A Uniform Framework for Handling Position Constraints in String Solving

YU-FANG CHEN, Academia Sinica, Taiwan
VOJTĚCH HAVLENA, Brno University of Technology, Czech Republic
MICHAL HEČKO, Brno University of Technology, Czech Republic
LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic and Aalborg University, Denmark
ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic

We introduce a novel decision procedure for solving the class of *position string constraints*, which includes string disequalities, ¬*prefixof*, ¬*suffixof*, *str.at*, and ¬*str.at*. These constraints are generated frequently in almost any application of string constraint solving. Our procedure avoids expensive encoding of the constraints to word equations and, instead, reduces the problem to checking conflicts on positions satisfying an integer constraint obtained from the Parikh image of a polynomial-sized finite automaton with a special structure. By the reduction to counting, solving position constraints becomes NP-complete and for some cases even falls into PTime. This is much cheaper than the previously used techniques, which either used reductions generating word equations and length constraints (for which modern string solvers use exponential-space algorithms) or incomplete techniques. Our method is relevant especially for automata-based string solvers, which have recently achieved the best results in terms of practical efficiency, generality, and completeness guarantees. This work allows them to excel also on position constraints, which used to be their weakness. Besides the efficiency gains, we show that our framework may be extended to solve a large fragment of ¬*contains* (in NExpTime), for which decidability has been long open, and gives a hope to solve the general problem. Our implementation of the technique within the Z3-Noodler solver significantly improves its performance on position constraints.

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

Additional Key Words and Phrases: string constraints, stabilization, word equations, SMT solving, disequality, length constraints, regular languages, monadic decomposition, not contains

ACM Reference Format:
Yu-Fang Chen, Vojtěch Havlena, Michal Hečko, Lukáš Holík, and Ondřej Lengál. 2025. A Uniform Framework for Handling Position Constraints in String Solving. *Proc. ACM Program. Lang.* 9, PLDI, Article 169 (June 2025), 26 pages. https://doi.org/10.1145/3729273

1 Introduction

Solving string constraints (string solving) has been motivated initially by the analysis of string manipulations in programs, especially in preventing security risks such as cross-site scripting or SQL injection in web-applications. In the last two decades, the lively research community has managed to develop a number of string constraints solvers (overviewed in Sec. 9) that may be

Authors' Contact Information: Yu-Fang Chen, Academia Sinica, Taipei, Taiwan, yfc@iis.sinica.edu.tw; Vojtěch Havlena, Brno University of Technology, Brno, Czech Republic, ihavlena@fit.vutbr.cz; Michal Hečko, Brno University of Technology, Brno, Czech Republic, ihecko@fit.vutbr.cz; Lukáš Holík, Brno University of Technology, Brno, Czech Republic and Aalborg University, Aalborg, Denmark, holik@fit.vutbr.cz; Ondřej Lengál, Brno University of Technology, Brno, Czech Republic, lengal@fit.vutbr.cz.

ACM 2475-1421/2025/6-ART169
https://doi.org/10.1145/3729273

capable of realizing this goal. String constraints solvers are being integrated within SMT solvers such as Z3, CVC4/5, or Princess [11, 33, 51, 69], and the string category has been introduced in the SMT competition [1]. Since string solving is ubiquitous and string constraint logics are general, string solving keeps finding also new applications, such as analysing Simulink models [42], verifying UML models [45], or checking cloud access policies at Amazon Web Services [70].

The competition of string-solving methods is lively. While it was long dominated by the strongest industrial grade SMT-solvers Z3 and CVC4/5, the leading positions have been recently taken by approaches based on finite automata. They include Z3-Noodler [24], a recent winner of string categories of SMT-COMP [2], OSTRICH [20, 22], which supports the richest palette of string constraints with strong completeness guarantees, Z3str3RE [14, 16], and loosely also one of the engines of Z3 [72]. Automata-based solvers excel especially in handling complex regular constraints with word equations and related constraints, such as transducer constraints or ReplaceAll.

Position constraints. In this paper, we aim at remedying a weakness of automata-based techniques, which is handling a class of constraints that we call *position constraints*. The so-called *existential* position constraints can be reduced to checking disequality of letters at two specific positions in strings (called a *mismatch*), where the positions are determined through measuring lengths of certain sub-strings—i.e., counting their letters and comparing the counts. The prime example of such constraints are string disequalities (negated word equations) like $xyx \neq yxz$, which can be reduced to the existence of mismatching positions in the two strings that are at the same distance from the strings' start.[1] Other existential position constraints include $\neg prefixof$, $\neg suffixof$, $str.at$, and $\neg str.at$. A special place among position constraints is occupied by $\neg contains(t_x, t_y)$ where t_x and t_y are concatenations of variables, which does not reduce to a simple existence of mismatching positions (i.e., it is not an existential constraints), but is equivalent to a string formula with *quantifier alternation*, much harder to solve than existential formulae. Informally, the formula says that there exists a string assignment to variables such that for the two strings w_x and w_y obtained by concatenating assignments of the variables in t_x and t_y, it holds that for all alignments of the start of w_x inside w_y, the letter at some position of w_x does not match the letter at the aligned position in w_y. The formula falls into the $\forall\exists$ fragment of the string theory, which is undecidable [35, 58]; the decidability of $\neg contains$ is a known open problem [3]. Solvers use heuristics that handle only very simple cases or rough approximations.

Position constraints are practically relevant, for instance, a disequality may be generated in symbolic execution at every else-branch of a program that tests the equality of strings. Some solvers may be able to guess the right solution for satisfiable position constraints with ease (CVC5 excels in this), but especially unsatisfiable position constraints may be much harder; no current solver can handle them satisfactorily.

The automata-based approach particularly reduces position constraints into combinations of equations and length constraints. The work [23] even shows a version of this reduction that allows one to freely add disequalities to one of the largest known decidable fragments of basic constraints (word equations, regular membership, and string length constraints), the *chain-free fragment* [8], while preserving decidability. However, since equations are expensive, the price of the reduction may be very high. Indeed, essentially all string solvers use exponential-space algorithms to deal with word equations, including the automata-based ones, and the problem is opaque even in theory: equations with regular memberships are in PSpace [44, 64], but decidability of their combination with length constraints is a long-standing open problem.

[1] On a high-level, we might write such a formula as $\exists i \in \mathbb{N}\colon i \leq \max(len(xyx), len(yxz)) \wedge (xyx)[i] \neq (yxz)[i]$ where len denotes the length of the corresponding string obtained by substituting into the variables and $t[i]$ denotes the i-th position in the string given by the term t.

The gist of our approach to solving position constraints. In contrast, the procedure we propose in this paper starts only *after* the rest of the constraint is solved—transformed into the *monadic decomposition* [22, 38, 74], i.e., a formula obtained by transforming word equations into regular constraints—and then solves position constraints quickly and efficiently by other means. The specific technical problem we are solving is therefore *satisfiability of Monadic-Position constraints*:

(MP) Satisfiability of a quantifier-free conjunction of a monadic constraint (a conjunction of regular membership constraints and linear integer arithmetic (LIA) constraints over string lengths, a.k.a. length constraints) and a conjunction of position constraints with string terms as parameters (e.g., $xyx \neq yxy$, $\neg contains(xyx, yxy)$).

Monadic decomposition is at the heart of how automata-based solvers, such as Z3-Noodler and OSTRICH, solve string constraints. The fact that position constraints can be ignored in this process is the main distinguishing feature of our approach and the key to its efficiency. Translating the entire MP to a monadic constraint first, as automata-based solvers currently do, takes *exponential* space, while our algorithm runs in *NP for existential position constraints* and in *PTime for a single one*.

Moreover, our framework also allows to make a step towards showing decidability of MP with $\neg contains$. Namely, it allows to translate $\neg contains(t, t')$ combined with a monadic constraint to LIA with a nested universal quantifier when the languages constraining variables in terms t and t' are *flat* (expressible as a concatenation of a fixed number of parts, where every part is an iteration of a single word). The resulting quantified LIA formula can be solved by an off-the-shelf solver (SMT-solvers seem to be capable of solving the obtained formulae efficiently). To the best of our knowledge, our algorithm is the first one for exact reasoning with $\neg contains$ that is complete for a large and interesting fragment of the problem.

The technical basis of our approach. Technically, given a set of position constraints and automata-represented regular membership constraints in the monadic decomposition, our procedure derives a LIA constraint that relates positions of mismatches and lengths of the strings assigned to variables. Repetition of variables, e.g., $xyx \neq yxy$ or $\neg contains(xyx, yxy)$, the main limiting factor of decidable fragments, is handled too: The length of every variable is extracted from a single run of its automaton, and its contribution to a mismatch position in the string is counted with the multiplicity equal to the number of the occurrences of the variable that precede the mismatch position. For a single existential position constraint, the LIA formula is constructed by taking the formula for the Parikh image of an automaton with a specific structure, enriched with constraints that enforce seeing a mismatch of symbols at at the proper positions.

The construction is more complex with multiple position constraints that share variables. The straightforward approach would be to enumerate an exponential number of cases corresponding to all orders of mismatches in the position constraints. E.g., for the constraint $D_a \wedge D_b \wedge D_c$, we would need to consider orders like $(D_a^1, D_a^2, D_b^1, D_b^2, D_c^1, D_c^2)$, $(D_a^1, D_b^1, D_c^1, D_a^2, D_c^2, D_b^2)$, etc., where D_x^i denotes the i-th mismatch in D_x. The generated LIA formula would then be exponential (more precisely in $2^{\Theta(n \log n)}$) to the number of position constraints. Our NP algorithm for MP depends on the discovery of an equivalent polynomial encoding. The encoding combines the Parikh image of a polynomial-size automaton that generates any number of mismatch positions in any order with an additional arithmetic constraint that rules out the "wrong" orderings.

Moreover, we show that MP with a *single* existential position constraint is in PTime by a reduction to 0-reachability in a one-counter automaton [9, 59].

Practical impact on performance. Our experiments show that our framework substantially improves the speed and effectiveness of the automata approach to string constraint solving whenever position constraints are involved. On position-heavy cases from our benchmark (obtained from symbolic executions of large software projects), the string solver Z3-Noodler extended with our

decision procedure for position constraints is the fastest of all solvers and has less timeouts than CVC5, the only other solver that handles these benchmarks well. The performance of CVC5 and the modified Z3-Noodler is orthogonal, reflecting the large difference between the two approaches, and together they solve all but 10 out of the ∼150,000 benchmarks. Z3-Noodler also demonstrates its ability to solve hard instances of ¬*contains* on an artificial benchmark made to test solvers on constraints involving ¬*contains*, where all other solvers fail.

Summary of contributions. Our contributions can be summarised as follows:

(1) We propose a procedure for handling position constraints in the automata-based approach efficiently, without the need of including them into a procedure for monadic decomposition.
(2) We show that MP is in NP for existential position constraints and in PTime for a single one. This contrasts with the exponential cost of monadic decomposition.
(3) We propose the first algorithm for exact reasoning with ¬*contains* that is guaranteed to work on a large and interesting fragment of the problem, namely, the MP where the regular languages restricting variables within the terms in the arguments of ¬*contains* are *flat*, and show that the problem is in NExpTime.
(4) Experimental results show that our techniques significantly improve the performance of one of the fastest string solvers, Z3-Noodler, on position constraints.

Context of our work. Our focus is on the basic constraints that must be handled by a universal practical string solver: combinations of word equations, regular constraints, and length constraints (as witnessed, e.g., by the SMT-COMP benchmark [65]). Besides the PSpace-completeness of classical decidability of equations with regular constraints by Makanin, Plandowski, and Jeż [44, 57, 64], decidability of the full basic logic is a long-standing open problem, and it is open even when equations are quadratic, where only two occurrences of every variable are allowed [55]. The known undecidability results are concerned with extensions of this logic with other than basic constraints (here called extended constraints), and are thus only marginally relevant to our current work. These fragments concern unrestricted ReplaceAll, transducer-defined relations, string-integer conversions, and other extended constraints [19–22, 31, 32, 54]. The approach that started at [6] led to a discovery of the largest decidable fragment of the basic constraints, the chain-free fragment [8]. It generalizes the earlier acyclic fragment of [54] and the larger straight-line fragment of [6]. The other solvers, especially Z3 and CVC5, which use different approaches usually revolving around the congruence closure [34], can handle similar class of constraints in practice, but do not come with strong theoretical guarantees. Limited extensions of straight-line or chain-free logics with extended constraints were shown decidable in [19–22, 31, 32, 54]. The automata-based approach transforms equations and regular constraints to a monadic decomposition (a disjunction of conjunctions of regular constraints of at most doubly exponential size), which is in turn transformed to a LIA formula and solved using a LIA solver. The decidable fragments are all based on prohibiting forms of "*cyclic dependencies*" of string variables in word equations, and range from PSpace-complete to 2-ExpSpace, depending on the allowed extended constraints. The position constraints that we discuss here are in this framework normally solved by a reduction to the basic constraints. Essentially, after obtaining doubly exponential monadic decomposition of the other basic constraints, the original approach needs to run the doubly exponential space procedure to solve the position constraints. Our work allows to replace the second 2-ExpSpace phase by an NP-algorithm (or NExpTime for our fragment of ¬*contains*), or even by a PTime one in the simplest case of one disequality.

2 Preliminaries

We fix a finite non-empty *alphabet* Γ. A *word* or *string* over Γ is a (finite) sequence of symbols $w = a_0 \dots a_{n-1}$ from Γ, its *length* $|w|$ is n, and the symbol at index $0 \leq i < n$ is denoted as $w[i] = a_i$.

$$
\begin{aligned}
\nu \models x_s \in L &\quad\Leftrightarrow \nu(x_s) \in L,\\
\nu \models t_i \leq t_i' &\quad\Leftrightarrow \nu(t_i) \leq \nu(t_i'),\\
\nu \models t_s = t_s' &\quad\Leftrightarrow \nu(t_s) = \nu(t_s'),\\
\nu \models x_s = str.at(t_s, t_i) &\quad\Leftrightarrow \begin{cases} \nu(x_s) = w_s[\nu(t_i)] \wedge \nu(t_s) = w_s & \text{if } \nu(t_i) < |w_s|,\\ \nu(x_s) = \epsilon & \text{otherwise,} \end{cases}\\
\nu \models prefixof(t_s, t_s') &\quad\Leftrightarrow \exists z_p \in \Gamma^* \colon \nu(t_s) \circ z_p = \nu(t_s'),\\
\nu \models suffixof(t_s, t_s') &\quad\Leftrightarrow \exists z_s \in \Gamma^* \colon z_s \circ \nu(t_s) = \nu(t_s'), \text{and}\\
\nu \models contains(t_s, t_s') &\quad\Leftrightarrow \exists z_c, z_c' \in \Gamma^* \colon z_c \circ \nu(t_s) \circ z_c' = \nu(t_s').
\end{aligned}
$$

Fig. 1. Semantics of atomic predicates for a variable assignment ν

We use ϵ to denote the *empty word* and $\circ$ denotes string concatenation (we sometimes omit the operator, i.e., $a \circ b = ab$). $\mathbb{N}$ denotes the set of natural numbers $\{0, 1, \ldots\}$.

A *nondeterministic finite automaton* (NFA) is a tuple $A = (Q, \Delta, I, F)$ where Q is a finite set of *states*, $\Delta \subseteq Q \times \Gamma \times Q$ is a *transition relation* with transitions denoted as $q \xrightarrow{a} p$ for $q, p \in Q$ and $a \in \Gamma$, and $I, F \subseteq Q$ are sets of *initial* and *final* states respectively. A *run* $\mathcal{R}$ of A over a word $w = a_1 \ldots a_n \in \Gamma^*$ is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n$ s.t. $q_0 \in I$ and $\forall 1 \leq i \leq n \colon q_{i-1} \xrightarrow{a_i} q_i \in \Delta$. The run $\mathcal{R}$ is *accepting* if $q_n \in F$, and the language of A is the set $L(A) = \{w \in \Gamma^* \mid \text{there exists an accepting run of } A \text{ over } w\}$. A language $L \subseteq \Gamma^*$ is *regular* iff there exists an NFA A such that $L = L(A)$. We sometimes define regular languages using regular expressions with the standard textbook notation.

Given a set U, let $\#U = \{\#u \mid u \in U\}$ denote the set of elements obtained from U's elements by prepending them with #. The *Parikh image* of a run $\mathcal{R}$, denoted as $PI_{\mathcal{R}}$, is a mapping $PI_{\mathcal{R}} \colon \#\Delta \to \mathbb{N}$ such that $PI_{\mathcal{R}}(\#t)$ denotes the number of occurrences of transition t in $\mathcal{R}$. We say that A is *flat*[2] if for every two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ it holds that if $PI_{\mathcal{R}_1} = PI_{\mathcal{R}_2}$, then $\mathcal{R}_1 = \mathcal{R}_2$. Structurally, this means that flat automata have the form of *directed acyclic graphs* (DAGs) connecting *simple* (i.e., non-nested) *loops*. We say that a regular language L is flat iff there exists a flat NFA A s.t. $L = L(A)$. For instance, the language $(ab)^*c((ab)^* + (ba)^*)$ is flat, while the language $(a + b)^*$ is not flat.

Let $\mathbb{X}$ and $\mathbb{I}$ be the sets of *string* and *integer variables*. String formulae are of the form:

$$
\begin{aligned}
\varphi &::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \varphi_{atom}\\
\varphi_{atom} &::= x_s \in L \mid t_s \leq t_i \mid t_s = t_s \mid x_s = str.at(t_s, t_i) \mid prefixof(t_s, t_s) \mid suffixof(t_s, t_s) \mid contains(t_s, t_s)\\
t_s &::= x_s \mid t_s \circ t_s\\
t_i &::= x_i \mid k \mid len(x_s) \mid t_i + t_i
\end{aligned}
$$

where φ_{atom} is an atomic formula, L is a regular language (given by a regular expression or an NFA), t_s is a string term consisting of a concatenation of string variables $x_s \in \mathbb{X}$[3], and t_i is an integer term composed of sums of integer variables x_i, integers $k \in \mathbb{Z}$, and lengths of string variables $len(x_s)$.

The semantics of formulae is defined as follows. A (variable) *assignment* is a mapping $\nu \colon (\mathbb{X} \to \Gamma^*) \cup (\mathbb{I} \to \mathbb{Z})$ and we lift ν to string terms t_s and integer terms t_i in the usual way ($t_s \circ t_s$ as string concatenation and $t_i + t_i$ as integer addition), with $\nu(len(x_s))$ being interpreted as the length of the string $\nu(x_s)$. Semantics of atomic predicates is given in Fig. 1.

[2]We note that our notion of *flatness* differs from the one from [4] and is similar to the one from [49].

[3]A string literal $u \in \Gamma^*$ can be encoded by a new string variable x_u and a regular constraint $u \in L_u$ for $L_u = \{u\}$.

When used within the DPLL(T) framework, it is sufficient to consider only conjunctions of atomic formulae and their negations. A *normal form* of such formula is $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{I} \wedge \mathcal{P}$ where

- $\mathcal{E}$ is a conjunction of word equations $t_s = t_s$,
- $\mathcal{R}$ is a conjunction of regular memberships $x_s \in L$[4] such that for every $x_s \in \mathbb{X}$, there is exactly one regular membership constraint in $\mathcal{R}$ (and we use $L(x_s)$ to denote it),
- $\mathcal{I}$ is a conjunction of integer constraints $t_i \leq t_i$ where t_i's do not contain $len(t_s)$ terms, and
- $\mathcal{P}$ is a conjunction of position constraints of the following form:

$$t_s \neq t_s \mid x_i = len(x_s) \mid x_s = str.at(t_s, t_i) \mid x_s \neq str.at(t_s, t_i) \mid$$
$$\neg prefixof(t_s, t_s) \mid \neg suffixof(t_s, t_s) \mid \neg contains(t_s, t_s)$$

We transform a conjunction of literals into the normal form in the following way: (i) We substitute every non-negated occurrence of the predicates $prefixof(u_p, v_p)$, $suffixof(u_s, v_s)$, and $contains(u_c, v_c)$ with the word equations $v_p = u_p z_p$, $v_s = z_s u_s$, and $v_s = z_c u_c z'_c$ respectively for fresh string variables z_p, z_s, z_c, and z'_c (note that we cannot do a similar thing for negated occurrences, since we would need to introduce universal quantifiers for the z-variables). (ii) For every string variable x, we compute a single NFA that represents all regular membership constraints for x. We will use $|\mathcal{R}|$ to denote the sum of numbers of states of all NFAs used for encoding the $\mathcal{R}$ constraint.

3 Overview

Given a formula in the normal form $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{I} \wedge \mathcal{P}$ in the considered fragment, the main idea of our approach is the following (focusing on the $\mathcal{P}$ part). Other automata-based approaches usually try to get rid of the predicates in $\mathcal{P}$ by transforming them into word equations and length constraints (e.g. [21, 23, 39]), thus making their word equations potentially much harder to process. Handling word equations is the most demanding task in string solving, as the best known algorithms for dealing with word equations work in PSpace [44, 64] and are not practical[5] (and in the presence of length constraints, the decidability of the problem is currently unknown).

In our approach, we take care of the constraints in $\mathcal{P}$ only after the word equations in $\mathcal{E}$ have been processed and the obtained constraint $\mathcal{R}' \wedge \mathcal{I}' \wedge \mathcal{P}'$ contains no more word equations. This is achieved by the stabilization-based procedure introduced in [23], which transforms $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{I}$ into a disjunction of constraints $\mathcal{R}' \wedge \mathcal{I}'$ extended with a substitution mapping variables from the original constraints to a concatenation of fresh variables occurring in $\mathcal{R}'$ and $\mathcal{I}'$. The transformation comes with the additional property that the resulting constraint is a *monadic decomposition* [22, 38, 74], which means that each choice of the fresh variable assignment given by $\mathcal{R}'$ forms a solution of the original system of equations (the substitution defines how to obtain values of variables occurring in $\mathcal{E}$ from the fresh variables)[6]. Using the substitution map, we can substitute variables in $\mathcal{P}$ in order to obtain a position constraint $\mathcal{R}' \wedge \mathcal{I}' \wedge \mathcal{P}'$. Therefore, we will now focus on solving formulae of the form $\mathcal{R}' \wedge \mathcal{I}' \wedge \mathcal{P}'$, which is the main contribution of this paper.

The main idea of solving a formula of the form $\mathcal{R}' \wedge \mathcal{I}' \wedge \mathcal{P}'$ is by transforming $\mathcal{R}' \wedge \mathcal{P}'$ into a LIA formula that is then added to $\mathcal{I}'$ to obtain a (potentially quantified) LIA constraint $\mathcal{I}''$, which can be solved by an off-the-shelf LIA solver. The procedure for transforming $\mathcal{R}' \wedge \mathcal{P}'$ into a LIA constraint is based on constructing a *tag automaton* A_{tag}, which is an NFA whose transitions are extended with *tags*. The tags do not affect the run of A_{tag}, but are used for counting "*positions*" where something interesting happens, e.g., for a string disequality $x \neq y$, we count the position ℓ

[4] A regular *non-membership* constraint can be translated into a *membership* constraint for a complement language.
[5] Existing string solvers usually deal with word equations by incomplete algorithms that do not guarantee termination.
[6] Strictly speaking, we only need monadic decomposition on the variables that occur in position constraints.

of a single character mismatch $x[\ell] \neq y[\ell]$. A_{tag} is constructed from the regular constraints in $\mathcal{R}'$ based on the atomic predicates in $\mathcal{P}'$.

4 Tag Automaton

In this section, we define *tag automata* (TAs) used in the later sections for encoding position constraints. We note that TAs are used just to simplify notation; one could build the framework on top of NFAs or some counter model, such as Parikh automata [47], cost-enriched finite automata [21], or simply vector addition systems with states [41], for the price of a more cumbersome notation.

Let $\mathbb{T}$ be a set of *tags*. A *tag automaton* (TA) over $\mathbb{T}$ is a quadruple $T = (Q, \Delta, I, F)$, where Q, I, F are as for an NFA and the set of transitions Δ is $\Delta \subseteq Q \times 2^{\mathbb{T}} \times Q$. We use $q \text{-}\{S\}\!\!\rightarrow p$ to denote a transition (q, S, p); we drop duplicate braces, so, e.g., for $S = \{a, b\}$, we write $q \text{-}\{a,b\}\!\!\rightarrow p$. A run $\mathcal{R}$ of T is a sequence of transitions $q_0 \text{-}\{S_1\}\!\!\rightarrow q_1 \text{-}\{S_2\}\!\!\rightarrow \cdots \text{-}\{S_n\}\!\!\rightarrow q_n$ such that $q_0 \in I$ and $q_{i-1} \text{-}\{S_i\}\!\!\rightarrow q_i \in \Delta$ for all $1 \leq i \leq n$. $\mathcal{R}$ is accepting if $q_n \in F$. The Parikh image of $\mathcal{R}$, $PI_{\mathcal{R}}$, is defined in the same way as for NFAs. We may write $Q(T)$, $\Delta(T)$, $I(T)$, and $F(T)$ to refer to the components of a TA T.

For an NFA $A = (Q, \Delta, I, F)$ and a variable x, we define the tag automaton $\mathsf{LenTag}_x(A) = (Q, \Delta', I, F)$ over a set of tags $\{\langle \mathbf{S}, a\rangle \mid a \in \Gamma\} \cup \{\langle \mathbf{L}, x\rangle\}$ where $\Delta' = \{q \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},x\rangle\}\!\!\rightarrow r \mid q \xrightarrow{a} r \in \Delta\}$. The used tags denote the **S**ymbol and **L**ength (i.e., we will use the number of occurrences of the **L** tag to derive the length of a word from the TA). Given two TAs $A = (Q_A, \Delta_A, I_A, F_A)$ and $B = (Q_B, \Delta_B, I_B, F_B)$ with disjoint sets of states, their ϵ-concatenation is the TA $A \circ_\epsilon B = (Q_A \cup Q_B, \Delta_A \cup \Delta_B \cup \Delta_\epsilon, I_A, F_B)$ with $\Delta_\epsilon = \{q \text{-}\{\}\!\!\rightarrow r \mid q \in F_A, r \in I_B\}$.

The *Parikh formula* of a TA T, denoted as $PF(T)$ is a *linear integer arithmetic* (LIA) formula with free variables $\#\delta$ corresponding to numbers of occurrences of transitions $\delta \in \Delta$. Models of $PF(T)$ are, therefore, assignments $\sigma\colon \#\Delta \to \mathbb{N}$ such that

$$\sigma \models PF(T) \quad \text{iff there is an accepting run } \mathcal{R} \text{ of } T \text{ s.t. } PI_{\mathcal{R}} = \sigma \tag{1}$$

Constructing $PF(T)$ can be done in the usual way [43] (cf. [28]). We will also work with the *Parikh tag formula* $PF_{tag}(T)$, which is a formula whose models are numbers of each tag seen on an accepting run in T, constructed as

$$PF_{tag}(T) \quad \overset{\text{def.}}{\Leftrightarrow} \quad PF(T) \wedge \bigwedge_{t \in \mathbb{T}} \#t = \sum\{\#\delta \in \#\Delta \mid \delta = q \text{-}\{S\}\!\!\rightarrow r \in \Delta, t \in S\}. \tag{2}$$

Note that in $PF_{tag}(T)$, the free variables are now also the counts of tags from $\mathbb{T}$ and a model is an assignment $\sigma'\colon (\#\Delta \cup \#\mathbb{T}) \to \mathbb{N}$.

5 Solving Disequalities

In this section, we will show how to solve a formula $\mathcal{R}' \wedge \mathcal{I} \wedge \mathcal{P}$ where $\mathcal{P}$ only contains disequalities. We will start from the simplest case (a single disequality with two different variables), proceed to an arbitrary single disequality, and finish with a system of disequalities.

5.1 I: A Single Disequality of Two Variables

First, we consider the simplest case, i.e., when the position constraint $\mathcal{P}$ contains a single disequality

$$x \neq y, \tag{3}$$

where x and y are two different string variables whose values are restricted to regular languages L_x and L_y. The constraint is satisfiable iff there exist strings $w_x \in L_x$ and $w_y \in L_y$ such that they are either (i) of a different length or (ii) they are of the same length ℓ and there exists a position $0 \leq p < \ell$ such that $w_x[p] \neq w_y[p]$.

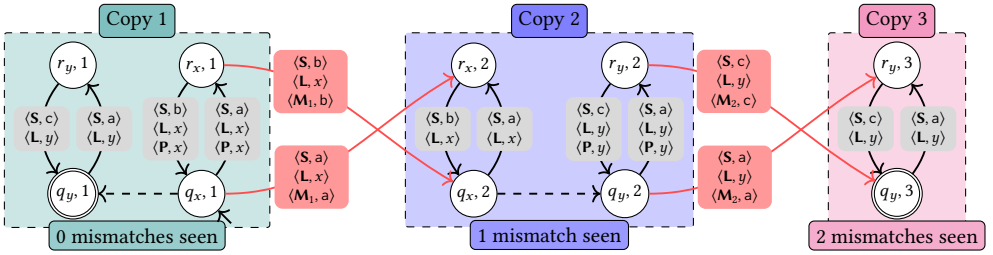


Fig. 2. Example of a tag automaton for the disequality $x \neq y$ with $L(A_x) = (ab)^*$ and $L(A_y) = (ac)^*$. States q_x, r_x belong to A_x, states q_y, r_y belong to A_y.

We will show how to construct a tag automaton and, from it, a LIA formula φ^{I} that is satisfiable iff the disequality is satisfiable. We assume that we are given NFAs $A_x = (Q_x, \Delta_x, I_x, F_x)$ and $A_y = (Q_y, \Delta_y, I_y, F_y)$ such that $L(A_x) = L_x$ and $L(A_x) = L_y$ with $Q_x \cap Q_y = \emptyset$. For this, we construct a TA A^{I}. Intuitively, A^{I} is obtained by first concatenating $\mathsf{LenTag}_x(A_x)$ with $\mathsf{LenTag}_y(A_y)$ using an ϵ-transition into a TA $A_\circ$. One can see $A_\circ$ as an encoding of all possible models of x and y w.r.t. only regular constraints[7]. Then, we take three copies of $A_\circ$ and connect them together with transitions that represent detected mismatches: the first copy is used for tracking the run of A_x before the position of the mismatch in x is encountered, the second copy is used for tracking the rest of the run in A_x and the first part of the run in A_y (before the mismatch in A_y), and the last copy tracks the rest of the run in A_y. Moreover, the automaton is enhanced with tags that keep track of the position of the mismatch in x and in y and the values of the mismatched symbols.

5.1.1 Tag Automaton Construction. Formally, let $A_\circ = (Q_\circ, \Delta_\circ, I_\circ, F_\circ)$ be a TA over $\mathbb{T}_\circ = \{\langle \mathbf{S}, a\rangle \mid a \in \Gamma\} \cup \{\langle \mathbf{L}, x\rangle, \langle \mathbf{L}, y\rangle\}$ obtained by the ϵ-concatenation of $\mathsf{LenTag}_x(A_x)$ and $\mathsf{LenTag}_y(A_y)$, i.e., $A_\circ = \mathsf{LenTag}_x(A_x) \circ_\epsilon \mathsf{LenTag}_y(A_y)$. The tags **S** are used for tracking the currently read symbol and **L**-tags are used for counting of the length of a word from the language of the corresponding variable. Then $A^{\mathrm{I}} = (Q_1 \cup Q_2 \cup Q_3, \Delta, I, F)$ is a TA over $\mathbb{T}^{\mathrm{I}} = \mathbb{T}_\circ \cup \{\langle \mathbf{M}_1, a\rangle, \langle \mathbf{M}_2, a\rangle \mid a \in \Gamma\} \cup \{\langle \mathbf{P}, x\rangle, \langle \mathbf{P}, y\rangle\}$, where the $\mathbf{M}_1$ and $\mathbf{M}_2$ tags denote the first and the second **M**ismatch respectively and $\langle \mathbf{P}, x\rangle$, $\langle \mathbf{P}, y\rangle$ are used to count the **P**ositions of the mismatch in x and y. A^{I} is constructed as follows:

- $Q_1 = Q_\circ \times \{1\}$, $Q_2 = Q_\circ \times \{2\}$, and $Q_3 = Q_\circ \times \{3\}$; intuitively, Q_1 are states where no mismatch was seen, Q_2 are states where only the first mismatch symbol was seen, and Q_3 are states where both mismatch symbols were seen,
- $I = I_\circ \times \{1\}$,
- $F = F_\circ \times \{1, 3\}$, and
- Δ is the union of the following sets of transitions:
  - $\{(q,1) \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{P},x\rangle, \langle \mathbf{L},x\rangle\}\!\mapsto (r,1) \mid q \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{L},x\rangle\}\!\mapsto r \in \Delta_\circ\}$ — transitions in A_x before the first mismatch,
  - $\{(q,1) \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{L},y\rangle\}\!\mapsto (r,1) \mid q \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{L},y\rangle\}\!\mapsto r \in \Delta_\circ\}$ — transitions in A_y if no mismatch symbols are seen and the disequality is satisfied due to x and y having different lengths,
  - $\{(q,1) \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{M}_1,a\rangle, \langle \mathbf{L},x\rangle\}\!\mapsto (r,2) \mid q \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{L},x\rangle\}\!\mapsto r \in \Delta_\circ\}$ — the first mismatch (in A_x),
  - $\{(q,2) \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{L},x\rangle\}\!\mapsto (r,2) \mid q \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{L},x\rangle\}\!\mapsto r \in \Delta_\circ\}$ — transitions in A_x after the first mismatch (we still need to finish reading x to make sure that it was accepted by A_x),
  - $\{(q,2) \text{-}\{\}\!\mapsto (r,2) \mid q \text{-}\{\}\!\mapsto r \in \Delta_\circ\}$ — jump from A_x to A_y,
  - $\{(q,2) \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{P},y\rangle, \langle \mathbf{L},y\rangle\}\!\mapsto (r,2) \mid q \text{-}\{\langle \mathbf{S},a\rangle, \langle \mathbf{L},y\rangle\}\!\mapsto r \in \Delta_\circ\}$ — transitions in A_y before the second mismatch,

[7] In fact, the order in which we do the concatenation does not really matter—the main objective is to obtain a TA such that an accepting run in it represents a model of regular constraints

- $\{(q,2) \xrightarrow{\{\langle \mathbf{S},a\rangle,\langle \mathbf{M}_2,a\rangle,\langle \mathbf{L},y\rangle\}} (r,3) \mid q \xrightarrow{\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},y\rangle\}} r \in \Delta_\circ\}$ — the second mismatch (in A_y),
- $\{(q,3) \xrightarrow{\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},y\rangle\}} (r,3) \mid q \xrightarrow{\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},y\rangle\}} r \in \Delta_\circ\}$ — transitions in A_y after the second mismatch.

Note that in Δ, the $\mathbf{M}_1$-tagged transitions denote the occurrence of the first mismatch (which causes a jump from Q_1 to Q_2) and the $\mathbf{M}_2$-tagged transitions denote the occurrence of the second mismatch (jumping from Q_2 to Q_3). For accepting runs of A^{I}, it holds that they either (i) stay in Q_1 and accept in some state from $F_\circ \times \{1\}$ (so we only keep track of the lengths of the words $w_x \in L_x$ and $w_y \in L_y$) or (ii) take a $\mathbf{M}_1$-tagged transition to Q_2 and then a $\mathbf{M}_2$-tagged transition to Q_3 and accept in some state from $F_\circ \times \{3\}$. An accepting run of the tag automaton encodes an assignment of x and y to words from L_x and L_y. An example of a constructed tag automaton is given in Fig. 2.

5.1.2 Formula Construction. After A^{I} is constructed, it remains to test whether there is a run of A^{I} starting in I and ending in F such that the number of occurrences of $\langle \mathbf{L}, x\rangle$ and $\langle \mathbf{L}, y\rangle$ differs (corresponding to the case $|x| \neq |y|$), or the number of occurrences of the $\langle \mathbf{P}, x\rangle$ and $\langle \mathbf{P}, y\rangle$ tags is the same and there is one occurrence of a $\langle \mathbf{M}_1, a\rangle$ tag and one occurrence of a $\langle \mathbf{M}_2, b\rangle$ tag with $a \neq b$. The means to this is via the Parikh tag formula of A^{I}. First, we define formulae $\varphi^{\mathrm{I}}_{sym}$ and $\varphi^{\mathrm{I}}_{mis}$, which express that the two sampled symbols are a mismatch and that there was a mismatch:

$$\varphi^{\mathrm{I}}_{sym} \overset{\text{def.}}{\Leftrightarrow} \bigwedge_{a\in\Gamma} (\#\langle \mathbf{M}_1, a\rangle + \#\langle \mathbf{M}_2, a\rangle < 2) \qquad \text{and} \qquad \varphi^{\mathrm{I}}_{mis} \overset{\text{def.}}{\Leftrightarrow} \sum_{a\in\Gamma} \#\langle \mathbf{M}_1, a\rangle > 0. \tag{4}$$

In the formula, the first sum is used to check that the mismatched symbols are indeed different (from the construction of A^{I}, there will be at most one $\mathbf{M}_1$ and one $\mathbf{M}_2$ tags in every accepting run) and the second sum makes sure that there was at least one mismatch (so that we can only accept in Q_3). Finally, we construct the formula φ^{I} equisatisfiable to the disequality $x \neq y$ as follows:

$$\varphi^{\mathrm{I}} \overset{\text{def.}}{\Leftrightarrow} PF_{tag}(A^{\mathrm{I}}) \wedge \left(\#\langle \mathbf{L}, x\rangle \neq \#\langle \mathbf{L}, y\rangle \vee \left(\#\langle \mathbf{P}, x\rangle = \#\langle \mathbf{P}, y\rangle \wedge \varphi^{\mathrm{I}}_{sym} \wedge \varphi^{\mathrm{I}}_{mis}\right)\right). \tag{5}$$

Theorem 5.1. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge x \neq y$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{\mathrm{I}}$. Moreover, the size of φ^{I} is polynomial to $|\mathcal{R}'|$.*

5.2 II: A Single Unrestricted Disequality

Let us now move to the case of an arbitrary disequality between concatenations of (potentially repeating) variables:

$$x_1 \dots x_n \neq y_1 \dots y_m. \tag{6}$$

This complex disequality is satisfiable if there are words $w_{x_i} \in L_{x_i}$ and $w_{y_j} \in L_{y_j}$ for all i, j such that either (i) both sides have different lengths (given by $\sum_{1\le i\le n} |w_{x_i}|$ and $\sum_{1\le j\le m} |w_{y_j}|$ respectively) or (ii) they are of the same length and there is a *mismatch* position ℓ s.t. $w_x[\ell] \neq w_y[\ell]$ where $w_x = w_{x_1} \cdots w_{x_n}$ and $w_y = w_{y_1} \cdots w_{y_m}$. We emphasize that there might be multiple occurrences of a single variable z, potentially on both sides of the disequality, and they all need to be assigned the same word from L_z.

We will again construct a tag automaton A^{II} checking whether one of the conditions to satisfy the disequality holds. In this case, an accepting run in the tag automaton encodes an assignment that maps every variable z from the disequality to a word from L_z. The mismatch may happen in any pair of occurrences of variables (x_i, y_j) and, moreover, the variables might have multiple occurrences in the disequality. In the tag automaton, a run encoding a mismatch needs to nondeterministically guess a pair of variables' occurrences where the mismatch happens, the mismatch positions within the variables, and the mismatch symbol itself. In order to check that the guess is valid, we then construct a formula that will use the Parikh tag image of A^{II} and use it to check that (i) the mismatch symbols are different and (ii) the *global* positions of both mismatches are equal, meaning that for

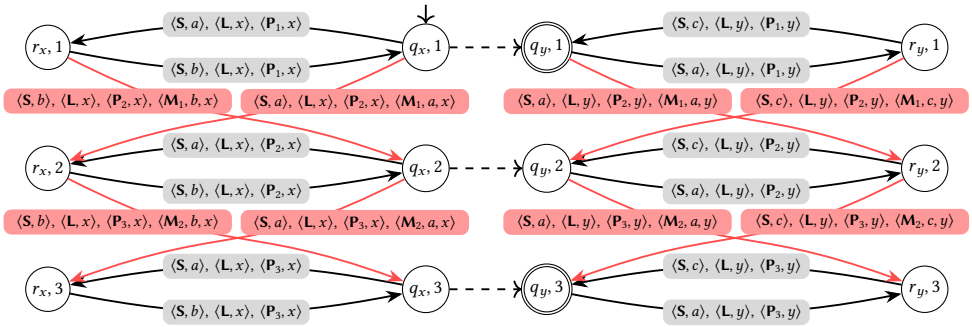


Fig. 3. Example of a tag automaton for the disequality $xy \neq yx$ with $L(A_x) = (ab)^*$ and $L(A_y) = (ac)^*$.

a guess of mismatch variables (x_i, y_j), the mismatch position in x_i plus lengths of assignments of $x_1 \cdots x_{i-1}$ is equal to the mismatch position in y_j plus lengths of assignments of $y_1 \cdots y_{j-1}$.

5.2.1 Tag Automaton Construction. Similarly to the previous section, we assume an NFA A_x for each variable x describing the language L_x. We use $\mathbb{X}$ to denote the set of all variables in the disequality. Without loss of generality, we assume that the sets of states of A_x's are pairwise disjoint. We also assume a fixed linear order on variables $\preccurlyeq$, which is further used to create a unique concatenation of tag automata for each variable. First, for each variable x we construct the TA T_x corresponding to A_x enriched with lengths, i.e., $T_x = \mathsf{LenTag}_x(A_x)$. Then, we construct $A_\circ = (Q_\circ, \Delta_\circ, I_\circ, F_\circ)$ over $\mathbb{T}_\circ$ as an ϵ-concatenation of all TAs T_x for $x \in \mathbb{X}$ in the order given by $\preccurlyeq$.

$A^{\text{II}} = (Q_1 \cup Q_2 \cup Q_3, \Delta, I, F)$ is a TA over $\mathbb{T}^{\text{II}} = \mathbb{T}_\circ \cup \{\langle \mathbf{M}_1, a, x\rangle, \langle \mathbf{M}_2, a, x\rangle \mid a \in \Gamma, x \in \mathbb{X}\} \cup \{\langle \mathbf{P}_1, x\rangle, \langle \mathbf{P}_2, x\rangle, \langle \mathbf{P}_3, x\rangle \mid x \in \mathbb{X}\}$, where the $\mathbf{M}_1$ and $\mathbf{M}_2$ tags again denote the first and the second mismatch respectively (note that, contrary to Sec. 5.1, the mismatch tags here are extended with variables). The tags $\langle \mathbf{P}_1, z\rangle$ and $\langle \mathbf{P}_2, z\rangle$ are used to count the local positions of the first and second mismatch in z respectively[8]. The $\langle \mathbf{P}_3, x\rangle$ tag will become important in Sec. 6.2 when reusing the automaton construction for the $\neg$*suffixof* predicate. A^{II} is constructed as follows:

- $Q_1 = Q_\circ \times \{1\}$, $Q_2 = Q_\circ \times \{2\}$, and $Q_3 = Q_\circ \times \{3\}$,
- $I = I_\circ \times \{1\}$,
- $F = F_\circ \times \{1, 3\}$, and
- Δ is the union of the following sets of transitions:
  - $\{(q,1) \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{P}_1,z\rangle,\langle \mathbf{L},z\rangle\}\!\!\to (r,1) \mid q \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle\}\!\!\to r \in \Delta_\circ\}$ — transitions in each A_z before the first mismatch,
  - $\{(q,1) \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{M}_1,a,z\rangle,\langle \mathbf{P}_2,z\rangle,\langle \mathbf{L},z\rangle\}\!\!\to (r,2) \mid q \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle\}\!\!\to r \in \Delta_\circ\}$ — the first mismatch,
  - $\{(q,2) \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{P}_2,z\rangle,\langle \mathbf{L},z\rangle\}\!\!\to (r,2) \mid q \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle\}\!\!\to r \in \Delta_\circ\}$ — transitions in each A_z before the second mismatch,
  - $\{(q,2) \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{M}_2,a,z\rangle,\langle \mathbf{P}_3,z\rangle,\langle \mathbf{L},z\rangle\}\!\!\to (r,3) \mid q \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle\}\!\!\to r \in \Delta_\circ\}$ — the second mismatch,
  - $\{(q,3) \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle,\langle \mathbf{P}_3,z\rangle\}\!\!\to (r,3) \mid q \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle\}\!\!\to r \in \Delta_\circ\}$ — transitions in each A_z after the second mismatch, and
  - $\{(q,i) \text{-}\{\}\!\!\to (r,i) \mid q \text{-}\{\}\!\!\to r \in \Delta_\circ, 1 \leq i \leq 3\}$ — transitions connecting variables on level i.

5.2.2 Formula Construction. For the satisfiability checking of the general disequality, we generalize the LIA reduction from the previous section. As in the previous case, the LIA formula speaks about properties of A^{II} using the Parikh tag formula $PF_{tag}(A^{\text{II}})$.

[8]We need to consider two possible mismatches in one variable because an occurrence of a mismatch can be between two positions in an assignment for z, one for the left-hand side and one for the right-hand side. E.g., consider the disequality $xy \neq yx$ and the assignment $\{x \mapsto ab, y \mapsto a\}$; the first mismatch is between the b in x on the left-hand side and the a in x on the right-hand side.

First, the formula expressing that lengths of both sides are different can be defined as follows:

$$\varphi^{\mathrm{II}}_{len} \quad \overset{\text{def.}}{\Leftrightarrow} \quad \sum_{1\leq i\leq n} \#\langle \mathbf{L}, x_i\rangle \neq \sum_{1\leq j\leq m} \#\langle \mathbf{L}, y_j\rangle. \tag{7}$$

Next, for the case the lengths are the same but there is a mismatch, we begin by defining a formula that checks that the particular mismatch symbols are different (and that there is at least one mismatch) by generalizing the formula $\varphi^{\mathrm{I}}_{sym}$ from the previous section:

$$\varphi^{\mathrm{II}}_{sym} \quad \overset{\text{def.}}{\Leftrightarrow} \quad \bigwedge_{a\in\Gamma}\left(\sum_{x\in\mathbb{X}} (\#\langle \mathbf{M}_1, x, a\rangle + \#\langle \mathbf{M}_2, x, a\rangle) < 2\right) \tag{8}$$

In order to check whether the global mismatch positions on both sides are equal, we need to make a case split ranging over all pairs (x_i, y_j) of occurrences of variables from the left-hand side and the right-hand of the disequality. For each such a pair, we define an auxiliary formula $\varphi_{pos(i,j)}$ comparing global mismatch positions when the mismatch is between the two occurrences.

(1) If x_i and y_j are occurrences of a different variable then:
- if $x_i \prec y_j$,

$$\varphi_{pos(i,j)} \quad \overset{\text{def.}}{\Leftrightarrow} \quad \#\langle \mathbf{P}_1, x_i\rangle + \sum_{1\leq u<i} \#\langle \mathbf{L}, x_u\rangle = \#\langle \mathbf{P}_2, y_j\rangle + \sum_{1\leq v<j} \#\langle \mathbf{L}, y_v\rangle, \tag{9}$$

- if $x_i \succ y_j$,

$$\varphi_{pos(i,j)} \quad \overset{\text{def.}}{\Leftrightarrow} \quad \#\langle \mathbf{P}_2, x_i\rangle + \sum_{1\leq u<i} \#\langle \mathbf{L}, x_u\rangle = \#\langle \mathbf{P}_1, y_j\rangle + \sum_{1\leq v<j} \#\langle \mathbf{L}, y_v\rangle. \tag{10}$$

The formulae express that the mismatch position is given by the sum of lengths of preceding variable assignments and the local mismatch position in the particular variable x_i or y_j.

(2) If x_i and y_j are occurrences of the same variable z, in order to get the position of the second local mismatch we have to add $\#\langle \mathbf{P}_1, z\rangle$ to $\#\langle \mathbf{P}_2, z\rangle$ since the second local mismatch in z has to be counted from the beginning of z and not from the beginning of the previous mismatch. Formally, the formula is given as

$$\begin{aligned}\varphi_{pos(i,j)} \quad \overset{\text{def.}}{\Leftrightarrow} \quad & \left(\#\langle \mathbf{P}_1, z\rangle + \sum_{1\leq u<i} \#\langle \mathbf{L}, x_u\rangle = \#\langle \mathbf{P}_1, z\rangle + \#\langle \mathbf{P}_2, z\rangle + \sum_{1\leq v<j} \#\langle \mathbf{L}, y_v\rangle\right) \vee \\ & \left(\#\langle \mathbf{P}_1, z\rangle + \#\langle \mathbf{P}_2, z\rangle + \sum_{1\leq u<i} \#\langle \mathbf{L}, x_u\rangle = \#\langle \mathbf{P}_1, z\rangle + \sum_{1\leq v<j} \#\langle \mathbf{L}, y_v\rangle\right).\end{aligned} \tag{11}$$

Then, for each $\varphi_{pos(i,j)}$, we need to combine it with a formula that says that there are indeed mismatches in x_i and y_j, to obtain formulae $\varphi_{i,j}$ as follows:

- if $x_i \preccurlyeq y_j$ (including the case when they are occurrences of the same variable),

$$\varphi_{i,j} \quad \overset{\text{def.}}{\Leftrightarrow} \quad \varphi_{pos(i,j)} \wedge \sum_{a\in\Gamma} \#\langle \mathbf{M}_1, x_i, a\rangle > 0 \wedge \sum_{a\in\Gamma} \#\langle \mathbf{M}_2, y_j, a\rangle > 0, \tag{12}$$

- if $x_i \succ y_j$,

$$\varphi_{i,j} \quad \overset{\text{def.}}{\Leftrightarrow} \quad \varphi_{pos(i,j)} \wedge \sum_{a\in\Gamma} \#\langle \mathbf{M}_1, y_j, a\rangle > 0 \wedge \sum_{a\in\Gamma} \#\langle \mathbf{M}_2, x_i, a\rangle > 0. \tag{13}$$

We do the case split based on the order of variables since the construction of A^{I} guarantees in which variable will be which mismatch. All $\varphi_{i,j}$ formulae are then collected into the formula

$$\varphi^{\mathrm{II}}_{mis} \quad \overset{\text{def.}}{\Leftrightarrow} \quad \bigvee_{\substack{1\leq i\leq n\\ 1\leq j\leq m}} \varphi_{i,j} \tag{14}$$

and the final formula equisatisfiable to $x_1 \dots x_n \neq y_1 \dots y_m$ is then defined as

$$\varphi^{\text{II}} \quad \overset{\text{def.}}{\Leftrightarrow} \quad PF_{tag}(A^{\text{II}}) \wedge \Big(\varphi^{\text{II}}_{len} \vee \big(\varphi^{\text{II}}_{sym} \wedge \varphi^{\text{II}}_{mis}\big)\Big). \tag{15}$$

Theorem 5.2. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge x_1 \dots x_n \neq y_1 \dots y_m$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{\text{II}}$. Moreover, the size of φ^{II} is polynomial to $nm \cdot |\mathcal{R}'|$.*

5.3 III: A System of Disequalities

We now move to the general case of a system of disequalities, which all need to be satisfied at the same time:

$$\bigwedge_{1 \leq i \leq n} L_i \neq R_i \tag{16}$$

where each L_i and R_i are arbitrary concatenations of variables, with potentially multiple occurrences in multiple disequalities. We, again, construct a tag automaton and a corresponding LIA formula for it. The tag automaton for this case will be more complex.

One could extend the construction of A^{II} from the previous section in a straightforward manner by creating more copies of $A_\circ$. With multiple disequalities, we need to keep track of satisfying position mismatches in particular disequalities, so that we do not count two mismatches in one disequality and zero mismatches in another disequality. If done in a straightforward way, we would need to consider all possible orders of mismatches in different disequalities, basically having one copy of the tag automaton A^{II} for each such an order. The number of these copies and the size of the resulting automaton would, however, be intractable. In particular, if we consider a set of disequalities $D = \{D_1, \dots, D_n\}$, we would need $\frac{(2n)!}{2^n} \in 2^{\Theta(n \log n)}$ such copies (obtained as the number of permutations of a set of n pairs of symbols $\mathbf{M}^i_1, \mathbf{M}^i_2$, respecting the order $\mathbf{M}^i_1 \prec \mathbf{M}^i_2$).

Another issue that we need to take into consideration is the fact that one mismatched symbol may be used for solving more than one disequality. For instance, for the system of disequalities $x \neq y \wedge x \neq z$ and its model $\{x \mapsto a, y \mapsto b, z \mapsto c\}$, the value of x is a mismatch for both disequalities. To deal with these issues, we take a more involved approach. Our approach is based on introducing two new types of tags:

(i) Instead of mismatch tags $\langle \mathbf{M}_i, a, x\rangle$ for $i \in \{1, 2\}$ from $\mathbb{T}^{\text{II}}$, we use more complex tags for mismatches of the form $\langle \mathbf{M}_i, x, D, s, a\rangle$ denoting that the i-th mismatched symbol for the disequality D on the side $s \in \{\mathcal{L}, \mathcal{R}\}$ tracked for the variable x was a. The mismatches can appear in an arbitrary order in an accepting run of the tag automaton (potentially also multiple or zero times), so we will need to extend the final LIA formula with a part that makes sure that we have a mismatch for both sides of every disequality.

(ii) We introduce **C**opy tags $\langle \mathbf{C}_i, x, D, s\rangle$, which express that the i-th mismatch symbol for disequality D and side $s \in \{\mathcal{L}, \mathcal{R}\}$ is given by the latest symbol sampled by a $\mathbf{M}$-tag for variable x.

With these two new kinds of tags and corresponding constraints added to the final LIA formula, we suffice with having a tag automaton with only $2n + 1$ copies of $A_\circ$.

5.3.1 Tag Automaton Construction. Let $A_\circ$ be the ϵ-concatenation of NFAs for all variables obtained in the same way as described in Sec. 5.2. Then $A^{\text{III}} = (Q, \Delta, I, F)$ is a TA over $\mathbb{T}^{\text{III}} = \mathbb{T}_\circ \cup \{\langle \mathbf{M}_i, x, D, s, a\rangle, \langle \mathbf{C}_i, x, D, s\rangle \mid a \in \Gamma, x \in \mathbb{X}, 1 \leq i \leq 2n, 1 \leq D \leq n, s \in \{\mathcal{L}, \mathcal{R}\}\} \cup \{\langle \mathbf{P}_i, x\rangle \mid x \in \mathbb{X}, 1 \leq i \leq 2n + 1\}$. A^{III} is constructed as follows:

- $Q = \{(q, i) \mid q \in Q_\circ, 1 \leq i \leq 2n + 1\}$,
- $I = I_\circ \times \{1\}$,
- $F = F_\circ \times \{1, 3, \dots, 2n + 1\}$, and
- Δ is the union of the following sets of transitions:

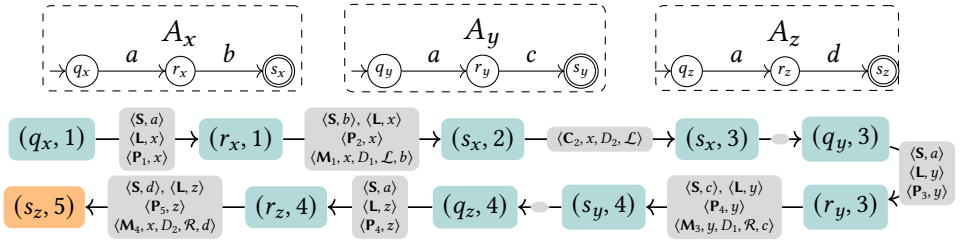


Fig. 4. An example of a run satisfying the system $D_1 \wedge D_2$ where $D_1 \overset{\text{def.}}{\Leftrightarrow} x \neq y$ and $D_2 \overset{\text{def.}}{\Leftrightarrow} x \neq z$.

- $\{(q,i) \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle,\langle \mathbf{P}_i,z\rangle\}\!\mapsto (r,i) \mid q \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle\}\!\mapsto r \in \Delta_\circ, 1 \leq i \leq 2n+1\}$,
- $\{(q,i) \text{-}\{\}\!\mapsto (r,i) \mid q \text{-}\{\}\!\mapsto r \in \Delta_\circ, 1 \leq i \leq 2n+1\}$,
- $\{(q,i) \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{M}_i,z,D,s,a\rangle,\langle \mathbf{L},z\rangle,\langle \mathbf{P}_{i+1},z\rangle\}\!\mapsto (r,i+1) \mid q \text{-}\{\langle \mathbf{S},a\rangle,\langle \mathbf{L},z\rangle\}\!\mapsto r \in \Delta_\circ, 1 \leq D \leq n, 1 \leq i \leq 2n, s \in \{\mathcal{L},\mathcal{R}\}\}$ — a mismatch guess for the disequality D and its side s, and
- $\{(q,i) \text{-}\{\langle \mathbf{C}_i,x,D,s\rangle\}\!\mapsto (q,i+1) \mid 1 \leq D \leq n, 2 \leq i \leq 2n, s \in \{\mathcal{L},\mathcal{R}\}\}$ — a guess that a mismatch previously seen in x is shared with the disequality D and its side s.

A run of A^{III} nondeterministically guesses possible mismatches for disequalities, as well as which mismatch is shared by multiple disequalities (the correctness of the guess is enforced by the final LIA formula). The run also guesses which disequalities are satisfied due to a mismatch and which are satisfied by the lengths violation (that is why F contains accepting states within all odd-labelled internal copies: each length-satisfied disequality removes the need for two mismatches). An example of a selected run of A^{III} is shown in Fig. 4.

5.3.2 Formula Construction. We construct a LIA formula equisatisfiable to the system of disequalities based on the tag automaton described above. Contrary to the case of a single disequality, the resulting formula is enhanced by subformulae ensuring consistency of each nondeterministic choice. For simplicity, we introduce auxiliary (integer) variables describing particular choices that are then used in the LIA subformulae: (i) $m_{D,s}$ variables containing the mismatch symbol for a disequality D and its side s and (ii) c_i variables containing the shared i-th mismatch symbol (the mismatch symbol preceding the $\mathbf{C}_i$-tag).

We start with auxiliary subformulae expressing that the mismatches are consistent, meaning that each disequality and side has at most one sampled mismatch and that these mismatches are sampled consistently for both sides. The first subformula φ_{Fair} checks that there is at most one mismatch for each side of each disequality:

$$\varphi_{Fair} \overset{\text{def.}}{\Leftrightarrow} \bigwedge_{\substack{D \in \{D_1,\ldots,D_n\} \\ s \in \{\mathcal{L},\mathcal{R}\}}} \Bigg(\sum_{\substack{1 \leq i \leq 2n \\ x \in \mathbb{X}, a \in \Gamma}} \#\langle \mathbf{M}_i, x, D, s, a\rangle + \#\langle \mathbf{C}_i, x, D, s\rangle \leq 1 \Bigg). \tag{17}$$

The subformula $\varphi_{Consistent}$ then ensures that the quantified variables containing the mismatch symbols are properly set, including the case of the copy tag, where the mismatch is inherited from the previous mismatch transition.

$$\begin{aligned} \varphi_{Consistent} \overset{\text{def.}}{\Leftrightarrow} \bigwedge_{\substack{D \in \{D_1,\ldots,D_n\} \\ s \in \{\mathcal{L},\mathcal{R}\}, a \in \Gamma, \\ 1 \leq i \leq 2n}} & \Bigg(\Bigg(\sum_{x \in \mathbb{X}} \#\langle \mathbf{M}_i, x, D, s, a\rangle = 1 \Bigg) \rightarrow c_i = m_{D,s} = a \Bigg) \wedge \\ & \Bigg(\Bigg(\sum_{x \in \mathbb{X}} \#\langle \mathbf{C}_i, x, D, s\rangle = 1 \Bigg) \rightarrow c_i = m_{D,s} = c_{i-1} \Bigg). \end{aligned} \tag{18}$$

Since not all disequalities have to be satisfied by the existence of a mismatch (but possibly also by a length violation), the values of $m_{D,s}$ and c_i variables for disequalities with a missing mismatch (one side or both) might hold arbitrary values. Therefore, it is not sufficient to compare only values

of $m_{D,\mathcal{L}}$ and $m_{D,\mathcal{R}}$ but it is necessary to take into account existing mismatches. It remains to check the consistency of copy tags. In particular, we need to ensure that copy tags for a variable x occur on a run only if the previous mismatch or copy transition for x was taken. We also need to check that a **C**-transition was taken immediately after the previous mismatch or copy transition (**M** or **C**).

$$\begin{aligned}\varphi_{Copies} \overset{\text{def.}}{\Leftrightarrow} & \bigwedge_{\substack{1\le i\le 2n\\ x\in\mathbb{X}}} \left(\Big(\sum_{\substack{D\in\{D_1,\ldots,D_n\}\\ s\in\{\mathcal{L},\mathcal{R}\},a\in\Gamma}} \#\langle \mathbf{M}_i, x, D, s, a\rangle + \#\langle \mathbf{C}_i, x, D, s\rangle = 0 \Big) \rightarrow \Big(\sum_{\substack{D\in\{D_1,\ldots,D_n\}\\ s\in\{\mathcal{L},\mathcal{R}\}}} \#\langle \mathbf{C}_{i+1}, x, D, s\rangle = 0 \Big) \right) \wedge \\ & \bigwedge_{\substack{2\le i\le 2n\\ x\in\mathbb{X}}} \left(\Big(\sum_{\substack{D\in\{D_1,\ldots,D_n\}\\ s\in\{\mathcal{L},\mathcal{R}\}}} \#\langle \mathbf{C}_i, x, D, s\rangle = 1 \Big) \rightarrow \#\langle \mathbf{P}_i, x\rangle - \sum_{\substack{D\in\{D_1,\ldots,D_n\}\\ s\in\{\mathcal{L},\mathcal{R}\},a\in\Gamma}} \#\langle \mathbf{M}_{i-1}, x, D, s, a\rangle = 0 \right).\end{aligned} \tag{19}$$

We note that the last expression in φ_{Copies}, $\#\langle \mathbf{P}_i, x\rangle - \sum \ldots$, is there since we need to make sure that if there is a $\mathbf{C}_i$-tag immediately after an $\mathbf{M}_{i-1}$-tag, the number of $\langle \mathbf{P}_i, x\rangle$ is one (because there was already one $\mathbf{P}_i$ tag on the $\mathbf{M}_{i-1}$-transition), but if a copy tag follows another copy tag, then the number of corresponding position tags is zero.

The final formula will be

$$\varphi^{\text{III}} \overset{\text{def.}}{\Leftrightarrow} PF_{tag}(A^{\text{III}}) \wedge \varphi_{Fair} \wedge \varphi_{Consistent} \wedge \varphi_{Copies} \wedge \bigwedge_{D\in\{D_1,\ldots,D_n\}} \big(\varphi^D_{len} \vee (\varphi^D_{mis} \wedge \varphi^D_{sym})\big), \tag{20}$$

where φ^D_{len}, φ^D_{mis}, and φ^D_{sym} are similar to their counterparts in Sec. 5.2 but using the $m_{D,s}$ variables instead of directly using **M**-tags. Details are in [28].

Theorem 5.3. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge \bigwedge_{1\le i\le n} L_i \neq R_i$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{\text{III}}$. Moreover, the size of φ^{III} is polynomial to $mn \cdot |\mathcal{R}'|$ where m is the maximum size of any L_i or R_i.*

6 Other Position Constraints

In this section, we show how the framework introduced in Sec. 5 can be extended for solving other considered position constraints.

6.1 Length Constraints

For conjunctions $\bigwedge_{1\le i\le n} x_i = len(y_{i1}\cdots y_{im_i})$, where x_i are integer variables, we create the ϵ-concatenation $A_\circ$ for all string variables occurring in the constraint and construct the formula

$$\varphi^{LEN} \overset{\text{def.}}{\Leftrightarrow} PF_{tag}(A_\circ) \wedge \bigwedge_{1\le i\le n} \Big(x_i = \sum_{1\le j\le m_i} \#\langle \mathbf{L}, y_{ij}\rangle \Big). \tag{21}$$

Theorem 6.1. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge \bigwedge_{1\le i\le n} x_i = len(y_{i1}\cdots y_{im_i})$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{LEN}$ and the size of φ^{LEN} is polynomial to $mn \cdot |\mathcal{R}'|$.*

6.2 Not Prefix and Not Suffix Predicates

The $\neg$*prefixof*$(x_1\cdots x_n, y_1\cdots y_m)$ and $\neg$*suffixof*$(x_1\cdots x_n, y_1\cdots y_m)$ predicates are similar to a disequality $x_1\cdots x_n \neq y_1\cdots y_m$ in that they are satisfied if there is a mismatch at the same global position between their first and second argument. Both predicates, however, have slightly different conditions than disequalities on satisfiability due to their sides having incompatible lengths—the first argument ($x_1\cdots y_n$) must be strictly longer than the second argument ($y_1\cdots y_m$). Therefore, the tag-automaton construction is the same as in the case of a single unrestricted disequality given in Sec. 5.2. Forming an equisatisfiable LIA formula also remains the same, save for small

differences. The different condition on satisfiability by $x_1 \cdots x_n$ and $y_1 \cdots y_m$ having incompatible lengths requires replacing the corresponding subformula $\varphi^{\text{II}}_{len}$ by $\varphi^{*\text{FIX}}_{len}$ defined as

$$\varphi^{*\text{FIX}}_{len} \overset{\text{def.}}{\Leftrightarrow} \sum_{1 \le i \le n} \#\langle \mathbf{L}, x_i \rangle > \sum_{1 \le j \le m} \#\langle \mathbf{L}, y_j \rangle. \tag{22}$$

Furthermore, the $\neg$*suffixof* predicate treats the mismatch position differently than $\neg$*prefixof*. Instead of being satisfied by a mismatch on the same global position starting from the beginning of its arguments, the $\neg$*suffixof* predicate counts the mismatch position from the end of its arguments. Therefore, we also need to replace the $\varphi_{pos(i,j)}$ subformulae with $\varphi^{\text{NS}}_{pos(i,j)}$ asserting that the mismatch positions in both arguments are the same, using the $\langle \mathbf{P}_3, x \rangle$-tags, which we already added into A^{II} in Sec. 5.2. We define $\varphi^{\text{NS}}_{pos(i,j)}$ as follows:

(1) If x_i and y_j are occurrences of a different variable, then

$$\varphi^{\text{NS}}_{pos(i,j)} \overset{\text{def.}}{\Leftrightarrow} \begin{cases} \#\langle \mathbf{P}_2, x_i \rangle + \#\langle \mathbf{P}_3, x_i \rangle + \sum_{1 \le u < i} \#\langle \mathbf{L}, x_u \rangle = \#\langle \mathbf{P}_3, y_j \rangle + \sum_{1 \le v < j} \#\langle \mathbf{L}, y_v \rangle & \text{if } x_i \prec y_j, \\ \#\langle \mathbf{P}_3, x_i \rangle + \sum_{1 \le u < i} \#\langle \mathbf{L}, x_u \rangle = \#\langle \mathbf{P}_2, y_j \rangle + \#\langle \mathbf{P}_3, y_j \rangle + \sum_{1 \le v < j} \#\langle \mathbf{L}, y_v \rangle & \text{otherwise.} \end{cases} \tag{23}$$

(2) If x_i and y_j are occurrences of the same variable z, then

$$\begin{aligned} \varphi^{\text{NS}}_{pos(i,j)} \overset{\text{def.}}{\Leftrightarrow} & \left(\#\langle \mathbf{P}_2, z \rangle + \#\langle \mathbf{P}_3, z \rangle + \sum_{1 \le u < i} \#\langle \mathbf{L}, x_u \rangle = \#\langle \mathbf{P}_3, z \rangle + \sum_{1 \le v < j} \#\langle \mathbf{L}, y_v \rangle \right) \vee \\ & \left(\#\langle \mathbf{P}_3, z \rangle + \sum_{1 \le u < i} \#\langle \mathbf{L}, x_u \rangle = \#\langle \mathbf{P}_2, z \rangle + \#\langle \mathbf{P}_3, z \rangle + \sum_{1 \le v < j} \#\langle \mathbf{L}, y_v \rangle \right). \end{aligned} \tag{24}$$

Intuitively, we start counting the mismatch position inside a variable *after* the mismatch has been sampled rather than counting *until* it has been sampled. We denote the corresponding constructed formulae as φ^{pre} (for $\neg$*prefixof*) and φ^{suf} (for $\neg$*suffixof*).

Theorem 6.2. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge \neg$prefixof$(x_1 \cdots x_n, y_1 \cdots y_m)$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{pre}$ and the formula $\mathcal{R}' \wedge \mathcal{I} \wedge \neg$suffixof$(x_1 \cdots x_n, y_1 \cdots y_m)$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{suf}$. The sizes of φ^{pre} and φ^{suf} are polynomial to $mn \cdot |\mathcal{R}'|$.*

6.3 Symbol (not) at a Position

Starting with the negative case first, let the input predicate be $x_s \neq str.at(y_1 \cdots y_m, x_i)$ where $x_s, y_1, \ldots y_n$ are string variables and x_i is an integer variable. We construct the tag automaton A in the same way as described in Sec. 5.2. To form an equisatisfiable LIA formula, we modify the reduction from Sec. 5.2 to capture that the mismatch position of the left-hand side is given by x_i rather than being nondeterministically given by a run in the automaton:

$$\varphi_{1,j} \overset{\text{def.}}{\Leftrightarrow} \begin{cases} x_i = \#\langle \mathbf{P}_1, y_j \rangle + \sum_{1 \le k < j} \#\langle \mathbf{L}, y_k \rangle & \text{if } y_i \prec x_s, \\ x_i = \#\langle \mathbf{P}_2, y_j \rangle + \sum_{1 \le k < j} \#\langle \mathbf{L}, y_k \rangle & \text{otherwise.} \end{cases} \tag{25}$$

We further introduce an auxiliary predicate $\varphi_{InBounds}$ checking that x_i is a valid position in $y_1 \cdots y_m$:

$$\varphi_{InBounds} \overset{\text{def.}}{\Leftrightarrow} 0 \le x_i < \sum_{1 \le j \le m} \#\langle \mathbf{L}, y_j \rangle. \tag{26}$$

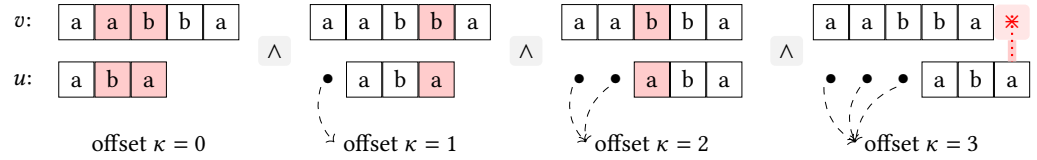


Fig. 5. Demonstration of how $\neg contains(u, v)$ for $u, v \in \mathbb{X}^*$ is satisfied by an assignment $\sigma = \{u \mapsto aba, v \mapsto aabba\}$. Symbols with red background present mismatches in the corresponding alignments.

The final formula $\varphi^{\neg str.at}$ is then modified to capture possible invalid positions of x_i:

$$\varphi^{\neg str.at} \overset{\text{def.}}{\Leftrightarrow} PF_{tag}(A) \wedge \Big(\big(\#\langle \mathbf{L}, x_s \rangle > 0 \wedge \neg\varphi_{InBounds}\big) \vee \#\langle \mathbf{L}, x_s \rangle > 1 \vee \big(\#\langle \mathbf{L}, x_s \rangle = 1 \wedge \varphi_{InBounds} \wedge \varphi_{sym} \wedge \bigvee_{1 \le j \le m} \varphi_{1,j}\big)\Big). \tag{27}$$

Theorem 6.3. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge x_s \neq str.at(y_1 \cdots y_m, x_i)$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{\neg str.at}$ and the size of $\varphi^{\neg str.at}$ is polynomial to $m \cdot |\mathcal{R}'|$.*

The $x_s = str.at(y_1 \cdots y_m, x_i)$ predicate can be reduced in a similar fashion, requiring us to replace φ_{sym} with φ'_{sym}, which enforces the sampled letters to be the same rather than being different.

$$\varphi^{str.at} \overset{\text{def.}}{\Leftrightarrow} PF_{tag}(A) \wedge \Big(\big(\#\langle \mathbf{L}, x_s \rangle = 0 \wedge \neg\varphi_{InBounds}\big) \vee \big(\#\langle \mathbf{L}, x_s \rangle = 1 \wedge \varphi_{InBounds} \wedge \varphi'_{sym} \wedge \bigvee_{1 \le j \le m} \varphi_{1,j}\big)\Big). \tag{28}$$

Theorem 6.4. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge x_s = str.at(y_1 \cdots y_m, x_i)$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{str.at}$ and the size of $\varphi^{str.at}$ is polynomial to $m \cdot |\mathcal{R}'|$.*

6.4 Not Contains Predicate

In this section, we extend the reasoning about the disequality tag automaton $A^{\neq}$ introduced in Sec. 5.2 to handling $\neg contains$ with flat languages. The constraint $\neg contains(u, v)$ for $u, v \in \mathbb{X}^*$ is satisfiable if there is a string assignment of variables from u and v yielding words w_u and w_v respectively such that for *every* alignment of w_u and w_v (i) there is a mismatch symbol of w_u and w_v or (ii) w_u overflows w_v. For example, considering the constraint $\neg contains(u, v)$, the assignment $\sigma = \{u \mapsto aba, v \mapsto aabba\}$ is a model—for every alignment of *aba* and *aabba*, there is either a mismatching symbol or a part of *aba* is outside *aabba* (cf. Fig. 5). The alignment of w_u and w_v can be characterized by the offset $\kappa \in \mathbb{N}$ of w_u counted from the beginning of w_v. Therefore, the semantics of $\neg contains$ implicitly involves a universal quantifier ranging over all possible offsets.

Since we need to consider mismatches for all offsets of w_u and w_v, one might consider the formula $\varphi^{\neq}$ from Sec. 5.2, but changed such that the position constraints $\varphi_{pos(i,j)}$ take into account a universally quantified offset variable κ. Such a solution, however, does not work, since we need that (i) for each value of κ, the string assignment remains the same (we want to shift the same assignment to different positions given by the offset and not obtain a different assignment for each offset), and (ii) for each offset the particular mismatch position and symbol (if any) may be different. The second property is problematic as the different offsets might involve different runs in the tag automaton that are, however, over the same string assignment. For this reason, we need to impose a *flat language restriction*, since for flat automata, the number of taken transitions (and so a model of $PF(A)$) uniquely determines the accepted word. On the other hand, for non-flat automata, this property does not hold. E.g,. consider an NFA with a single state q (which is both initial and

accepting) and two transitions: $q \xrightarrow{a} q$ and $q \xrightarrow{b} q$. The two words *aabb* and *bbaa* accepted by the NFA are different, but they have the same Parikh images. Our restriction to flat languages gives us the guarantee that a model of the Parikh formula uniquely determines the string assignment.

6.4.1 Formula Construction. Let $\varphi = \neg contains(u, v)$ where $u = u_1 \dots u_n$ and $v = v_1 \dots v_m$ are sequences of variables from $\mathbb{X}$ such that $L(\mathrm{Aut}(x))$ is flat for every $x \in \mathbb{X}$, and let A^{II} be a tag automaton constructed as described in Sec. 5.2. Since we now need to speak about different runs of A^{II}, we lift the definition of the Parikh tag image to explicitly speak about the Parikh variables, i.e., $PF_{tag}(T, \#)$ where # denotes the set of all #-prefixed variables in PF_{tag}. Since we need to speak about alignments of assignments, we also refine the formulae $\varphi_{pos(i,j)}$ from Sec. 5.2 to $\varphi_{pos(i,j)}(\kappa, \#)$, explicitly relating the used Parikh variables and the particular offset κ. The offset κ is added to the left-hand side of every equation occurring inside $\varphi_{pos(i,j)}$, in order to express that the assignments of the left-hand side are shifted to the right by κ. The formula $\varphi^{\text{II}}_{mis}$ from Sec. 5.2 is then also changed into $\varphi_{mis}(\kappa, \#)$, which uses $\varphi_{pos(i,j)}(\kappa, \#)$ instead of $\varphi_{pos(i,j)}$.

Let us start by defining some auxiliary predicates used later in the resulting formula:

$$\pi(q \xrightarrow{\langle \mathbf{S}, a\rangle, \langle \mathbf{L}, x\rangle} p) \quad \stackrel{\text{def.}}{=} \quad \left\{(q, i) \xrightarrow{U} (p, j) \mid (q, i) \xrightarrow{U} (p, j) \in \Delta(A^{\text{II}}), \langle \mathbf{S}, a\rangle \in U\right\} \text{ and} \tag{29}$$

$$EqualWords(\#_1, \#_2) \quad \stackrel{\text{def.}}{\Leftrightarrow} \quad \bigwedge_{t \in \Delta(A_\circ)} \left(\sum_{r \in \pi(t)} \#_1 r = \sum_{r \in \pi(t)} \#_2 r \right). \tag{30}$$

Let $\#_1$ and $\#_2$ be two sets of Parikh variables encoding accepting runs of A^{II}, i.e., $PF_{tag}(A^{\text{II}}, \#_1)$ and $PF_{tag}(A^{\text{II}}, \#_2)$ hold. Intuitively, $EqualWords(\#_1, \#_2)$ is satisfied iff $\#_1$ and $\#_2$ correspond to the same sets of runs in the ϵ-concatenation $A_\circ$ serving as a basis for A^{II}, and, therefore, since we assume flat automata, $\#_1$ and $\#_2$ correspond to the same string assignments. Further, we define $LenDiff(\#)$ expressing the difference $|w_v| - |w_u|$ where w_u and w_v are concatenated assignments of $\neg contains$'s arguments given by the Parikh image:

$$LenDiff(\#) \quad \stackrel{\text{def.}}{=} \quad \left(\sum_{1 \le j \le m} \#\langle \mathbf{L}, v_j \rangle \right) - \left(\sum_{1 \le i \le n} \#\langle \mathbf{L}, u_i \rangle \right). \tag{31}$$

The resulting formula equisatisfiable to the $\neg contains$ is then given as

$$\begin{aligned} \varphi^{NC} \quad \stackrel{\text{def.}}{\Leftrightarrow} \quad PF_{tag}(A^{\text{II}}, \#_1) \wedge \forall \kappa \exists \#_2 \Big(& \big(PF_{tag}(A^{\text{II}}, \#_2) \wedge EqualWords(\#_1, \#_2) \wedge \varphi_{mis}(\kappa, \#_2) \big) \vee \\ & \kappa < 0 \vee \kappa > LenDiff(\#_1) \Big). \end{aligned} \tag{32}$$

Note that in the formula, we use $\exists \#$ to denote the existential quantification over all variables in #. Intuitively, φ^{NC} is satisfied iff there is a string assignment corresponding to $\#_1$ such that for every offset κ, we can find some other model $\#_2$ satisfying $PF_{tag}(A^{\text{II}}, \#_2)$ that encodes the same string model (but potentially a different run through A^{II}. Moreover, φ^{NC} says that the run corresponding to $\#_2$ contains a mismatch for the offset κ. Alternatively, the offset κ might be larger than the difference between lengths of $\neg contains$ arguments or negative, in which case the corresponding alignment is trivially satisfied.

Theorem 6.5. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge \neg contains(u_1 \dots u_n, v_1 \dots v_m)$ where the language of each u_i and v_j is flat is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{NC}$. Moreover, the size of φ^{NC} is polynomial to $mn \cdot |\mathcal{R}'|$.*

6.5 Arbitrary Combination of Position Predicates

The construction of the tag automaton for multiple disequalities and the subsequent LIA reduction can be easily extended to a system $\psi \equiv \bigwedge_{1\leq k\leq K} P_k(x_{k,1}\cdots x_{k,n_k}, y_{k,1}\cdots y_{k,m_k})$ where $P_k \in \{\neq, \neg\mathit{prefixof}, \neg\mathit{suffixof}, \mathit{str.at}, \neg\mathit{str.at}, \neg\mathit{contains}\}$. From a high-level perspective, a single ϵ-concatenation $A_\circ$ of all automata of variables occurring in ψ is created. The tag automaton is formed in the same way as described in Sec. 5.3, containing $2K+1$ copies of $A_\circ$ to track up to $2K$ possible mismatch symbols (one for each side of every predicate). The resulting LIA formula is then

$$\varphi^{comb} \overset{\text{def.}}{\Leftrightarrow} \varphi_{Parikh} \wedge \varphi_{Consistent} \wedge \varphi_{Copies} \wedge \bigwedge_{1\leq i\leq K} \varphi^i_{Sat} \tag{33}$$

where φ^i_{Sat} is a LIA formula specific to the type of i-th constraint described in previous sections expressing that the predicate is satisfied. Note that φ^i_{Sat} needs to be modified in the same way as in the case of a system of multiple disequalities (cf. Sec. 5.3) to make use of the $p_{D,s}$ and $m_{D,s}$ variables. Furthermore, all variables present in any $\neg\mathit{contains}$ predicate must have a flat language to maintain soundness, similar as in the case of a single $\neg\mathit{contains}$ predicate.

Theorem 6.6. *The formula $\mathcal{R}' \wedge \mathcal{I} \wedge \psi$ is equisatisfiable to the formula $\mathcal{I} \wedge \varphi^{comb}$, provided all variables occurring in $\neg$contains constraints within ψ are assigned flat languages by $\mathcal{R}'$. In addition, the size of φ^{comb} is polynomial to $mn \cdot |\mathcal{R}'|$ where n is the number of constraints in ψ and m is the maximum size of any side of the constraints in ψ.*

7 Decidability and Complexity

This section covers theoretical results that follow from tag-automaton constructions based on position predicates and subsequent reductions into LIA described in previous sections. We will formulate our results as instances of the following parametrized decision problem.

PosRegSAT$(\mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P})$	
INPUT:	• a set of string variables $\mathbb{X} = \{x_1, x_2, \ldots, x_k\}$, • a conjunction of word equations $\mathcal{E}$, • a conjunction of regular constraints $\mathcal{R} = \{x \in L(A_x) \mid x \in \mathbb{X}\}$, • a conjunction of length constraints $\mathcal{I}$, and • a conjunction of position constraints $\mathcal{P}$.
QUESTION:	Is there an assignment $\sigma\colon \mathbb{X} \to \Gamma^*$ satisfying $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{I} \wedge \mathcal{P}$?

In the following, we write $\mathcal{R}$ to denote an arbitrary conjunction of regular constraints of the form $\mathcal{R} = \bigwedge_{x\in\mathbb{X}} x \in L(A_x)$ where A_x is an NFA associated with the variable x if not specified otherwise.

Theorem 7.1. *The time complexity of PosRegSAT$(\emptyset, \mathcal{R}, \emptyset, \mathcal{P})$ with $\mathcal{P} = P(x_1\cdots x_n, y_1\cdots y_m)$ for $P \in \{\neq, \neg\mathit{suffixof}, \neg\mathit{prefixof}\}$ is in PTime, more concretely in $O(nm \cdot |\Gamma|^3 \cdot |\mathcal{R}|^6)$.*

Proof outline. We outline the proof for P being a disequality; the other cases are similar. We construct a one-counter automaton C with a counter $\mathbf{c}$ with updates limited to $\{-1, 0, +1\}$ such that there is an accepting state reachable with $\mathbf{c} = 0$ iff the input combination of regular constraints and P is satisfiable. We show that C has a polynomial size to the input and using the result of [9, Lemma 11] stating that 0-reachability of a state in a one-counter automaton can be decided in PTime, we get the theorem. The full proof is given in [28]. □

Lemma 7.2. *PosRegSAT$(\emptyset, \mathcal{R}, \emptyset, \mathcal{P})$ with $\mathcal{P} = \bigwedge_{1\leq i\leq K}(x_{i,1}\cdots x_{i,n_i} \neq y_{i,1}\cdots y_{i,m_i})$ is NP-hard.*

Proof. By reduction from 3-SAT. Let φ be an input 3-SAT formula. For each Boolean variable x_i in φ, we create a string variable y_i with $\mathsf{Aut}(y_i)$ being a DFA with 2 states accepting the language

$\{0, 1\}$. For each clause in φ, we create a new disequality such that, e.g., for a clause $(x_1 \vee \neg x_2 \vee x_3)$, we create the disequality $y_1 y_2 y_3 \neq 010$. Then the system of disequalities is equisatisfiable to φ. □

Theorem 7.3. *PosRegSAT*$(\emptyset, \mathcal{R}, \emptyset, \mathcal{P})$ *with* $\mathcal{P} = \bigwedge_{1 \leq i \leq K} P_i(x_{i,1} \cdots x_{i,n_i}, y_{i,1} \cdots y_{i,m_i})$ *for* $P_i \in \{\neq, \neg suffixof, \neg prefixof, str.at, \neg str.at\}$ *is NP-complete.*

Proof. From Theorem 7.2 we have that the problem is NP-hard. NP-membership follows from constructing an equisatisfiable quantifier-free LIA formula ψ as described in Sec. 6.5 and observing that ψ is of polynomial size. Satisfiability of quantifier-free LIA is in NP [63]. □

The following theorem states that position constraints with structurally limited languages of variables occurring in $\neg contains$ predicates can be decided in NExpTime.

Theorem 7.4. *PosRegSAT*$(\emptyset, \mathcal{R}, \emptyset, \mathcal{P})$ *with* $\mathcal{P} = \bigwedge_{1 \leq i \leq K} P_i(x_{i,1} \cdots x_{i,n_i}, y_{i,1} \cdots y_{i,m_i})$ *for* $P_i \in \{\neq, \neg suffixof, \neg prefixof, \neg contains, str.at, \neg str.at\}$ *such that* $L(A_x)$ *of any variable* x *that occurs in a* $\neg contains$ *predicate is flat can be decided in NExpTime.*

Proof. We can observe that, in the presence of $\neg contains$ predicates, the resulting formula ψ constructed as described in Sec. 6.5 falls into the $\exists\forall\exists$-fragment of LIA (after transforming into the prenex normal form). As the number of quantifier alternations is 2, we obtain that ψ is decidable in $\Sigma_1^{\text{Exp}} = \text{NExpTime}$ [37], where Σ_1^{Exp} is the first level of the weak exponential hierarchy. □

Contrary to deciding a single disequality (which is in PTime), deciding a single $\neg contains$ is already NP-hard, as stated by the following theorem (proven in [28]).

Theorem 7.5. *PosRegSAT*$(\emptyset, \mathcal{R}, \emptyset, \mathcal{P})$ *with* $\mathcal{P} = \neg contains(x_1 \ldots x_n, y_1 \ldots y_m)$ *is NP-hard.*

Finally, we obtain the decidability of the whole fragment considered in the paper for chain-free word equations.

Theorem 7.6. *PosRegSAT*$(\mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P})$ *with* $\mathcal{E}$ *being chain-free [8],* $\mathcal{P} = \bigwedge_{1 \leq i \leq K} P_i(x_{i,1} \cdots x_{i,n_i}, y_{i,1} \cdots y_{i,m_i})$ *for* $P_i \in \{\neq, \neg suffixof, \neg prefixof, \neg contains, str.at, \neg str.at\}$ *and* $\mathcal{R} = \bigwedge_{x \in \mathbb{X}} x \in L(A_x)$ *such that* $L(A_x)$ *of any variable* x *that occurs in a* $\neg contains$ *predicate is flat is decidable.*

Proof. As $\mathcal{E}$ is chain-free, we start by solving only $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{I}$ using the approach described in [23], obtaining a new set of variables $\mathbb{X}'$ along with a length constraint $\mathcal{I}'$ (extension of $\mathcal{I}$ with equalities relating lengths of the original variables and the variables from $\mathbb{X}'$), a monadic decomposition $\mathcal{R}' = \bigwedge_{x' \in \mathbb{X}'} x' \in L(A_{x'})$ and a length constraint (we note that the *noodlification* procedure in [23] preserves flatness of languages), and a substitution map $\sigma \colon \mathbb{X} \to (\mathbb{X}')^*$ mapping original variables to (potentially concatenations of) new variables. Applying σ to $\mathcal{P}$, we obtain a conjunction $\mathcal{P}'$ of new position predicates. We are left to solve a new system $\mathcal{R}' \wedge \mathcal{I}' \wedge \mathcal{P}'$, for which we can construct an equisatisfiable LIA formula using the techiniques presented in this work. Therefore, we obtain an equisatisfiable formula in a decidable theory, concluding the proof. □

8 Experimental Evaluation

We implemented the proposed decision procedure in the Z3-Noodler solver version 1.3 [24]. We call the modified version Z3-Noodler-pos. Since we need a monadic decomposition for dealing with position constraints, the proposed decision procedure was integrated to the stabilization-based procedure [23], which computes the monadic decomposition. For an input formula, we separate the position constraints, apply the stabilization-based procedure on the remaining constraints, and for each of the possible obtained monadic decompositions (there might be more depending on the case-splits within the stabilization), we add the LIA formula describing satisfiability of those position constraints to the LIA formula provided by the stabilization-based procedure (this

Table 1. Results of experiments on all benchmarks. For each benchmark we give the number of cases the tool runs out of resources (column "OOR", the number of timeouts and memory outs), the number of unknowns (column "Unk"), and total time in seconds on finished instances (column "Time") and on all instances (i.e., when taking the time 120 s for OOR/Unk instances; column "TimeAll"). The best TimeAll results are **bold**.

	biopython (77,222)				django (52,643)				thefuck (19,872)				position-hard (550)				All (150,287)			
	OOR	Unk	Time	TimeAll	OOR	Unk	Time	TimeAll	OOR	Unk	Time	TimeAll	OOR	Unk	Time	TimeAll	OOR	Unk	Time	TimeAll
Z3-Noodler-pos	171	0	3,490	24,010	39	0	3,325	8,005	0	0	665	**665**	0	0	124	**124**	210	0	7,604	**32,804**
Z3-Noodler	171	336	3,545	64,385	37	108	3,473	20,873	1	375	637	45,757	234	246	1,912	59,512	443	1,065	9,567	190,527
cvc5	69	0	12,834	**21,114**	0	0	4,515	**4,515**	0	0	690	690	550	0	–	66,000	619	0	18,039	92,319
Z3	1,047	0	15,661	141,301	502	0	7,501	67,741	47	0	9,457	15,097	550	0	–	66,000	2,146	0	32,619	290,139
OSTRICH	2,986	0	749,986	1,108,306	4,404	0	979,326	1,507,806	967	0	120,152	236,192	550	0	–	66,000	8,907	0	1,849,464	2,918,304

LIA formula may contain additional subformulae speaking, e.g., about lengths of the solution or string-integer conversions [39]). For satisfiability checking of quantifier-free LIA formulae, Z3-Noodler uses Z3's internal LIA solver based on the Simplex method extended with a branch-and-cut strategy to obtain integer solutions [33]. For universally quantified formulae (those obtained by the reduction of $\neg contains$), we use an additional Z3's internal solver based on the *model-based quantifier instantiation* [36] approach.

For representing the tag automaton structure, we use the Mata library [30]. A tag automaton is represented as a nondeterministic finite automaton with additional mapping of Mata's integer symbols to sets of tags. The LIA formula is generated in Z3-Noodler's internal format, which is then converted to Z3's formula representation. Except of the proposed procedure, Z3-Noodler-pos implements heuristics for simple cases of the $\neg contains$ predicate. In particular, if $|u| < |v|$, then $\neg contains(u, v)$ is satisfied. Therefore, if we get a $\neg contains$ with not-flat languages, we apply this underapproximation. For the case when the language of v is finite, we enumerate words from the language and check them separately instead of generating complex quantified formulae.

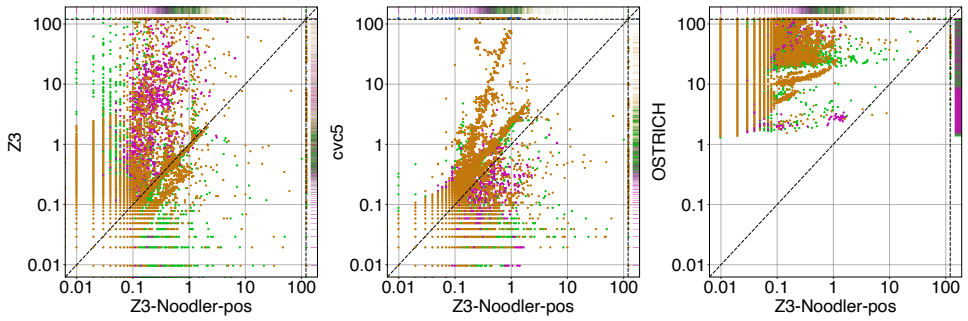
8.1 Experimental Settings

We evaluated Z3-Noodler-pos on benchmarks containing heavy position constraints. We collected 4 benchmark sets (the number of formulae is in parentheses): (i) biopython (77,222) obtained by a symbolic execution of Python tools for bioinformatics [3], (ii) django (52,643) from a symbolic execution of the Django web application framework [3], (iii) thefuck (19,872) from a symbolic execution of a tool correcting command mistakes [3][9], and (iv) position-hard (550) containing difficult hand-crafted formulae with disequalities and $\neg contains$ predicates.[10] Altogether, we collected 150,287 formulae for evaluation. We note that these formulae (in particular the first three sets) are *from the wild*, i.e., they are not of the form $\mathcal{R}' \wedge \mathcal{I} \wedge \mathcal{P}$, which we consider in Secs. 5 and 6. Instead, they have a more general Boolean structure with word equations and other constraints (not even necessarily chain-free), which are (in our case) handled and transformed into the monadic decomposition and our input form by the stabilization-based procedure of Z3-Noodler. Moreover, a single input formula might cause multiple calls to the string solver with formulae from different fragments. Since the main goal of this evaluation is to show the improvement that our contribution brings to automata-based techniques being able to handle position constraints, we excluded benchmarks from `SMT-LIB` [12], where the number of position-heavy constraints is small.

We compared Z3-Noodler-pos with Z3-Noodler (version 1.3) [24], cvc5 (version 1.2.0) [11], Z3 (version 4.13.3) [33], and OSTRICH (version 1.4) [22]. We excluded Z3-Trau [5] as it gives incorrect results on some benchmarks and Z3-Alpha [56] as it fails with an error on a large fraction

[9]The three benchmark sets biopython, django, and thefuck are from [3], where they were obtained by running the symbolic executor PyCT [73] on the respective projects and keeping formulae that contained at least one position string constraint or a constraint that is naturally translated to a position constraint, such as `indexof`.

[10]These are simple formulae inspired by the problem of testing primitiveness of a word. They contain one $\neg contains$ or $\neq$ predicate over concatenations of string variables (with possible repetitions, e.g., $xyz \neq xxy$) constrained by simple regular languages (e.g., a^* or $(abc)^*$). Despite their apparent simplicity, a solution cannot be easily found by systematically trying different assignments, which seems to be the reason why these formulae are unsolvable by state-of-the-art solvers.



(a) Z3-Noodler-pos vs. Z3 (b) Z3-Noodler-pos vs. cvc5 (c) Z3-Noodler-pos vs. OSTRICH

Fig. 6. Comparison of Z3-Noodler-pos with Z3-Noodler, cvc5, Z3, and OSTRICH. Times are in seconds, axes are logarithmic. Dashed lines represent timeouts (120 s). Colours distinguish benchmarks: • biopython, • django, • thefuck, and • position-hard.

of the benchmark. The experiments were executed on a server with an AMD EPYC 9124 64-Core Processor (16 cores were used by our experiment) with 125 GiB of RAM running Ubuntu 22.04.5. The timeout was set to 120 s (from our experience, higher limit has only a negligible effect on the number of solved instances) and the memory limit was set to 8 GiB (except for OSTRICH, where we set the limit to 16 GiB, since OSTRICH refuses to run with less than 8 GiB of memory).

8.2 Results

The overall results comparing Z3-Noodler-pos with other tools are shown in Table 1. From the table you can see that Z3-Noodler-pos has the smallest number (210) of OORs (i.e., time/memory-outs) followed by Z3-Noodler (443; however, it answers Unk for 1,065 instances) and cvc5 (619). Z3 and OSTRICH have a bit more OORs than these tools (2,146 and 8,907 respectively). For the biopython and django benchmark sets, cvc5 is the best solver, having slightly less OORs than Z3-Noodler-pos (171 vs. 69 on biopython and 39 vs. 0 on django, which is ~0.1 % of the two benchmark sets). On the thefuck set, the overall performance of Z3-Noodler-pos and cvc5 is roughly the same (Z3-Noodler-pos is faster by 25 s on the whole benchmark set, which is negligible), though the performance on individual formulae in the set can differ by a lot (cf. Fig. 6b). On the position-hard benchmark, Z3-Noodler-pos can solve all formulae while no other solver except Z3-Noodler can solve any of them (and Z3-Noodler can solve only 70). Note that concerning unsolved instances, Z3-Noodler-pos is quite orthogonal to cvc5 (only 10 formulae can be solved neither by Z3-Noodler-pos nor cvc5). Regarding the overall running time, Z3-Noodler-pos has the smallest time of all other tools.

From a comparison of Z3-Noodler-pos and Z3-Noodler, it is evident that the proposed decision procedure significantly helps in solving instances that were unknown for the original Z3-Noodler without any performance regression. The OORs of Z3-Noodler-pos on benchmarks obtained from symbolic execution are caused mainly by non-chain-freeness of the input constraint where the stabilization-based procedure was not able to get a stable solution before the time limit. In Fig. 6 we show scatter plots comparing the performance of Z3-Noodler-pos with Z3, cvc5, and OSTRICH. It can be seen from the figures that Z3-Noodler-pos can significantly outperform other state-of-the-art solvers on many instances. In Fig. 7 we give a cactus plot comparing sorted running times on all benchmarks, showing the superior performance of Z3-Noodler-pos.

9 Related Work

Approaches and tools for string solving are numerous and diverse, with a variety of constraint representations, algorithms, and input types. Many approaches use automata, e.g., Stranger [76–78], Z3-Noodler [18, 24, 26], Norn [6, 7], OSTRICH [19–21, 21, 54], Trau [3–5, 8], Sloth [40],

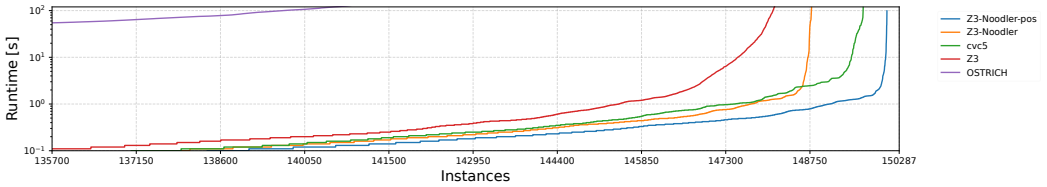


Fig. 7. Cactus plot comparing sorted runtimes of Z3-Noodler-pos with other tools. The y-axis denotes the time in seconds (the axis is logarithmic), the x-axis denotes the number of solved formulae ordered by their runtime (we show only the ∼14,500 hardest formulae for each solver).

Slog [75], Z3str3RE [14, 16], Retro [25, 29]. The most important tools focused on word equations include cvc4/5 [13, 51–53, 62, 67, 68], Z3 [17, 33]. Bit vectors are commonly used in tools like Z3Str/2/3/4 [15, 60, 79, 80] and HAMPI [46], while PASS [50] utilizes arrays, and G-strings [10] and GECODE+S [71] use a SAT solver. Z3-Alpha [56] synthesizes efficient strategies for Z3 in order to improve the performace.

The chain-free fragment [8], which we extend in this paper, represents the largest fragment of string constraints for which any string solver offers formal completeness guarantees. Quadratic equations, addressed by tools like Retro [25, 29] and $\texttt{Kepler}_{22}$ [48], are incomparable but have less practical relevance, though some tools, such as Z3-Noodler or OSTRICH, implement Nielsen's algorithm [61] to handle quadratic cases. Most other solvers guarantee completeness on smaller fragments (e.g., OSTRICH [54], Norn [6, 7], and Z3str3RE [16]), or use incomplete heuristics that work in practice by over-/under-approximating or by sacrificing termination guarantees.

When it comes to handling position constraints, existing tools generally employ a similar approach—reducing these constraints to equations and length constraints, which are then solved using exponential-space algorithms or incomplete techniques in modern string solvers. However, this approach cannot be even used for ¬*contains* as it cannot be directly reduced to quantifier-free combination of equations and length constraints. cvc-4/5 transforms ¬*contains* to quantified string formula, which is then solved by quantifier instantiation [66]. Probably the closest approach to ours is [3] converting the ¬*contains* into a LIA formula. The main differences are threefold: (i) the approach of [3] builds on the flattening underapproximation, while our approach is precise. (ii) our framework is more general that we can reduce to LIA all combinations of position constraints and not just ¬*contains*. (iii) The approach in [3] avoids considering repetitions of variables, which is a central part of our work, by an aggressive overapproximation based on replacing repeating variables by fresh ones. The idea of using counting to determine positions in strings was also used in [21], where cost enriched automata similar to our tag automata were used, though [21] aspires only to solve a substantially simpler problem of computing pre-images of basic constraints and does not consider position constraints. The inspiration for our use of tag automata was originally drawn from methods used in functional equivalence checking of streaming string transducers [9].

Data Availability Statement

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

Acknowledgements

We thank the anonymous reviewers for careful reading of the paper and their suggestions that greatly improved its quality. This work was supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 25-18318S, and the FIT BUT internal project FIT-S-23-8151. The work of Michal Hečko, a Brno Ph.D. Talent Scholarship Holder, is funded by the Brno City Municipality.

References

[1] 2024. SMT-COMP'24. https://smt-comp.github.io/2024/

[2] 2024. SMT-COMP'24, QF_Strings. https://smt-comp.github.io/2024/results/qf_strings-single-query/

[3] 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 *Proc. of APLAS'21 (LNCS, Vol. 13008)*, Hakjoo Oh (Ed.). Springer, 305–320. doi:10.1007/978-3-030-89051-3_17

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. 2017. Flatten and conquer: a framework for efficient analysis of string constraints. In *Proc. of PLDI'17*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 602–617. doi:10.1145/3062341.3062384

[5] 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 *Proc. of FMCAD'18*, Nikolaj S. Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–5. doi:10.23919/FMCAD.2018.8602997

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String constraints for verification. In *Proc. of CAV'14 (LNCS, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 150–166. doi:10.1007/978-3-319-08867-9_10

[7] 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 *Proc. of CAV'15 (LNCS, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 462–469. doi:10.1007/978-3-319-21690-4_29

[8] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukáš Holík, and Petr Janků. 2019. Chain-Free String Constraints. In *Proc. of ATVA'19 (LNCS, Vol. 11781)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer, 277–293. doi:10.1007/978-3-030-31784-3_16

[9] Rajeev Alur and Pavol Cerný. 2011. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 599–610. doi:10.1145/1926385.1926454

[10] 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 (LNCS, Vol. 10416)*, J. Christopher Beck (Ed.). Springer, 3–20. doi:10.1007/978-3-319-66158-2_1

[11] 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 *Proc. of TACAS'22 (LNCS, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

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

[13] Clark W. Barrett, Cesare Tinelli, Morgan Deters, Tianyi Liang, Andrew Reynolds, and Nestan Tsiskaridze. 2016. 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. doi:10.1145/2898375.2898393

[14] 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. doi:10.1016/j.tcs.2022.12.009

[15] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A string solver with theory-aware heuristics. In *Proc. of FMCAD'17*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 55–59. doi:10.23919/FMCAD.2017.8102241

[16] 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 *Proc. of CAV'21 (LNCS, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 289–312. doi:10.1007/978-3-030-81688-9_14

[17] Nikolaj S. Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *Proc. of TACAS'09 (LNCS, Vol. 5505)*, Stefan Kowalewski and Anna Philippou (Eds.). Springer, 307–321. doi:10.1007/978-3-642-00768-2_27

[18] Frantisek Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtech Havlena, Lukás Holík, Ondrej Lengál, and Juraj Síc. 2023. Word Equations in Synergy with Regular Constraints. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (LNCS, Vol. 14000)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer, 403–423. doi:10.1007/978-3-031-27481-7_23

[19] 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. doi:10.1145/3158091

[20] 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. doi:10.1145/3498707

[21] Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In *Proc. of ATVA'20 (LNCS, Vol. 12302)*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer, 325–342. doi:10.1007/978-3-030-59152-6_18

[22] 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. doi:10.1145/3290362

[23] Yu-Fang Chen, David Chocholatý, Vojtech Havlena, Lukás Holík, Ondrej Lengál, and Juraj Síc. 2023. Solving String Constraints with Lengths by Stabilization. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 2112–2141. doi:10.1145/3622872

[24] Yu-Fang Chen, David Chocholatý, Vojtech Havlena, Lukás Holík, Ondrej Lengál, and Juraj Síc. 2024. Z3-Noodler: An Automata-based String Solver. In *Proc. of TACAS'24 (LNCS, Vol. 14570)*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer, 24–33. doi:10.1007/978-3-031-57246-3_2

[25] Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. 2020. A symbolic algorithm for the case-split rule in string constraint solving. In *Proc. of APLAS'20 (LNCS, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, 343–363. doi:10.1007/978-3-030-64437-6_18

[26] Yu-Fang Chen, David Chocholatỳ, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2023. Solving string constraints with lengths by stabilization. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 2112–2141.

[27] Yu-Fang Chen, Vojtěch Havlena, Michal Hečko, Lukáš Holík, and Ondřej Lengál. 2025. *Artifact: A Uniform Framework for Handling Position Constraints for String Solving.* doi:10.5281/zenodo.15050654

[28] Yu-Fang Chen, Vojtěch Havlena, Michal Hečko, Lukáš Holík, and Ondřej Lengál. 2025. A Uniform Framework for Handling Position Constraints in String Solving (Technical Report). arXiv:2504.07033 [cs.LO] https://arxiv.org/abs/2504.07033

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

[30] David Chocholatý, Tomás Fiedor, Vojtech Havlena, Lukás Holík, Martin Hruska, Ondrej Lengál, and Juraj Síc. 2024. Mata: A Fast and Simple Finite Automata Library. In *Proc. of TACAS'24 (LNCS, Vol. 14571)*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer, 130–151. doi:10.1007/978-3-031-57249-4_7

[31] Joel D. Day, Vijay Ganesh, Nathan Grewal, Matthew Konefal, and Florin Manea. 2024. A Closer Look at the Expressive Power of Logics Based on Word Equations. *Theory Comput. Syst.* 68, 3 (2024), 322–379. doi:10.1007/S00224-023-10154-8

[32] 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

[33] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08 (LNCS, Vol. 4963)*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[34] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Proc. of CADE'07 (LNCS, Vol. 4603)*, Frank Pfenning (Ed.). Springer, 183–198. doi:10.1007/978-3-540-73595-3_13

[35] V. G. Durnev. 1995. The undecidability of the positive $\forall\exists^3$ theory of a free semigroup. *Sibirsk. Mat. Zh.* 36, 5 (1995), 1067–1080, ii. doi:10.1007/BF02112533

[36] Yeting Ge and Leonardo Mendonça de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories. In *Proc. of CAV'09 (LNCS, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 306–320. doi:10.1007/978-3-642-02658-4_25

[37] Christoph Haase. 2014. Subclasses of Presburger arithmetic and the weak EXP hierarchy. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 47:1–47:10. doi:10.1145/2603088.2603092

[38] Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2020. Monadic Decomposition in Integer Linear Arithmetic. In *Proc. of IJCAR'20 (LNCS, Vol. 12166)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, 122–140. doi:10.1007/978-3-030-51074-9_8

[39] Vojtech Havlena, Lukás Holík, Ondrej Lengál, and Juraj Síc. 2024. Cooking String-Integer Conversions with Noodles. In *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India (LIPIcs, Vol. 305)*, Supratik Chakraborty and Jie-Hong Roland Jiang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:19. doi:10.4230/LIPICS.SAT.2024.14

[40] 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. doi:10.1145/3158092

[41] John E. Hopcroft and Jean-Jacques Pansiot. 1979. On the Reachability Problem for 5-Dimensional Vector Addition Systems. *Theor. Comput. Sci.* 8 (1979), 135–159. doi:10.1016/0304-3975(79)90041-0

[42] Daisuke Ishii, Takashi Tomita, Toshiaki Aoki, The Quyen Ngo, Thi Bich Ngoc Do, and Hideaki Takai. 2022. SMT-Based Model Checking of Industrial Simulink Models. In *Formal Methods and Software Engineering: 24th International Conference on Formal Engineering Methods (ICFEM 2022) (LNCS, Vol. 13478)*. Springer, 156–172. doi:10.1007/978-3-031-17244-1_10

[43] Petr Janku and Lenka Turonová. 2019. Solving String Constraints with Approximate Parikh Image. In *Proc. of EUROCAST'19 (LNCS, Vol. 12013)*, Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia (Eds.). Springer, 491–498. doi:10.1007/978-3-030-45093-9_59

[44] Artur Jeż. 2016. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM* 63, 1 (2016), 4:1–4:51. doi:10.1145/2743014

[45] Ankit Jha, Rosemary Monahan, and Hao Wu. 2023. Verifying UML Models Annotated with OCL Strings. In *Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 123–132. doi:10.1145/3652620.3687822

[46] 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. doi:10.1145/2377656.2377662

[47] Felix Klaedtke and Harald Rueß. 2003. Monadic Second-Order Logics with Cardinalities. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings (LNCS, Vol. 2719)*, Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger (Eds.). Springer, 681–696. doi:10.1007/3-540-45061-0_54

[48] Quang Loc Le and Mengda He. 2018. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In *Proc. of APLAS'18 (LNCS, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 350–372. doi:10.1007/978-3-030-02768-1_19

[49] Jérôme Leroux and Grégoire Sutre. 2005. Flat Counter Automata Almost Everywhere!. In *Proc. of ATVA'05 (LNCS, Vol. 3707)*, Doron A. Peled and Yih-Kuen Tsay (Eds.). Springer, 489–503. doi:10.1007/11562948_36

[50] 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 (LNCS, Vol. 8244)*, Valeria Bertacco and Axel Legay (Eds.). Springer, 15–31. doi:10.1007/978-3-319-03077-7_2

[51] 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 *Proc. of CAV'14 (LNCS, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 646–662. doi:10.1007/978-3-319-08867-9_43

[52] 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. doi:10.1007/s10703-016-0247-6

[53] 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 (LNCS, Vol. 9322)*, Carsten Lutz and Silvio Ranise (Eds.). Springer, 135–150. doi:10.1007/978-3-319-24246-0_9

[54] 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. doi:10.1145/2837614.2837641

[55] 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). doi:10.46298/lmcs-17(4:4)2021

[56] Zhengyang Lu, Stefan Siemer, Piyush Jha, Joel Day, Florin Manea, and Vijay Ganesh. 2024. Layered and Staged Monte Carlo Tree Search for SMT Strategy Synthesis. In *Proc. of IJCAI'24*, Kate Larson (Ed.). International Joint Conferences on Artificial Intelligence Organization, 1907–1915. doi:10.24963/ijcai.2024/211 Main Track.

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

[58] S. S. Marchenkov. 1982. Undecidability of the positive ∀∃-theory of a free semigroup. *Sibirsk. Mat. Zh.* 23, 1 (1982), 196–198, 223.

[59] David Melski and Thomas Reps. 1997. Interconvertibility of Set Constraints and Context-Free Language Reachability. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*

(PEPM). ACM, 74–89. doi:10.1145/258993.259006

[60] Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. 2021. Z3str4: A Multi-armed string solver. In *Proc. of FM'21 (LNCS, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 389–406. doi:10.1007/978-3-030-90870-6_21

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

[62] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Clark W. Barrett, and Cesare Tinelli. 2022. Even faster conflicts and lazier reductions for string solvers. In *Proc. of CAV'22 (LNCS, Vol. 13372)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 205–226. doi:10.1007/978-3-031-13188-2_11

[63] Christos H. Papadimitriou. 1981. On the complexity of integer programming. *J. ACM* 28, 4 (1981), 765–768. doi:10.1145/322276.322287

[64] Wojciech Plandowski. 1999. Satisfiability of Word Equations with Constants is in PSPACE. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, 495–500. doi:10.1109/SFFCS.1999.814622

[65] Mathias Preiner, Hans-Jörg Schurr, Clark Barrett, Pascal Fontaine, Aina Niemetz, and Cesare Tinelli. 2024. *SMT-LIB release 2024 (non-incremental benchmarks)*. doi:10.5281/zenodo.11061097

[66] Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019. High-level abstractions for simplifying extended string constraints in SMT. In *Proc. of CAV'19 (LNCS, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 23–42. doi:10.1007/978-3-030-25543-5_2

[67] Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2020. Reductions for strings and regular expressions revisited. In *Proc. of FMCAD'20*. IEEE, 225–235. doi:10.34727/2020/isbn.978-3-85448-042-6_30

[68] 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 *Proc. of CAV'17 (LNCS, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 453–474. doi:10.1007/978-3-319-63390-9_24

[69] Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (LNCS, Vol. 5330)*. Springer, 274–289. doi:10.1007/978-3-540-89439-1_20

[70] Neha Rungta. 2022. A billion SMT queries a day (invited paper). In *Proc. of CAV'22 (LNCS, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 3–18. doi:10.1007/978-3-031-13185-1_1

[71] Joseph D. Scott, Pierre Flener, Justin Pearson, and Christian Schulte. 2017. Design and implementation of bounded-length sequence variables. In *Proc. of CPAIOR'17 (LNCS, Vol. 10335)*, Domenico Salvagnin and Michele Lombardi (Eds.). Springer, 51–67. doi:10.1007/978-3-319-59776-8_5

[72] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In *Proc. of PLDI'21* (Virtual, Canada). Association for Computing Machinery, New York, NY, USA, 620–635. doi:10.1145/3453483.3454066

[73] Wei-Lun Tsai. 2021. PyCT. https://github.com/alan23273850/PyCT

[74] Margus Veanes, Nikolaj S. Bjørner, Lev Nachmanson, and Sergey Bereg. 2017. Monadic Decomposition. *J. ACM* 64, 2 (2017), 14:1–14:28. doi:10.1145/3040488

[75] 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 *Proc. of CAV'16 (LNCS, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 241–260. doi:10.1007/978-3-319-41528-4_13

[76] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. Stranger: An automata-based string analysis tool for PHP. In *Proc. of TACAS'10 (LNCS, Vol. 6015)*, Javier Esparza and Rupak Majumdar (Eds.). Springer, 154–157. doi:10.1007/978-3-642-12002-2_13

[77] 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. doi:10.1007/s10703-013-0189-1

[78] 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. doi:10.1142/S0129054111009112

[79] 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 *Proc. of CAV'15 (LNCS, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 235–254. doi:10.1007/978-3-319-21690-4_14

[80] 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. doi:10.1145/2491411.2491456

Received 2024-11-15; accepted 2025-03-06