# Solving String Constraints with Lengths by Stabilization

YU-FANG CHEN, Academia Sinica, Taiwan
DAVID CHOCHOLATÝ, Brno University of Technology, Czech Republic
VOJTĚCH HAVLENA, Brno University of Technology, Czech Republic
LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic
ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic
JURAJ SÍČ, Brno University of Technology, Czech Republic

We present a new algorithm for solving string constraints. The algorithm builds upon a recent method for solving word equations and regular constraints that interprets string variables as languages rather than strings and, consequently, mitigates the combinatorial explosion that plagues other approaches. We extend the approach to handle linear integer arithmetic length constraints by combination with a known principle of equation alignment and splitting, and by extension to other common types of string constraints, yielding a fully-fledged string solver. The ability of the framework to handle unrestricted disequalities even extends one of the largest decidable classes of string constraints, the chain-free fragment. We integrate our algorithm into a DPLL-based SMT solver. The performance of our implementation is competitive and even significantly better than state-of-the-art string solvers on several established benchmarks obtained from applications in verification of string programs.

CCS Concepts: • **Theory of computation → Automated reasoning**; **Logic and verification**; **Regular languages**.

Additional Key Words and Phrases: string constraints, stabilization, word equations, SMT solving, length constraints, regular languages

**ACM Reference Format:**
Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2023. Solving String Constraints with Lengths by Stabilization. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 296 (October 2023), 30 pages. https://doi.org/10.1145/3622872

## 1 INTRODUCTION

*String constraint solving* has been receiving much attention in recent years, particularly in the context of analyzing web programs. The main source of motivation has been verification of the absence of security vulnerabilities, especially against SQL injection or cross-site scripting (XSS) attacks, which are still considered to be among the most frequent and problematic sources of security bugs related to web applications [OWASP 2013, 2017, 2021]. String constraint solving is useful in verification of programs that manipulate strings in general, particularly programs written in scripting languages like Python or JavaScript, where strings are a primary data type.

Authors' addresses: Yu-Fang Chen, Academia Sinica, Institute of Information Science, Taiwan, yfc@iis.sinica.edu.tw; David Chocholatý, Faculty of Information Technology, Brno University of Technology, Czech Republic, xchoch08@stud.fit.vutbr.cz; Vojtěch Havlena, Faculty of Information Technology, Brno University of Technology, Czech Republic, ihavlena@fit.vutbr.cz; Lukáš Holík, Faculty of Information Technology, Brno University of Technology, Czech Republic, holik@fit.vutbr.cz; Ondřej Lengál, Faculty of Information Technology, Brno University of Technology, Czech Republic, lengal@fit.vutbr.cz; Juraj Síč, Faculty of Information Technology, Brno University of Technology, Czech Republic, sicjuraj@fit.vutbr.cz.


2475-1421/2023/10-ART296
https://doi.org/10.1145/3622872

In addition, string constraints are a natural and general formalism that is likely to keep finding new applications, similarly as, e.g., regular expressions. A recent and prominent example of this claim is Amazon Web Service's use of a portfolio of string solvers to analyze user policies controlling access to cloud resources. String solvers are in AWS solving billions of queries a day [Backes et al. 2018; Liana Hadarean 2019; Rungta 2022]. Other emergent applications of string solving can be found in analyzing smart contracts [Alt et al. 2022], Microsoft Azure resource manager policies [Microsoft 2020; Stanford et al. 2021], graph databases [Barceló and Muñoz 2017], or regular document spanners [Freydenberger and Peterfreund 2021].

Numerous approaches to string solving and string solving tools have been developed in recent years. CVC4/5 [Barbosa et al. 2022; Barrett et al. 2016b; Liang et al. 2014, 2016, 2015; Nötzli et al. 2022; Reynolds et al. 2020, 2017] and Z3 [Bjørner et al. 2009; de Moura and Bjørner 2008; Stanford et al. 2021] are mature and reliable solvers used in industrial applications. Other tools, such as OSTRICH [Chen et al. 2018, 2022, 2020a, 2019; Lin and Barceló 2016], Trau [Abdulla et al. 2021, 2017, 2018, 2019], Noodler [Blahoudek et al. 2023], Z3Str/2/3/4/3RE [Berzish et al. 2023, 2017, 2021; Berzish, Murphy 2021; Zheng et al. 2015, 2013], and Slog [Wang et al. 2016] challenge the state of the art with their innovative approaches. Many other solvers have also been developed, such as Norn [Abdulla et al. 2014, 2015], S3 [Trinh et al. 2014], $\texttt{Kepler}_{22}$ [Le and He 2018], Sloth [Holík et al. 2018], Stranger [Yu et al. 2010, 2014, 2011], ABC [Aydin et al. 2015; Bultan et al. [n. d.]], HAMPI [Kiezun et al. 2012], Kaluza/Kudzu [Saxena et al. 2010], Retro [Chen et al. 2020b, 2023b], Woorpje [Day et al. 2019], PASS [Li and Ghosh 2013], to name a few, and many others [Amadini et al. 2017; Cox and Leasure 2017; Fu and Li 2010; Hooimeijer et al. 2011; Hooimeijer and Weimer 2012; Scott et al. 2017; Trinh et al. 2016; Veanes et al. 2012; Wang et al. 2018].

String solvers have evolved to handle a diverse range of string constraints, including not only *basic string constraints* such as word equations, regular, and length constraints, but also a rich palette of *extended constraints* such as string transformations (given in various forms, e.g. as transducers), replace-all/substring/index-of functions, and conversions between integers and strings. The best industrial-strength solvers are now capable of handling most of these constraints in most of the cases encountered in known benchmarks coming from practical applications.

Nevertheless, dealing with basic constraints remains a central and challenging task. Not only do basic constraints appear most frequently, but extended constraints are often transformed into combinations of basic ones. The worst-case complexity of basic constraints is high, already equations with regular constraints are PSPACE-complete [Jeż 2016; Plandowski 1999]. Moreover, despite general algorithms for solving equations exist, namely Makanin's [Makanin 1977] and Jeż's recompression [Jeż 2016], their efficient implementation seems problematic and remains an open challenge. String solvers have therefore resorted to implementing partial algorithms that work in practical cases, such as chain-free [Abdulla et al. 2019], straight-line [Lin and Barceló 2016], and quadratic forms [Chen et al. 2020b; Le and He 2018; Nielsen 1917]. Nevertheless, these fragments are still PSPACE-hard, already due to boolean combinations of regular constraints, and remain difficult to solve. Despite ongoing efforts to improve the efficiency of string solving, the handling of basic constraints continues to be a limiting factor in practice, particularly for solvers that rely on a large number of specialized heuristics. These solvers often struggle when dealing with complex combinations of regular constraints and equations, particularly on benchmarks for which they were not optimized.

Addition of length constraints to equations and regular constraints brings another layer of complexity. Despite promising attempts reported in [Le and He 2018; Lin and Majumdar 2021], the decidability of word equations combined with length constraints has remained an open problem for many years, even when restricted to quadratic word equations (word equations with at most two occurrences of each variable). Some approaches, such as [Abdulla et al. 2014, 2019; Lin and Barceló

2016], extract lengths from regular constraints and solve them using a *linear integer arithmetic* (LIA) solver after solving word equations and regular constraints. Other approaches, such as [Abdulla et al. 2018; Berzish et al. 2023, 2021; Mora et al. 2021; Reynolds et al. 2019], manage to even take advantage of input length constraints to simplify reasoning about regular properties and equations in some cases. Integrating reasoning about lengths into string solving in a general way, however, remains a complex problem.

This paper presents a novel string constraint solving algorithm that integrates reasoning about length constraints with a recent technique [Blahoudek et al. 2023], called Stabilization, which excels in solving equations and regular constraints. Unlike most other solvers, which attempt to avoid complex questions about equations and regular expressions, Stabilization works with equations and regular constraints from the beginning. Equations are approached as equivalences of concatenated languages of the variables on the left-hand side and variables on the right-hand side. The variable languages are iteratively refined until the equations stabilise (hence the name), that is, the concatenated languages of their sides become equivalent. If all the languages are still non-empty, then a solution of the original string equation exists. This approach significantly reduces the combinatorial blow-up that other algorithms suffer from when tackling equations and regular constraints separately.

Our principal technical contribution lies in extending Stabilization with reasoning about string lengths. Since Stabilization is in its basic form incompatible with other methods for reasoning about lengths, we had to merge it with an older approach called Align&Split. First implemented in the string solver Norn [Abdulla et al. 2014, 2015] and later used in the approaches behind the solvers OSTRICH [Chen et al. 2018, 2022, 2020a, 2019; Lin and Barceló 2016], Z3str3RE [Berzish et al. 2023, 2021], Sloth [Holík et al. 2018], and, to some extent, Trau [Abdulla et al. 2021, 2017, 2018, 2019], it splits automata and enumerates equation alignments. Align&Split may be slower, but it can convert regular constraints into length constraints using standard automata techniques. The obtained length constraints can then be solved alongside the input length constraint by an off-the-shelf LIA solver. We designed a fine-grained interaction between the two approaches, preserving most of the efficiency of Stabilization. In the essence, variables are eliminated from equations using Align&Split only when they are related to the length constraint.

Secondly, we observed that by including length constraints into regular constraints and word equations, we can solve word disequalities more generally than in previous approaches, such as [Abdulla et al. 2014, 2015]. Those approaches reduce a disequality between the left-hand and the right-hand side to saying that after a common (potentially empty) prefix, the next characters differ. We propose a modified encoding saying that the differing characters appear after (not necessarily identical) prefixes of the same length. This approach allows to extend one of the largest decidable fragments of string constraints, the *chain-free fragment* of [Abdulla et al. 2019]. The chain-free fragment prohibits cycles in a *graph* constructed based on the equations. As the disequalities are transformed into length constraints (unlike [Abdulla et al. 2014, 2015], where they spawn new equations), they do not affect the graph. Thus, we can extend the chain-free fragment with unrestricted disequalities while preserving decidability.

Third, we show how our new algorithm can be efficiently incorporated into an SMT solver. Utilising a relatively simple but efficient implementation of nondeterministic finite automata, we have implemented it within Z3 and obtained a new fully-fledged string solver. The integration of our algorithm in Z3 includes some design choices that are quite different from other fast solvers such as cvc5, Z3, or Z3str4/3RE. The different design comes with unique optimization opportunities. Since, as mentioned earlier, a large part of the efficiency of established string solvers comes from a large number of heuristics and optimizations tuned to characteristics of existing benchmarks,

the fact that our approach offers different optimization opportunities is highly practically relevant (we indeed confirm experimentally that it is performance-wise orthogonal to the best solvers).

After implementing some of these optimizations, we arrived at a solver with good performance. We compared it with most SMT string solvers, such as Z3, CVC5, TRAU, Z3STR3RE, Z3STR4, and OSTRICH, on the benchmark from SMT-COMP. We provide a transparent evaluation of the comparison and do not generate new benchmarks tailored to the strengths of our tool. From 10 benchmark sets of SMT-COMP'22, our solver is clearly the best on three, by a large margin, and second best on one. This success is mainly due to the superior handling of equations and regular constraints. Our tool has the least timeouts and its weaknesses seem orthogonal to those of the other tools. This result is the main selling point of our method. The tool is also the best on a recent benchmark of [Stanford et al. 2021] targeting boolean combinations of regular properties.

The benchmarks where our tool is worse are generally of two kinds: (1) our tool does not support some of the used string constraints, such as string-integer conversion or the replace-all function, or (2) the benchmark contains many similar examples that are hard for a generic algorithm but easy for a specialized heuristic, which we have not implemented yet. Some of these weaknesses represent interesting engineering or research problems, they are not principal weaknesses of our approach, and we briefly outline how they can be eliminated. Moreover, when the best solvers are combined into a portfolio solver (as used, e.g., in Amazon [Backes et al. 2018; Rungta 2022]), inclusion of our solver significantly improves the overall performance. Considering also the relative infancy of our solver, its relatively small size, and apparent space for optimization, we believe that this experiment demonstrates a significant shift in the state of the art of practical string solving.

We summarize our contribution into the following four points:

(1) A string solving algorithm extending the algorithm of [Blahoudek et al. 2023] with handling length constraints, via a fine-grained combination with the principles behind [Abdulla et al. 2014]. It allows to turn the algorithm of [Blahoudek et al. 2023] into a fully-fledged string solver, while preserving most of its efficiency.
(2) Extension of one of the largest decidable fragments of string constraints, the chain-free fragment, with unrestricted disequalities.
(3) An integration of the algorithm into the SMT solver Z3, with a number of practical heuristics and optimizations, some of them unique to our approach.
(4) An experimental evaluation showing that our approach significantly improves the state of the art of string solving technology in practice.

## 2 PRELIMINARIES

*Sets and strings.* We use $\mathbb{N}$ to denote the set of natural numbers (including 0). We fix a finite *alphabet* $\Sigma$ of *symbols/letters* (usually denoted $a, b, c, \ldots$) for the rest of the paper. A sequence of symbols $w = a_1 \cdots a_n$ from $\Sigma$ is a *word* or a *string* over $\Sigma$, with its *length* $n$ denoted by $|w|$. The set of all words over $\Sigma$ is denoted as $\Sigma^*$. $\epsilon \notin \Sigma$ is the *empty word* with $|\epsilon| = 0$. The *concatenation* of words $u$ and $v$ is denoted $u \cdot v$, $uv$ for short ($\epsilon$ is the neutral element). A set of words over $\Sigma$ is a *language*, the concatenation of languages is $L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1 \wedge v \in L_2\}$, $L_1L_2$ for short. *Bounded iteration* $x^i$, for $i \in \mathbb{N}$, of a word or a language $x$ is defined by $x^0 = \epsilon$ for a word, $x^0 = \{\epsilon\}$ for a language, and $x^{i+1} = x^i \cdot x$. Then $x^* = \bigcup_{i \in \mathbb{N}} x^i$ for languages and $x^* = \{x\}^*$ for words. The *shuffle* of languages $L_1$ and $L_2$ is the language $L_1 \sqcup\!\sqcup L_2$ of words $a_1 \ldots a_n$ for which there exists a set $P = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ such that $a_{i_1} \ldots a_{i_k} \in L_1$ with $i_1 < \ldots < i_k$ and $a_{j_1} \ldots a_{j_\ell} \in L_2$ with $\{j_1, \ldots, j_\ell\} = \{1, \ldots, n\} \setminus P$ and $j_1 < \ldots < j_\ell$. We denote regular languages using regular expressions $E$ with the following standard notation:

$$E ::= \epsilon \mid a \mid (E) \mid EE \mid E + E \mid E^* \mid E^+ \mid E^n \mid E? \tag{1}$$

for $a \in \Sigma$, $n \in \mathbb{N}$ (in particular, $E^+$ is a syntactic sugar for $EE^*$ and $E?$ is a syntactic sugar for $\epsilon + E$).

*Automata.* A *(nondeterministic) finite automaton (NFA)* over $\Sigma$ is a tuple $A = (Q, \Delta, I, F)$ where $Q$ is a finite set of *states*, $\Delta$ is a set of *transitions* of the form $q \text{-}\{a\}\!\!\rightarrow r$ with $q, r \in Q$ and $a \in \Sigma \cup \{\epsilon\}$, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A run of $A$ over a word $w \in \Sigma^*$ is a sequence $p_0 \text{-}\{a_1\}\!\!\rightarrow p_1 \text{-}\{a_2\}\!\!\rightarrow \ldots \text{-}\{a_n\}\!\!\rightarrow p_n$ where for all $1 \leq i \leq n$ it holds that $a_i \in \Sigma \cup \{\epsilon\}$, $p_{i-1} \text{-}\{a_i\}\!\!\rightarrow p_i \in \Delta$, and $w = a_1 \cdot a_2 \cdots a_n$. The run is *accepting* if $p_0 \in I$ and $p_n \in F$, and the language $L(A)$ of $A$ is the set of all words for which $A$ has an accepting run. A language $L$ is called *regular* if it is accepted by some NFA. Two NFAs with the same language are called *equivalent*. An automaton without $\epsilon$-transitions is called *$\epsilon$-free*. An automaton with each state belonging to some accepting run is *trimmed*. To concatenate languages of two NFAs $A = (Q, \Delta, I, F)$ and $A' = (Q', \Delta', I', F')$ with a symbol $\sharp$, we construct their *$\sharp$-concatenation* $A\sharp A' = (Q \uplus Q', \Delta \uplus \Delta' \uplus \{p \text{-}\{\sharp\}\!\!\rightarrow q \mid p \in F, q \in I'\}, I, F')$. 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 $a \in \Sigma$, $q \text{-}\{a\}\!\!\rightarrow r \in \Delta, q' \text{-}\{a\}\!\!\rightarrow r' \in \Delta'$. Finally, the *shuffle* of the two automata is the automaton $A \sqcup A' = (Q \times Q', \Delta^{\sqcup\!\sqcup}, I \times I', F \times F')$ where $(q, q') \text{-}\{a\}\!\!\rightarrow (r, r') \in \Delta^{\sqcup\!\sqcup}$ iff $a \in \Sigma$ and either $q' = r'$ and $q \text{-}\{a\}\!\!\rightarrow r \in \Delta$ or $q = r$ and $q' \text{-}\{a\}\!\!\rightarrow r' \in \Delta'$.

*Basic string constraints.* The basic string constraints are word equations, regular membership constraints, and arithmetic relations over lengths of strings. That is, given a set $\mathbb{X}$ of *string variables* (denoted $u, v, \ldots, z$), fixed for the rest of the paper, basic *string constraints* are boolean combinations of atomic constraints of the following three types: (i) a *word equation* of the form $s = t$ where $s$ and $t$ are *string terms*, i.e., words[1] from $\mathbb{X}^*$, (ii) an atomic *regular constraint* of the form $x \in L$, where $x \in \mathbb{X}$ and $L$ is a regular language, and (iii) an atomic *length constraint*: an atomic *linear integer arithmetic* (LIA) predicate containing length terms $|x|$ (for $x \in \mathbb{X}$) for variables[2]. An *equation/regular/length constraint* is a boolean combination of constraints of the respective type. Given a constraint, term, or generally any object $\alpha$, we use $\mathtt{Var}(\alpha)$ to denote the set of variables occurring in $\alpha$.

A *string assignment* is a map $\nu \colon \mathbb{X} \to \Sigma^*$. The assignment is a solution for a word equation $s = t$ if $\nu(s) = \nu(t)$ where $\nu(t')$ for a term $t' = x_1 \ldots x_n$ is defined as $\nu(x_1) \cdots \nu(x_n)$, it is a solution for an atomic regular constraint $x \in L$ if $\nu(x) \in L$, and it is a solution of an atomic length constraint $\varphi$ if the map $\{|x| \mapsto |\nu(x)| : x \in \mathtt{Var}(\varphi)\}$ is its (integer) solution. A solution of a boolean combination $\Phi$ of atomic constraints is then defined inductively as usual. We denote by $\nu \models \Phi$ that $\nu$ is a solution of $\Phi$ and we use $[\![\Phi]\!]$ to denote the set of all solutions of $\Phi$. We say that $\Phi$ is *satisfiable* if $[\![\Phi]\!] \neq \emptyset$.

A substitution is a map $\sigma \colon \mathbb{X} \to \mathbb{X}^*$ of string variables $\mathbb{X}$ to terms over variables. The image of a string constraint under $\sigma$, denoted as $\sigma(\Phi)$, is the constraint in which every occurrence of every $x \in \mathbb{X}$ is replaced by $\sigma(x)$. We use $\Phi[x/y]$ to represent constraint $\Phi$ in which every $x$ is replaced by $y$.

We will sometimes abuse notation and treat conjunctive constraints $\Phi$ as sets of their conjuncts, i.e., atomic string constraints $\varphi$, and write, for instance, $\varphi \in \Phi$. A *language assignment* is a function $\mathcal{R} \colon \mathbb{X} \to \mathcal{P}(\Sigma^*)$ assigning to every variable a regular language over $\Sigma$. Given a string term $t = x_1 \ldots x_n$, we define $\mathcal{R}(t) = \mathcal{R}(x_1) \cdots \mathcal{R}(x_n)$. We say that a language assignment $\mathcal{R}_1$ is a *refinement* of a language assignment $\mathcal{R}_2$ iff $\bigwedge_{x \in \mathbb{X}} \mathcal{R}_1(x) \subseteq \mathcal{R}_2(x)$, and $\mathcal{R}$ is called *feasible* iff $\bigwedge_{x \in \mathbb{X}} \mathcal{R}(x) \neq \emptyset$ and *infeasible* otherwise. A *normalized regular constraint* is a conjunction $\bigwedge_{x \in \mathbb{X}} x \in \mathcal{R}(x)$ where $\mathcal{R}$ is a language assignment (any regular constraint can be normalized by using the standard intersection/union/complement operations on the corresponding regular languages). We often treat a normalized regular constraint as its language assignment and the other way around.

[1] Note that terms with string literals from $\Sigma^*$, sometimes used in our examples, can be encoded by replacing each occurrence $o$ of a string literal $\ell$ by a *fresh* variable $x_o$ and a regular constraint $x_o \in \{\ell\}$.

[2] Generally, length constraints can contain length terms $|t|$ for $t = x_1 \cdots x_n$; we replace them with $|x_1| + \cdots + |x_n|$.

## 3 TWO METHODS FOR SOLVING EQUATIONS WITH REGULAR CONSTRAINTS

In this section, we explain the two basic algorithms for solving string constraints composed of word equations and regular constraints that we combine in our new algorithm later. The first one, Stabilization from [Blahoudek et al. 2023], is faster, but cannot be easily combined with reasoning about length constraints. The second one, called Align&Split, originates from the algorithms of [Abdulla et al. 2014] and is more expensive. On the other hand, it completely eliminates word equations from the constraint and leaves only regular constraints, which can then be used to obtain a LIA formula precisely characterizing lengths of their solutions.

We will first discuss the basics of the two algorithms, after which we describe our combined approach in Sec. 4.2. For the rest of this section, assume a string constraint $\Phi\colon \mathcal{E} \wedge \mathcal{R}$ where $\mathcal{E}$ is a conjunction of word equations and $\mathcal{R}$ is a normalized regular constraint.

### 3.1 Stabilization

The Stabilization algorithm of [Blahoudek et al. 2023] is based on Theorem 3.1 (Theorem 1 in [Blahoudek et al. 2023]) stated below that refers to *stability* of equations. Given a word equation $s = t$, we say that a language assignment $\mathcal{R}$ is *stable* for $s = t$ if $\mathcal{R}(s) = \mathcal{R}(t)$ and, for a set of word equations $\mathcal{E}$, we say that $\mathcal{R}$ is stable for $\mathcal{E}$ iff it is stable for every equation in $\mathcal{E}$.

Theorem 3.1. *Consider a string constraint $\Phi\colon \mathcal{E} \wedge \mathcal{R}$ where $\mathcal{E}$ is a conjunction of word equations and $\mathcal{R}$ is a normalized regular constraint. If $\mathcal{R}$ is stable for $\mathcal{E}$, then $\Phi$ is satisfiable.*

Proof. (Sketch of the proof in [Blahoudek et al. 2023]) The proof gives a backtracking procedure that constructs a model of a word equation by choosing *minimum-length* words from $\mathcal{R}$ for every variable. It is possible to encode a set of word equations into a single equation whose left-hand (right-hand) side is obtained by concatenating left-hand (right-hand) sides of all equations in $\mathcal{E}$, separated by delimiters. □

Theorem 3.1 essentially allows to test satisfiability of a string constraint while interpreting its variables in the domain of *languages* instead of the domain of *strings*. Equations are interpreted as equalities of concatenated languages of the variables on the respective sides, regular constraints as inclusions of the variable values in the specified regular languages. The stability of the constraint in the domain of languages corresponds to satisfiability in the domain of words.

Usually, the language assignment given by the input constraint $\mathcal{R}$ is not stable for $\mathcal{E}$ and needs to be *refined* first. For refinement, Stabilization looks at each equation $s = t$ as a pair of *inclusions/inclusion terms* $s \subseteq t$ and $t \subseteq s$ (remember that now we are looking at $s$ and $t$ as languages) that all need to be satisfied; the whole system of equations becomes a set of inclusion terms $\mathcal{I}$. We use $\mathcal{R} \models s \subseteq t$ to denote that $\mathcal{R}(s) \subseteq \mathcal{R}(t)$ and, given a string assignment $\nu$, we write $\nu \models s \subseteq t$ to denote $\nu \models s = t$ (in the algorithm, we only work with inclusion terms). A single refinement step of Stabilization takes one inclusion $s \subseteq t$ s.t. $\mathcal{R} \not\models s \subseteq t$ from $\mathcal{I}$ and refines $\mathcal{R}$ for every variable from $s$ into $\mathcal{R}'$ so that $\mathcal{R}'(s) \subseteq \mathcal{R}(t)$. Refinement is performed using the intersection $\mathcal{R}(s) \cap \mathcal{R}(t)$ as follows.

Consider the inclusion term $s \subseteq t$ and assume that $s = x_1 x_2 \dots x_n$ and $t = y_1 y_2 \dots y_m$. We first construct, for the left-hand side of the inclusion term, the language $\mathcal{R}(x_1)\triangleleft\mathcal{R}(x_2)\triangleleft \cdots \triangleleft\mathcal{R}(x_n)$, where $\triangleleft \notin \Sigma$ is a special symbol marking the boundaries between languages corresponding to occurrences of variables. Similarly, for the right-hand side of the inclusion term, we construct the language $\mathcal{R}(y_1)\mathcal{R}(y_2)\cdots\mathcal{R}(y_m)$ (note that no markers are present on the right-hand side). Both of these languages will be represented by NFAs. We then compute the product NFA *l-prod* of $\mathcal{R}(x_1)\triangleleft\mathcal{R}(x_2)\triangleleft \cdots \triangleleft\mathcal{R}(x_n)$ and $\mathcal{R}(y_1)\mathcal{R}(y_2)\cdots\mathcal{R}(y_m)$ where $\triangleleft$ is treated as an $\epsilon$-symbol (but preserved in the output). This can be seen formally as computing an NFA for the following language:

$$\textit{l-prod} = \left(\mathcal{R}(x_1)\triangleleft\mathcal{R}(x_2)\triangleleft \cdots \triangleleft\mathcal{R}(x_n)\right) \cap \left(\mathcal{R}(y_1)\mathcal{R}(y_2)\cdots\mathcal{R}(y_m) \sqcup \triangleleft^{n-1}\right) \tag{2}$$

*Example 3.2.* Assume a single equation $xuy = yzx$ and the regular constraint $\mathcal{R}\colon x \in d?(a+b)^+ \wedge y \in c + (ab)^+ \wedge u \in d \wedge z \in d?$ (with $a, b, c, d \in \Sigma$). The equation can be seen as a pair of inclusion terms $xuy \subseteq yzx$ and $yzx \subseteq xuy$. We start with the language assignment $\mathcal{R}$ from the initial regular constraint. Suppose we start processing the inclusion term $xuy \subseteq yzx$. For this, a refinement step of Stabilization will create the following pair of NFAs:

(i) The first one will be $A_{\mathsf{RHS}}$ obtained by concatenating the NFAs for the regular languages on the right-hand side of the inclusion term, i.e., $c + (ab)^+$ (for $y$), $d?$ (for $z$), and $d?(a+b)^+$ (for $x$). That is, the language of $A_{\mathsf{RHS}}$ will be $(c + (ab)^+)d?d?(a+b)^+$.

(ii) The second one will be $A_{\mathsf{LHS}}$ obtained by concatenating the NFAs for the regular languages on the left-hand side of the inclusion, i.e., $d?(a+b)^+$, $d$, and $c + (ab)^+$, respectively, using $\lhd$. In our case, the language of $A_{\mathsf{LHS}}$ will be $d?(a+b)^+ \lhd d \lhd (c + (ab)^+)$.

Then, the two obtained NFAs, i.e., $A_{\mathsf{LHS}}$ and $A_{\mathsf{RHS}}$, are intersected in an NFA *l-prod* by the product construction described above. The language of *l-prod* will be $(ab)^+ \lhd d \lhd (ab)^+$. □

After *l-prod* is computed, we extract from it new language assignments $\mathcal{R}'_i$ in four steps (there will potentially be a disjunction of several language assignments).

(1) First, we split *l-prod* into several NFAs $A_1, \ldots, A_k$ (without useless states) so that each $A_i$ contains exactly $n-1$ $\lhd$-transitions and the union of languages of $A_1, \ldots, A_k$ gives $L(\textit{l-prod})$. We call $A_1, \ldots, A_k$ *noodles*; each of them represents one possible way how the languages of variables occuring in $s$ change by the intersection with $t$.

(2) Second, for each NFA $A_i$, we split it into NFAs $A_{i,1}, \ldots, A_{i,n}$ by treating the source states of $\lhd$-transitions as final states and the target states of $\lhd$-transitions as initial states. As a result, for every occurrence of a variable $x_j$ in the left-hand side $s = x_1 \ldots x_n$, there will be a corresponding NFA $A_{i,j}$.

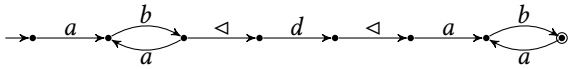
(3) Third, for a sequence of NFAs $A_{i,1}, \ldots, A_{i,n}$ and every variable $x \in \mathtt{Var}(s)$ (there might, in general, be multiple occurrences of $x$ in $s$), we take the intersection of all NFAs that correspond to $x$ and obtain $A^i_x = \bigcap\{A_{i,j} \mid x_j = x\}$.

(4) Finally, for the given $A_i$, we obtain $\mathcal{R}'_i$ from $\mathcal{R}$ by changing the NFA for the language of every variable $x \in \mathtt{Var}(s)$ to $A^i_x$.

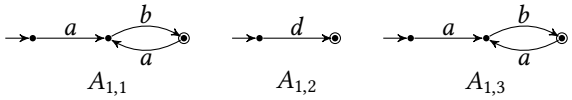
We have obtained $k$ new language assignments $\mathcal{R}'_1, \ldots, \mathcal{R}'_k$ (which can, again, be interpreted as regular language constraints) such that they together represent the refinement of $\mathcal{R}$. This new set of language assignments will be called $\mathtt{L\text{-}noodlify}(s \subseteq t, \mathcal{R}) = \{\mathcal{R}'_1, \ldots, \mathcal{R}'_k\}$. The following lemma states that the refinement operation preserves the solutions of a string constraint.

Lemma 3.3. *$v \models s \subseteq t \wedge \mathcal{R}$ iff $v \models s \subseteq t \wedge \mathcal{R}'$ for some $\mathcal{R}' \in \mathtt{L\text{-}noodlify}(s \subseteq t, \mathcal{R})$.*

*Example 3.4.* Continuing in refining the inclusion $xuy \subseteq yzx$ from Example 3.2, the obtained language $(ab)^+ \lhd d \lhd (ab)^+$ can be represented by the following NFA *l-prod*:




The NFA is a *noodle* $A_1$ itself, so it does not need to be split into noodles (we would need to perform splitting in the case that, e.g., the target state of the $d$-transition had several outgoing $\lhd$-transitions). Next, we need to split $A_1$ into NFAs for each occurrence of a variable on the left-hand side of the inclusion (marked by the $\lhd$-transitions). We obtain the following three NFAs:



Finally, we refine $\mathcal{R}$ into $\mathcal{R}'$: both variables $x$ and $y$ (corresponding to $A_{1,1}$ and $A_{1,3}$ respectively) will be assigned the language $(ab)^+$ in $\mathcal{R}'$ and the language of $u$ (corresponding to $A_{1,2}$) will still be $d$ (we do not touch the language of $z$ since $z$ does not occur on the left-hand side of the inclusion). □

The top-level algorithm of Stabilization starts with a language assignment $\mathcal{R}$ from the input constraint $\Phi\colon \mathcal{E} \wedge \mathcal{R}$ and keeps refining $\mathcal{R}$ using unsatisfied inclusion terms from $\mathcal{I}$, potentially spawning multiple branches of the computational tree. The procedure terminates when either (i) *one branch* of the computational tree stabilizes with a *feasible* language assignment (for a satisfiable constraint) or (ii) *all branches* of the computational tree stabilize with *infeasible* language assignments (for an unsatisfiable constraint).

*Example 3.5.* Continuing in Example 3.4, we can proceed to another refinement using, e.g., the inclusion term $yzx \subseteq xuy$), which will change the language of $z$ from $d?$ to $d$ (the rest will stay the same). After that, further refinements will not change the language assignment any more—the constraint is stable. A possible solution might be, e.g., $x \mapsto ab$, $y \mapsto ab$, $u \mapsto d$, and $z \mapsto d$. □

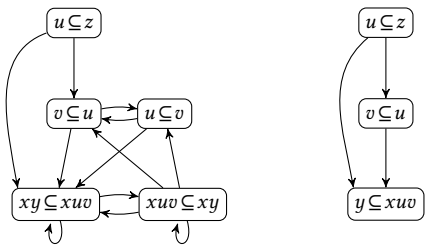
The presented procedure is correct, but in practice often inefficient. In particular, the termination condition (stabilization on the set of all inclusion terms) is in many cases too restrictive and can be weakened. For this, we work with the *inclusion graph* of a set of word equations $\mathcal{E}$.

An inclusion graph for $\mathcal{E}$ is given as a set of inclusion terms $\mathcal{I}$ where the edge relation is induced by $\mathcal{I}$ as follows: there is an edge $s \subseteq t$ to $s' \subseteq t'$ iff $s$ and $t'$ share a variable. For $\mathcal{I}$ to induce an inclusion graph, the following three conditions need to hold:

(IG1) For each $s = t$ in $\mathcal{E}$, $s \subseteq t \in \mathcal{I}$ or $t \subseteq s \in \mathcal{I}$.
(IG2) If $s \subseteq t \in \mathcal{I}$ s.t. $t$ has a variable with multiple occurrences on right-hand sides of inclusions in $\mathcal{I}$, then also $t \subseteq s \in \mathcal{I}$.[3]
(IG3) If $s \subseteq t \in \mathcal{I}$ lies on a cycle in the graph consisting of edges induced by $\mathcal{I}$, then also $t \subseteq s \in \mathcal{I}$.

We use $\mathcal{I}(s \subseteq t)$ for the set of successors of a node $s \subseteq t$ w.r.t. the edge relation induced by $\mathcal{I}$.

Given a set of word equations $\mathcal{E}$, there might exist several sets $\mathcal{I}$ that induce an inclusion graph using the definition above (e.g., the conditions are trivially satisfied by taking for every word equation both corresponding inclusion terms). We, however, try to take a set of the smallest size that satisfies the conditions and is *acyclic* (or at least contains as few cycles as possible; see [Blahoudek et al. 2023] for an algorithm for selecting a suitable $\mathcal{I}$). Examples of inclusion graphs are shown in Fig. 1.




(a) Inclusion graph for $\mathcal{E}_1$ (b) Inclusion graph for $\mathcal{E}_2$

Fig. 1. Examples of inclusion graphs. (a) A cyclic inclusion graph for $\mathcal{E}_1\colon z = u \wedge u = v \wedge xy = xuv$. $\mathcal{E}_1$ does not have an acyclic inclusion graph. (b) An acyclic inclusion graph for $\mathcal{E}_2\colon z = u \wedge u = v \wedge y = xuv$. Both inclusion graphs in (a) and (b) are induced by a set of inclusions of the smallest size.

We say that a language assignment $\mathcal{R}$ is *stable for a set of inclusions* $\mathcal{I}$ if it satisfies its every inclusion. The following theorem (Theorem 3 from [Blahoudek et al. 2023]) states that it is sufficient to stabilize a language assignment w.r.t. a set of inclusions that induce an inclusion graph. We say that a string constraint $\mathcal{E}$ is *chain-free* if there exists at least one acyclic inclusion graph for $\mathcal{E}$.[4]

[3] E.g., $\mathcal{I} = \{xx \subseteq yy\}$ does not induce an inclusion graph (it can be fixed by adding $yy \subseteq xx$). Similarly for $\{u \subseteq x, v \subseteq x\}$ (it can be fixed by either adding $x \subseteq u$ and $x \subseteq v$ or by inverting all inclusions, i.e., $\mathcal{I}' = \{x \subseteq u, x \subseteq v\}$).
[4] The equivalence of this definition with the original definition of chain-free constraints from [Abdulla et al. 2019] was shown in [Blahoudek et al. 2023].

Theorem 3.6. *A string constraint* $\Phi\colon \mathcal{E} \wedge \mathcal{R}$ *is satisfiable iff there exists a set of inclusions* $\mathcal{I}$ *inducing an inclusion graph for* $\mathcal{E}$ *such that* $\mathcal{R}$ *is stable for* $\mathcal{I}$.

Given $\mathcal{I}$, edges in the induced inclusion graph tell us the sufficient order of refinements that need to be performed. If the inclusion graph is acyclic, then the algorithm is guaranteed to terminate (cf. Theorems 6 and 7 in [Blahoudek et al. 2023]).

### 3.2 Align&Split

Align&Split is an algorithm from [Abdulla et al. 2014] that transforms a system $\Phi$ of word equations and regular constraints into a purely regular constraint. From this regular constraint, it is then possible to extract a precise LIA formula characterizing lengths of strings in satisfying assignments to the variables of $\Phi$ by standard means (in particular, we use the lasso automaton construction, used, e.g., in [Abdulla et al. 2014; Berzish et al. 2023, 2021; Chen et al. 2019]; an alternative is the computation of the Parikh image of the languages [Esparza 1997; Parikh 1966]). Equation alignment and automata splitting introduced in [Abdulla et al. 2014] transform the constraint into a purely regular one. In a nutshell, when Align&Split is called on a conjunction $\Phi\colon \mathcal{E} \wedge \mathcal{R}$ of a word equation constraint $\mathcal{E}$ and a regular constraint $\mathcal{R}$, it performs the following three steps: (1) It transforms $\mathcal{E}$ into the so-called *solved form* [Ganesh et al. 2012], a disjunction of conjunctions $\bigwedge_{x\in\mathtt{Var}(\mathcal{E})} x = t_x$ where the terms $t_x$ are concatenations of fresh variables (not in $\Phi$), (2) each $x \in \mathtt{Var}(\mathcal{E})$ is in $\mathcal{R}$ substituted by $t_x$ and the equations are removed. The substitution $\sigma = \{x \mapsto t_x \mid x \in \mathtt{Var}(\mathcal{E})\}$ can be understood as an *alignment* of the equations that specifies the relative positions of borders in their left and right-hand side variables (for instance, one possible disjunct of the solved form of $xy = uz$ would be $x = v_1v_2 \wedge y = v_3 \wedge u = v_1 \wedge z = v_2v_3$, corresponding to aligning $u$ as a prefix of $x$). The substitution transforms atomic regular constraints $x \in L_x$ into regular constraints over terms: $x_1 \dots x_n \in L$. (3) The last step, *automata splitting*, transforms these term-regular constraints back into ordinary regular constraints as follows: each term-regular constraint is transformed into a disjunction $\bigvee_{q_0,\dots,q_n\in Q, q_0\in I, q_n\in F} \bigwedge_{i=0}^{n-1} x_i \in L(A_{q_i}^{q_{i+1}})$, where $A = (Q, \delta, I, F)$ is an NFA accepting $L_x$ and $A_r^q$ is the NFA $(Q, \delta, \{q\}, \{r\})$ (intuitively, $x_i$ is given the language between states $q_i$ and $q_{i+1}$ of $A$).

The alignment and splitting combines two expensive case splits, computing the full solved form with all alignments and then splitting over all $n$-tuples of automata states. In Sec. 4.2, we will therefore propose a fine grained integration of steps from Stabilization and Align&Split, where the latter are used only when absolutely necessary. For that, let us first formulate a version of Align&Split that is more compatible with Stabilization.

*Align&Split step.* The basic Align&Split step is similar to a refinement step for one inclusion of Stabilization. In the presentation here, on top of the functionality of Stabilization, it also makes the alignment of the borders between left and right-hand side variables explicit in the product of the left and the right-hand side languages (in Eq. (2), we only tracked borders of left-hand side variables), and applies the alignment substitution that fixes the alignment of the inclusion for the rest of the computation. Over several Align&Split steps, these substitutions converge to an alignment for all inclusions.

In particular, for $s \subseteq t$ with $s = x_1 \dots x_n$ and $t = y_1 \dots y_m$, we compute an NFA for the intersection of languages of $s$ and $t$ with marked borders between left-hand as well as right-hand side variables:

$$\mathit{lr\text{-}prod} = \left(\mathcal{R}(x_1)\triangleleft\mathcal{R}(x_2)\triangleleft\cdots\triangleleft\mathcal{R}(x_n) \sqcup \triangleright^{m-1}\right) \cap \left(\mathcal{R}(y_1)\triangleright\mathcal{R}(y_2)\triangleright\cdots\triangleright\mathcal{R}(y_m) \sqcup \triangleleft^{n-1}\right) \quad (3)$$

The product *lr-prod* is then decomposed into a set of noodles (using the vocabulary of Stabilization), automata with languages

$$L_1 \mid_1 L_2 \mid_2 \cdots \mid_{n+m} L_{n+m} \quad \text{where each } \mid_i \text{ is } \triangleleft \text{ or } \triangleright \text{ and each } L_i \text{ is in } \Sigma^*$$

such that union of the languages of the noodles is the original language $L(\mathit{lr\text{-}prod})$.

*Example 3.7.* As in Example 3.2, we again start processing the inclusion term $xuy \subseteq yzx$ with $x \in d?(a+b)^+ \wedge y \in c + (ab)^+ \wedge u \in d \wedge z \in d?$. Align&Split creates the following pair of NFAs:

(i) $A_{\mathsf{RHS}}$ is obtained by concatenating the right-hand side languages, divided by the right-hand side separator symbol $\triangleright$. The language of $A_{\mathsf{RHS}}$ is $(c + (ab)^+)\triangleright d?\triangleright d?(a+b)^+$.

(ii) $A_{\mathsf{LHS}}$ is obtained by concatenating the left-hand side languages, divided by the left-hand side separator symbol $\triangleleft$. The language of $A_{\mathsf{LHS}}$ is $d?(a+b)^+\triangleleft d\triangleleft(c+(ab)^+)$.

The $A_{\mathsf{LHS}}$ and $A_{\mathsf{RHS}}$ are intersected in NFA *lr-prod* by a specialized product construction that treats the separator symbols $\triangleleft$ and $\triangleright$ as $\epsilon$ and remembers their position. *lr-prod* accepts the union of the two languages that correspond to $z$ on the right-hand side of the inclusion being either $d$ or $\epsilon$:

$$\left((ab)^+\triangleleft\triangleright d\triangleleft\triangleright(ab)^+\right) \cup \left((ab)^+\triangleleft\triangleright\triangleright d\triangleleft(ab)^+\right) \tag{4}$$

The two languages then become two noodles. Recall that in Stabilization, only the left-hand side separators $\triangleleft$ were used and only one noodle was created, illustrating that Align&Split generates larger case splits. □

From each noodle, we infer an alignment substitution $\sigma$ and a refinement $\mathcal{R}'$ of languages of variables. Intuitively, each occurrence of a variable will be split into several segments by the separators $\triangleleft$ and $\triangleright$. Each of the segments will be represented by a fresh variable and the original variables will be substituted by the sequences of the fresh variables.

Consider the sequence

$$v_1 \mid_1 v_2 \mid_2 \cdots \mid_{n+m} v_{n+m} \tag{5}$$

where the $v_i$'s are fresh variables, called *alignment variables* with $\mathcal{R}'(v_i) = L_i$ (for original variables $x$, $\mathcal{R}'(x) = \mathcal{R}(x)$). For every index $j$ of a left-hand side position of $s$, with $1 \leq j \leq n$, let $\mathtt{s}_j$ denote the sequence $v_k v_{k+1} \ldots v_l$ of the fresh variables in between the $j$-th and the $(j+1)$-th occurrence of $\triangleleft$ (corresponding to $x_j$ on the left-hand side of the inclusion), and for $1 \leq j \leq m$, let $\mathtt{t}_j$ be the sequence of the fresh variables $v_k v_{k+1} \ldots v_l$ in between the $j$-th and the $(j+1)$-th occurrence of $\triangleright$ (corresponding to $y_j$ on the right-hand side of the inclusion). As an optimization, segments with languages equivalent to $\epsilon$ can be omitted (the corresponding fresh variables are removed from Eq. (5)). Then, the set *Align* of *alignment inclusions* will consist of, for every index $1 \leq j \leq m$ on the right-hand side, the inclusion $\mathtt{s}_j \subseteq y_j$, and for every index $1 \leq j \leq n$ on the left-hand side, the inclusion $x_j \subseteq \mathtt{t}_j$.

*Example 3.8.* Let us continue processing the inclusion $xuy \subseteq yzx$ from Example 3.7 and consider the noodle $(ab)^+ \triangleleft\triangleright \triangleright d \triangleleft (ab)^+$ obtained from the right-hand side operand of Eq. (4). This noodle corresponds to the following alignment of variables:

$$\overbrace{\underbrace{v_1}_{y}\,\underbrace{v_2\,v_3}_{z}\,\underbrace{v_4\,v_5}_{x}}^{x\quad u\quad y} \quad\rightsquigarrow\quad \overbrace{\underbrace{v_1}_{y}\,\underbrace{\epsilon}_{z}\,\underbrace{\epsilon\,v_4}_{x}\,\underbrace{v_5}_{y}}^{x\quad u\quad y}$$

Note that $v_2$ and $v_3$ were removed because the subsequence $\ldots\triangleleft\triangleright\triangleright\ldots$ in the noodle assigns them to $\epsilon$. Therefore, we obtain the set $Align = \{x \subseteq v_1, u \subseteq v_4, y \subseteq v_5, v_1 \subseteq y, \epsilon \subseteq z, v_4 v_5 \subseteq x\}$. □

The new string assignment together with the alignment inclusions preserve solutions in the sense formalised in the following lemma. Let $\mathtt{LR\text{-}noodlify}(s \subseteq t, \mathcal{R})$ be the set of pairs $\left\{(Align_i, \mathcal{R}_i)\right\}_{i=1}^{k}$.

Lemma 3.9. *$v \models s \subseteq t \wedge \mathcal{R}$ iff $v' \models Align_i \wedge \mathcal{R}_i$ for some $(Align_i, \mathcal{R}_i) \in \mathtt{LR\text{-}noodlify}(s \subseteq t, \mathcal{R})$ and some $v'$ extending $v$ to $\mathtt{Var}(Align_i \wedge \mathcal{R}_i)$.*

Now, for every variable $x \in \mathtt{Var}(s \subseteq t)$, we choose one alignment inclusion in *Align*, called a *substitution inclusion*, of the form either $x \subseteq t_x$ or $t_x \subseteq x$ (mind that $x$ can occur in $s \subseteq t$ multiple

times, leading to multiple alignment inclusions with $x$). The selected substitution inclusions are removed from *Align*, resulting in the set *Align*$'$, and we define the *alignment substitution* based on them as $\sigma = \{x \mapsto t_x \mid x \in \mathtt{Var}(s = t)\}$. The original inclusion is replaced by *Align*$'$, and the alignment substitution is applied on the entire string constraint $\Phi$. The Align&Split steps continue until all inclusions are eliminated (transformed to trivial inclusions and removed).

*Example 3.10.* Continuing in Example 3.8, we can take $\sigma = \{y \mapsto v_1, z \mapsto \epsilon, u \mapsto v_4, x \mapsto v_4 v_5\}$ as an alignment substitution and then $Align' = \{x \subseteq v_1, y \subseteq v_5\}$. After applying the substitution and removing trivial inclusions (such as $v_1 \subseteq v_1$), we obtain from *Align*$'$ inclusions $\{v_4 v_5 \subseteq v_1, v_1 \subseteq v_5\}$ and language assignment $\{v_1 \mapsto (ab)^+, v_4 \mapsto d, v_5 \mapsto (ab)^+\}$. A follow-up Align&Split applied on $v_4 v_5 \subseteq v_1$ would terminate with an empty intersection of the left-hand and the right-hand side languages ($d(ab)^+ \cap (ab)^+ = \emptyset$). This means that the right-hand side operand of Eq. (4), which we chose for the presentation, does not lead to a solution.

If we continued from the left-hand side operand of Eq. (4) with Align&Split steps, we would obtain $Align = \{x \subseteq v_1, u \subseteq v_3, y \subseteq v_5, v_1 \subseteq y, v_3 \subseteq z, v_5 \subseteq x\}$. We can select, e.g., the alignment substitution $\sigma = \{x \mapsto v_1, u \mapsto v_3, y \mapsto v_5, z \mapsto v_3\}$, leaving $Align' = \{v_1 \subseteq y, v_5 \subseteq x\}$. From this, after applying the substitution, we would obtain inclusions $\{v_1 \subseteq v_5, v_5 \subseteq v_1\}$ and language assignment $\mathcal{R}' = \{v_1 \mapsto (ab)^+, v_3 \mapsto d, v_5 \mapsto (ab)^+\}$. After one more Align&Split step, we would terminate with no equations and language assignment $\mathcal{R}'$. □

As mentioned at the beginning of this section, since Align&Split returns a purely regular constraint, it is possible to return a formula that precisely characterises the lengths of all solutions of the original constraint. We can compute the arithmetic formula expressing the lengths of solution assignments for every variable and take their conjunction (by computing the Parikh images of the automata [Esparza 1997] or the lasso automaton construction used, e.g., in [Abdulla et al. 2014]).

Align&Split is costlier than Stabilization because (1) it introduces more variables, creating larger equations and (2) it generates noodles with explicit boundaries of variables on both sides. The number of noodles generated by splitting is thus higher. In the following section, we therefore propose a combined algorithm that uses splitting only when necessary and uses Stabilization refinement steps instead whenever possible.

## 4 SOLVING BASIC STRING CONSTRAINTS

We will now explain our string solving algorithm that handles basic string constraints: equations, regular, and length constraints. Concretely, it takes a conjunction $\Phi\colon \mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L}$ on the input, where $\mathcal{E}$ is a conjunction of equations, $\mathcal{R}$ is a normalized regular constraint, and $\mathcal{L}$ is a length constraint. The algorithm reduces the word equations and regular constraints to a LIA formula, which is then solved (together with $\mathcal{L}$) by a LIA solver using the DPLL($T$) framework for combining theories.

### 4.1 The Overall Structure of the Basic String Solver

Essentially, the string solver generates the strongest length constraint $\psi$ on the variables of $\mathcal{L}$ implied by $\mathcal{E} \wedge \mathcal{R}$, and the LIA solver solves it in conjunction with $\mathcal{L}$. The string solver for basic constraints, discussed in detail in Sec. 4.2, takes on the input constraints $\mathcal{E}$, $\mathcal{R}$, and a set of *length variables* $\mathtt{Len} = \mathtt{Var}(\mathcal{L})$. In order to be able to generate $\psi$ from the input, the solver first "solves" the equations and regular constraints by transforming them to the disjunction

$$\bigvee_{i=1}^{n} \mathcal{E}_i \wedge \mathcal{R}_i, \tag{6}$$

where each $\mathcal{E}_i$ is a conjunction of equations and each $\mathcal{R}_i$ is a normalised regular constraint. Every disjunct $\mathcal{E}_i \wedge \mathcal{R}_i$ is paired with an alignment substitution $\sigma_i$, which substitutes variables of $\mathcal{E}$

by fresh variables appearing in $\mathcal{E}_i \wedge \mathcal{R}_i$. We call $\langle \mathcal{E}_i, \mathcal{R}_i, \sigma_i \rangle$ a *candidate solution triple*. The set of candidate solution triples is required to satisfy the following two properties:

(1) Triples in the set are together *equivalent to* $\mathcal{E} \wedge \mathcal{R}$ *modulo the substitutions*, i.e.,

$$[\![\mathcal{E} \wedge \mathcal{R}]\!] = \bigcup_{i=1}^{n} \sigma_i^\bullet([\![\mathcal{E}_i \wedge \mathcal{R}_i]\!]), \tag{7}$$

where for a substitution $\sigma$ and an assignment $\nu$, $\sigma^\bullet$ denotes the assignment $\sigma^\bullet(\nu) = \{x \mapsto \nu(v_1) \cdots \nu(v_\ell) \mid x \in \mathbb{X}, \sigma(x) = v_1 \dots v_\ell\}$ ($\sigma^\bullet$ is extended to sets of assignments as usual).

(2) In each triple $\langle \mathcal{E}_i, \mathcal{R}_i, \sigma_i \rangle$, the word equation constraint $\mathcal{E}_i$ *does not restrict length variables*, i.e., variables from the set $\mathtt{Len}_i = \mathtt{Var}(\sigma_i(\mathcal{L}))$. In other words, in any solution of $\mathcal{E}_i$, the value of any length variable $x$ can be replaced by an arbitrary value from $\mathcal{R}_i(x)$ and the resulting assignment will still be a solution of $\mathcal{E}_i$. Formally, it holds when the formula $\forall \mathtt{Len}_i \exists \overline{\mathtt{Len}_i} : \mathcal{E}_i$ is valid, where $\overline{\mathtt{Len}_i} = \mathtt{Var}(\mathcal{E}_i) \setminus \mathtt{Len}_i$ are the non-length variables in $\mathcal{E}_i$ (note that, generally speaking, the variables in each $\mathcal{E}_i$ will be different from those in $\mathcal{E}$; in particular some non-length variables will be split using the alignment rule and their parts substituted by length variables).

For each triple $\langle \mathcal{E}_i, \mathcal{R}_i, \sigma_i \rangle$, we generate for each variable $x$ of $\sigma_i(\mathcal{L})$ the precise length characterisation of $\mathcal{R}_i(x)$. This is a LIA formula $\psi_i^x$ with only one free variable $x$ (used in the length term $|x|$) such that it has a solution $\ell_x$ if and only if $\mathcal{R}_i(x)$ contains a word of the length $\ell_x$. The LIA formula $\psi_i^x$ is generated from $\mathcal{R}_i(x)$ by well-known automata constructions, such as the Parikh image construction [Esparza 1997; Parikh 1966], or from one-letter deterministic lasso automaton construction, see, e.g., [Abdulla et al. 2014] (we use the latter as it seems faster and generates simpler formulae). For the $i$-th candidate solution triple, the string solver returns the formula

$$\mathcal{L}_i : \bigwedge_{x \in \mathtt{Len}_i} \psi_i^x \wedge \bigwedge_{x \in \mathtt{Len}, \sigma_i(x) = x_1 \cdots x_m} |x| = |x_1| + \cdots + |x_m|. \tag{8}$$

The first conjunction describes the lengths derived from $\mathcal{R}_i$ and the second conjunct links the lengths of the original variables with the new variables substituting them. Note that $\mathcal{L}_i \wedge \mathcal{L}$ is satisfiable if and only if $\mathcal{E}_i \wedge \mathcal{R}_i \wedge \sigma_i(\mathcal{L})$ is satisfiable. A naive implementation of this translation would return the entire conjunction $\bigwedge_{i=1}^{n} \mathcal{L}_i$ at once. In Sec. 7, we discuss a way of generating the conjuncts lazily on demand.

### 4.2 Combined Algorithm for Solving Equations and Regular Constraints

The core of our technical contribution is the string solving algorithm. It takes $\mathcal{E}$, $\mathcal{R}$, and the set of length variables $\mathtt{Len} = \mathtt{Var}(\mathcal{L})$ on the input. The set of length variables is used to keep track of variables directly involved in $\mathcal{L}$ or possibly influencing their value. The algorithm works with equations represented again as a set of inclusions $\mathcal{I}$, which are initially obtained from equations $\mathcal{E}$ to satisfy the requirements for an inclusion graph in Sec. 3.1. The algorithm builds a proof tree by repeatedly applying the proof rules below. The nodes of the proof tree are tuples of the form $(\mathcal{I}, \mathcal{R}, W, \sigma, \mathtt{Len})$ consisting of inclusions $\mathcal{I}$, language assignment $\mathcal{R} : \mathtt{Var}(\mathcal{I}) \to \mathcal{P}(\Sigma^*)$, a worklist of inclusions $W \subseteq \mathcal{I}$, a partial alignment substitution $\sigma$, and a set of length variables $\mathtt{Len} \subseteq \mathtt{Var}(\mathcal{I})$.

Essentially, on every branch of the proof tree, the inclusions and languages of variables are updated by a combination of Stabilization and Align&Split in order to achieve stability of inclusions not relevant to length constraints and elimination of inclusions that are relevant. Every branch of the proof tree uses its worklist $W$ to track its progress in stabilization of its inclusions.[5] For the worklist, we use the notation $s \subseteq t :: W$ to denote that $s \subseteq t$ is at the top of the worklist (whose tail is $W$) and the notation $S :: W$ for a set $S$ to denote a worklist obtained by pushing all elements

[5]Strictly speaking, the worklist is used for efficiency purposes; its omission does not affect soundness. A simple alternative would be to use any rule enabled by its side condition.

from $S$ on the top of $W$ in an arbitrary order. Applying substitution $\sigma$ on a worklist substitutes all occurrences of the variables within elements of the worklist.

A leaf $(\mathcal{I}, \mathcal{R}, \emptyset, \sigma, \mathtt{Len})$ with the empty worklist and a feasible language assignment $\mathcal{R}$ will be guaranteed to have $\mathcal{R}$ stable for $\mathcal{I}$ and so denote satisfiability of the input constraint. The candidate solution triple of Eq. (6) for the given branch is obtained as $\langle \mathcal{E}_{\mathcal{I}}, \mathcal{R}, \sigma \rangle$ where $\mathcal{E}_{\mathcal{I}}$ is obtained from $\mathcal{I}$ by replacing inclusion signs by equality. On the other hand, all leafs of a complete proof tree having an infeasible language assignment $\mathcal{R}$ will denote unsatisfiability of the constraint.

We write the inference rules in the form $\textsc{Name} : \frac{P}{\{C_i\}_{i=1}^k}\ \mathit{cond}$, where Name is the rule's *name*, $P$ is the *premise*, *cond* is a *side condition* on $P$ necessary for the rule to be applicable, and $\{C_i\}_{i=1}^k$ is a *set of conclusions* of the rule. Each application consumes a premise and produces a set of conclusions, each conclusion spawning a new branch of the tree.

To simplify the proof rules, we will take advantage of that only lenght variables need to be completely eliminated from equations by means of Align&Split, but not the non-lenght variables. Hence, before the algorithm starts, we substitute every occurrence of a maximal sequence $z_1 \cdots z_n$ of non-length variables in all inclusions in $\mathcal{I}$ by a fresh (non-length) variable $y$ and add into $\mathcal{I}$ the inclusion $y \subseteq z_1 \cdots z_n$ (if the sequence occurred on the left-hand side) or $z_1 \cdots z_n \subseteq y$ (if the sequence occurred on the right-hand side). This establishes the following invariant, which will be preserved by the proof rules:

If the right-hand side of an inclusion in $\mathcal{I}$ contains a length variable,
then neither of its sides contains two non-length variables next to each other. (9)

Note that after replacing sequences of non-lenght variables in an equaion, `LR-noodlify` of the Align&Split algorithm produces less and shorter noodles (we will also comment on the role of the invariant from Eq. (9) where it is relevant). The output of the preprocessing satisfied the invariant because (i) all non-trivial sequences of non-length variables in the original inclusions were substituted by a new variable and (ii) the newly introduced inclusions do not contain any length variables.

*Stabilization rules.* We now formulate proof rules that combine Stabilization and Align&Split to solve equations in a way minimizing the use of Align&Split steps but still creating a constraint where the length constraint is *separated* from equations and restricted only by regular constraints. We will first describe the more straightforward rules corresponding to the steps in Stabilization, which do not deal with length variables.

The first rule, Refine, is easily explained as one refinement step of Stabilization. It is applied on inclusions that are not yet satisfied and have no length variables on the right-hand side. One proof tree child is generated for each of the $k$ noodles obtained by decomposing *l-prod* (cf. Sec. 3.1), each child with a different update of the regular constraint. The inclusion is in the worklist replaced by its successors. Refinement neither aligns equation sides nor substitutes variables, hence $\mathcal{I}$ and $\sigma$ remain unchanged.

$$\textsc{Refine} : \frac{(\ \mathcal{I},\ s \subseteq t :: W,\ \mathcal{R},\ \sigma,\ \mathtt{Len}\ )}{\{(\ \mathcal{I},\ \mathcal{I}(s \subseteq t) :: W,\ \mathcal{R}'_i,\ \sigma,\ \mathtt{Len}\ )\}_{i=1}^k}\ \varphi_{\textsc{Refine}}$$

$$\text{where } \varphi_{\textsc{Refine}} \overset{\text{def.}}{\Leftrightarrow} \mathtt{Len} \cap \mathtt{Var}(t) = \emptyset \wedge \mathcal{R} \not\models s \subseteq t \wedge \mathtt{L\text{-}noodlify}(s \subseteq t, \mathcal{R}) = \{\mathcal{R}'_i\}_1^k.$$

Rule Skip simply allows to skip processing of an inclusion that is already stable, i.e., satisfied in the current language assignment $\mathcal{R}$, if it does not contain length variables on its right-hand side.

$$\textsc{Skip} : \frac{(\ \mathcal{I},\ s \subseteq t :: W,\ \mathcal{R},\ \sigma,\ \mathtt{Len}\ )}{\{(\ \mathcal{I},\ W,\ \mathcal{R},\ \sigma,\ \mathtt{Len}\ )\}}\ \mathtt{Len} \cap \mathtt{Var}(t) = \emptyset \wedge \mathcal{R} \models s \subseteq t$$

Rule Stable applies when the worklist is empty and the language assignment is feasible. By the construction of the rules, the inclusions are stable w.r.t. the language assignment, and the current inclusions, assignment, and substitution represent one candidate solution triple.

$$\textsc{Stable}: \frac{(\quad \mathcal{I},\quad \emptyset,\quad \mathcal{R},\quad \sigma,\quad \texttt{Len}\quad)}{\texttt{SAT}(\langle \mathcal{I}, \mathcal{R}, \sigma\rangle)}\ \mathcal{R} \text{ is feasible}$$

Rule Infeasible applies when the assignment is infeasible, and it terminates the proof tree branch as unsuccessful.

$$\textsc{Infeasible}: \frac{(\quad \mathcal{I},\quad W,\quad \mathcal{R},\quad \sigma,\quad \texttt{Len}\quad)}{\texttt{UNSAT}}\ \mathcal{R} \text{ is infeasible}$$

*Combined rules.* The next set of rules is used for inclusions with length variables. The rules are only used if the stabilization rules above do not apply. Note that the rules are not just a reformulation of the Align&Split algorithm of Sec. 3.2 but, instead, they are a fine-grained combination of Align&Split and Stabilization that introduces as few new alignment variables as possible.

Rule AlignSplit encapsulates the first part of the Align&Split step. It aligns the left-hand and right-hand side variables, generating the set *Align* of alignment inclusions. Rule AlignSplit is the purpose of intruducing Eq. (9): it reduces sequences of non-length variables into a single variable, which in turn reduces the number of newly introduced variables (corresponding to the "length" of noodles) and the number of noodles produced by $\texttt{LR-noodlify}(s \subseteq t, \mathcal{R})$ used within AlignSplit. The second part of Align&Split, which turns alignment equations into substitutions, is then implemented by rules LSubst, LSubstLen, and RSubstLen.

$$\textsc{AlignSplit}: \frac{(\quad \mathcal{I} \uplus \{s \subseteq t\},\quad s \subseteq t{::}W,\quad \mathcal{R},\quad \sigma,\quad \texttt{Len}\quad)}{\left\{(\quad \mathcal{I} \cup \mathit{Align}_i,\quad \mathit{Align}_i{::}W,\quad \mathcal{R}'_i,\quad \sigma,\quad \texttt{Len}\quad)\right\}_{i=1}^{k}}\ \texttt{LR-noodlify}(s \subseteq t, \mathcal{R}) = \left\{(\mathcal{R}'_i, \mathit{Align}_i)\right\}_{i=1}^{k}$$

When one side of the processed inclusion contains a single variable $x$, then $x$ is, after refinement of the languages into noodles, substituted by the term on the other side. Rule LSubst is applied when the $x$ is on the left-hand side and it is not a length variable.

$$\textsc{LSubst}: \frac{(\quad \mathcal{I} \uplus \{x \subseteq t\},\quad x \subseteq t{::}W,\quad \mathcal{R},\quad \sigma,\quad \texttt{Len}\quad)}{\left\{(\quad \sigma'_i(\mathcal{I} \cup \mathit{Align}),\quad \sigma'_i(\mathit{Align}{::}W),\quad \mathcal{R}'_i,\quad \sigma'_i \cup \sigma,\quad \texttt{Len}\quad)\right\}_{i=1}^{k}}\ \varphi_{\textsc{LSubst}}$$

where $\varphi_{\textsc{LSubst}} \overset{\text{def.}}{\Leftrightarrow} x \notin \texttt{Len} \wedge \texttt{LR-noodlify}(x \subseteq t, \mathcal{R}) = \left\{(\mathcal{R}'_i, \mathit{Align} \uplus \{t'_i \subseteq x\})\right\}_{i=1}^{k} \wedge \bigwedge_{i=1}^{k} \sigma'_i = \{x \mapsto t'_i\}$.

Rule LSubstLen is the same as LSubst, but $x$ is a length variable now, in which case the variables of the term on the right-hand side become length variables too (a change of their values influences a length variable). LSubstLen takes precedence before AlignSplit.

$$\textsc{LSubstLen}: \frac{(\quad \mathcal{I} \uplus \{x \subseteq t\},\quad x \subseteq t{::}W,\quad \mathcal{R},\quad \sigma,\quad \texttt{Len}\quad)}{\left\{(\quad \sigma'_i(\mathcal{I} \cup \mathit{Align}),\quad \sigma'_i(\mathit{Align}{::}W),\quad \mathcal{R}'_i,\quad \sigma'_i \cup \sigma,\quad \texttt{Len} \cup \texttt{Var}(t'_i)\quad)\right\}_{i=1}^{k}}\ \varphi_{\textsc{LSubstLen}}$$

where $\varphi_{\textsc{LSubstsLen}} \overset{\text{def.}}{\Leftrightarrow} x \in \texttt{Len} \wedge \texttt{LR-noodlify}(x \subseteq t, \mathcal{R}) = \left\{(\mathcal{R}'_i, \mathit{Align} \uplus \{x \subseteq t'_i\})\right\}_{i=1}^{k} \wedge \bigwedge_{i=1}^{k} \sigma'_i = \{x \mapsto t'_i\}$.

Rule RSubstLen is symmetrical to LSubstLen, the difference is that $t \subseteq x$ is replaced by $x \subseteq t$. Note that there is no symmetrical counterpart of LSubst. This is because the case with a non-length $x$ on the right-hand side is handled by Refine from stabilization rules.

*Local soundness and completeness of the rules.* A proof rule is called *sound* if satisfiability of the premise implies satisfiability of one of the conclusions and it is *locally complete* if satisfiability of one of the conclusions implies satisfiability of the premise. If all proof rules are locally complete, and if $\Phi$ or one of the produced conclusions turns out to be satisfiable, then $\Phi$ is also satisfiable. If all the proof rules are sound and none of the produced conclusions is satisfiable, then $\Phi$ is unsatisfiable.

Soundness and local completeness of our rules can be concretized as follows. Let a rule produce for a premise $(\mathcal{I}, W, \mathcal{R}, \sigma, \mathsf{Len})$ the set of conclusions $\{(\mathcal{I}_i, W_i, \mathcal{R}_i, \sigma_i \circ \sigma, \mathsf{Len}_i)\}_{i=1}^{k}$. The rule is sound and locally complete if $\nu \models \sigma_i(\mathcal{I} \wedge \mathcal{R})$ if and only if $\nu \models \mathcal{I}_i \wedge \mathcal{R}_i$ for some $1 \leq i \leq k$.

Lemma 4.1. *All rules of the combined algorithm are sound and locally complete.*

Proof. (sketch) Soundness and completeness of Refine and AlignSplit follow directly from Lemma 3.3 and Lemma 3.9, respectively. Similarly, soundness and completeness LSubst, LSubstLen, and RSubstLen follow from Lemma 3.9 and from the fact that substituting $x$ by $t_i'$ and removing $x \subseteq t_i'$ or $t_i' \subseteq x$, depending on the rule, preserves solutions modulo the substitution $\sigma_i'$ since $t_i'$ consists of fresh variables. Soundness and completeness of the rest of the rules is trivial. □

We also argue that upon termination, the variables in the remaining equations (inclusions) do not influence the length constraints. This means that we can indeed obtain a precise characterisation of the lengths of length variables.

Lemma 4.2. *When* $\mathsf{SAT}(\langle \mathcal{I}', \mathcal{R}', \sigma' \rangle)$ *is derived,* $\mathcal{I}'$ *does not restrict* $\sigma'(\mathcal{L})$.

Proof. (sketch) The lemma's statement follows from the fact that upon termination (achieving stability), length variables can appear only on the left-hand sides of the inclusions. This means that any combination of values from languages of the length variables can be completed into a (string) solution of the inclusions (equations).

The claim that no length variable can appear on the right-hand side of any inclusion upon termination can be argued as follows: Processing an inclusion with a single variable on the right-hand side eliminates that inclusion (by processing, we mean removing the inclusion as the head of the worklist in a premise of a rule). Indeed, the used rule must be either AlignSplit or RSubstLen. For AlignSplit, the inclusion is replaced by the alignment equations, where it holds that if a variable occurs in the right-hand side, then it appears there alone. RSubstLen then eliminates inclusions with a single length variable $x$ on the right-hand side.

RSubstLen, however, also substitutes $x$ by $t_i'$, and makes all its variables into length variables. We need to make sure that also the variables of $t_i'$ will not remain on the right-hand side of some inclusion until stabilisation. Intuitively, RSubstLen is sending length awareness of variables along paths in the inclusion graph. The worklist mechanism ensures that inclusions are processed along the paths in the graph. The variables of $t_i'$ in right-hand sides of inclusions will therefore be eventually eliminated by processing those inclusions. □

Finally, we argue that the algorithm terminates whenever the initial set of equations $\mathcal{E}$ is chain-free, which gives us completeness for the chain-free fragment of equations, regular constraints, and length constraints.

Theorem 4.3. *If* $\mathcal{E}$ *is chain-free, then the combined algorithm terminates.*

Proof. (sketch) The inclusions obtained from the equations form an acyclic inclusion graph. The combined algorithm essentially explores the graph in the depth-first manner using the worklist $W$. Since there are no cycles, the exploration terminates. □

## 5 EXTENDED CONSTRAINTS

Established string constraint benchmarks contain a large amount of extended string constraints, hence we extend our algorithm with a basic handling of a minimal set of these extensions in order to be able to compare and compete with other solvers. We handle the extended constraints almost exclusively by generic translations to the basic constraints. Interesting is the case of disequalities,

where a slight modification of a known translation allows us to extend the decidable chain-free fragment of string constraints with unrestricted disequalities.

### 5.1 Disequalities

Disequalities $s \neq t$ occur sometimes in real-word benchmarks and they are generated by rewriting of extended string functions and predicates. Treating them efficiently is therefore important.

Consider a disequality of the form $s \neq t$. Already [Abdulla et al. 2014] noticed that the disequality can be converted to equations, namely, to

$$\bigvee_{a\in\Sigma}(s = tax \vee t = sax) \vee \bigvee_{a_1,a_2\in\Sigma,a_1\neq a_2}(s = xa_1y_1 \wedge t = xa_2y_2). \tag{10}$$

Such a formula, however, involves large disjunctions over all characters. It was improved upon in [Abdulla et al. 2015], which eliminates the large disjunction and uses length constraints and a special disequality of the so-called witness variables and with lengths restricted to length one:

$$|s| \neq |t| \vee (s = xa_1y_1 \wedge t = xa_2y_2 \wedge |a_1| = |a_2| = 1 \wedge |y_1| = |y_2| \wedge a_1 \neq a_2) \tag{11}$$

where $x, y_1, y_2, a_1, a_2$ are fresh string variables. The latter approach is more efficient, but the newly created equations could potentially create a cycle in the inclusion graph. The original disequalities $s \neq t$ must hence be taken into account in the definition of the chain-free fragment in the same way as equations.

Here, we notice that this dependence of chain-freeness on disequalities can be completely removed by using yet another translation of disequalities. Essentially, the different $a_1$ and $a_2$ do not have to appear after a common prefix $x$, but after any two prefixes $x_1$ and $x_2$ of the same length. Then, instead of new equations, the disequality is reduced to essentially only length constraints, which do not impact chain-freeness:

$$|s| \neq |t| \vee \Big( \overbrace{s = x_1a_1y_1 \wedge t = x_2a_2y_2 \wedge |x_1| = |x_2| \wedge a_1 \in \Sigma \wedge a_2 \in \Sigma}^{\mathsf{disstr}(x_1,x_2,y_1,y_2,a_1,a_2)} \wedge \overbrace{a_1 \neq a_2}^{\mathsf{dist}(a_1,a_2)} \Big). \tag{12}$$

Here, $x_1$, $x_2$, $y_1$, $y_2$, $a_1$, and $a_2$ are fresh variables. The formula is then processed by the SAT solver of a DPLL($T$) procedure. The procedure may produce a query to our string solver in the form of a conjunction of basic string constraints, along with character disequalities $\mathsf{dist}(a_1, a_2)$.

Our string solver leaves processing $\mathsf{dist}(a_1, a_2)$ after it solves the conjunction of the basic string constraints by the combined algorithm of Sec. 4.2 and obtains a stable language assignment $\mathcal{R}$ and an alignment substitution $\sigma$. During the algorithm, $a_1$ and $a_2$ are handled as length variables so that we get the precise set of symbols in $\mathcal{R}$ and $\sigma$ for $a_1$ and $a_2$. This also means that if $a_1$ and $a_2$ are substituted in $\sigma$, then they can each be only susbtituted by one variable, as they contain exactly one symbol. Let $a'_1$ and $a'_2$ that are substituted instead of $a_1$ and $a_2$ by $\sigma$. The solver then checks that the disequality is satisfiable by transforming it into the arithmetic constraint

$$i_{a'_1} \neq i_{a'_2} \wedge \bigvee_{a\in\mathcal{R}(a'_1)} i_{a'_1} = \mathsf{code}(a) \wedge \bigvee_{a\in\mathcal{R}(a'_2)} i_{a'_2} = \mathsf{code}(a) \tag{13}$$

where $i_v$ is a fresh integer variable corresponding to the string variable $v$ and $\mathsf{code}(a)$ is the Unicode code point of the character $a$. As we use substituted variables, this transformation works even for multiple disequalities.

Lemma 5.1. *If $\varphi$ is a chain-free string constraint, then $\varphi \wedge \mathsf{disstr}(x_1, x_2, y_1, y_2, a_1, a_2)$ is chain-free.*

Proof. Let $\varphi$ be a chain-free constraint. Since all variables on the right side of the newly added string equations are fresh, we can conclude that the constraint $\varphi \wedge \mathsf{disstr}(x_1, x_2, y_1, y_2, a_1, a_2)$ from Eq. (12) is also chain-free. Indeed, adding the equations $s = x_1a_1y_1$ and $t = x_2a_2y_2$ cannot introduce

a cycle in the inclusion graph since the variables on the right-hand sides $x_1a_1y_1$ and $x_2a_2y_2$ have only the single occurrence, hence they cannot have incoming edges. □

Overall, the disequality is reduced to length equations, word equations that cannot occur in a cycle of the inclusion graph, and a character disequality constraint that is easy to solve. This gives us a decidable extension of the chain-free fragment with unrestricted disequalities.

Theorem 5.2. *Chain-free constraints with unrestricted disequalities are decidable.*

### 5.2 String Functions and Predicates

Apart from disequalities, we currently support the following extended string predicates and functions from the SMT-LIB format [Barrett et al. 2016a]: (i) `str.at`, (ii) `str.substr`, (iii) `str.replace`, (iv) `str.indexof`, (v) `str.contains`, (vi) `str.prefixof`, and (vii) `str.suffixof`. These constraints are reduced to the basic constraints by generic constructions. We use a similar way of rewriting predicates as is implemented, e.g., in Z3 [de Moura and Bjørner 2008]. We illustrate the translation on the example of $\texttt{str.substr}(s, i, n)$. The function is supposed to return the longest substring of $s$ of the length at most $n$ starting at position $i$. It evaluates to the empty string if $n$ is negative or $i$ is not in the interval $[0, |s| - 1]$. We substitute all occurrences of $\texttt{str.substr}(s, i, n)$ in the input constraint by a fresh string variable $v$ and add a conjunction of the following formulae (where $x_1, x_2$ are fresh string variables):

(1) $v = \texttt{str.substr}(s, i, n)$
(2) $0 \leq i \leq |s| \Rightarrow x_1 v x_2 = s$
(3) $0 \leq i \leq |s| \Rightarrow |x_1| = i$
(4) $(0 \leq i \leq |s| \wedge 0 \leq n \leq |s| - i) \Rightarrow |v| = n$
(5) $(0 \leq i \leq |s| \wedge |s| - i < n) \Rightarrow |v| = |s| - i$
(6) $i \geq |s| \Rightarrow v = \epsilon$
(7) $i < 0 \Rightarrow v = \epsilon$
(8) $n \leq 0 \Rightarrow v = \epsilon$

The negative form, $\neg\texttt{str.contains}$ is supported only in the case when the parameter expressing what should be checked is an explicit string (not a variable or other more complicated term). For instance, we support $\neg\texttt{str.contains}(s, \texttt{"}abc\texttt{"})$ expressing that $s$ does not contain the string "$abc$" (which we reduce to the regular non-membership constraint $s \notin \Sigma^* abc \Sigma^*$ and change to membership constraint by NFA complementation), but we do not support $\neg\texttt{str.contains}(s, x)$ where $x$ is a string variable. The unrestricted form of $\neg\texttt{str.contains}$ involves universal quantification over strings. Theory of strings with universal quantification is generally undecidable [Day et al. 2018]. Although $\neg\texttt{str.contains}$ may be a simpler problem, its decidability is currently not known, and, together with practical methods for solving this type of constraint, is an area of active research (see, e.g. [Abdulla et al. 2021]).

## 6 PREPROCESSING

Before running the combined algorithm from Sec. 4.2, we apply series of preprocessing steps. The aim is to simplify the equations and hence reduce the number of cases generated from Align&Split steps by converting equations and disequalities to regular and length constraints.

The preprocessing steps are formulated using rewriting rules (with a notation similar to the proof rules in Sec. 4.2), which work on the level of a whole string constraint, a conjunction of word equations and disequalities $\mathcal{E}$, a normalised regular constraint $\mathcal{R}$, and a length constraint $\mathcal{L}$. Seeing the full context of the whole conjunction allows us to use optimizations that do not seem to be easily available in solvers that use the eager approach, such as cvc5 [Nötzli et al. 2022], where equations and disequalities are processed separately. We use $x, y, x_i, \ldots \in \mathbb{X}$ for variables and $S, S_L, S_R, S_i, \ldots \in \mathbb{X}^*$ for concatenations of variables. The rewriting rules follow.

The first rule eliminates simple equations of the form $x = y$ by substituting $y$ by $x$, intersecting the language of $x$ with that of $y$, and generating the length constraint $|x| = |y|$. Note that in the

preprocessing rules, we conjunct newly generated length constraints instead of a direct modification of $\mathcal{L}$ (as for $|x| = |y|$ where we could replace $x$ with $y$ in $\mathcal{L}$) because we want to get as close as possible to the employment of our procedure in the SMT framework.

$$\textsc{PropVars} : \frac{\mathcal{E} \uplus \{x = y\} \quad \mathcal{R} \uplus \{x \mapsto L_x, y \mapsto L_y\} \quad \mathcal{L}}{\mathcal{E}[y/x] \quad \mathcal{R} \cup \{x \mapsto L_x \cap L_y\} \quad \mathcal{L} \wedge |x| = |y|}$$

If the language of one side contains only $\epsilon$, we can propagate it to each variable on the other side (by multiple applications of the rule PropEps).

$$\textsc{PropEps} : \frac{\mathcal{E} \uplus \{S_1 x S_2 = S_R\} \quad \mathcal{R} \uplus \{x \mapsto L_x\} \quad \mathcal{L}}{\mathcal{E} \uplus \{S_1 x S_2 = S_R\} \quad \mathcal{R} \cup \{x \mapsto L_x \cap \{\epsilon\}\} \quad \mathcal{L}} \; \mathcal{R}(S_R) = \{\epsilon\}$$

A variable $x$ whose language is just the empty word $\epsilon$ can be safely removed. Rule RemoveEps is applied after the rule PropEps saturates the set of $\epsilon$-variables. The length constraint $|x| = 0$ propagates the changes to lengths.

$$\textsc{RemoveEps} : \frac{\mathcal{E} \quad \mathcal{R} \uplus \{x \mapsto \{\epsilon\}\} \quad \mathcal{L}}{\mathcal{E}[x/\epsilon] \quad \mathcal{R} \quad \mathcal{L} \wedge |x| = 0}$$

Trivial equations, i.e., equations with identical sides, can be removed. This is especially useful after applying PropEps and RemoveEps, which often create trivial equations.

$$\textsc{RemoveTrivial} : \frac{\mathcal{E} \uplus \{S = S\} \quad \mathcal{R} \quad \mathcal{L}}{\mathcal{E} \quad \mathcal{R} \quad \mathcal{L}}$$

*Regular equations*, i.e., equations of the form $x = x_1 \cdots x_n$, where $x_1, \ldots, x_n$ have exactly one occurrence in the constraint (so they do not occur in the length constraint either), can be transformed into the regular constraint $x \in \mathcal{R}(x_1) \cdots \mathcal{R}(x_n)$. Then, we only need to intersect the language of $x$ with $\mathcal{R}(x_1) \cdots \mathcal{R}(x_n)$.

$$\textsc{RemoveReg} : \frac{\mathcal{E} \uplus \{x = x_1 \cdots x_n\} \quad \mathcal{R} \uplus \{x \mapsto L_x\} \uplus \{x_i \mapsto L_i\}_{i=1}^{n} \quad \mathcal{L}}{\mathcal{E} \quad \mathcal{R} \uplus \{x \mapsto (L_x \cap (L_1 \cdots L_n))\} \quad \mathcal{L}} \varphi_{\textsc{RemoveReg}}$$

$$\text{where } \varphi_{\textsc{RemoveReg}} \overset{\text{def.}}{\Leftrightarrow} \forall 1 \leq i,j \leq n (i \neq j \Rightarrow x \neq x_i \wedge x_i \neq x_j \wedge x_i \notin \mathtt{Var}(\mathcal{E} \wedge \mathcal{L}))$$

Equations $S_L = x_1 \cdots x_n$, where variables $x_1, \ldots, x_n$ have no restriction on their languages (all are $\Sigma^*$) and have only the single occurrence in $\mathcal{E}$ can be removed. This is because if we find a satisfying solution for other variables, we can always complete it with values for each $x_i$, as there are no restrictions on them except the one equation. This works even if these variables occur in the length constraints $\mathcal{L}$, but in this case, we have to add the length constraint $|S_L| = |x_1| + \cdots + |x_n|$ to propagate the equation to $\mathcal{L}$.

$$\textsc{RemoveLenSat} : \frac{\mathcal{E} \uplus \{S_L = x_1 \cdots x_n\} \quad \mathcal{R} \uplus \{x_i \mapsto \Sigma^*\}_{i=1}^{n} \quad \mathcal{L}}{\mathcal{E} \quad \mathcal{R} \quad \mathcal{L} \wedge |S_L| = |x_1| + \cdots + |x_n|} \varphi_{\textsc{RemoveLenSat}}$$

$$\text{where } \varphi_{\textsc{RemoveLenSat}} \overset{\text{def.}}{\Leftrightarrow} \forall 1 \leq i,j \leq n (i \neq j \Rightarrow x_i \neq x_j \wedge x_i \notin \mathtt{Var}(\mathcal{E}) \cup \mathtt{Var}(S_L))$$

We can remove disequalities whose sides have disjoint languages, as these disequalities will always hold under $\mathcal{R}$ or any refinement of $\mathcal{R}$.

$$\textsc{RemoveDis1} : \frac{\mathcal{E} \uplus \{S_L \neq S_R\} \quad \mathcal{R} \quad \mathcal{L}}{\mathcal{E} \quad \mathcal{R} \quad \mathcal{L}} \; \mathcal{R}(S_L) \cap \mathcal{R}(S_R) = \emptyset$$

For the special case of a disequality $x \neq S_R$ that contains only one variable on one side, we can improve upon RemoveDis1 by using equations in $\mathcal{E}$. Specifically, if an equation $x = S'_R$ is in $\mathcal{E}$ and $\mathcal{R}(S_R)$ and $\mathcal{R}(S'_R)$ are disjoint, then the disequality $x \neq S_R$ must hold. We can take even more general rule where we take all equations $x = S_1, \ldots, x = S_n$, and check whether $\mathcal{R}(S_R) \cap \mathcal{R}(S_1) \cap \cdots \cap \mathcal{R}(S_n) \cap \mathcal{R}(x)$ is empty.

$$\textsc{RemoveDis2} : \frac{\mathcal{E} \uplus \{x \neq S_R\} \uplus \{x = S_i\}_{i=1}^{n} \quad \mathcal{R} \quad \mathcal{L}}{\mathcal{E} \uplus \{x = S_i\}_{i=1}^{n} \quad \mathcal{R} \quad \mathcal{L}} \; \mathcal{R}(S_R) \cap \mathcal{R}(S_1) \cap \cdots \cap \mathcal{R}(S_n) \cap \mathcal{R}(x) = \emptyset$$

Rule GenIdent generates the identity $x = y$ from equations $S_1xS_2 = S$ and $S_1yS_2 = S$. The identity can be then propagated by the rule PropVars. Note that this is also applied for the case of one equation $S_1xS_2 = S_1yS_2$ in which case after the rule PropVars we can apply RemoveTrivial to remove the equation.

$$\textsc{GenIdent}: \frac{\mathcal{E} \uplus \{S_1xS_2 = S, S_1yS_2 = S\} \quad \mathcal{R} \quad \mathcal{L}}{\mathcal{E} \cup \{S_1xS_2 = S, S_1yS_2 = S, x = y\} \quad \mathcal{R} \quad \mathcal{L}}$$

If $\mathcal{R}$ is infeasible (the language of some variable is empty), we can immediately finish with UNSAT without even running the combined algorithm.

$$\textsc{Unsat}: \frac{\mathcal{E} \quad \mathcal{R} \uplus \{x \mapsto \emptyset\} \quad \mathcal{L}}{\texttt{UNSAT}}$$

The last rule SatLang is an underapproximating rule applicable if a variable $x$ is assigned a *co-finite* language, i.e., a language whose complement is a finite set $T$. The rule removes the regular constraint on $x$ and replaces it with length contraints excluding *all* words with the same length as one of the words from $T$. Removing regular constraints helps the combined algorithm stabilize faster and the rule may also enable RemoveLenSat.

Unlike the previous rules, SatLang underapproximates the set of solutions. It often leads to a faster identification of SAT. However, an UNSAT answer after using the rule is inconclusive, and we have to run the procedure from the beginning without SatLang.

$$\textsc{SatLang}: \frac{\mathcal{E} \quad \mathcal{R} \uplus \{x \mapsto L_x\} \quad \mathcal{L}}{\mathcal{E} \quad \mathcal{R} \uplus \{x \mapsto \Sigma^*\} \quad \mathcal{L} \wedge \bigwedge_{i=1}^{n} |x| \neq |\ell_i|} \; \Sigma^* \setminus L_x = \{\ell_1, \ldots, \ell_n\}$$

*Length variables.* Rules PropVars, RemoveEps, and RemoveLenSat introduce new length constraints, and, consequently, more length variables. Having more length variables, however, harms the combined algorithm of Sec. 4.2. Fortunately, it can be argued that length variables that were introduced by the rewriting rules above can safely be removed from the set Len of length variables used in the combined algorithm (except $x$ from PropVars for $x = y$ in case $y$ was a length variable).

*Preference of regular constraints.* In our implementation, we take advantage of the efficiency of our decision procedure in handling of regular constraints. Unlike length constraints, whose feasibility is being checked after obtaining a stable assignment, regular constraints are used from the start to prune the state space. Moreover, length constraints bring length variables, which may cause more splitting in the combined procedure of Sec. 4.2. Therefore, during rewriting of string functions and predicates, we prefer to use regular constraints. For instance, instead of $|c| = 1$, we use the equivalent regular constraint $c \in \Sigma$.

## 7 IMPLEMENTATION IN Z3

We implemented our decision procedure into the DPLL($T$)-based SMT solver Z3 [de Moura and Bjørner 2008] (version 4.11.2) as a plugin for the theory of strings. The DPLL($T$) framework combines a SAT solver with multiple theory solvers for conjunctions of constraints in certain theories. The SAT solver is responsible for computing a model of the boolean skeleton of the input SMT formula. Based on the propositional model, the theory checks satisfiability and, in the negative case, returns a *conflict clause* (true in the theory but violating the propositional model), which forces the SAT solver to provide another solution. Our theory implementation checks satisfiability of the string constraint after the whole propositional model is constructed (referred to as the *lazy* approach [Nieuwenhuis et al. 2006]). The lazy reasoning enables us to use more aggressive preprocessing utilizing the information about the whole constraint (cf. Sec. 6). When we obtain a propositional model (which assigns a truth value to every boolean variable of the skeleton), we remove from it assignments that are irrelevant to satisfiability. For instance, if the input formula

Table 1. Results of experiments on all benchmarks. For each benchmark and tool, we give the number of unsolved instances categorized by the number of timeouts ("TOs"), errors ("Es"), and unknowns ("Us"), the total runtime (in seconds), and the runtime without timeouts ("Time–TOs"). Results for tools with the lowest number of unsolved instances (the sum of columns TOs, Es, Us) and lowest runtime (with and without timeouts) on the given benchmark are in **bold**. Number of errors with * contain also wrongly solved instances.

| | Sygus-Qgen (343) | | | | | Norn (1 027) | | | | | Slent (1 128) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TOs | Es | Us | Time | Time–TOs | TOs | Es | Us | Time | Time–TOs | TOs | Es | Us | Time | Time–TOs |
| Z3-Noodler | **0** | **0** | **0** | **5.7** | **5.7** | **0** | **0** | **0** | **18.7** | **18.7** | **7** | **0** | **0** | 982.3 | 142.3 |
| cvc5 | **0** | **0** | **0** | 188.2 | 188.2 | 84 | 0 | 0 | 10 883.3 | 803.3 | 28 | 0 | 0 | 4 763.7 | 1 403.7 |
| Z3 | **0** | **0** | **0** | 34.2 | 34.2 | 127 | 0 | 0 | 15 318.7 | 78.7 | 73 | 0 | 0 | 9 313.0 | 553.0 |
| Z3str3RE | 1 | 0 | 0 | 163.9 | 43.9 | 133 | 0 | 0 | 15 986.2 | 26.2 | 87 | 0 | 0 | 10 457.3 | **17.3** |
| Z3-Trau | 2 | 41 | 0 | 6 065.8 | 5 825.8 | | | | N/A | | 5 | *53 | 4 | **662.2** | 62.2 |
| Z3str4 | **0** | **0** | **0** | 65.9 | 65.9 | 75 | 0 | 0 | 9 113.6 | 113.6 | 77 | 0 | 0 | 9 271.5 | 31.5 |
| OSTRICH | **0** | **0** | **0** | 962.1 | 962.1 | **0** | **0** | **0** | 8 985.7 | 8 985.7 | 155 | 1 | 0 | 23 547.0 | 4 827.0 |

| | Slog (1 976) | | | | | Leetcode (2 652) | | | | | Kaluza (19 432) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TOs | Es | Us | Time | Time–TOs | TOs | Es | Us | Time | Time–TOs | TOs | Es | Us | Time | Time–TOs |
| Z3-Noodler | **0** | **0** | **0** | 36.2 | 36.2 | 35 | 0 | 0 | 4 779.2 | 579.2 | 192 | 0 | 0 | 24 226.9 | 1 186.9 |
| cvc5 | **0** | **0** | **0** | **12.1** | **12.1** | **0** | **0** | **0** | 149.3 | 149.3 | **6** | **0** | **0** | **1 914.4** | 1 194.4 |
| Z3 | 33 | 0 | 0 | 4 297.1 | 337.1 | **0** | **0** | **0** | **142.4** | 142.4 | 188 | 0 | 0 | 23 418.5 | 858.5 |
| Z3str3RE | 58 | 0 | 0 | 8 279.5 | 1 319.5 | 2 | 0 | 190 | 275.3 | **35.3** | 132 | 0 | 8 | 16 133.1 | **293.1** |
| Z3-Trau | 45 | 0 | 1 | 7 827.6 | 2 427.6 | **0** | **0** | **0** | 162.0 | 162.0 | 125 | 0 | 0 | 20 587.7 | 5 587.7 |
| Z3str4 | 22 | 0 | 0 | 3 816.3 | 1 176.3 | 2 | 0 | 2 | 400.9 | 160.9 | 132 | 0 | 46 | 17 752.9 | 1 912.9 |
| OSTRICH | 6 | *5 | 0 | 9 323.7 | 8 603.7 | 185 | 26 | 0 | 33 308.9 | 8 108.9 | 305 | 0 | 0 | 88 056.3 | 51 456.3 |

is $s_1 = t_1 \vee s_2 = t_2$, then a boolean model might be, e.g., the assignment $[s_1 = t_1] \wedge \neg[s_2 = t_2]$. Such an assignment would make us solve the constraint $s_1 = t_1 \wedge s_2 \neq t_2$, where the disequality $s_2 \neq t_2$ is irrelevant for the satisfiability of the input formula, so we remove it. This has the following two benefits: (i) in each call of our solver, we solve easier constraints and (ii) the conflict clause we output is more general, which restricts the total number of calls.

As a specific feature, we use *iterative* computation of feasible solutions together with a tight cooperation with the LIA solver. A naive approach is to generate a length formula describing possible lengths of all solutions of a string formula and then use the LIA solver to check if the length formula is satisfiable w.r.t. the input length constraints. This would mean to explore the entire proof tree of the combined algorithm of Sec. 4.2. It might, however, happen that already the first solution satisfies the length constraints, making the computation of the remaining ones redundant. Therefore, we iteratively generate solutions of the string equations and eagerly check if their lengths are feasible in conjunction with the input length formula.

*Efficient Automata Representation.* A foundation stone of our decision procedure is efficient handling of finite automata. We use nondeterministic automata instead of the deterministic ones since they allow more concise representation, which is beneficial particularly for the noodlification (the procedures `L-noodlify` and `LR-noodlify` discussed in Sec. 3). As the background automata library we use eNFA, a simple yet surprisingly competitive implementation of NFAs [Fiedor et al. 2023]. In order to keep automata as small as possible, we use simulation-based minimization [Bustan and Grumberg 2003]. We apply the reduction eagerly on every result of a noodlification. We also try to keep the alphabet as small as possible by using only the symbols used in the input formula. For this to work correctly, we, however, need to add a dummy symbol for each disequality and regular non-membership constraint, to allow these negative constraints be satisfied with symbols not used in the input.

## 8 EXPERIMENTAL EVALUATION

*Used tools and environment.* We compared the performance of our implementation, called Z3-Noodler[6], with the following state-of-the-art tools: cvc5 [Barbosa et al. 2022] (version 1.0.5), Z3 [de Moura and Bjørner 2008] (version 4.12.1), Z3str3RE [Berzish et al. 2023, 2021], Z3-Trau [Abdulla et al. 2020] (version 1.1), Z3str4 [Mora et al. 2021], and OSTRICH [Chen et al. 2019] (version 1.2). The experiments were executed on a workstation with an Intel Core i5 661 CPU at 3.33 GHz with 16 GiB of RAM running Debian GNU/Linux. The timeout was set to 120 s.

*Benchmarks.* For the comparison, we used supersets of several benchmark sets that were used in SMT-COMP'22 [SMT-COMP'22 2022] (it seems some benchmarks in the repository were not used for competition runs). In particular, we took datasets from division QF_Strings that use the following two logics:

**QF_S** [SMTLib 2023a] containing only string constraints without explicit length constraints.

- Slog ([Wang et al. 2016], 1,976 formulae): formulae obtained from real web applications using static analysis tools JSA [Christensen et al. 2003] and Stranger [Yu et al. 2010].
- Sygus-Qgen (343 formulae): formulae with equations, disequalities, and regular constraints.

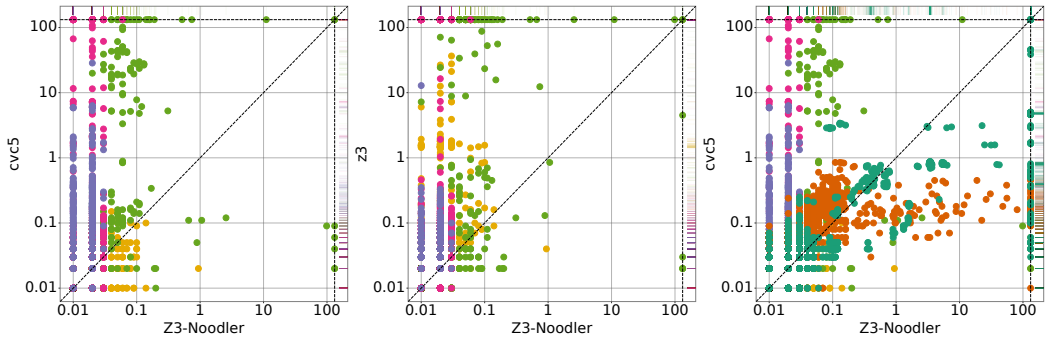
**QF_SLIA** [SMTLib 2023b] containing also length constraints.

- Norn ([Abdulla et al. 2014, 2015], 1,027 formulae): formulae representing queries generated during verification of string-processing programs [Abdulla et al. 2014].
- Slent ([Wang et al. 2018], 1,128 formulae): formulae generated from Kaluza [Saxena et al. 2023] and Stranger [Yu et al. 2010] benchmarks; they cover security analysis of string manipulating web applications.
- Leetcode (2,652 formulae): formulae generated by concolic execution engines Conpy [Chen et al. 2014] and PyExZ3 [Ball and Daniel 2015] from interview questions at https://leetcode.com.
- Kaluza ([Liang et al. 2016; Saxena et al. 2023], 19,432 formulae): formulae obtained from JavaScript operations on real-world AJAX web applications generated by Kudzu, a symbolic executor for JavaScript [Saxena et al. 2010].

The benchmarks contain a wide range of complex string equations, disequalities, regular constraints (all benchmarks), length constraints (QF_SLIA), together with string functions and predicates. In particular, `str.replace` (Slog, Slent) and `str.indexof`, `str.substr`, and `str.at` (Leetcode). In our experiments, we did not use all datasets from QF_SLIA. Namely, we did not use Kepler as it contains only hand-crafted quadratic equations (many of them easily solvable by Nielsen transformation [Nielsen 1917]), Full-str-int as it contains unsupported functions (conversions between strings and integers), and Pyex and Str-small-rw as they consist mostly of combinations of string functions and predicates. Although we support a general way of handling these functions and predicates (see Sec. 5.2), the aim of this paper is not an optimization of these specific features. We leave their optimizations as a future work (see the discussion below).

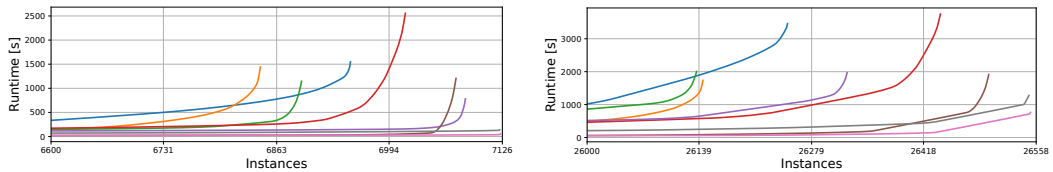
*Results.* The results of the experiments are shown in Table 1. We show the total number of unsolved instances divided by timeouts, errors (segfaults for Z3-Trau or problems with model extraction for OSTRICH and wrong results), and unknown results, the total runtime, and the total runtime without timeouts. We do not show the results of Z3-Trau on Norn as nearly all formulae in this benchmark contain the string constraint `re.range`, which Z3-Trau does not support correctly. This constraint also occurs in Slent benchmark, but less frequently, where Z3-Trau gave 6 incorrect results. Furthermore, OSTRICH, whose version 1.2 should work even on

[6]The tool is available at https://github.com/VeriFIT/z3-noodler.




(a) Z3-Noodler vs. cvc5 on selection (b) Z3-Noodler vs. Z3 on selection (c) Z3-Noodler vs. cvc5 on all

Fig. 2. Comparison of Z3-Noodler with cvc5 (a), Z3 (b) on benchmarks Sygus-Qgen, Slog, Slent, and Norn; and cvc5 on all benchmarks (c). Times are in seconds, axes are logarithmic. Dashed lines represent timeouts (120 s). Colors distinguish benchmarks: • Slog, • Slent, • Norn, • Sygus-Qgen, • Leetcode, and • Kaluza.




(a) The hardest 526 formulae without Kaluza (b) The hardest 558 formulae with Kaluza

Fig. 3. Cactus plots of solvers and portfolio solvers. The $y$-axis is the cumulative time taken on benchmarks, the $x$-axis denotes the number of solved benchmarks ordered by runtime. Colors distinguish solvers: • cvc5, • Z3, • Z3str3RE, • Z3str4, • Z3-Noodler, and portfolio solvers: • cvc5 + Z3 + Z3str3RE + Z3str4, • cvc5 + Z3 + Z3str3RE + Z3str4 + Z3-Noodler, and • cvc5 + Z3-Noodler.

formulae outside the *straight-line fragment* [Lin and Barceló 2016], gave 5 incorrect results on the Slog benchmark.

The results show that Z3-Noodler outperforms other tools on Sygus-Qgen, Norn, and Slent benchmarks by a large margin, while being closely behind the best tool (cvc5) on Slog. On the last two benchmarks, performance of Z3-Noodler is weaker, however, on Kaluza, the performance is still comparable with Z3.

In Fig. 2, we show scatter plots comparing the performance of Z3-Noodler with cvc5 and Z3. We selected cvc5 and Z3, as they are heavily used in industry, cvc5 behaves better than the remaining tools we compare with, and Z3 takes the second place together with Z3str3RE and Z3str4, which have a similar performance. Moreover, Z3-Noodler is built on Z3. Figs. 2a and 2b show the comparison on Sygus-Qgen, Slog, Slent, and Norn, on which Z3-Noodler has the best performance. Z3-Noodler is here clearly much faster than its competitors. Fig. 2c compares Z3-Noodler with cvc5 on all benchmarks (comparisons with other tools look mostly similar or better for Z3-Noodler). The plot shows that these two tools behave complementary to each other, each formula can be usually solved by either Z3-Noodler or cvc5 under one second and there are only 8 formulae in the whole benchmark set that neither can solve.

The cactus plots in Fig. 3 compare the sorted cumulative runtimes of all tools (except Z3-Trau and OSTRICH, as they give wrong results) and *portfolio solvers*, which run these tools in parallel and take the best result. The cactus plot in Fig. 3a shows the hardest 526 instances of all benchmarks except Kaluza as it is much larger than other benchmarks and would dominate the figure. It also

shows that, without Kaluza, Z3-Noodler is better than a portfolio solver composed of all the other tools. In Fig. 3b, we provide the cactus plot of the hardest 558 instances of all benchmarks, including Kaluza. In both cactus plots, the portfolio solver composed of only Z3-Noodler and cvc5 is already nearly as good as the portfolio solver composed of all solvers. Moreover, it is evident that the portfolio solver with Z3-Noodler significantly improves the performance of the portfolio solver without Z3-Noodler.

*Regex-heavy benchmark.* To showcase the strength of our algorithm on formulae that contain only regex operations, we add results from one more benchmark, which was not used in SMT-COMP'22, but was used to show the strengths of the derivation-based solving method used in Z3 [Stanford et al. 2021]. This benchmark contains 110 handwritten formulae encoding date and password problems, problems where boolean operations interact with concatenation and iteration, and problems with exponential determinization and 155 problems encoding intersection and containment of regexes from https://regexlib.com. Table 2 shows the results for this benchmark. We do not show results for Z3-Trau as the benchmark contains operations that were not standardized during the development of Z3-Trau and it cannot support them. Furthermore, Z3str3RE and OSTRICH returned 5 and 6 wrong results, respectively. The benchmark also contains some formulae containing comparison of regular languages, which is not supported by cvc5 or Z3-Noodler (the errors and unknowns are all from these formulae). As Z3-Noodler is automata-based, implementing support for these constraints will not be hard, and we plan to add support for them in the future.

Table 2. Results of experiments on Regex. The notation is the same as in Table 1.

| | Regex (265) | | | | |
|---|---|---|---|---|---|
| | TOs | Es | Us | Time | Time–TOs |
| Z3-Noodler | **5** | **4** | **7** | 1992.7 | 1392.7 |
| Z3 | 47 | 0 | 1 | 6031.3 | 391.3 |
| cvc5 | 69 | 9 | 0 | 8648.4 | 368.4 |
| Z3str3RE | 30 | *5 | 158 | 3921.2 | 321.2 |
| Z3-Trau | N/A | | | | |
| Z3str4 | 25 | 0 | 158 | 3238.4 | **238.4** |
| OSTRICH | 1 | *31 | 0 | **1302.9** | 1182.9 |

The results show that Z3-Noodler has the lowest number of unsolved instances and, disregarding results of OSTRICH (whose errors are not included in total time), has the best total solving time.

*Discussion.* We can see from the experiments that our approach applies best on complex combinations of (dis)equalities with regular and length constraints having just a few other string functions and predicates (i.e., benchmarks Slog, Sygus-Qgen, Norn, Slent, and Regex). On these benchmarks, Z3-Noodler works better compared to all other tools, including the industrial tools cvc5 and Z3.

A weakness of Z3-Noodler currently is handling of combined (especially nested) string functions and predicates (as it is the case for Leetcode). Although we generally support them, we have not yet implemented heuristics needed to handle frequent special cases efficiently. For instance, the predicate $\texttt{str.substr}(s, 2, 3)$ where $s$ is a string variable can be rewritten to a simpler formula than the general case of $\texttt{str.substr}(s, i, n)$ where $i$ and $n$ are integer variables. On top of that, $\texttt{str.substr}(s, 2, 3)$ allows replacement of some lengths with regular constraints. Careful instantiation may significantly improve the performance on predicate-intensive formulae.

Performance of Z3-Noodler on Kaluza can in principle be improved also. The difficult string constraints we have inspected are rather simple straight-line constraints with many variables, unrestricted by regular constraints, that occur also in length constraints—a scenario in which our general algorithm is weak as it must generate many noodles. Solving these constraints however can be broken down into enumerating shuffles of two or several short words where each letter corresponds to either a literal or an unrestricted segment. Knowing the length of the literals, each case can be easily converted to a length constraint. The examples we tried can be quickly solved this way on a whiteboard. Generally, Z3-Noodler can be combined with a plethora of techniques used in other mature solvers as cvc5 or Z3Str4/3RE, such as pruning the solutions of equations and regular constraints through input length constraints and their models, approximation using lengths, etc. Implementing such heuristics and fine tuning the tool is a part of our future work.

## 9 RELATED WORK

Approaches and tools for string solving are numerous and diverse, with various representations of constraints, algorithms, or sorts of inputs. Many approaches use automata, e.g., Stranger [Yu et al. 2010, 2014, 2011], Norn [Abdulla et al. 2014, 2015], OSTRICH [Chen et al. 2018, 2022, 2020a, 2019; Lin and Barceló 2016], Trau [Abdulla et al. 2021, 2017, 2018, 2019], Sloth [Holík et al. 2018], Slog [Wang et al. 2016], Slent [Wang et al. 2018], Z3str3RE [Berzish et al. 2023, 2021], Retro [Chen et al. 2020b, 2023b], ABC [Aydin et al. 2015; Bultan et al. [n. d.]], Qzy [Cox and Leasure 2017], BEK [Hooimeijer et al. 2011], or [Zhu et al. 2019]. Around word equations are centered tools such as cvc4/5 [Barrett et al. 2016b; Liang et al. 2014, 2016, 2015; Nötzli et al. 2022; Reynolds et al. 2020, 2017], Z3 [Bjørner et al. 2009; de Moura and Bjørner 2008], S3 [Trinh et al. 2014], $\texttt{Kepler}_{22}$ [Le and He 2018], StrSolve [Hooimeijer and Weimer 2012], Woorpje [Day et al. 2019]; bit vectors are (among other things) used in Z3Str/2/3/4 [Berzish et al. 2017; Mora et al. 2021; Zheng et al. 2015, 2013], HAMPI [Kiezun et al. 2012]; PASS uses arrays [Li and Ghosh 2013]; G-strings [Amadini et al. 2017] and GECODE+S [Scott et al. 2017] use a SAT solver. With regard to equations and regular constraints, the fragment of chain-free constraints [Abdulla et al. 2019], which we extend, handled also by Trau, is the largest for which any string solver offers formal completeness guarantees, with the exception of quadratic equations, handled, e.g., by [Chen et al. 2020b, 2023b; Le and He 2018], which are incomparable but of a smaller practical relevance (although some tools implement Nielsen's algorithm [Nielsen 1917] to handle simple quadratic cases). The other solvers guarantee completeness on smaller fragments, notably that of OSTRICH (straight-line), Norn, and Z3str3RE; or use incomplete heuristics that work in practice (giving up guarantees of termination, over-/under-approximating by various means). Most string solvers tend to avoid handling regular expressions by postponing them as much as possible or abstracting them into arithmetic/length and other constraints (e.g., Trau, Z3str3RE, Z3str4, cvc4/5, S3). An important point of our work is that taking the opposite approach may work even better if automata are approached from the right angle and implemented carefully, although heuristics that utilise length information or Parikh images would probably speed up our algorithm too.

The algorithm of [Blahoudek et al. 2023], which we extend with lengths, is an improvement of the automata-based algorithm first proposed in [Abdulla et al. 2014], which is, at least in part, used as the basis of several string solvers, namely, Norn [Abdulla et al. 2014, 2015, 2019], Trau [Abdulla et al. 2020, 2021, 2017, 2018], OSTRICH [Chen et al. 2018, 2019; Lin and Barceló 2016], and Z3str3RE [Berzish et al. 2023, 2021]. The original algorithm first transforms equations to the disjunction of their solved forms [Ganesh et al. 2012] through generating alignments of variable boundaries on the equation sides (essentially an incomplete version of Makanin's algorithm). Second, it eliminates concatenation from regular constraints by *automata splitting*. Trau uses this algorithm within its unsatisfiability check. Trau's main solution finding algorithm also performs a step similar to our refinement, though with languages underapproximated as arithmetic formulae (representing Parikh images of the languages). Sloth [Holík et al. 2018] implements a compact version of automata splitting through alternating automata. OSTRICH has a way of avoiding the variable boundary alignment for the straight-line formulae, but still uses it for formulae outside the fragment. Z3str3RE optimises the algorithm of [Abdulla et al. 2014] heavily by the use of length-aware heuristics.

The approaches descending from equation alignment and automata splitting generally derive lengths from regular/transducer constraints by means of deterministic lasso automata or Parikh image construction [Abdulla et al. 2014; Esparza 1997; Parikh 1966]. Z3str3RE [Berzish et al. 2021] improves this by a fine-grained integration of reasoning about lengths with reasoning about

equations and regular properties, utilising the information from input length constraints to prune automata constructions.

Z3str4 also extensively uses heuristics that reduce the decision problem to reasoning about lengths, including approximation and concrete length assignments. cvc5/cvc4 handles length constraints in the congruence-based approach. Their algorithm eagerly builds equivalence classes of length terms and uses a LIA solver to detect inconsistencies (yielding UNSAT). Length equivalence classes are saturated during the computation by length axioms and other consequences derived by rules handling string (dis)equalities [Liang et al. 2014; Nötzli et al. 2022]. The information about Z3's string solver is limited, but to the best of our knowledge, it also uses principles behind congruence-based reasoning with a similar handling of lengths, based partially on an earlier solver S3 [Trinh et al. 2014] (descendant of Z3-str [Zheng et al. 2013]). The recent work [Stanford et al. 2021] enriches Z3 with techniques of symbolic derivatives of regular expressions and alternating/boolean automata to handle combinations of regular properties. Slent [Wang et al. 2016] reduces string constraints to alternating automata and tests their language emptiness by model checking tools. Slent is extended for lengths in [Wang et al. 2018], which augments the automata with counting on transitions. The approach is interesting, but the tool does not take SMT-LIB format on the input and so we were not able to include it into the comparison.

Altogether, a comprehensive overview of techniques would be difficult to provide due to the width of and the fast pace of evolution in the field. The mentioned solvers use plethora of heuristics and implementation techniques, accumulated over long lines of publications, which interact and influence each other. A comparison beyond the overall performance of the tools would be difficult.

## 10 CONCLUSIONS

We have presented a new string solving algorithm that generalises a recent algorithm for solving equations and regular constraints with reasoning about string lengths and other extensions. Implementing the approach in Z3, we have obtained a string solver that can compete with the best industrial-strength solvers on established benchmarks. The solver is significantly better than all other tools on several benchmark sets from SMT-COMP and on a benchmark used recently to evaluate techniques specialising on combinations of regular constraints. In the context of years of development and stacks of publications leading to the current state of the other solvers, our tool is still an infant. There clearly are many possibilities for optimization. Techniques that leverage reasoning about lengths used in other tools can most probably be adapted to our framework, the implementation of non-deterministic automata can be optimized, very promising are possibilities of using automata with registers to integrate reasoning about lengths directly with automata algorithms, or represent compactly disjunctions of noodles. We will also work on integrating the so-far unsupported string constraints into the framework.

## DATA-AVAILABILITY STATEMENT

An environment with the tools and data used for the experimental evaluation in the current study is available at [Chen et al. 2023a].

## ACKNOWLEDGEMENT

We thank the anonymous reviewers of the paper and the artifact on their comments on how to improve the quality of the paper (and the artifact) and we thank Michal Hečko for his help with the artifact's testing. This work was supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, the Czech Science Foundation project GA23-07565S, the FIT BUT internal project FIT-S-23-8151, and the project 109-2628-E-001-001-MY3 from National Science and Technology Council, Taiwan.

## REFERENCES


Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Wei-Cheng Wu. 2020. Efficient handling of string-number conversion. In *Proc. of PLDI'20*. ACM, 943–957. https://doi.org/10.1145/3385412

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Denghang Hu, Wei-Lun Tsai, Zhilin Wu, and Di-De Yen. 2021. Solving not-substring constraint with flat abstraction. In *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings (Lecture Notes in Computer Science)*, Hakjoo Oh (Ed.), Vol. 13008. Springer, 305–320. https://doi.org/10.1007/978-3-030-89051-3_17

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. 2017. Flatten and conquer: A framework for efficient analysis of string constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 602–617. https://doi.org/10.1145/3062341.3062384

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. 2018. Trau: SMT solver for string constraints. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj S. Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–5. https://doi.org/10.23919/FMCAD.2018.8602997

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String constraints for verification. In *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)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 150–166. https://doi.org/10.1007/978-3-319-08867-9_10

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT solver for string constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 462–469. https://doi.org/10.1007/978-3-319-21690-4_29

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukáš Holík, and Petr Janků. 2019. Chain-Free String Constraints. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.), Vol. 11781. Springer, 277–293. https://doi.org/10.1007/978-3-030-31784-3_16

Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. 2022. SolCMC: Solidity compiler's model checker. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science)*, Sharon Shoham and Yakir Vizel (Eds.), Vol. 13371. Springer, 325–338. https://doi.org/10.1007/978-3-031-13185-1_16

Roberto Amadini, Graeme Gange, Peter J. Stuckey, and Guido Tack. 2017. A novel approach to string constraint solving. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings (Lecture Notes in Computer Science)*, J. Christopher Beck (Ed.), Vol. 10416. Springer, 3–20. https://doi.org/10.1007/978-3-319-66158-2_1

Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-based model counting for string constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 255–272. https://doi.org/10.1007/978-3-319-21690-4_15

John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based automated reasoning for AWS access policies using SMT. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj S. Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–9. https://doi.org/10.23919/FMCAD.2018.8602994

Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. 40 (2015), 26–41. https://doi.org/10.3233/978-1-61499-495-4-26

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science)*, Dana Fisman and Grigore Rosu (Eds.), Vol. 13243. Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Pablo Barceló and Pablo Muñoz. 2017. Graph Logics with Rational Relations: The Role of Word Combinatorics. *ACM Trans. Comput. Log.* 18, 2 (2017), 10:1–10:41. https://doi.org/10.1145/3070822

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016a. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org.

Clark W. Barrett, Cesare Tinelli, Morgan Deters, Tianyi Liang, Andrew Reynolds, and Nestan Tsiskaridze. 2016b. Efficient solving of string constraints for security analysis. In *Proceedings of the Symposium and Bootcamp on the Science of Security, Pittsburgh, PA, USA, April 19-21, 2016*, William L. Scherlis and David Brumley (Eds.). ACM, 4–6. https://doi.org/10.1145/2898375.2898393

Murphy Berzish, Joel D. Day, Vijay Ganesh, Mitja Kulczynski, Florin Manea, Federico Mora, and Dirk Nowotka. 2023. Towards More Efficient Methods for Solving Regular-expression Heavy String Constraints. *Theor. Comput. Sci.* 943 (2023), 50–72. https://doi.org/10.1016/j.tcs.2022.12.009

Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 55–59. https://doi.org/10.23919/FMCAD.2017.8102241

Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. 2021. An SMT solver for regular expressions and linear arithmetic over string length. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science)*, Alexandra Silva and K. Rustan M. Leino (Eds.), Vol. 12760. Springer, 289–312. https://doi.org/10.1007/978-3-030-81688-9_14

Berzish, Murphy. 2021. *Z3str4: A Solver for Theories over Strings.* Ph.D. Dissertation. http://hdl.handle.net/10012/17102

Nikolaj S. Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Kowalewski and Anna Philippou (Eds.), Vol. 5505. Springer, 307–321. https://doi.org/10.1007/978-3-642-00768-2_27

František Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2023. Word equations in synergy with regular constraints. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (Lecture Notes in Computer Science)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.), Vol. 14000. Springer, 403–423. https://doi.org/10.1007/978-3-031-27481-7_23

Tevfik Bultan et al. [n. d.]. ABC string solver. https://github.com/vlab-cs-ucsb/ABC

Doron Bustan and Orna Grumberg. 2003. Simulation-based Minimization. *ACM Trans. Comput. Log.* 4, 2 (2003), 181–206. https://doi.org/10.1145/635499.635502

Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2018. What is decidable about string constraints with the ReplaceAll function. *Proc. ACM Program. Lang.* 2, POPL (2018), 3:1–3:29. https://doi.org/10.1145/3158091

Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2022. Solving string constraints with Regex-dependent functions through transducers with priorities and variables. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498707

Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020a. A decision procedure for path feasibility of string manipulating programs with integer data type. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings (Lecture Notes in Computer Science)*, Dang Van Hung and Oleg Sokolsky (Eds.), Vol. 12302. Springer, 325–342. https://doi.org/10.1007/978-3-030-59152-6_18

Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.* 3, POPL (2019), 49:1–49:30. https://doi.org/10.1145/3290362

Ting Chen, Xiao-song Zhang, Rui-dong Chen, Bo Yang, and Yang Bai. 2014. Conpy: Concolic execution engine for Python applications. In *Algorithms and Architectures for Parallel Processing - 14th International Conference, ICA3PP 2014, Dalian, China, August 24-27, 2014. Proceedings, Part II (Lecture Notes in Computer Science)*, Xian-He Sun, Wenyu Qu, Ivan Stojmenovic, Wanlei Zhou, Zhiyang Li, Hua Guo, Geyong Min, Tingting Yang, Yulei Wu, and Lei (Chris) Liu (Eds.), Vol. 8631. Springer, 150–163. https://doi.org/10.1007/978-3-319-11194-0_12

Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. 2020b. A symbolic algorithm for the case-split rule in string constraint solving. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science)*, Bruno C. d. S. Oliveira (Ed.), Vol. 12470. Springer, 343–363. https://doi.org/10.1007/978-3-030-64437-6_18

Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2023a. Artifact for the OOPSLA'23 paper "Solving String Constraints with Lengths by Stabilization". https://doi.org/10.5281/zenodo.8289595

Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. 2023b. A symbolic algorithm for the case-split rule in solving word constraints with extensions. *Journal of Systems and Software* 201 (2023), 111673. https://doi.org/10.1016/j.jss.2023.111673

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise analysis of string expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot (Ed.), Vol. 2694. Springer, 1–18. https://doi.org/10.1007/3-540-44898-5_1

Arlen Cox and Jason Leasure. 2017. Model Checking Regular Language Constraints. *CoRR* abs/1708.09073 (2017). arXiv:1708.09073 http://arxiv.org/abs/1708.09073

Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. 2019. On solving word equations using SAT. In *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings (Lecture Notes in Computer Science)*, Emmanuel Filiot, Raphaël M. Jungers, and Igor Potapov (Eds.), Vol. 11674. Springer, 93–106. https://doi.org/10.1007/978-3-030-30806-3_8

Joel D. Day, Vijay Ganesh, Paul He, Florin Manea, and Dirk Nowotka. 2018. The Satisfiability of Extended Word Equations: The Boundary Between Decidability and Undecidability. *CoRR* abs/1802.00523 (2018). arXiv:1802.00523 http://arxiv.org/abs/1802.00523

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Javier Esparza. 1997. Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes. *Fundam. Informaticae* 31, 1 (1997), 13–25. https://doi.org/10.3233/FI-1997-3112

Tomáš Fiedor, Lukáš Holík, Martin Hruška, Adam Rogalewicz, Juraj Síč, and Pavol Vargovčík. 2023. Reasoning about regular properties: A comparative study. In *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings (Lecture Notes in Computer Science)*, Brigitte Pientka and Cesare Tinelli (Eds.), Vol. 14132. Springer, Cham, 286–306. https://doi.org/10.1007/978-3-031-38499-8_17

Dominik D. Freydenberger and Liat Peterfreund. 2021. The theory of concatenation over finite models. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference) (LIPIcs)*, Nikhil Bansal, Emanuela Merelli, and James Worrell (Eds.), Vol. 198. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 130:1–130:17. https://doi.org/10.4230/LIPIcs.ICALP.2021.130

Xiang Fu and Chung-Chih Li. 2010. Modeling regular replacement for string constraint solving. In *Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings (NASA Conference Proceedings)*, César A. Muñoz (Ed.), Vol. NASA/CP-2010-216215. 67–76.

Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. 2012. Word equations with length constraints: What's decidable?. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers (Lecture Notes in Computer Science)*, Armin Biere, Amir Nahir, and Tanja E. J. Vos (Eds.), Vol. 7857. Springer, 209–226. https://doi.org/10.1007/978-3-642-39611-3_21

Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. 2018. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.* 2, POPL (2018), 4:1–4:32. https://doi.org/10.1145/3158092

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association. http://static.usenix.org/events/sec11/tech/full_papers/Hooimeijer.pdf

Pieter Hooimeijer and Westley Weimer. 2012. StrSolve: solving string constraints lazily. *Autom. Softw. Eng.* 19, 4 (2012), 531–559. https://doi.org/10.1007/s10515-012-0111-x

Artur Jeż. 2016. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM* 63, 1, Article 4 (feb 2016), 51 pages. https://doi.org/10.1145/2743014

Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2012. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4 (2012), 25:1–25:28. https://doi.org/10.1145/2377656.2377662

Quang Loc Le and Mengda He. 2018. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science)*, Sukyoung Ryu (Ed.), Vol. 11275. Springer, 350–372. https://doi.org/10.1007/978-3-030-02768-1_19

Guodong Li and Indradeep Ghosh. 2013. PASS: String solving with parameterized array and interval automaton. In *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings (Lecture Notes in Computer Science)*, Valeria Bertacco and Axel Legay (Eds.), Vol. 8244. Springer, 15–31. https://doi.org/10.1007/978-3-319-03077-7_2

Liana Hadarean. 2019. String Solving at Amazon. https://mosca19.github.io/program/index.html. Presented at MOSCA'19.

Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. 2014. A DPLL(T) theory solver for a theory of strings and regular expressions. In *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)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 646–662. https://doi.org/10.1007/978-3-319-08867-9_43

Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. 2016. An efficient SMT solver for string constraints. *Formal Methods Syst. Des.* 48, 3 (2016), 206–234. https://doi.org/10.1007/s10703-016-0247-6

Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark W. Barrett. 2015. A decision procedure for regular membership and length constraints over unbounded strings. In *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings (Lecture Notes in Computer Science)*, Carsten Lutz and Silvio Ranise (Eds.), Vol. 9322. Springer, 135–150. https://doi.org/10.1007/978-3-319-24246-0_9

Anthony Widjaja Lin and Pablo Barceló. 2016. String solving with word equations and transducers: Towards a logic for analysing mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 123–136. https://doi.org/10.1145/2837614.2837641

Anthony W. Lin and Rupak Majumdar. 2021. 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

G. S. Makanin. 1977. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik* 32, 2 (1977), 147–236. (in Russian).

Microsoft. 2020. Azure Resource Manager documentation. https://docs.microsoft.com/en-us/azure/azure-resource-manager/

Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. 2021. Z3str4: A Multi-armed string solver. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.), Vol. 13047. Springer, 389–406. https://doi.org/10.1007/978-3-030-90870-6_21

Jakob Nielsen. 1917. Die Isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. *Math. Ann.* 78, 1 (1917), 385–397.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM* 53, 6 (nov 2006), 937–977. https://doi.org/10.1145/1217856.1217859

Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Clark W. Barrett, and Cesare Tinelli. 2022. Even faster conflicts and lazier reductions for string solvers. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II (Lecture Notes in Computer Science)*, Sharon Shoham and Yakir Vizel (Eds.), Vol. 13372. Springer, 205–226. https://doi.org/10.1007/978-3-031-13188-2_11

OWASP. 2013. Top 10. https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf.

OWASP. 2017. Top 10. https://owasp.org/www-project-top-ten/2017/.

OWASP. 2021. Top 10. https://owasp.org/Top10/.

Rohit J. Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (oct 1966), 570–581. https://doi.org/10.1145/321356.321364

Wojciech Plandowski. 1999. Satisfiability of word equations with constants is in NEXPTIME. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC '99)*. Association for Computing Machinery, New York, NY, USA, 721–725. https://doi.org/10.1145/301250.301443

Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019. High-level abstractions for simplifying extended string constraints in SMT. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11562. Springer, 23–42. https://doi.org/10.1007/978-3-030-25543-5_2

Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2020. Reductions for strings and regular expressions revisited. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 225–235. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_30

Andrew Reynolds, Maverick Woo, Clark W. Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling up DPLL(T) string solvers using context-dependent simplification. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 453–474. https://doi.org/10.1007/978-3-319-63390-9_24

Neha Rungta. 2022. A billion SMT queries a day (invited paper). In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science)*, Sharon Shoham and Yakir Vizel (Eds.), Vol. 13371. Springer, 3–18. https://doi.org/10.1007/978-3-031-13185-1_1

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 513–528. https://doi.org/10.1109/SP.2010.38

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2023. Kaluza web site. https://webblaze.cs.berkeley.edu/2010/kaluza/

Joseph D. Scott, Pierre Flener, Justin Pearson, and Christian Schulte. 2017. Design and implementation of bounded-length sequence variables. In *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings (Lecture Notes in Computer Science)*, Domenico Salvagnin and Michele Lombardi (Eds.), Vol. 10335. Springer, 51–67. https://doi.org/10.1007/978-3-319-59776-8_5

SMT-COMP'22. 2022. https://smt-comp.github.io/2022/.

SMTLib. 2023a. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S.

SMTLib. 2023b. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA.

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 620–635. https://doi.org/10.1145/3453483.3454066

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1232–1243. https://doi.org/10.1145/2660267.2660372

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. Progressive reasoning over recursively-defined strings. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, 218–240. https://doi.org/10.1007/978-3-319-41528-4_12

Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj S. Bjørner. 2012. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 137–150. https://doi.org/10.1145/2103656.2103674

Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. 2016. String analysis via automata manipulation with logic circuit representation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, 241–260. https://doi.org/10.1007/978-3-319-41528-4_13

Hung-En Wang, Shih-Yu Chen, Fang Yu, and Jie-Hong R. Jiang. 2018. A symbolic model checking approach to the analysis of string and length constraints. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 623–633. https://doi.org/10.1145/3238147.3238189

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. Stranger: An automata-based string analysis tool for PHP. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Javier Esparza and Rupak Majumdar (Eds.), Vol. 6015. Springer, 154–157. https://doi.org/10.1007/978-3-642-12002-2_13

Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods Syst. Des.* 44, 1 (2014), 44–70. https://doi.org/10.1007/s10703-013-0189-1

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. 2011. Relational String Verification Using Multi-Track Automata. *Int. J. Found. Comput. Sci.* 22, 8 (2011), 1909–1924. https://doi.org/10.1142/S0129054111009112

Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. 2015. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 235–254. https://doi.org/10.1007/978-3-319-21690-4_14

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based string solver for web application analysis. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 114–124. https://doi.org/10.1145/2491411.2491456

Qizhen Zhu, Hitoshi Akama, and Yasuhiko Minamide. 2019. Solving String Constraints with Streaming String Transducers. *J. Inf. Process.* 27 (2019), 810–821. https://doi.org/10.2197/ipsjjip.27.810

Received 2023-04-14; accepted 2023-08-27