') \text{-}\{a\}\!\!\rightarrow (r, r') \in \delta^\times$ iff $q \text{-}\{a\}\!\!\rightarrow r \in \delta$ and $q' \text{-}\{a\}\!\!\rightarrow r' \in \delta'$. The *union* of two NFAs, denoted $A \cup A'$, is given as the piece-wise disjoint union of their components. The *complement* of A is given as $A^\complement = (2^Q, \delta^D, \{I\}, F^D)$ where $\delta^D(S, a) = \bigcup_{q \in S} \delta(q, a)$ and $F^D = \{S \subseteq Q \mid S \cap F = \emptyset\}$.

Basic string constraints. In this paper, we consider *basic string constraints* over alphabet Σ, string variables $\mathbb{X}$, and integer variables $\mathbb{I}$. The string variables range over Σ^* and integer variables over $\mathbb{Z}$. In the paper, we use the letters x, y, z to denote variables. The syntax of a string constraint φ is given as follows:

$$\begin{aligned} \varphi &::= t_i \le t_i \mid t_s = t_s \mid t_s \in \mathcal{R} \mid \varphi \wedge \varphi \mid \neg\varphi \\ t_s &::= x_s \mid a \mid t_s \cdot t_s \\ t_i &::= x_i \mid k \mid t_i + t_i \mid \mathtt{len}(t_s) \end{aligned}$$

where t_s is a string term, t_i is linear-integer arithmetic (LIA) term, $x_s \in \mathbb{X}$, $a \in \Sigma$, $x_i \in \mathbb{I}$, $k \in \mathbb{Z}$, and $\mathcal{R}$ is an (extended) regular expression (regex) as defined by the SMT-LIB standard [7] (classical regular expressions extended with operations such as `re.compl`, `re.inter`, ...). The semantics and satisfiability of string constraints are defined in the usual way ($\mathtt{len}(t_s)$ represents the length of the string term t_s). A formula without string terms is called a *LIA formula* and the set of all LIA formulae is denoted as Φ_{LIA}. We usually use the letters u, v, w to denote concatenations of string terms, i.e., words from $(\mathbb{X} \cup \Sigma)^*$. A *string literal* (or just *literal*) is a string term containing only symbols from Σ. We use $\mathtt{Var}(\varphi)$ to denote the set of variables occurring in the string constraint φ.

Other than the basic string constraints, Z3-Noodler can also handle extended string constraints (e.g., `prefixof`, `indexof`, `from_int`, ...) with the semantics specified by the SMT-LIB standard. We do not include these extended constraints in the definition as their solving is not subject of this paper (we briefly discuss their handling in Section 3.2).

3 Shepherding Decision Procedures

Since handling string constraints is complex and there is no universal procedure efficient on every possible constraint, Z3-Noodler implements several decision procedures, each one suitable for a different class of constraints. In this section, we describe a framework for handling the decision procedures and its position within Z3-Noodler's architecture.

3.1 Integration to Z3

Z3-Noodler replaces the string theory plugin of the DPLL(T)-based SMT solver Z3 [38] with the string theory handler that takes care of choosing, running, and processing the results of decision procedures. From a high-level point of view, the main solver, using the internal SAT solver, iteratively provides a conjunction of string atoms corresponding to a SAT solution of the input Boolean skeleton. The core of the string theory handler then works on a conjunction of string (dis)equations, regular constraints, and extended string predicates/functions that could not be axiomatized in preprocessing. The string theory plugin communicates with the main solver using theory lemmas, which steer the generation of further satisfiable assignments of the Boolean skeleton. See [24] for further details of Z3-Noodler's architecture.

3.2 Handling of Extended Constraints

Handling of extended constraints is performed in Z3-Noodler using axiomatization. Extended string functions and predicates (such as `indexof`, `substr`, `contains`, ...) are saturated with axioms consisting of string (dis)equations, and regular, and length constraints. In some special cases, it is not possible (e.g., the general $\neg$`contains` predicate or string-integer conversions). In such cases, we receive these complex constraints as a part of the input conjunction and they are handled by the given decision procedure (if it supports the particular extended constraint). See [24] for more details.

3.3 Handling of Decision Procedures

General interface. In order to maintain extensibility of Z3-Noodler, we propose a plugin architecture for string decision procedures, which all need to implement the following simple interface:

- $\mathsf{isSuitable}(\psi) \to \mathbb{B}$
- $\mathsf{init}(\psi)$
- $\mathsf{preprocess}()$
- $\mathsf{nextSolution}() \to \mathbb{B}_3$
- $\mathsf{getLIA}() \to \Phi_{\mathsf{LIA}} \times \{\mathsf{precise}, \mathsf{underapprox}\}$
- $\mathsf{getModel}(\theta, x) \to \Sigma^*$

where ψ is a string constraint, θ is a LIA model, an assignment of integers to LIA terms (especially the lengths of string terms), and x is a string variable. The meaning of each part of this interface is explained in the rest of this section.

Procedure selection. The handler of decision procedures in Z3-Noodler selects a proper procedure by using the suitability check $\mathsf{isSuitable}(\psi)$. This check takes a string constraint ψ and decides whether a given decision procedure is suitable for it. The first suitable procedure is chosen, ordered from the most specific to the most general ones, starting with the procedure for pure regular constraints (Section 4), followed by the Nielsen transformation (Section 5) and length-based procedure (Section 6). If none of these procedures are suitable, then the stabilization-based procedure (Section 7) is chosen. Therefore, this procedure must always returns $\top$ in $\mathsf{isSuitable}$. Furthermore, some of the decision procedures can be incomplete, i.e., they may lead to an inconclusive state. If this happens, the handler invokes the next suitable procedure.

Procedure execution. A simplified schema for an execution of the selected decision procedure $\mathcal{D}$ on a string constraint ψ is shown in Algorithm 1. It starts with initializing the decision procedure with the string constraint using $\mathsf{init}(\psi)$ followed by the application of preprocessing steps tailored for given decision procedure in $\mathsf{preprocess}$, which may (significantly) simplify the formula and make the solving easier. Note that preprocessing of input formulae is performed at two levels: (i) simplifications during formula rewriting done in the core Z3 solver, and (ii) preprocessing of conjunctions of atomic string constraints done here in $\mathsf{preprocess}$. For the latter case, the preprocessing rules are independent of Z3 's rules, as they are tightly integrated to the string theory decision procedure.

Algorithm 1: Decision procedure handler

Input: String constraint ψ, decision procedure $\mathcal{D}$
Output: Satisfiability of ψ and a LIA formula φ describing relevant solutions of ψ

$\varphi := \bot$; $\mathcal{D}.\mathsf{init}(\psi)$; $\mathcal{D}.\mathsf{preprocess}()$;
while $r := \mathcal{D}.\mathsf{nextSolution}(); r = \top$ **do**
    $(\beta, p) := \mathcal{D}.\mathsf{getLIA}()$;
    if $p = \mathsf{underapprox}$ **then**
        $\mathsf{solver.precision} := \mathsf{underapprox}$;
    if β *is SAT* **then return** (sat, β) ;
    $\varphi := \varphi \lor \beta$;
if $r = \mathsf{undef}$ **then return** $(\mathsf{unknown}, \bot)$;
if $r = \bot$ **then return** $(\mathsf{unsat}, \varphi)$;

The algorithm then iteratively computes solutions using $\mathsf{nextSolution}$, which moves the internal state of the decision procedure to the point before the next LIA check. It is the point where the decision procedure found a possible satisfiable solution, a solution of the non-LIA part of the input formula, and we need to check if it is compatible with the LIA part of the formula. The return value of nextSolution represents whether the generation of possible solution is finished, and if it is finished, whether it was exhaustive.

The value $\top$ means that the generation is not finished, and we can continue with a LIA satisfiability check. This check is accomplished using getLIA, which returns the LIA formula β describing the LIA part of the currently found solution with the indication p of its precision. We check the satisfiability of β using the Z3's LIA solver (which also contains the input LIA formula, and formulae generated during the solver run) and if it is satisfiable, the string theory handler returns sat to the main solver with the theory lemma β. The LIA formula β can sometimes be underapproximating (p = underapprox), which is useful for the stabilization and length-based procedures. If β is underapproximating, Z3-Noodler utilizes an approximation module interacting with decision procedures via the solver variable solver.precision. At the very end, before the final result of Z3-Noodler is given, the approximation module checks if the value of solver.precision is compatible with the final answer. More precisely, if we ever underapproximated any LIA formula from getLIA, then a final unsat becomes unknown instead.

On the other hand, the return values $\bot$ and undef of nextSolution represent that the the generation of possible solutions is finished. For $r = \bot$, the generation was exhaustive, therefore we return unsat with the length formula φ describing the LIA part of all string solutions provided by nextSolution. The string theory handler then adds a new theory lemma of the form $\psi \rightarrow \varphi$, which is used to force the internal SAT solver to find another satisfiable assignment. For r = undef, the generation was not exhaustive, which means the decision procedure $\mathcal{D}$ is not complete, and the string theory handler repeats this step with the next suitable deicision procedure. Note that for the stabilization-based procedure, nextSolution never returns undef, as it is the last procedure.

Model generation. After a decision procedure $\mathcal{D}$ returns (sat, φ), the string theory handler pushes the LIA formula φ as a new theory lemma, which forces Z3 to generate the correct LIA model θ which maps LIA terms into integers (especially integer variables and length and string-integer conversion terms). Following this, Z3 iteratively asks the string theory handler for a model of some string term, which is translated into its corresponding string variable x (based on axiomatization). The handler then calls $\mathcal{D}$.getModel(θ, x) and returns the computed model for x based on the LIA model θ.

4 Efficient Handling of Regular Constraints

In this section, we propose a procedure for handling pure regular constraints, i.e., regular constraints without (dis-)equations or length-constraints. Solving these constraints can be done just by basic automata/regex-based reasoning. Here, the most difficult operation is automata complementation, corresponding to negation in the constraint, since it may cause a state blow-up during determinization of the automaton (this happens especially for automata obtained from regexes containing loop bounds). Therefore, our procedure tries to avoid explicit complementation and handle such constraints in a different way.

Automata construction. For a regular expression $\mathcal{R}$ we use a procedure aut for an inductive construction of (nondeterministic) automaton corresponding to $\mathcal{R}$. The procedure aut uses eager simulation-based reduction [19], which is applied after each inductive step. In the case that some sub-expression requires future complementation (because it is under the `re.compl` regex operator), we eagerly determinize and minimize the automaton for the sub-expression (using Brzozowski-based minimization [18]). We use the automata library Mata [26] to handle finite automata and operations over them.

4.1 General Regular Constraint

We want to decide the satisfiability of the string constraint on the right. To do this, we first construct the product $P = \bigcap_{1 \leq i \leq n} \mathtt{aut}(\mathcal{S}_i)$ of automata on the left side (if $n = 0$ we set P to be the universal automaton having the language Σ^*). We do this iteratively, with regexes $\mathcal{S}_i$ sorted according to an estimated size of the corresponding NFA (with the smallest being the first one). The estimation is based particularly on regex loop bounds as they affect the resulting size the most. It is possible that the product P becomes empty during this construction and then we can immediately decide unsatisfiability, without having to construct the product for larger regexes. For the right side, we would need to compute the product of complemented automata, which we want to avoid as it might be expensive. Instead, we construct the union $U = \bigcup_{1 \leq i \leq m} \mathtt{aut}(\mathcal{R}_i)$ (if $m = 0$ we set U to be empty automaton with the language $\emptyset$). We then want to check if the difference of P and U (i.e., $P \cap U^{\complement}$) is non-empty, which is the same as checking if the inclusion $L(P) \subseteq L(U)$ does *not* hold. This can be accomplished by using antichain-based algorithms [54,4], which perform well on real-world instances.

$$\bigwedge_{1 \leq i \leq n} x \in \mathcal{S}_i \wedge \bigwedge_{1 \leq i \leq m} x \notin \mathcal{R}_i$$

Model generation. A model of x is any word w belonging to the language of $P \cap U^{\complement}$. We construct the product $P \cap U^{\complement}$ (including the complement $U^{\complement}$) lazily, until some word (the returned model) belonging to the language is found. This seems to work reasonably well, as the found models are usually quite short. Note that if there are no negated regular constraints, the model is any word from P, which can be easily found by applying depth-first search algorithm until some final state is reached.

4.2 Single Regular Constraint

For a single regular constraint, either in the positive ($x \in \mathcal{R}$) or the negative ($x \notin \mathcal{R}$) form, we try to postpone the automaton construction and instead gather information about the regex based on its structure. In particular, we propagate triples (e, u, ℓ) where $e \in \mathbb{B}_3$ is a flag denoting whether the regex includes the empty word, $u \in \mathbb{B}_3$ is a flag denoting whether the regex is universal, and $\ell \in \mathbb{N} \cup \{\mathtt{undef}\}$ denotes the minimum length of a word recognized by the regex. The value undef represents that it is not possible to compute the value of the flag or the length from the given information. We then use the flag e for the positive constraint (or u for the negative one) to decide if it is satisfiable, completely avoiding automaton construction. If the flag is undefined, we continue as with the general case.

Example 1. Consider the regex $\mathtt{re.++}(R_1, R_2)$, where the propagated value for R_1 is (e_1, u_1, ℓ_1) and for R_2 it is (e_2, u_2, ℓ_2). The resulting propagated value corresponding to the concatenation is given as $(e_1 \wedge e_2, u, \ell_1 + \ell_2)$ where $u = \bot$ if $\ell_1 + \ell_2 > 0$, otherwise $u = \mathtt{undef}$. Note that undef behaves as an annihilating element in operations (i.e., if undef occurs in the expression, the result is undef).

Model generation. If the regex is in the positive form and does not contain more complex operations (intersection, complement, or difference), then we construct the model directly from the regex. Otherwise, we need to construct the automaton from the regex and get the model similarly as for the general case.

4.3 Implementation

The function $\mathsf{isSuitable}(\psi)$ returns $\top$ if ψ contains only regular constraints. There is no preprocessing and because we do not work with LIA constraints, getLIA always returns $(\top, \mathsf{precise})$. The functions $\mathsf{nextSolution}$ and $\mathsf{getModel}$ implement the procedure and model generation as explained in this section (the LIA model θ is ignored in $\mathsf{getModel}$).

5 Nielsen Transformation

Another decision procedure used in Z3-NOODLER is the Nielsen transformation [39]. We use the Nielsen transformation for satisfiability checking of a conjunction of string equations $\mathcal{E}$ that are not suitable for the stabilization-based procedure. After a brief description of the Nielsen transformation, we propose an approach used for a (partial) handling of length constraints within Nielsen transformation as it is currently used in Z3-NOODLER. We also discuss particular implementation details, including preprocessing details, optimizations, and suitability conditions when Nielsen transformation is applied.

Let e be an equation. By $\mathsf{trim}(e)$ we denote the equation obtained by removing the longest common prefix and suffix from both sides of e. For instance, the result of $\mathsf{trim}(abxzwb = abxnvb)$ is the equation $zw = nv$. We lift trim to a set of equations as usual. In this section, we represent a conjunction of string equations by a set of equations $\mathcal{E}$. We say that a set of equations $\mathcal{E}$ is *quadratic* if each variable has at most two occurrences in $\mathcal{E}$.

Nielsen rules. The transformation uses two meta-rules, which are used to generate a (Nielsen) graph. Nodes of the graph are sets of equations and the directed edges capture the effects of the applied rules. The rules are based on the following observation: if an equation $xu = yv$ is satisfiable, then there is a couple of (not necessarily disjoint) cases that may occur: (i) $x = y$ meaning that $u = v$, or (ii) the variable x or y is ϵ, or (iii) $\mathtt{len}(x) \leq \mathtt{len}(y)$ (the other case is analogous); in that case we have that $y = xy'$ where y' is a fresh variable and we can apply a substitution y/xy' in the equation, followed by the substitution y/y' to avoid generation of isomorphic equations. The Nielsen rules then mimic the cases (ii) and (iii), combined with an implicit handling of the case (i). Formally, the two rules are given as

$$(x \hookrightarrow \alpha x) : \frac{\mathcal{E}' \uplus \{xu = \alpha v\}}{\mathsf{trim}(\mathcal{E}[x/\alpha x])} \mathcal{E} = \mathcal{E}' \uplus \{xu = \alpha v\}, \quad (x \hookrightarrow \epsilon) : \frac{\mathcal{E}' \uplus \{xu = v\}}{\mathsf{trim}(\mathcal{E}[x/\epsilon])} \mathcal{E} = \mathcal{E}' \uplus \{xu = v\}.$$

The rule $(x \hookrightarrow \alpha x)$, where $\alpha \in \Sigma \cup \mathbb{X}$, rewrites all occurrences of x in $\mathcal{E}$ by αx. Since this rule is applied if the left-hand side of an equation starts with x while the right-hand side starts with α, after trimming, the first occurrence of α from the right-hand side is removed. The second rule $x \hookrightarrow \epsilon$ removes all occurrences of x from the system.

Nielsen graph. Nielsen graph $\mathcal{G}_{\mathcal{E}}$ of a set of equations $\mathcal{E}$ is a (possibly infinite) graph induced by Nielsen rules, meaning that vertices are sets of equations and edges are labeled by particular Nielsen rules. The initial vertex is $\mathcal{E}$. The system $\mathcal{E}$ is satisfiable iff the vertex $\{\epsilon = \epsilon\}$ is reachable in $\mathcal{G}_{\mathcal{E}}$. If $\mathcal{E}$ is a quadratic system, $\mathcal{G}_{\mathcal{E}}$ is finite [39].

5.1 Preprocessing

The number of variables and literals of an equation directly affects the size of the corresponding Nielsen graph. To reduce the size, we use the LenSplit rule to split an equation into several ones according to prefixes with the same length.

$$\textsc{LenSplit} : \frac{\mathcal{E} \uplus \{u_1 u_2 \cdots u_k = v_1 v_2 \cdots v_k\}}{\mathcal{E} \cup \{u_i = v_i \mid 1 \leq i \leq k\}} \bigwedge_{i=0}^{k} \mathtt{len}(u_i) = \mathtt{len}(v_i)$$

This preprocessing rule allows not only to generate smaller Nielsen graphs, but if the new equations do not share variables with the other ones, it is possible to divide $\mathcal{E}$ into several independent sets (cf. Section 5.2). In Z3-Noodler, we approximate the length-equality check $\mathtt{len}(u) = \mathtt{len}(v)$ by comparing the number of occurrences of each variable and comparing the total lengths of all literals occurring in u and v.

5.2 Optimizations

As optimizations, we propose two rules pruning the generated state space of the Nielsen graph, focusing on cutting off nodes that would not lead to the satisfiable node $\{\epsilon = \epsilon\}$:

$$\textsc{SymUnsat} : \frac{\mathcal{E} \uplus \{au = bv\}}{\emptyset} a \neq b, \quad \textsc{LenUnsat} : \frac{\mathcal{E} \uplus \{u = v\}}{\emptyset} \mathtt{len}(u) \neq \mathtt{len}(v)$$

The rule SymUnsat skips vertices containing trivially unsatisfiable equations that differ in the first symbol of each side, while LenUnsat is used to avoid vertices containing length-unsatisfiable equations. The check $\mathtt{len}(u) \neq \mathtt{len}(v)$ is approximated in a similar way as in the LenSplit rule.

In order to further reduce the state space, we split $\mathcal{E}$ into several sets that do not share variables and we construct Nielsen graphs for them separately. For instance, we split $\mathcal{E} = \{x = yy, z = wa\}$ to $\mathcal{E}_1 = \{x = yy\}$, $\mathcal{E}_2 = \{z = wa\}$ and then check the satisfiability of $\mathcal{E}_1$ and $\mathcal{E}_2$ separately.

Example 2. Let $\{xaby = yxxbca\}$ be a vertex of a Nielsen graph. Since $\mathtt{len}(xaby) = \mathtt{len}(x)+\mathtt{len}(y)+2 < 2\cdot\mathtt{len}(x)+\mathtt{len}(y)+3 = \mathtt{len}(yxxbca)$, we can skip the generation of successors of this vertex as it is length-unsatisfiable.

5.3 Length Constraints

If we have a length formula ψ, we need to check if ψ is satisfiable for a string solution generated by a constructed satisfiable Nielsen graph $\mathcal{G}_{\mathcal{E}}$. In order to fit into Z3-Noodler's decision procedure handling, we need to infer a LIA formula describing possible lengths of string solutions induced by $\mathcal{G}_{\mathcal{E}}$. In Z3-Noodler, we utilize the approach of [35], converting the Nielsen graph into a counter abstraction. We then saturate the counter system with self-loops and enumerate particular flat paths that can be directly converted to a LIA formula. We consider the counter system to be an NFA with counter updates on edges modifying the counter values during a run (we assume no guards).

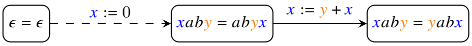


Fig. 1: A run of the counter system corresponding to Nielsen rules ($x \hookrightarrow yx$) and ($x \hookrightarrow \epsilon$). The counter values obtained during this run are $x = 0$ and $y = 0$. The sequence of corresponding Nielsen rules, however, describes all string solutions where $x = y$.

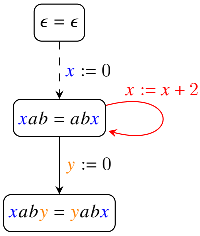
Counter system construction. For the Nielsen graph $\mathcal{G}_{\mathcal{E}}$ we construct a counter system C s.t. states of C are vertices of $\mathcal{G}_{\mathcal{E}}$, transitions are *reverted* edges of $\mathcal{G}_{\mathcal{E}}$ where the update transition action is obtained as (i) from $x \hookrightarrow ax$ where $a \in \Sigma$ we get $x := x+1$, (ii) from $x \hookrightarrow yx$ we get $x := x + y$, and (iii) from $x \hookrightarrow \epsilon$ we get $x := 0$. Note that contrary to [35] where the counter system has the same direction of edges with the subtracting semantics, we use C with reversed edges and additive semantics for a better fit to usual counter system definition. The initial state of C is $\{\epsilon = \epsilon\}$ and the accepting state is $\mathcal{E}$. Each run of C corresponds to a satisfiable length assignment. Counters of a run of C, however, do not represent all string solutions, as shown in Fig. 1.

Generating a LIA formula. In order to check if there exists an accepting run in C that satisfies the length formula ψ, we need to construct a LIA formula ϕ_C describing all possible valuations of each counter on all accepting paths of C. In general, such a formula cannot be constructed [35], therefore, we use an under-approximation enumerating *extended runs* of C. An extended run is a sequence of states occurring on a run of C empowered with the possibility of *simple self-loops* on states. Simple self-loops allow only update actions of the form $x := x + \ell$ where $\ell \in \mathbb{N}$ and x is a counter. For the extended runs, we are able to construct a LIA formula precisely describing the counter values. In particular, for the LIA formula, we create a vector of fresh variables $\boldsymbol{x}$ expressing counter values after each step of the extended run. Then, we connect the variables using conjunction of formulae describing counter actions on each transition: (i) for a non-self-loop transition with the counter update $x := x + y$, the corresponding formula looks like $\phi(\boldsymbol{x}', \boldsymbol{x}) \Leftrightarrow \boldsymbol{x}'_i = \boldsymbol{x}_i + \boldsymbol{x}_j \wedge id_{\{i,j\}}(\boldsymbol{x}', \boldsymbol{x})$ where $\boldsymbol{x}_i = x$ and $\boldsymbol{x}_j = y$, and $id_I(\boldsymbol{x}', \boldsymbol{x}) \Leftrightarrow \bigwedge_{i \notin I} \boldsymbol{x}_i = \boldsymbol{x}'_i$ (other updates are given analogously), and (ii) for a simple self-loop transition with the counter update $x := x + \ell$, the formula is given as $\phi(\boldsymbol{x}', \boldsymbol{x}) \Leftrightarrow id_{\{i\}}(\boldsymbol{x}', \boldsymbol{x}) \wedge \boldsymbol{x}'_i = \boldsymbol{x}_i + k \cdot \ell$ where $\boldsymbol{x}_i = x$ and k is a fresh LIA variable counting the number of times the self-loop was taken (we do not use existential quantification as the value of k is important for model generation).

Enumeration of extended runs. Since there might be infinitely many extended runs of C, we use a heuristic enumeration algorithm preferring runs having self-loops as it means that they describe more behaviour. We mark states containing simple self-loops and enumerate extended runs that contain these states. For each such run, we construct the corresponding LIA formula and check if it is satisfiable with the length constraint ψ.

Self-loop saturation. Since the extended runs with self-loops yield weaker LIA formulae, we apply saturation of self-loops on the original counter system in order to generate new simple self-loops. In particular, we select cycles starting and ending in a state q, having counter updates of the same variable x with the counter updates on the cycle of the form $x := x + \ell_1, \ldots, x := x + \ell_n$ where $\ell_i \in \mathbb{N}$. For each such a cycle, we introduce a new self-loop of q labeled by the counter update $x := x + \sum_{i=1}^{n} \ell_i$.

Example 3. Consider the string constraint $xaby = yabx \wedge \mathtt{len}(x) \geq 50$. Example of an enumerated extended run in the counter system is shown in the right. The red self-loop was added during the self-loop saturation. The LIA formula corresponding to this extended run obtained by the procedure above is then given as follows:



$$\varphi(x, y) \Leftrightarrow x_0 = 0 \wedge y_0 = 0 \wedge x_1 = 0 \wedge y_1 = y_0 \wedge x_2 = x_1 + 2k \wedge \\ y_2 = y_1 \wedge y_3 = 0 \wedge x_3 = x_2 \wedge x = x_3 \wedge y = y_3.$$

Since the formula $\varphi(\mathtt{len}(x), \mathtt{len}(y)) \wedge \mathtt{len}(x) \geq 50$ is satisfiable, so is the string constraint.

5.4 Implementation

During the first call of nextSolution, the Nielsen graph together with the counter abstraction with saturated self-loops is constructed. Then, during each call of nextSolution, another extended run containing self-loops is generated. If there are no more suitable extended runs left, nextSolution returns undef (the procedure is incomplete). The method getLIA then returns the length constraint corresponding to the current extended run. The preprocess function implements the rule LenSplit. Suitability checking function isSuitable checks if there are only equations and length constraints in the system and the system is quadratic. If there are no length constraints and the system is *not* chain-free [3] we also use this procedure (the Nielsen graph is in such cases usually smaller that the proof graph generated by the stabilization-based procedure).

5.5 Model Generation

The method getLIA generates the LIA formula describing values of counters of a current extended run. For each transition of the extended run, we remember the Nielsen rule corresponding to the counter updates. The rule for self-loops that were saturated by the extended rule is of the form $x \hookrightarrow wx$, where $w \in \Sigma^+$ (the rule was obtained by concatenating the symbols from Nielsen rules that were used for the saturation). Moreover, for each simple self-loop in the extended run, we also remember the fresh LIA variable counting the number of times the self-loop was taken. For simplicity, for a state q of the extended graph, we define $sl(q) = q^{sl}$ if q has a self-loop and q otherwise. The method getModel(θ, x) then builds during the first call the model for all variables and then just returns the computed value for the particular variable x. The model is constructed by following the current extended run starting from the initial state q_0 and the initial model $\nu_{q_0} : \{x \mapsto \epsilon \mid x \in \mathbb{X}\}$. For a transition $q \to q'$, where $q \neq q'$, there are three possibilities for the label. If it is labeled by the counter update $x := 0$, we construct the next model as $\nu_{q'} = \nu_{sl(q)} \triangleleft \{x \mapsto \epsilon\}$. For a transition labeled by the counter update $x := x + y$, we construct the next model as $\nu_{q'} = \nu_{sl(q)} \triangleleft \{x \mapsto \nu_{sl(q)}(x) \cdot \nu_{sl(q)}(y)\}$. Finally, for the update $x := x + 1$, we construct the model as $\nu_{q'} = \nu_{sl(q)} \triangleleft \{x \mapsto \alpha \cdot \nu_{sl(q)}(x)\}$, where $x \hookrightarrow \alpha x$ is the Nielsen rule corresponding to the transition. For a self-loop $q \to q$ that is labeled by the counter update $x := x + \ell$, for $\ell \in \mathbb{N}$, we construct the next model as $\nu_{q^{sl}} = \nu_q \triangleleft \{x \mapsto w^{\theta(k)} \cdot \nu_q(x)\}$ where k is the LIA variable of the self-loop and $x \hookrightarrow wx$ is the corresponding extended rule.

6 Length-based Decision Procedure

Even though the stabilization-based procedure can be fast, it may suffer from an explosion in the number of alignments, especially for large systems of equations with many unrestricted variables and literals. To deal with such formulae, we propose a length-based decision procedure, which can symbolically encode all possible alignments using LIA formulae. Solving of the string formula is hence converted to the solving of a LIA formula, which might be easier.

Block-acyclic string constraints. An *equational block* (or just a *block* for short) of a variable x is a conjunction of string equations of the form shown in the right, where for each $i \neq j$, $R_i \in (\mathbb{X} \cup \Sigma)^*$, each variable of R_i has at most one occurrence in R_i, $x \notin \mathtt{Var}(R_i)$, and $\mathtt{Var}(R_i) \cap \mathtt{Var}(R_j) = \emptyset$.

$$\bigwedge_{1 \leq i \leq n} x = R_i$$

A conjunction of equational blocks is called a *block string constraint*. We abuse the notation and treat $\mathcal{E}_x$ as a set of atoms of the conjunction. A directed graph $G = (\{\mathcal{E}_x \mid x \in \mathbb{X}\}, \{(\mathcal{E}_x, \mathcal{E}_y) \mid x \neq y \wedge (y \in \mathtt{Var}(\mathcal{E}_x) \vee (\mathtt{Var}(\mathcal{E}_x) \cap \mathtt{Var}(\mathcal{E}_y)) \setminus \{x, y\} \neq \emptyset)\})$ is called the block-graph of the block string constraint. Block-graph connects blocks s.t. the values of variables of the adjacent blocks affect the value of the block variable of the predecessor. A string constraint is called *block-acyclic* if the corresponding block-graph is acyclic. As an example, Fig. 2 shows the block-graph for the block-acyclic constraint $x = abyc \wedge x = zw \wedge x = uddc \wedge y = vad \wedge y = as$.

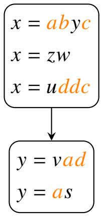


Fig. 2: A block-graph

6.1 Decision Procedure for Block-acyclic Constraints

In this section, we propose a decision procedure for block-acyclic string constraints extended with length constraints based on a translation of the string system to a LIA formula. The LIA formula symbolically encodes alignments of literals occurring in the system. Since block-acyclic constraints do not contain repetitions of variables (except the block variables connecting the blocks), every compatible alignment of literals forms a solution as the unrestricted variables adapt to the alignment of variables.

Let $\mathcal{E}_x\colon x = abyab \wedge x = zabucdw$ be an equational block. We need to find the positions of different occurrences of literals ab, ab, ab, and cd within the word represented by x, so that they do not clash with each other, nor with the words represented by the variables y, z, u, and w occurring in $\mathcal{E}_x$. To encode this, we use fresh integer variables B^x_{ab}, B^x_{ab}, B^x_y, etc., that represent the starting position of literals/variables within the word x. Then we just need to encode three things: (i) the literals/variables follow each other in the equation (for example, $B^x_{ab} = B^x_y + \mathtt{len}(y)$), (ii) the literals are not mismatched (for example, $B^x_{ab} = 5$ and $B^x_{cd} = 4$ is not valid, as this would force both a and d to be at the fifth position), and (iii) literals occurring inside variables y, z, u, and w follow the same rules. For the last one, we need to also define starting positions of literals that occur within those variables. For this reason, we use $\mathsf{lit}(\mathcal{E}_x)$ to denote *occurrences of literals* in the block $\mathcal{E}_x$ (for our example it would be $\mathsf{lit}(\mathcal{E}_x) = \{ab, ab, ab, cd\}$) and, for a block graph $G = (V, E)$, we define $\mathsf{litall}_G(\mathcal{E}_x)$ as the union of all $\mathsf{lit}(\mathcal{E}_y)$ such that there is a path from $\mathcal{E}_x$ to $\mathcal{E}_y$ in G.

The construction of a LIA formula for a block graph G is given in Algorithm 2. It describes the construction of the LIA formula φ compactly encoding all alignments of literals occurring in the string constraint. The formula ψ_{pos} introduces the equational length constraint for each equation of the block $\mathcal{E}_x$ and with the subformula

Algorithm 2: Encoding alignments of $\mathcal{E}$

Data: Block-graph $G = (V = \{\mathcal{E}_x \mid x \in \mathbb{X}\}, E)$
Result: LIA formula φ encoding all models of $\mathcal{E}$

$\varphi := \top;$
for $\mathcal{E}_x \in V$ **do**
  $\psi_{pos} := \bigwedge_{x=t_1\cdots t_n \in \mathcal{E}_x} \mathtt{len}(x) = \sum_{1\leq i\leq n} \mathtt{len}(t_i) \wedge \mathsf{pos}_x(t_1\cdots t_n);$
  $\psi_{match} := \bigwedge_{\ell_1,\ell_2 \in \mathsf{litall}_G(\mathcal{E}_x), \ell_1\neq\ell_2} \mathsf{comp}_x(\ell_1,\ell_2) \vee \mathsf{mis}_x(\ell_1,\ell_2);$
  $\varphi := \varphi \wedge \psi_{pos} \wedge \psi_{match};$
for $(\mathcal{E}_x, \mathcal{E}_y) \in E$ *s.t.* $y \in \mathtt{Var}(\mathcal{E}_x)$ **do**
  $\psi_{beg} := \bigwedge_{\ell \in \mathsf{litall}_G(\mathcal{E}_y)} B^x_\ell = B^x_y + B^y_\ell;$
  $\varphi := \varphi \wedge \psi_{beg};$
return φ;

$$\mathsf{pos}_x(t_1\cdots t_n) \Leftrightarrow B^x_{t_1} = 0 \wedge \bigwedge_{2\leq i\leq n} B^x_{t_i} = B^x_{t_{i-1}} + \mathtt{len}(t_{i-1})$$

it sets the beginnings of each literal/variable in the correct order for each equation.

If equations of $\mathcal{E}_x$ contain a variable y of the block $\mathcal{E}_y$ (hence there is an edge $(\mathcal{E}_x, \mathcal{E}_y)$ in E), it is necessary to propagate literals of $\mathcal{E}_y$ also to the block $\mathcal{E}_x$. Therefore, literals occurring in $\mathcal{E}_y$ (meaning that they occur in a possible model of y) transitively appear in a possible model of x through the equivalence. Since the literals of $\mathcal{E}_y$ in a possible model may occur in the same model of x only on positions determined by the occurrence of y in the block $\mathcal{E}_x$, we generate the formula ψ_{beg} expressing that literals occurring in y are shifted by B^x_y. See Fig. 3 for a schematic example.

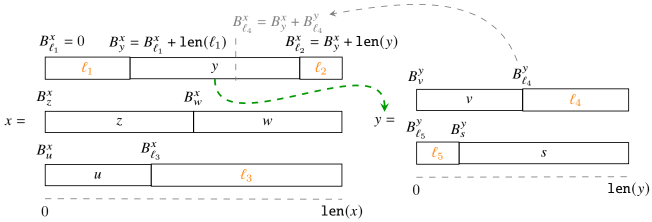


Fig. 3: Schematic example of encoding between $\mathcal{E}_x$ and $\mathcal{E}_y$.

The last subformula ψ_{match} expresses that two literals ℓ_1 and ℓ_2 are completely misaligned (mis_x), or they are aligned only in a compatible way (comp_x). Formally, these predicates are defined as

$$\mathsf{mis}_x(\ell_1, s, e) \Leftrightarrow B^x_{\ell_1} + \mathtt{len}(\ell_1) \leq s \vee B^x_{\ell_1} \geq e, \qquad \mathsf{comp}_x(\ell_1,\ell_2) \Leftrightarrow \bigvee_{i\in\mathsf{align}(\ell_1,\ell_2)} B^x_{\ell_1} + i = B^x_{\ell_2}.$$

We abuse the notation and use $\mathsf{mis}_x(\ell_1,\ell_2)$ to denote $\mathsf{mis}_x(\ell_1, B^x_{\ell_2}, B^x_{\ell_2} + \mathtt{len}(\ell_2))$. The formula $\mathsf{mis}_x(\ell_1, s, e)$ expresses that the literal ℓ_1 is not intersecting the interval (s, e). The set $\mathsf{align}(\ell_1,\ell_2)$ contains matching shiftments of ℓ_1 relative to ℓ_2. Formally, $\mathsf{align}(\ell_1,\ell_2) = \{i \mid \exists u\colon \ell_2 = \ell_1^{i:}u\} \cup \{-i \mid \exists u\colon \ell_1 = \ell_2^{i:}u\}$ where for a literal $\ell = a_0\cdots a_n$, we use $\ell^{i:}$ to denote the string $a_i\cdots a_n$.

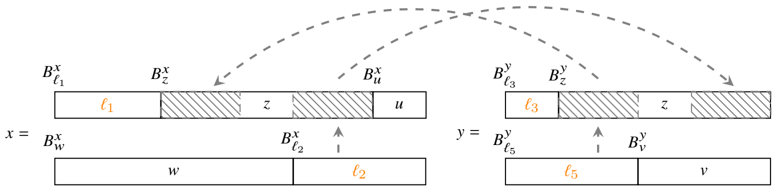


Fig. 4: A schematic example of a shared variable underapproximation of the string constraint $x = \ell_1 z u \land x = w\ell_2 \land y = \ell_3 z \land y = \ell_5 v$. Positions of z covered by parts of literals ℓ_2 and ℓ_5 are marked by hatching. These positions are excluded for possible alignments of other literals.

Because satisfiability checking of quantifier-free LIA formulae (the occurrences of $\texttt{len}(x)$ can be replaced with a pure integer variable) is in **NP**, it is easy to see that the following lemma holds.

Lemma 1. *Satisfiability checking of block-acyclic string constraints is in* **NP**.

6.2 Underapproximation of a Shared Variable

In this section, we generalize the length-based decision procedure to a block string fragment containing two blocks sharing a *single* variable that is different from block variables. In particular, in the following text, we assume two blocks $\mathcal{E}_x$ and $\mathcal{E}_y$ s.t. $y \in \texttt{Var}(\mathcal{E}_x)$ and $\texttt{Var}(\mathcal{E}_x) \cap \texttt{Var}(y) \supseteq \{z\}$ meaning that z is a variable that is shared among the blocks $\mathcal{E}_x$ and $\mathcal{E}_y$ (the block-graph has a cycle between $\mathcal{E}_x$ and $\mathcal{E}_y$). We further assume that z does not have its own block (in general there might be more shared variables but between two blocks only).

For instance, consider the string constraint $x = ayz \land x = ab \land y = bz$ with the shared variable z. For this system, we underapproximate the solution by ensuring that literals occurring inside a potential model of z are all misaligned with all other possible literals (since z has the same value among occurrences, parts of literals placed inside z are propagated among different occurrences of z). This yields an underapproximation since some of these completely excluded literals might be aligned with literals occurring inside z. See Fig. 4 for a schematic example. Formally, for blocks $\mathcal{E}_x$ and $\mathcal{E}_y$ sharing the variable z, the formula excluding alignments of literals inside occurrences of z is given as

$$\psi^{x,y}_{\mathsf{excl}}(z) \Leftrightarrow \bigwedge_{\ell \in \mathsf{litall}_{\overline{G}}(\mathcal{E}_y)} \left(\ell \in_y z \rightarrow \bigwedge_{\ell' \in \mathsf{litall}_{\overline{G}}(\mathcal{E}_x), \ell' \neq \ell} \mathsf{out}^y_x(\ell', z, \ell) \right)$$

where $\overline{G}$ is the block graph obtained from G by removing edges induced by the shared variables, $\ell \in_y z \Leftrightarrow \neg\mathsf{mis}_y(\ell, B^z_y, B^z_y + \texttt{len}(y))$ and $\mathsf{out}^y_x(\ell', z, \ell)$ is the formula expressing that ℓ' in the block x is placed on a different position than the propagated ℓ occurring inside z in y. Formally, $\mathsf{out}^y_x(\ell', z, \ell) \Leftrightarrow \mathsf{mis}_x\left(\ell', B^x_z + j, B^x_z + k\right)$, where $j = B^z_y - \max(B^z_y, B^\ell_y)$ and $k = B^z_y - \min(B^z_y + \texttt{len}(z), B^\ell_y + \texttt{len}(\ell))$. The formula $\psi^{x,y}_{\mathsf{excl}}(z)$ is then conjoined with the formula obtained from Algorithm 2.

6.3 Implementation

Since the length-based procedure generates a LIA formula describing all models of the input string constraint, the method nextSolution generates the formula together with the precision, which are then returned by getLIA. The precision is set to underapprox if an undeapproximation preprocessing rule was used, or the approach for a shared variable was applied. Further calls of nextSolution then return $\bot$. On top of that, during the first call of nextSolution, it checks whether the formula obtained by preprocessing is block acyclic (possibly with a shared variable). If not, nextSolution returns unknown (the length-based procedure is skipped). The preprocess method utilizes the same preprocessing rules as the preprocessing stabilization-based procedure does, except for rules introducing complex regular constraints, which are avoided. The method isSuitable checks whether the input constraints contain only equations and length constraints.

6.4 Model Generation

For the length-based procedure, the model of each variable (provided that the generated LIA formula is satisfiable) is determined by positions of literals. For each variable x we allocate a string skeleton of length $\theta(\texttt{len}(x))$. Fields of the skeleton will be filled with symbols from literals occurring of the corresponding positions in a block. Starting from a possible shared variable z, we take blocks containing occurrences of z and fill skeleton fields corresponding to symbols of literals involving the value of z (given by values of the begin variables for literals of the block). Then, we iteratively process blocks that do not contain block variables of still unprocessed blocks. For the block variable we update the fields of the skeleton given by positions of literals in the block and already filled skeletons of other variables. Then, we propagate the values of the block variable to variables occurring inside the block. After each block is processed, we fill fields of each variable that remained empty with the symbol a.

7 Stabilization-based procedure

The main and the most general decision procedure is the *stabilization-based procedure* applicable for any constraint. It was first introduced in [15] for handling word equations with regular constraints and then extended for handling length constraints [25] and string-integer conversions [29]. The procedure follows the framework from Section 3.3, with preprocessing rules described in [25], the nextSolution function follows the procedure from [25], and getLIA is based on [25,29]. As the procedure is explained in these papers, we do not give details here, instead, we focus on model generation, which was not done before.

7.1 Model generation

The model generation is implemented recursively (i.e., the model of a variable might be constructed from models of different variables) and a single top-level call of getModel may result in computing models of more variables. In such a case, we memoize the results and return their values directly if they are required.

After the stabilization-based procedure finds a solution it ends with: (i) the set of all variables divided into three disjoint sets: $\mathbb{X}_I$ are variables whose length or string-integer conversion value is important, $\mathbb{X}_N$ are variables for which these values are *not* important, and a set of fresh variables $\mathbb{X}_F$, (ii) the set of inclusions $I = \{u_1 \subseteq v_1, \ldots, u_n \subseteq v_n\}$ that contain only variables from $\mathbb{X}_N \cup \mathbb{X}_F$, (iii) the substitution map $\sigma\colon \mathbb{X}_I \to \mathbb{X}_F^*$ that substitutes variables from $\mathbb{X}_I$ with a concatenation of fresh variables, and (iv) the language assignment $\mathsf{Lang}\colon (\mathbb{X}_N \cup \mathbb{X}_F) \to 2^{\Sigma^*}$ such that the languages of fresh variables are precise, i.e., for each combination of words from the languages of fresh variables, there is a selection of words for variables from $\mathbb{X}_N$ such that each inclusion from I holds.

At the start of the model generation, during the very first call of getModel, we restrict the language assignments of fresh variables so that they follow the lengths/conversion values given by the LIA model θ. In particular, for $y \in \mathbb{X}_F$, we restrict the language $\mathsf{Lang}(y)$ only to the words of the length of $\theta(\mathtt{len}(y))$ and for conversion values, we restrict it to the singleton language containing exactly the string converted to that value.

The method $\mathsf{getModel}(\theta, x)$ used to get a model of x then first checks whether $x \in \mathbb{X}_I$. If true, it recursively calls $\mathsf{getModel}(\theta, x_i)$ on all fresh variables from $\sigma(x) = x_1 \cdots x_n$ and then constructs the model of x by their concatenation. Because the values of fresh variables are restricted by the LIA model θ, this means that the value of x will be correctly restricted too. If $x \notin \mathbb{X}_I$, we check if the variable is *not* on the right-hand side of any inclusion of I. In such case, we just return some word from $\mathsf{Lang}(x)$.

In the last case, when x is on the right-hand side of some inclusion $y_1 \cdots y_n \subseteq v$, we first recursively get models of all variables on the left-hand side (by calling $\mathsf{getModel}(\theta, y_i)$ for each $1 \leq i \leq n$). The concatenated models for the left-hand side yield the word $w = \mathsf{getModel}(\theta, y_1) \cdots \mathsf{getModel}(\theta, y_n)$. Subsequently, we find models of all variables on the right-hand side (including x) in a way that their concatenation matches w. This is implemented using a backtracking algorithm that reads the word w and checks whether w can be split into subwords (each subword corresponding to a variable of the right-hand side) that belong to the languages of the particular variables. Note that this algorithm works only if the variable x occurs at most once on the right-hand side of *any* inclusion and, furthermore, there is no cycle (e.g. for the inclusion $xy \subseteq zx$, we cannot get a model for x as shown above). For such cases we would need to use the algorithm from the proof of [16, Theorem 5], which is currently not implemented in Z3-NOODLER. However, as experiments show, this has almost no practical impact as the stabilization-based procedure usually does not finish for such cases anyway.

8 Experiments

We implemented the presented decision procedures and model generation in version 1.3 of Z3-NOODLER and evaluated them on all string benchmarks from SMT-LIB [7]. We split the benchmarks into three categories: In **Regex** we gather the benchmark sets that contain mostly regular and length constraints: AutomatArk [12], Denghang, Redos, StringFuzz [17], and Sygus-qgen. The benchmark sets Kaluza [46,33], Kepler [30], Norn [1,2], Omark, Slent [52], Slog [53], Webapp, and Woorpje [28], consisting of mostly word equations and length constraints with some small number of more complex constraints, are in the **Equations** category. The last category, **Predicates**, contains benchmark sets FullStrInt, LeetCode, PyEx [43], StrSmallRw [41], and Transducer+ [22], which heavily feature more complex string constraints. The experiments were executed on a workstation with an AMD Ryzen 5 5600G CPU @ 3.8 GHz with 100 GiB of RAM running Ubuntu 22.04.4. The timeout was set to 120 s, memory limit was 8 GiB.

Table 1: The impact of decision procedures on solving for each benchmark set/category for *solved formulae*. Second column shows the number of times a string solver was called within Z3's DPPL(T) procedure. Next columns show how many times (relative to the number of calls) was each decision procedure *called* and how many of these calls were *solved* by the decision procedure.

	number of calls	Regex proc.		Nielsen transf.		Length-based		Stabillization-based	
		called	solved	called	solved	called	solved	called	solved
Sygus-qgen	747	100%	100%	0%	0%	0%	0%	0%	0%
Denghang	999	0.10%	0.10%	0%	0%	96.10%	96.10%	3.80%	3.80%
AutomatArk	20,062	99.97%	99.97%	0%	0%	0.02%	0.02%	0.01%	0.01%
StringFuzz	9,941	46.45%	46.45%	0%	0%	27.98%	27.96%	25.58%	25.58%
Redos	2,952	70.02%	70.02%	0%	0%	11.21%	11.21%	18.77%	18.77%
Full **Regex**	34,701	79.21%	79.21%	0%	0%	11.75%	11.74%	9.04%	9.04%
LeetCode	874	1.37%	1.37%	0%	0%	59.27%	16.70%	81.92%	81.92%
StrSmallRw	6,327	0%	0%	0%	0%	4.85%	3.75%	96.25%	96.25%
PyEx	26,045	0.10%	0.10%	0%	0%	0.08%	0.08%	99.82%	99.82%
FullStrInt	9,003	0.04%	0.04%	0%	0%	0.26%	0.26%	99.70%	99.70%
Transducer+	0	-		-		-		-	
Full **Predicates**	42,249	0.10%	0.10%	0%	0%	2.06%	1.01%	98.89%	98.89%
Norn	918	11.76%	11.76%	0%	0%	6.86%	6.86%	81.37%	81.37%
Slog	1,565	25.37%	25.37%	0%	0%	0.13%	0.13%	74.50%	74.50%
Slent	1,489	0.40%	0.40%	0%	0%	35.19%	30.09%	69.51%	69.51%
Omark	9	0%	0%	11.11%	11.11%	11.11%	0%	88.89%	88.89%
Kepler	579	0%	0%	99.83%	99.83%	0%	0%	0%	0%
Woorpje	478	0.84%	0.84%	43.10%	42.47%	30.96%	27.20%	20.50%	20.50%
Webapp	381	0.52%	0.52%	0%	0%	2.36%	0.26%	99.21%	99.21%
Kaluza	11,222	35.31%	35.31%	0%	0%	63.45%	61.78%	2.91%	2.91%
Full **Equations**	16,641	26.92%	26.92%	4.72%	4.70%	47.27%	45.53%	22.59%	22.59%
All	93,591	34.20%	34.20%	0.84%	0.84%	13.69%	12.91%	52.01%	52.01%

Procedures comparison. Table 1 shows the impact of various decision procedures within Z3-Noodler on solving string constraints. We compare the number of times a particular procedure was used for solving. Note that the total number of calls might be different to the number of formulae in the particular benchmark as some formulae might have been solved directly by the theory rewriter or the initial phase (and therefore no call of a decision procedure was used) while some formulae might have resulted in multiple calls. The table shows that the decision procedure for regex constraints is (unsurprisingly) strong on the **Regex** benchmark. Furthermore, the length-based procedure is quite strong on **Equations** (and on regex-heavy benchmarks due to the StringFuzz benchmark set as it contains parts with pure equations). Even though the impact of the Nielsen transformation seems low, without it, Z3-Noodler cannot solve most of the formulae the procedure solves. Note that for **Equations**, there are 44 calls that were not solved by any presented procedure. Instead, they were solved by a simple procedure that is used for benchmarks that contain equations with exactly one symbol and length constraints. This procedure just asks if the lengths of both sides of each equation are the same (as there is only one symbol, it is the same as asking if the equation holds). All in all, Z3-Noodler with all procedures enabled takes 3,583 seconds less to solve 944 formulae more than Z3-Noodler with only stabilization-based procedure, which is a significant improvement (see tables in the appendix for more detailed results). From

Table 2: Numbers of solved instances and the time (in seconds) needed to solve them for each tool and benchmark category. The versions of tools with $\mathcal{M}$ were run with model generation turned on. The total number of formulae for each benchmark category is given in parentheses.

	Regex (32,242)		**Equations** (25,727)		**Predicates** (45,436)		All (103,405)	
	solved	time	solved	time	solved	time	solved	time
Z3-Noodler	32,232	3,688	25,301	1,147	45,035	6,353	**102,568**	**11,118**
Z3-Noodler$^{\mathcal{M}}$	32,228	4,010	25,299	1,456	45,035	7,321	**102,562**	**12,787**
cvc5	29,290	59,705	25,214	2,529	45,337	11,627	99,841	73,861
cvc5$^{\mathcal{M}}$	29,287	59,892	25,214	2,756	45,337	12,220	99,838	74,868
Z3	29,075	51,379	24,569	3,240	44,101	74,094	97,745	128,712
Z3$^{\mathcal{M}}$	29,064	51,830	24,571	4,013	44,096	74,708	97,731	130,551

Table 1 it is also evident that the values of *called* and *solved* are close to each other, i.e., the suitability check can precisely identify a suitable decision procedure.

Comparison with other tools. In Table 2 and Fig. 5, we compare Z3-Noodler with cvc5 (version 1.2.0) and Z3 (version 4.13.0). The table also shows the impact of model generation. From the results we can see that Z3-Noodler is significantly better (in both the number of solved instances and the time) than other tools on **Regex**. Furthermore, it is better than other tools on **Equations**, while slightly worse than cvc5 on **Predicates**. Comparing the impact of model generation, we can see that for all three tools, it is not significant, there is usually some slight slowdown with a few less solved instances. All in all, even with model generation, Z3-Noodler can solve the most number of instances the fastest.

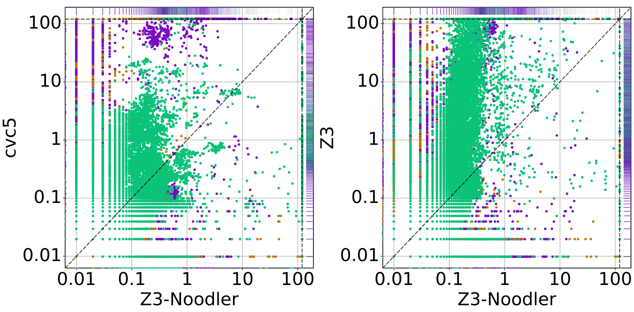


Fig. 5: Comparison with cvc5 and Z3. Times are in seconds, axes are logarithmic. Dashed lines are timeouts. Colours distinguish groups: • **Regex**, • **Equations**, and • **Predicates**.

Data availability statement. An environment with the tools and data used for the experimental evaluation in the current study is available at [27].

Acknowledgments

This work has been supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 23-07565S, and the FIT BUT internal project FIT-S-23-8151. The work of David Chocholatý, Brno Ph.D. Talent Scholarship Holder, is Funded by the Brno City Municipality. This work has been executed under the project VASSAL: "Verification and Analysis for Safety and Security of Applications in Life" funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 150–166. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_10, https://doi.org/10.1007/978-3-319-08867-9_10
2. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 462–469. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_29, https://doi.org/10.1007/978-3-319-21690-4_29
3. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P.: Chain-free string constraints. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 277–293. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_16, https://doi.org/10.1007/978-3-030-31784-3_16
4. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: TACAS'10. LNCS, vol. 6015, pp. 158–174. Springer (2010)
5. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for aws access policies using smt. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8602994
6. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength smt solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer International Publishing, Cham (2022)
7. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
8. Barrett, C.W., Tinelli, C., Deters, M., Liang, T., Reynolds, A., Tsiskaridze, N.: Efficient solving of string constraints for security analysis. In: HotSoS'16. pp. 4–6. ACM Trans. Comput. Log. (2016)
9. Bernardo, P., Veronese, L., Valle, V.D., Calzavara, S., Squarcina, M., Adão, P., Maffei, M.: Web platform threats: Automated detection of web security issues with WPT. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 757–774. USENIX Association, Philadelphia, PA (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/bernardo
10. Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: Towards more efficient methods for solving regular-expression heavy string constraints. Theor. Comput. Sci. **943**, 50–72 (2023). https://doi.org/10.1016/j.tcs.2022.12.009, https://doi.org/10.1016/j.tcs.2022.12.009
11. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 55–59 (2017). https://doi.org/10.23919/FMCAD.2017.8102241
12. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An SMT solver for regular expressions and linear arithmetic over string length. In: Silva,

A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 289–312. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_14, https://doi.org/10.1007/978-3-030-81688-9_14

13. Berzish, Murphy: Z3str4: A Solver for Theories over Strings. Ph.D. thesis (2021), http://hdl.handle.net/10012/17102
14. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: TACAS'09. LNCS, vol. 5505, pp. 307–321. Springer (2009)
15. Blahoudek, F., Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Word equations in synergy with regular constraints. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Formal Methods. pp. 403–423. Springer International Publishing, Cham (2023)
16. Blahoudek, F., Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Word equations in synergy with regular constraints (technical report) (2022), an extended version of the paper published at FM'23
17. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: StringFuzz: A fuzzer for string solvers. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 45–51. Springer International Publishing, Cham (2018)
18. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Proc. of Symposium on Mathematical Theory of Automata (1962)
19. Bustan, D., Grumberg, O.: Simulation based minimization. In: Proceedings of CADE-17. LNCS, vol. 1831, pp. 255–270. Springer (2000)
20. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proc. ACM Program. Lang. **2**(POPL), 3:1–3:29 (2018). https://doi.org/10.1145/3158091, https://doi.org/10.1145/3158091
21. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498707, https://doi.org/10.1145/3498707
22. Chen, T., Hague, M., He, J., Hu, D., Lin, A.W., Rümmer, P., Wu, Z.: A decision procedure for path feasibility of string manipulating programs with integer data type. In: Hung, D.V., Sokolsky, O. (eds.) Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12302, pp. 325–342. Springer (2020). https://doi.org/10.1007/978-3-030-59152-6_18, https://doi.org/10.1007/978-3-030-59152-6_18
23. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019). https://doi.org/10.1145/3290362, https://doi.org/10.1145/3290362
24. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-Noodler: An automata-based string solver. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 24–33. Springer Nature Switzerland, Cham (2024)
25. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Solving string constraints with lengths by stabilization. Proc. ACM Program. Lang. **7**(OOPSLA2) (oct 2023). https://doi.org/10.1145/3622872
26. Chocholatý, D., Fiedor, T., Havlena, V., Holík, L., Hruška, M., Lengál, O., Síč, J.: Mata: A fast and simple finite automata library. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 130–151. Springer Nature Switzerland, Cham (2024)

27. Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-Noodler 1.3: Shepherding decision procedures for strings with model generation (Oct 2024). https://doi.org/10.5281/zenodo.13989789, https://doi.org/10.5281/zenodo.13989789
28. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In: Filiot, E., Jungers, R.M., Potapov, I. (eds.) Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11674, pp. 93–106. Springer (2019). https://doi.org/10.1007/978-3-030-30806-3_8, https://doi.org/10.1007/978-3-030-30806-3_8
29. Havlena, V., Holík, L., Lengál, O., Síč, J.: Cooking String-Integer Conversions with Noodles. In: Chakraborty, S., Jiang, J.H.R. (eds.) 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024). Leibniz International Proceedings in Informatics (LIPIcs), vol. 305, pp. 14:1–14:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). https://doi.org/10.4230/LIPIcs.SAT.2024.14, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2024.14
30. Le, Q.L., He, M.: A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: Ryu, S. (ed.) Programming Languages and Systems. pp. 350–372. Springer International Publishing, Cham (2018)
31. Liana Hadarean: String solving at Amazon. https://mosca19.github.io/program/index.html (2019), presented at MOSCA'19
32. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 646–662. Springer International Publishing, Cham (2014)
33. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. Formal Methods in System Design **48**(3), 206–234 (2016)
34. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: FroCoS'15. LNCS, vol. 9322, pp. 135–150. Springer (2015)
35. Lin, A.W., Majumdar, R.: Quadratic word equations with length constraints, counter systems, and presburger arithmetic with divisibility. Log. Methods Comput. Sci. **17**(4) (2021). https://doi.org/10.46298/lmcs-17(4:4)2021, https://doi.org/10.46298/lmcs-17(4:4)2021
36. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 123–136. ACM (2016). https://doi.org/10.1145/2837614.2837641, https://doi.org/10.1145/2837614.2837641
37. Loring, B., Mitchell, D., Kinder, J.: Sound regular expression semantics for dynamic symbolic execution of javascript. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 425–438. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314645, https://doi.org/10.1145/3314221.3314645
38. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
39. Nielsen, J.: Die isomorphismen der allgemeinen, unendlichen gruppe mit zwei erzeugenden. Mathematische Annalen **78**(1), 385–397 (1917)
40. Nötzli, A., Reynolds, A., Barbosa, H., Barrett, C., Tinelli, C.: Even faster conflicts and lazier reductions for string solvers. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 205–226. Springer International Publishing, Cham (2022)

41. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C., Tinelli, C.: Syntax-guided rewrite rule enumeration for SMT solvers. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing – SAT 2019. pp. 279–297. Springer International Publishing, Cham (2019)
42. Reynolds, A., Notzlit, A., Barrett, C., Tinelli, C.: Reductions for strings and regular expressions revisited. In: 2020 Formal Methods in Computer Aided Design (FMCAD). pp. 225–235 (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_30
43. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 453–474. Springer International Publishing, Cham (2017)
44. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 3–18. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_1, https://doi.org/10.1007/978-3-031-13185-1_1
45. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: 2010 IEEE Symposium on Security and Privacy. pp. 513–528. IEEE (2010)
46. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: Kaluza web site (2023), https://webblaze.cs.berkeley.edu/2010/kaluza/
47. SMT-COMP'24: https://smt-comp.github.io/2024/ (2024)
48. Stanford, C., Veanes, M., Bjørner, N.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 620–635. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454066, https://doi.org/10.1145/3453483.3454066
49. Trinh, M., Chu, D., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: CCS. pp. 1232–1243. ACM Trans. Comput. Log. (2014)
50. Trinh, M., Chu, D., Jaffar, J.: progressive reasoning over recursively-defined strings. In: CAV'16. LNCS, vol. 9779, pp. 218–240. Springer (2016)
51. Trinh, M.T., Chu, D.H., Jaffar, J.: Inter-theory dependency analysis for smt string solvers. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428260, https://doi.org/10.1145/3428260
52. Wang, H.E., Chen, S.Y., Yu, F., Jiang, J.H.R.: A symbolic model checking approach to the analysis of string and length constraints. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p. 623–633. ASE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3238147.3238189, https://doi.org/10.1145/3238147.3238189
53. Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R.: String analysis via automata manipulation with logic circuit representation. In: CAV'16. LNCS, vol. 9779, pp. 241–260. Springer (2016)
54. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.: Antichains: A new algorithm for checking universality of finite automata. In: CAV'06. LNCS, vol. 4144, pp. 17–30. Springer (2006)
55. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods in System Design **44**(1), 44–70 (2014)
56. 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: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 235–254. Springer International Publishing, Cham (2015)
57. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A Z3-based string solver for web application analysis. In: ESEC/FSE'13. pp. 114–124. ACM Trans. Comput. Log. (2013)