On Solving String Equations via Powers and Parikh Images

Clemens Eisenhofer[1(✉)], Theodor Seiser[1], Nikolaj Bjørner[2], and Laura Kovács[1]

[1] TU Wien, Vienna, Austria
clemens.eisenhofer@tuwien.ac.at
[2] Microsoft Research, Redmond, USA

Abstract. We present a new approach for solving string equations as extensions of Nielsen transformations. Key to our work are the combination of three techniques: a power operator for strings; generalisations of Parikh images; and equality decomposition. Using these methods allows us to solve complex string equations, including less commonly encountered SMT inputs over strings.

Keywords: String Equation Solving · SMT · Nielsen Transformation · String Powers · Parikh Image · Equality Decomposition

1 Introduction

String solving is used in a wide range of applications, including formal verification [1,9], security analysis [14,15,33,34], and automated reasoning [2,17]. Modern Satisfiability Modulo Theories (SMT) solvers [5], such as Z3 [24], CVC5 [3], or Princess [29], support string constraints via specialised solving techniques for handling length constraints [26], regular expressions [7,30], containment predicates [27], and many others. However, solving string (word) equations that involve long repeated subsequences or contain mutually dependent string variables remains very challenging for state-of-the-art solvers [1,9,10,12,20,23,24,33].

Example 1 (Motivating Example). Consider the conjunction of the following two string equations:

$$x_3x_3x_4bx_5b \simeq x_5x_5x_5x_5x_4bb, \quad (E_1)$$
$$x_1x_1acx_2x_4x_2x_5x_3bax_5x_3x_4x_3 \simeq x_2x_2abcx_1x_1x_3x_3x_3x_4x_4ax_4 \quad (E_2)$$

where $\simeq$ denotes (first-order) equality, x_1, x_2, x_3, x_4, x_5 are string variables and a, b, c are constant characters. Solving such equations requires reasoning about string variables depending on themselves or mutually on each other needs, which is mostly out of scope of current techniques.

G. L. Pozzato and T. Uustalu (Eds.): TABLEAUX 2025, LNAI 15980, pp. 120–138, 2026.
https://doi.org/10.1007/978-3-032-06085-3_7

Our work addresses challenges similar to solving equations (E_1)–(E_2). Our approach uses *rewriting and generating rules over string equations* (Table 1) as Nielsen transformations (Sect. 3) over Nielsen graphs [17,22,26]. We extend Nielsen transformation rules with:

Equality decomposition – to decompose string equations during reasoning into subequations. Using equality decomposition greatly increases the applicability of string power reasoning and Parikh images.

Explicit power representation – to solve string equations in which string variables depend on themselves. Furthermore, it allows us to compactly reason on long, but repetitive, strings (Sect. 5).

Generalized Parikh images – to detect unsatisfiability when both Nielsen rules and string power reasoning fail (Sect. 6). We adjust generalised forms of Parikh images [25] to string solving.

We implemented our framework as a prototype called ZIPT using the user-propagation framework [8] of the SMT solver Z3 [24]. Our experiments demonstrate the practical potential of our method (Sect. 7) on `SMT-LIB2` benchmarks [4] containing equations only.

2 Preliminaries

Strings. For our purposes, we will use an extended definition of a string term.

Definition 1 (Token & String Terms). *We fix a set $\mathcal{K}$ of tokens and denote by $\mathcal{T}$ the set of string terms. A* string term *$u \in \mathcal{T}$ is a finite sequence of* tokens *$t \in \mathcal{K}$ and we distinguish between*

- concrete character *tokens, denoted as a, b, c, d, whose set is denoted by Σ;*
- symbolic characters *tokens, written as o, whose set is $\mathcal{V}_C$.*
- (string) variable *tokens, denoted as x, y, z, whose set is $\mathcal{V}_S$;*
- power *tokens of the form u^m, where $u \in \mathcal{T}$ does not contain string variables and m is an arbitrary integer term potentially containing integer variables. We call u the* base *and m the* exponent *of a power token.*

Our token notation might use indices, and we write $@ \in \Sigma \cup \mathcal{V}_C$ to denote (concrete or symbolic) character tokens and $t \in \mathcal{T}$ for a token of any kind. The sets Σ, $\mathcal{V}_C$, and $\mathcal{V}_S$ are mutually disjoint and Σ is additionally finite and non-empty. The set $\mathcal{T}$ of string terms is the set of all finite sequences of tokens. Using regular expression notation, we have $\mathcal{T} := \mathcal{K}^*$ and reserve ε for the empty sequence of tokens. ε represents the neutral element of concatenation, and we denote concatenation by juxtaposing tokens. The power token u^m expresses that the token u is repeated m times; that is, $u^0 = \varepsilon$ and $u^m = uu^{m-1}$ if $m > 0$ where m is some integer term. Given how power terms are introduced, negative values for the exponents can be excluded. We denote by $|u|$ the *length* with $u \in \mathcal{T}$; clearly, $|u|$ represents a natural number. We write u^R to denote the reverse string term of u. For example, $(ax(ab)^m)^R = (ba)^m x^R a$. Finally, we note that a symbolic character o is equivalent to a string variable x with $|x| = 1$. We nonetheless use symbolic characters to have simpler definitions later on.

String Equations and Substitutions. A *string equation* is $u_1 \simeq u_2$ over string terms $u_1, u_2 \in \mathcal{T}$; here, we refer to u_1 as the left-hand-side (LHS) of the equation, whereas u_2 is the right-hand-side (RHS) of the equation. String equations that only contain string variables and concrete characters are called *plain*; string equations that might also contain symbolic characters and powers are called *extended*. We use 2^u to denote the set of *consecutive tokens* within u. $||u||$ denotes the number of tokens in u, where power tokens in u are considered atomic; we call this the *symbolic length* of u: $||\varepsilon|| := 0$ and $||tv|| := 1 + ||v||$. For example, in $u = (abc)^m xb$ we have $2^u = \{\ \varepsilon, a, b, c, (abc)^m, x, ab, bc, (abc)^m x, xb, abc, (abc)^m xb\ \}$ and $||u|| = 3$.

String terms $u \in \mathcal{T}$ containing no string variables are called *ground*. For ground terms u we thus have $2^u \cap \mathcal{V}_S = \emptyset$. Nonetheless, ground terms might contain power and symbolic character tokens.

A *substitution* σ maps (i) string variables $x \in \mathcal{V}_S$ to string terms $u \in \mathcal{T}$; and (ii) symbolic characters $o \in \mathcal{V}_C$ to characters $@ \in \Sigma \cup \mathcal{V}_C$. The set of variables $\{x \in \mathcal{V}_S \mid \sigma(x) \neq x\}$ is finite. Similarly, for symbolic characters. We write $(x/u) \in \sigma$ to denote $\sigma(x) = u$. A substitution σ is *eliminating* a string variable x if $(x/u) \in \sigma$ and u is ground. The *application of a substitution* σ to an expression C is denoted by $C[\sigma]$, where C is a string term or a (set of) string/integer (in)equations.

Substitutions σ are assumed to be acyclic and fully extended: For any non-identity $(x/u) \in \sigma$ and $y \in 2^u$ we have $(y/y) \in \sigma$. We write $m_1 \simeq m_2$ and $m_1 \leq m_2$ for integer (in)equations over integer terms m_1, m_2.

Interpretations and Models. Given a string equation $u \simeq v$, an *interpretation* $\mathcal{M}$ of this equation is a substitution together with an assignment of integer variables to integer values. For every $x \in \mathcal{V}_S \cap (2^u \cup 2^v)$ we have $(x/w) \in \mathcal{M}$, where $w \in \Sigma^*$; further, for every $o \in \mathcal{V}_C \cap (2^u \cup 2^v)$ we have $(o/a) \in \sigma$ with $a \in \Sigma$. An interpretation $\mathcal{M}$ is called a *plain model* of a set S of plain string equations if the application of $\mathcal{M}$ to S results in a simplified set of equations of the form $u \simeq u$ with $u \in \Sigma^*$. If S has a model $\mathcal{M}$, then S is *satisfiable*; otherwise S is *unsatisfiable*. Similarly, an *extended model* of a set of extended string equations and integer (in)equalities simplifies every extended string equation to $u \simeq u$ with $u \in \mathcal{T}^*$ and satisfies all integer constraints. All string variables and symbolic characters in u are implicitly assumed to be universally quantified. Any extended model of a set of plain string equations can be made into a plain model by substituting all symbolic characters with some character of Σ and string variables with some element of Σ^*. Power tokens can be completely unwound by using the integer value of their evaluated power term.

3 String Solving – Workflow

Given a non-empty *set S of plain string equations, our task is to decide its satisfiability.* Doing so, we solve extended string equations corresponding to S, potentially containing symbolic characters and powers as well as integer (in)equalities.

To this end, we expand Nielsen graphs of string equations (Sects. 3.1) by repeatedly applying an extended set of Nielsen transformation rules (Sects. 3.2). We simplify extended string equations and integer (in)equations, by extending Nielson transformations to arbitrary string terms $u \in \mathcal{T}$ rather than only plain ones. Our approach is summarised in Fig. 1 and detailed in the following.

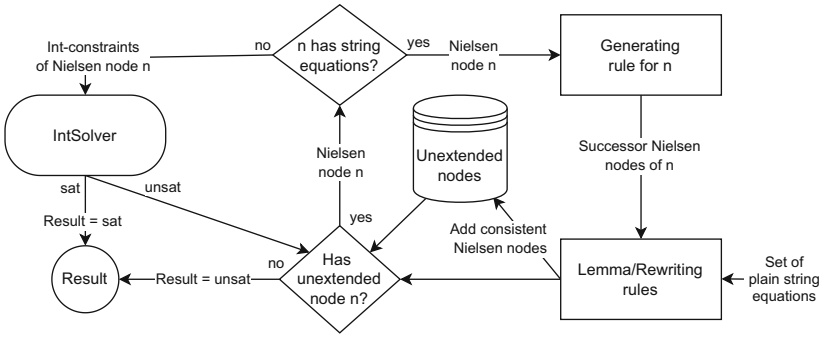

Fig. 1. String solving workflow.

3.1 Expansion of Nielsen Graphs

We detect the satisfiability of a set of string equations S with the help of Nielsen graphs [26], which we expand and simplify using our Nielsen transformation rules from Sect. 3.2.

Nielsen Graphs. Simply put, we consider a *(Nielsen) node* n to be the tuple $\langle \mathcal{E}(n), \mathcal{I}(n) \rangle$, where $\mathcal{E}(n)$ is a finite set of extended string equations and $\mathcal{I}(n)$ is its finite set of integer (in)equalities. In particular, the Nielsen node corresponding to the input set S of plain string equations is $\langle S, \emptyset \rangle$; this node is the *root node* of the Nielsen graph of S.

We assume that trivially satisfied constraints in $\mathcal{E}(n)$ and $\mathcal{I}(n)$, such as $\varepsilon \simeq \varepsilon$ or $0 \leq 1$, are implicitly removed from n. Given some integer (in)equation C, we write $n \models C$ to denote that we can derive C from $\mathcal{I}(n)$.

A Nielsen node n has a finite number of successor nodes and is called *inconsistent* if either (i) $\bot \in \mathcal{E}(n)$, or (ii) $\mathcal{I}(n) \models \bot$, or (iii) all successors nodes of n are inconsistent; where $\bot$ denotes the always false formula. Contrary, n is *satisfied* if (i) $\mathcal{E}(n) = \emptyset$, and (ii) $\mathcal{I}(n) \not\models \bot$. For checking $n \models C$, we assume access to a sound integer reasoner IntSolver.

The node n is *extended* if its successor nodes are already added to the Nielsen graph, given that n is neither satisfied nor inconsistent. Dually, a node that is not extended yet is called *unextended*. A Nielsen graph captures dependencies between Nielsen nodes and, as such, witnesses the (un)satisfiability of constraints by a *subgoal reduction*: if the constraints $\mathcal{E}(n) \cup \mathcal{I}(n)$ of node n have a model,

then at least one of the successors of n has a model as well. Nielsen graphs can thus be seen as a proof object or a variant of a tableau derivation.

String Solving via Expansion of Nielsen Graphs. We use Nielsen graphs to represent and manipulate our (initial) set S of (plain) string equations. Doing so, we apply extended Nielsen transformation rules (Sect. 3.2 and its extension in the succeeding sections) to simplify Nielsen nodes. In the sequel, let $\text{simpl}(n)$ denote the simplified version of Nielsen node n.

Satisfiability of the input set S of plain string equations is decided via recursively expanding the Nielsen graph of S, as shown in Fig. 1. We start by (i) adding the plain set of equations S to a Nielsen graph in the form of an unextended and simplified Nielsen node $\text{simpl}(\langle S, \emptyset \rangle)$ as the root of the Nielsen graph; here, we use the rewriting and lemma rules of Sect. 3.2. Next, (ii) we choose an arbitrary unextended Nielsen node n of the Nielsen graph and generate its successors $n_1, \ldots, n_k$ via some generating rule from Sect. 3.2. Then, (iii) we consider the simplified versions of the successor nodes $\text{simpl}(n_i)$ of n that are not inconsistent. These simplified successors of n are added as new unexpanded nodes to the Nielsen graph by connecting n_i with n via a respective edge – usually, we loop back to (ii). Once, (iv) a satisfied node n, reachable from the root node, is found, we report the initial set of equations S to be satisfiable. Satisfiability of S is implied by the construction of the expanded Nielsen graph: if a node n is satisfied, all its string equations $\mathcal{E}(n)$ have been removed and its remaining integer constraints $\mathcal{I}(n)$ are satisfiable. On the other hand, (v) if there are no unextended Nielsen nodes reachable from the root node anymore and no satisfied node has been found, we conclude the unsatisfiability of the input S.

Remark 1 (Termination of expanding Nielsen graphs). If a set of plain string equations S is satisfiable, there exists at least one satisfied Nielsen node n that is reachable from the root $\text{simpl}(\langle S, \emptyset \rangle)$. Yet, a fully extended Nielsen graph might be infinitely large and might contain infinitely many satisfied nodes. Termination of unsatisfiable instances is not guaranteed. As our rules (Sect. 3.2) are usually not invertible – n being satisfiable implies only one successors to be satisfiable as well – we require a fair tree traversal methods, such as breadth first or iterative deepening, in practice to make our approach terminate more robustly on satisfiable instances S.

3.2 Extended Nielsen Transformations Rules

Our string solving expands Nielsen graphs of string equations, by rewriting and generating Nielsen nodes n within the respective Nielsen graphs, as discussed in Sect. 3.1. Namely, we simplify some node n by $\text{simpl}(n)$ via (i) lemma rules, (ii) term rewriting rules, and (iii) equation rewriting rules, in order to obtain the "easier-to-handle" node $\text{simpl}(n)$. We use (iv) generating rules to introduce the successor nodes n' of n, where the constraints of n' result from the application of a substitution $\sigma_{n'}$ to the constraints of n and the addition of further constraints $C_{n'}$. That is, $n' = \text{simpl}(\langle \mathcal{E}(n)[\sigma_{n'}],\ \mathcal{I}(n)[\sigma_{n'}] \cup C_{n'} \rangle)$.

In the sequel, we fix an arbitrary Nielsen node n and introduce the following rules relative to n.

Lemma Rules. If $\mathcal{E}(n) \cup \mathcal{I}(n)$ contains the power tokens u^m, string equations $u \simeq v$, or string variables x, we add the respective integer constraints $m \geq 0$, $|u| \simeq |v|$, and $|x| \geq 0$ to $\mathcal{I}(n)$.

Term Rewriting Rules. In the sequel, we write $e_1 \rightsquigarrow e_2$ to mean that e_1 is replaced by e_2 everywhere in the considered Nielsen node. Term rewriting rules are used to simplify length constraints and rewrite power terms, as follows:

$$
\begin{array}{lll}
|uv| \rightsquigarrow |u| + |v| & |u^m| \rightsquigarrow m|u| & \varepsilon^m \rightsquigarrow \varepsilon \\
|\varepsilon| \rightsquigarrow 0 & |@| \rightsquigarrow 1 & v^m \rightsquigarrow \varepsilon, \text{ if } n \models m \simeq 0 \\
(v^{m_1})^{m_2} \rightsquigarrow v^{m_1 m_2} & v^{m_1} v^{m_2} \rightsquigarrow v^{m_1 + m_2} & v^m \rightsquigarrow v, \text{ if } n \models m \simeq 1
\end{array}
$$

A term is *fully rewritten* if no more rewriting rule can be applied to it. An (in)equality constraint is *fully rewritten* if all its terms are fully rewritten.

Example 2 (Computing simpl(n) *for Example 1).* Consider the two input equations of Example 1 and let n be their respective node n in the corresponding Nielsen graph. We drop the common b suffix in (E_1) and use lemma rules to add a length constraints for each string equation which can be rewritten into the fully rewritten constraints $2|x_3| \simeq 3|x_5|$ and $2|x_5| \simeq |x_4|$, using respective term rewriting rules. Finally, we add the side constraints $|x_i| \geq 0$ for all $1 \leq i \leq 5$. The resulting set of constraints defines simpl(n).

Equation Rewriting and Generating Rules. Equation rewriting rules remove some prefix of a string equation of $\mathcal{E}(n)$, sometimes under additional side conditions. In case no rewriting rule is applicable over n anymore, a generating rule over n is triggered to add additional information, which usually happens by considering multiple cases in the form of successor nodes.

Equation rewriting/generating rules for an extended string equation $u \simeq v \in \mathcal{E}(n)$ are given based on the first tokens of u and v. Our rewriting and generating rules are summarised in Table 1, listed per possible combinations of first tokens, or ε, of the LHS and RHS (columns 1–2 of Table 1). Primed variables x' in the table denote fresh string variables. A rule can be applied if the equation is of the given form and the side condition is satisfied in the current Nielsen node. We note that we can also apply all rules to $v \simeq u$, $u^R \simeq v^R$, and $v^R \simeq u^R$. Finally, any equation rewritten to $\varepsilon \simeq \varepsilon$ is removed from $\mathcal{E}(n)$, and similarly any trivially satisfied integer (in)equation.

Equation rewriting rules replace some equation $u \simeq v$ either by a new equation or detect a conflict $\perp$ (column 4), potentially requiring some side condition to be satisfied (column 3). On the other hand, generating rules apply substitutions (column 5) to n in order to generate successor node(s) n' of n or add additional new (integer) constraints to n' (column 6) to make further equation rewriting rules applicable.

Table 1. Equation rewriting rules and generating rules for an extended equation $u \simeq v$.

LHS	RHS	Rewriting rules		Generating rules	
		Side Cond.	New Equation	Substitution	New Cnstr.
tu	tv	–	$u \simeq v$	–	–
ε	av	–	$\bot$	–	–
ε	ov	–	$\bot$	–	–
ε	xv	–	–	x/ε	–
ε	$w^m v$	–	–	–	$\{w \simeq \varepsilon\}$
				–	$\{m \simeq 0\}$
au	bv	–	$\bot$	–	–
ou	$@v$	–	–	$o/@$	–
$^\dagger @u$	xv	–	–	x/ε	–
				$x/@x'$	–
$^\ddagger @u$	$w^m v$	$n \models m \simeq 0$	$@u \simeq v$	–	$\{m \simeq 0\}$
		$n \models m > 0$	$@u \simeq ww^{m-1}v$	–	$\{m > 0\}$
xu	yv	–	–	x/ε	–
				y/ε	$\{\|x\| > 0\}$
				y/xy'	$\{\|x\| > 0\}$
				x/yx'	$\{\|y\| > 0, \|x'\| > 0\}$
$^\dagger xu$	$w^m v$	–	–	$x/w^m x'$	–
				$x/w^{m'} p_1$	$s_1 \cup \{0 \leq m' < m\}$
				...	...
				$x/w^{m'} p_k$	$s_k \cup \{0 \leq m' < m\}$
$^\ddagger w^{m_1} u$	$w^{m_2} v$	$n \models m_1 \geq m_2$	$w^{m_1 - m_2} u \simeq v$	–	$\{m_1 \geq m_2\}$
		$n \models m_1 < m_2$	$u \simeq w^{m_2 - m_1} v$	–	$\{m_1 < m_2\}$
$^\ddagger w_1^{m_1} u$	$w_2^{m_2} v$	$n \models m_2 \simeq 0$	$w_1^{m_1} u \simeq v$	–	$\{m_2 \simeq 0\}$
		$n \models m_2 > 0$	$w_1^{m_1} u \simeq w_2 w_2^{m_2 - 1} v$	–	$\{m_2 > 0\}$

Example 3 (Generating rules for simpl(n) *in Example 1).* Consider the simplified node simpl(n) computed in Example 2 for Example 1. The set of applicable generating rules is given by the LHS/RHS token combinations of x_3 and x_5; x_5^R and b; x_1 and x_2; and x_3^R and x_4^R.

We stress that expanded Nielsen graphs using the rules of Table 1 correspond to a proof "tree", as illustrated next.

Example 4 (A fully expanded Nielsen graph for $xx \simeq yb$). Consider the plain string equation $xx \simeq yb$. We establish its satisfiability by constructing its expanded Nielsen graph, as presented in Sect. 3.1 and using the rules of Table 1. The fully expanded Nielsen graph of $xx \simeq yb$ is shown in Fig. 2. By reusing variable names, a single node could be used rather than multiple isomorphic ones.

Hence, $x' \simeq \varepsilon \wedge |x'| \simeq 0$ and $x'' \simeq \varepsilon \wedge |x''| \simeq 0$ could be contracted to the same node.

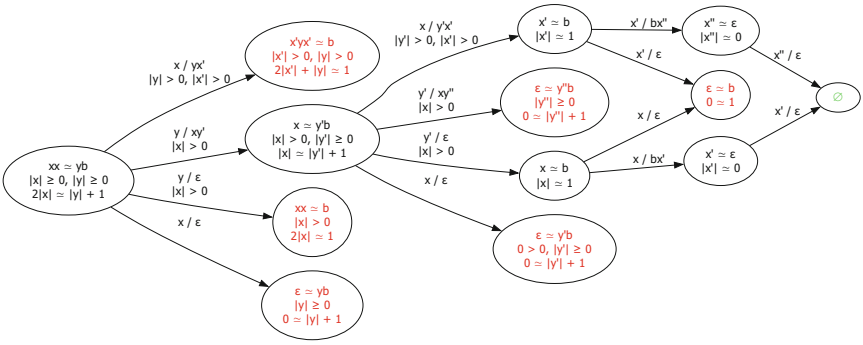

Fig. 2. Fully expanded Nielsen graph for the plain string equation $xx \simeq yb$, representing a variant of a tableau derivation.

We conclude this section by noting that the rules of Table 1 annotated by † and ‡ require special care, as discussed in Sect. 5. In particular, the case with xu on the LHS and $w^m v$ on the RHS requires $k+1$ branches given by the finite set $\mathrm{pre}(w) = \{\langle p_1, s_1\rangle, \ldots, \langle p_k, s_k\rangle\}$ which denotes the set of all syntactic prefixes p_i of w together with potential side conditions s_i. The set $\mathrm{pre}(w)$ is detailed in Sect. 5. We optimise the application of those rules by using more extended rules (Sects. 4–6) discussed next.

4 Equality Decomposition

This section provides a remedy for some cases when string equations cannot be solved using the Nielsen transformation rules of Sect. 3.2. Intuitively, we decompose string equations into subequations that can be further solved using our transformation rules. Note that the rules of Table 1 can only be applied upon the first (last) tokens of the LHS/RHS of string equations. With equality decomposition, we split equations into smaller ones. As such, the equality decomposition enables us to apply Table 1 on tokens at positions different from the first tokens of LHS/RHS.

In a nutshell, we proceed as follows. Consider an extended string equation $u_1 u_2 \simeq v_1 v_2 \in \mathcal{E}(n)$ and let $d = |u_1| - |v_1|$ be a known integer constant. Equality decomposition is applied by splitting the equation into two equations, potentially padding one side each by d characters. If $d = 0$, the equation $u_1 u_2 \simeq v_1 v_2$ can be decomposed into the set $\{u_1 \simeq v_1, u_2 \simeq v_2\}$ of two smaller equations. If all string terms u_1, u_2, v_1, v_2 contain at least one string variable or a power token, we replace $u_1 u_2 \simeq v_1 v_2$ in $\mathcal{E}(n)$ by $\{u_1 \simeq v_1, u_2 \simeq v_2\}$. For example, we decompose $xayw \simeq ybxz$ into $xay \simeq ybx$ and $w \simeq z$, as $|xay| - |ybx| = 0$.

On the other hand, if $d > 0$, then decomposing $u_1u_2 \simeq v_1v_2$ comes with the additional requirement that the last d characters of u_1 are the same as the first d characters of v_2. To do so, we introduce $|d|$ fresh symbolic character tokens $\bar{o_d}' = o'_1, \ldots, o'_d$ to represent these d characters of u_1. Note again, this is equivalent to introducing one fresh string variable x with $|x| = d$. Then, $u_1u_2 \simeq v_1v_2$ is decomposed into the equations:

$$\{\ u_1 \simeq v_1\bar{o'_d},\ \bar{o'_d}u_2 \simeq v_2\ \}.$$

Our equality decomposition approach thus uses length constraints $|u_1| - |v_1|$, which can be facilitated by the integer constraints $\mathcal{I}(n)$ of n. Our approach can be seen as a strengthened variant of the usual decomposition rule without padding, primarily used as a preprocessing rule in other solvers: We use d symbolic character tokens for padding string equations in order to enable their decomposition.

Example 5 (Equality decomposition in Example 1). Using the lemma and term rewriting rules of Example 2, we derive:

$$2|x_3| \simeq 3|x_5| \wedge 2|x_5| \simeq |x_4|.$$

Consequently, we infer $d = |x_1x_1acx_2x_4x_2x_5| - |x_2x_2abcx_1x_1x_3x_3| = 1$. We decompose (E_2) using a single symbolic character o and replace (E_2) by the resulting two new equations (E_3) and (E_4):

$$x_1x_1acx_2x_4x_2x_5o \simeq x_2x_2abcx_1x_1x_3x_3, \qquad (E_3)$$
$$x_3bax_5x_3x_4x_3 \simeq ox_3x_4x_4ax_4 \qquad (E_4)$$

In other words, (E_2) is replaced by (E_3) and (E_4) in $\mathcal{E}(n)$, and we further simplify n by lemma and term rewriting rules.

5 Ground Power Introduction

We introduce power terms to eliminate lengthy and repetitive substrings in string equations, and compress sequences of rewriting and generating rule applications into power terms. Power terms allow us to eliminate a potentially unbounded number of equation decompositions when applying substitutions of the form $x/@x'$; here, x' is again a fresh string variable. For example, when $xu = axv$ is satisfiable, all its models need x to be a^m, for some $m \geq 0$. Finding such models using equality decomposition in combination with the rules of Table 1 requires analysing infinitely many token combinations for some unsatisfiable equations. Even in the case of satisfiable problem instances, the required analysis can become very long. In order to circumvent this, we use rules based on the property that $xu = vx$, with $v \neq \varepsilon$, implies $x = (w_1w_2)^mw_1$, $v = w_1w_2$, and $u = w_2w_1$ for some natural number m and words w_1 and w_2 [21].

Power Introduction and Rules † *of Table 1.* The rules of Table 1 that are annotated by † are only applied when power introduction is not applicable. In such cases, given a ground string term $w \neq \varepsilon$ and constraint $xu \simeq wxv$, we have:

$$\exists m.\ \bigvee_{\langle p,s\rangle \in \mathrm{pre}(w)} \left(x \simeq w^m p \wedge \bigwedge s \right) \vee w \simeq \varepsilon \tag{1}$$

where $\mathrm{pre}(w)$ denotes the set of pairs $\langle p, s\rangle$ of possible strict prefixes p of w, such that there is only a finite set s of side conditions to constrain p. Formally, $\mathrm{pre}(w)$ is defined for a ground string term w as:

- $\mathrm{pre}(uv) := \mathrm{pre}(u) \cup \{\langle up, s\rangle \mid \langle p, s\rangle \in \mathrm{pre}(v)\}$,
- $\mathrm{pre}(\varepsilon) := \emptyset$,
- $\mathrm{pre}(@) := \{\langle \varepsilon, \emptyset\rangle\}$,
- $\mathrm{pre}(u^m) := \{\langle u^{m'} p, \{0 \leq m' < m\} \cup s\rangle \mid \langle p, s\rangle \in \mathrm{pre}(u)\}$

where m' is a fresh integer variable.

Let us make the following observation on the interplay between power introductions and prefix $\mathrm{pre}(w)$ analysis. When considering equations $xu \simeq wxv$, an unsoundness problem may arise when w represents ε in a potential model, as w would have cancelled out. Such cases may happen when w consists of power tokens all becoming ε because their exponents are zero. Based on the definition of $\mathrm{pre}(w)$, we would introduce a fresh m' integer variable with $0 \leq m' < m$, which yields a conflict when $m = 0$ and thus falsely eliminates a model. As a remedy, we also explicitly split upon $w \simeq \varepsilon$ to cover this case. On the other hand, if w is not ε, power introduction allows us to eliminate string variables x by replacing them with the help of integer constraints, as shown next.

Example 6 ($xbxa \simeq axbx$). Without introducing powers, applying the rules of Table 1 would fail to solve the unsatisfiable equation $xbxa \simeq axbx$. Applying the rules from Table 1 leads to an infinite chain of substitutions of the form x/ax', resulting in increasingly longer string terms. By introducing power terms, we eliminate x via the single substitution x/a^m: applying this substitution to $xbxa \simeq axbx$ yields $a^m b a^m a \simeq a a^m b a^m$ that can be easily simplified to $\bot$.

Power Introduction and Rules ‡ of Table 1. We next comment on the use of power introduction together with the Nielsen transformation rules annotated by ‡ in Table 1. When the first token of some side of an equation is a power token, we first check whether the other side of the equation can be rewritten so that it starts with a power token with the same base, so that they can cancel out. For doing so, we apply additional term rewriting rules:

$$\begin{array}{rcl@{\qquad}rcl} ww^m & \rightsquigarrow & w^{m+1} & w_2\,(w_1 w_2)^m & \rightsquigarrow & (w_2 w_1)^m\, w_2 \\ w^m w & \rightsquigarrow & w^{m+1} & (w_1 w_2)^m\, w_1 & \rightsquigarrow & w_1\,(w_2 w_1)^m \\ ww & \rightsquigarrow & w^2 & & & \end{array}$$

These additional rules considerably reduce the number of equational decompositions when introducing powers. However, more importantly, they are necessary to handle equations like $a(ba)^m u \simeq (ab)^m v$. In case the solver does not realize that the prefixes can be rewritten into the same power, the rules of Table 1 would suggest an explicit unwinding for each possible value of m.

Strengthening Power Reductions. We generalize the rule of (1) over $xu \simeq wxv$ to be applicable over sets of equations $EQ \subseteq \mathcal{E}(n)$ given by:

$$EQ := \{\ w_1 x_k u_1 \simeq x_1 v_1,\ w_2 x_1 u_2 \simeq x_2 v_2,\ \ \ldots,\ w_k x_{k-1} u_k \simeq x_k v_k\ \},$$

with $k \geq 1$, and all terms $w_1, \ldots, w_k$ being ground. Similarly to (1), the set EQ enforces the more general form of power introduction

$$\exists m.\ \bigvee_{1 \leq i \leq k}\ \bigvee_{\langle p,s\rangle \in \mathrm{pre}(\bar{w}_i)} \left(x_i \simeq \bar{w}_i^m p \wedge \bigwedge s \right) \vee \bar{w}_1 \simeq \varepsilon, \tag{2}$$

where $\bar{w}_i := w_i w_{i-1} \ldots w_1 w_k \ldots w_{i+1}$.

Example 7 (Power introduction in Example 5). After decomposing (E_2) into (E_3) and (E_4), we introduce a power token that is justified by the prefix of (E_4): the LHS of (E_4) starts with x_3 and the RHS with ox_3. As the string term consisting only of the symbolic character o is ground and cannot be ε, we apply power introduction and consider only one successor node as $\mathrm{pre}(o) = \{\langle \varepsilon, \emptyset \rangle\}$: the successor is generated by x_3/o^{m_1} for some fresh constant m_1. From a lemma rule, we get $m_1 \geq 0$ and we rewrite the length constraints $|x_3[x_3/o^{m_1}]|$ to m_1. The string equations (E_1), (E_3) and (E_4) respectively become:

$$\begin{aligned}
o^{2m_1} x_4 b x_5 &\simeq x_5 x_5 x_5 x_5 x_4 b, && (E_1')\\
x_1 x_1 a c x_2 x_4 x_2 x_5 o &\simeq x_2 x_2 a b c x_1 x_1 o^{2m_1}, && (E_3')\\
o^{m_1} b a x_5 o^{m_1} x_4 o^{m_1} &\simeq o o^{m_1} x_4 x_4 a x_4. && (E_4')
\end{aligned}$$

Thanks to the additional rewrite rules introduced previously in this section, token oo^{m_1} on the RHS of (E_4') can be rewritten to o^{m_1+1}. As such, common prefixes of (E_3') are removed, and we apply o/b. We get the respective equations of (E_1'), (E_3') and (E_4') as:

$$\begin{aligned}
b^{2m_1} x_4 b x_5 &\simeq x_5 x_5 x_5 x_5 x_4 b, && (E_1'')\\
x_1 x_1 a c x_2 x_4 x_2 x_5 b &\simeq x_2 x_2 a b c x_1 x_1 b^{2m_1}, && (E_3'')\\
a x_5 b^{m_1} x_4 b^{m_1} &\simeq x_4 x_4 a x_4. && (E_4'')
\end{aligned}$$

Further powers can be introduced, either by using (E_4'') because of the suffix $x_4 b^{m_1}$ on the LHS and x_4 on the RHS, or via the generalised form of power introduction (2) using equations (E_1'') and (E_4''). In the latter case, we end up with the four successor nodes given by:

$$\{\ x_4/(ab^{2m_1})^{m_2},\ x_4/(ab^{2m_1})^{m_2} a b^{m_3},\ x_5/(b^{2m_1} a)^{m_2} b^{m_3},\ x_5/(b^{2m_1} a)^{m_2} b^{2m_1}\ \}$$

and $m_3 < 2m_1$. In the former case, we have two cases: applying $x_4/b^{m_3}(b^{m_1})^{m_2}$ with $m_3 < m_1$ – which simplifies to $b^{m_1 m_2 + m_3}$ – or adding $b^{m_1} \simeq \varepsilon$. We chose

the former case and consider the first successor first: applying $x_4/b^{m_3}(b^{m_1})^{m_2}$.

$$b^{2m_1+m_1m_2+m_3}bx_5 \simeq x_5x_5x_5x_5b^{m_1m_2+m_3}b, \quad (E_1''')$$
$$x_1x_1acx_2b^{m_1m_2+m_3}x_2x_5b \simeq x_2x_2abcx_1x_1b^{2m_1}, \quad (E_3''')$$
$$ax_5b^{2m_1} \simeq b^{2m_1m_2+2m_3}a. \quad (E_4''')$$

As $0 \leq m_3 < m_1$, we derive $m_1 > 0$ and thus we unwind b^{2m_1} to $b^{2m_1-1}b$ in (E_4''') using the respective rule from Table 1, resulting in a conflict. We therefore consider the successor node generated by adding the constraint $b^{m_1} \simeq \varepsilon$. This implies $m_1 = 0$, eliminating all power tokens and allowing us to derive $|x_4| = |x_5| = 0$. After further simplifications, we derive x_4/ε and x_5/ε. As such, we end up with the Nielsen node containing the single plain string equation:

$$x_1x_1acx_2x_2b \simeq x_2x_2abcx_1x_1.$$

6 Parikh Image

Parikh images [25] can be an effective method to prove sets of string constraints to be unsatisfiable by extracting constraints on the number of occurrences of terminal characters [31]. It represents an abstraction of string equations that elides information about character positions, retaining only the number of occurrences. In the following, we establish a method that retains information about the relative positions of characters, resulting in a tighter abstraction. Our method computes a value we call *(error-bounded) multi-sequence Parikh image*.

To formalise the Parikh images, let $\pi_a(u)$ be a function that represents the set of positions on which there is an $a \in \Sigma$ in u. The set $\{ (a, \pi_a(u)) \mid a \in \Sigma \}$ is a complete representation of u. Let $\alpha_a(u) := |\pi_a(u)|$, then an equation $u \simeq v$ can be over-approximated by $\bigwedge_{a\in\Sigma} \alpha_a(u) \simeq \alpha_a(v)$. For example, $\alpha_a(xay) = \alpha_a(x) + 1 + \alpha_a(y)$ and $\alpha_a(xby) = \alpha_a(x) + \alpha_a(y)$, so the equation $xay \simeq ybx$ can be recognised as unsatisfiable immediately. We note that neither equality decomposition nor ground power introduction combined with the rules of Table 1 can detect this equation as unsatisfiable. Still, Parikh constraints based on single concrete characters fail to capture the infeasibility of equations such as the equation in the only Nielsen node of our running example: $x_1x_1acx_2x_2b \simeq x_2x_2abcx_1x_1$. Our method is going to accomplish this.

6.1 Parikh Images for Unbordered Patterns

In the following, we consider Parikh images for strings $w \in \Sigma^+$ of length greater than 1 – we call these w *patterns* – and assume that we have plain string equations. For the scope of this paper, we consider symbolic characters and power tokens as string variables when computing the Parikh image. For example, in the plain equation $xabcy \simeq ybacx$, pattern bc occurs on the LHS of the equation $\alpha_{bc}(xabcy) = \alpha_{bc}(x) + 1 + \alpha_{bc}(y)$ times and $\alpha_{bc}(ybacx) = \alpha_{bc}(y) + \alpha_{bc}(x)$ times on the RHS. The equation is therefore unsatisfiable. Our running example also offers an opportunity to use such a pattern:

Example 8 (Parikh image in Example 7). Consider the equation in our running Example 7. We can consider pattern $w = abc$ and thus get $\alpha_w(x_1x_1acx_2x_2b) \simeq \alpha_w(x_2x_2abcx_1x_1)$. This equation can be intuitively rewritten to $\alpha_w(x_1x_1) + \alpha_w(x_2x_2) \simeq \alpha_w(x_2x_2) + 1 + \alpha_w(x_1x_1)$ and further to $0 = 1$ which proves the equation itself, as well as the overall running example unsatisfiable, as all successor Nielsen nodes of the root node are inconsistent. In the following, we will formally define this "intuitive rewriting".

In contrast to usual Parikh images over singleton characters, there is no general notion of $\alpha_w(u)$ that decomposes over string constants and string variables in u. For example, we cannot decompose $\alpha_{ab}(ybacx)$ into a sum over the components of $ybacx$ because the value depends on whether x ends with a or not.

We address this limitation by

1. Restricting Parikh images to character sequence patterns w such that no proper suffix of w is a prefix of w. Such words are called *unbordered*.
2. Defining over- $\alpha^{\uparrow}_w(u)$ and under-approximations $\alpha^{\downarrow}_w(u)$ such that unsatisfiability of a string equation $u \simeq v$ can be established when $\alpha^{\uparrow}_w(u) - \alpha^{\downarrow}_w(v)$ or, symmetrically, $\alpha^{\uparrow}_w(v) - \alpha^{\downarrow}_w(u)$ rewrite to a negative constant.

Our rewriting rules for $\alpha^{\uparrow}_w(u)$ and $\alpha^{\downarrow}_w(u)$ satisfy the following properties:

Lemma 1 (Correctness of Parikh images for Unbordered Patterns). *Assume w is an unbordered pattern.*

- *If $u \in \Sigma^*$ then $\alpha^{\uparrow}_w(u) = \alpha^{\downarrow}_w(u) =$ number of occurrences of w in u.*
- *Assume $\alpha^{\uparrow}_w(u)$ rewrites to $k + \sum_x c_x\alpha_w(x)$ for natural numbers c_x and k. Then for every substitution σ from string variables to Σ^*, $\alpha^{\uparrow}_w(u[\sigma]) \leq k + \sum_x c_x\alpha^{\uparrow}_w(x[\sigma])$. A symmetric property holds for $\alpha^{\downarrow}_w(u)$.*

It remains to define the over- and under-approximations. As the examples suggested, the restriction to unbordered patterns means that there are many cases where the under- and over-approximations coincide.

Definition 2 (d-gaps and $w \bowtie u$).

***d*-gaps** *We call a string term u a d-*gap *(with $d > 1$) iff it is of one of the following forms: $u = xvy$ (with $v \in \Sigma^*$), $u = xv$, or $u = vy$ ($v \in \Sigma^+$ in the latter two cases) and $0 < ||u|| \leq d$.*

w* ⋈ *u *If w is a pattern and u is a $|w|$-gap, we write $w \bowtie u$ to denote that there can be crossing occurrences of w within a $|w|$-gap u. Formally, $w \bowtie u$ is true if $u = xw_2y$ and we can decompose $w = w_1w_2w_3$ ($\{w_1, w_3\} \neq \varepsilon$), otherwise it is false. Similarly, the cases $u = w_1y$ requires $w = w_1w_2$ ($w_2 \neq \varepsilon$) and in cases $u = xw_2$ requires $w = w_1w_2$ ($w_1 \neq \varepsilon$)*

Using the definition of a gap, apply the following term rewriting rules to rewrite stepwise $\alpha_w^{\downarrow}(u)/\alpha_w^{\uparrow}(u)$ into the desired grouping for an unbordered pattern w:

$$\begin{aligned}
\alpha_w^{\updownarrow}(x) &\rightsquigarrow \alpha_w(x) && \\
\alpha_w^{\updownarrow}(w_2) &\rightsquigarrow 0 && \text{if } w_2 \in \Sigma^* \wedge |w_2| < |w| \\
\alpha_w^{\updownarrow}(uwv) &\rightsquigarrow 1 + \alpha_w^{\updownarrow}(u) + \alpha_w^{\updownarrow}(v) && \\
\alpha_w^{\updownarrow}(ua_1w_2a_2v) &\rightsquigarrow \alpha_w^{\updownarrow}(ua_1w_2) + \alpha_w^{\updownarrow}(w_2a_2v) && \text{if } a_1w_2a_2 \in \Sigma^+ \wedge \\
& && \quad |a_1w_2a_2| = |w| \wedge a_1w_2a_2 \neq w \\
\alpha_w^{\updownarrow}(uxw_2yv) &\rightsquigarrow \alpha_w^{\updownarrow}(uxw_2) + \alpha_w^{\updownarrow}(w_2yv) && \text{if } xw_2y \text{ is } |w|\text{-gap and } w \not\bowtie xw_2y \\
\alpha_w^{\updownarrow}(aw_2yv) &\rightsquigarrow \alpha_w^{\updownarrow}(w_2yv) && \text{if } aw_2y \text{ is } |w|\text{-gap and } w \not\bowtie aw_2y \\
\alpha_w^{\updownarrow}(uxw_2a) &\rightsquigarrow \alpha_w^{\updownarrow}(uxw_2) && \text{if } xw_2a \text{ is } |w|\text{-gap and } w \not\bowtie xw_2a
\end{aligned}$$

We now define under- and over-approximations for cases not covered by the previous exact rewrite rules. They are:

$$\begin{aligned}
\alpha_w^{\downarrow}(x) &\rightsquigarrow \alpha_w(x) & \alpha_w^{\uparrow}(x) &\rightsquigarrow \alpha_w(x) & \\
\alpha_w^{\downarrow}(w_2u) &\rightsquigarrow \alpha_w^{\downarrow}(u) & \alpha_w^{\uparrow}(w_2xu) &\rightsquigarrow 1 + \alpha_w^{\uparrow}(xu) & \text{if } w_2 \in \Sigma^+ \\
\alpha_w^{\downarrow}(xw_2) &\rightsquigarrow \alpha_w(x) & \alpha_w^{\uparrow}(xw_2) &\rightsquigarrow 1 + \alpha_w(x) & \text{if } w_2 \in \Sigma^+ \\
& & \alpha_w^{\uparrow}(xw_2yv) &\rightsquigarrow 1 + \alpha_w(x) + \alpha_w^{\uparrow}(yv) & \text{if } w_2 \in \Sigma^*
\end{aligned}$$

The correctness of the rewrite rules relies on first applying the non-approximate rules exhaustively before considering the cases presented by the approximations. Informally speaking, we require all u in the remaining $\alpha_w(u)$ to be "concatenations of $|w|$-gaps" that can contain crossing occurrences.

Our procedure for filtering unsatisfiable equations $u \simeq v$ is now as follows:

(i) Enumerate maximal unbordered patterns $w \in \Sigma^*$ occurring in u and v. The patterns are maximal only w.r.t. the pattern within the considered side u or v.
(ii) For each of these w rewrite $\alpha_w^{\uparrow}(u) - \alpha_w^{\downarrow}(v)$ to a sum $k + \sum_x c_x \alpha_w(x)$. If each c_x is 0 and $k < 0$, we can conclude the equation $u \simeq v$ to be unsatisfiable.

Example 9 ($xaxaabbby \simeq xyababababx$). Consider the unbordered pattern $w = ab$, which is maximal within the RHS. Then $\alpha_w^{\uparrow}(xaxaabbby) \rightsquigarrow 2\alpha_w(x) + 2 + \alpha_w(y)$ and $\alpha_w^{\downarrow}(xyababababx)$ rewrites to $\alpha_w(y) + 3 + 2\alpha_w(x)$. Thus, $\alpha_w^{\uparrow}(xaxaabbby) - \alpha_w^{\downarrow}(xyababababx)$ is -1; witnessing that the equation is unsatisifable.

7 Implementation and Experiments

Implementation. We implemented our approach within a new prototype ZIPT. Our implementation[1] uses the Z3 [24] SMT solver both as the auxiliary integer

[1] Available at https://github.com/CEisenhofer/ZIPT.

solver, as well as for the general CDCL($\mathcal{T}$) framework, calling our string solving procedure with the set of plain string equations using user-propagation [8].

Our implementation follows the workflow of Fig. 1, by using the Nielsen transformation rules of Table 1 in combination with equality decomposition 4, power introduction Sect. 5, and Parikh images Sect. 6. If multiple generating rules of Table 1 are applicable, we choose the one predicted to cause faster termination. For example, rules eliminating string variables are prioritised over those that do not.

In addition to the generating rules presented, we use look-ahead heuristics to prefer generating rules in which all but one generated successor node results in a conflict. Examples for such cases are:

- Eliminating some string variable x using x/ε with $xu_1v_1 \simeq u_2v_2$, where $u_1 \in (\{x\} \cup \mathcal{V}_C \cup \Sigma)^+$ and $u_2 \in (\mathcal{V}_C \cup \Sigma)^+$. Unless a power introduction rule is applicable, we can compute the longest prefix w of u_2 such that we can safely apply x/wx'. For example, $xxbau \simeq abv$ gives x/abx' without branching, as both x/ε and x/a would yield a conflict immediately. A similar case applies if x was a power token;
- Unwinding a power token $w_1^m u \simeq w_2 v$ with $w_2 \in (\mathcal{V}_C \cup \Sigma)^+$. Whenever w_2 cannot be a prefix or a suffix of any word we can get by unwinding w_1^m at least once, we conclude $m = 0$. For example, $(ab^{m_2}c)^{m_1}u = aav$ enforces $m_1 = 0$ as aa is not a prefix of $ab^{m_2}c$ for any m_2 nor is it a suffix;
- Length constraints where we can deduce $n \models |x| > |y|$ in our current Nielsen node n. If we consider $xu \simeq yv$, we have to apply x/yx'. Similarly, if $n \models |x| = |y|$, we chose x/y.

For traversing and expanding the Nielsen graph, we rely on a variant of iterative deepening, in which successors of node n that strictly reduce the constraints of n are expanded in more depth.

Experimental Setup. As such, we ran our solver ZIPT on the four `QF_S` tracks of the `woorpje` benchmark set [12] of SMT-LIB [4]; this benchmark set consists of 409 benchmark files containing only string equations. We compared our work against the state-of-the-art solvers competing in the most recent SMT-COMP competition[2]. We used a 10 seconds timeout, 8 GB of RAM, and default solver configurations, based on a dedicated core of an Intel i7-13850HX CPU.

Experimental Analysis. Our results are summarized in Table 2, showcasing that `ZIPT` outperforms the state-of-the-art. When solving problems within `track 02`, our approach benefits from power introduction; this is so because models described by these benchmarks are exponential in the number of string variables. Using (nested) power tokens allows us to solve such examples without taking exponentially many steps, which is a key difference compared to other works.

[2] https://smt-comp.github.io/2024/.

Table 2. Number of solved problems using string solving benchmarks from the `woorpje` benchmark set of SMT-LIB, organised in four tracks. The number of overall problems in each track is listed in column 8. We compare our solver `ZIPT` to the related approaches of Z3 [24], CVC5 [3], OSTRICH [9], Z3-NOODLER [10], and Z3STR3 [6].

	ZIPT	Z3	CVC5	OSTRICH	Z3-NOODLER	Z3STR3	Total
`track 01`	200	198	191	198	200	200	200
`track 02`	9	4	1	4	6	6	9
`track 03`	195	176	164	127	190	190	200
`track 04`	200	198	200	198	199	200	200

8 Related Work

Decomposing string equations into subsequences is typically used mainly as a preprocessing step in string solvers. Even though it is primarily only applied in cases where no padding is required, a different variation of our padding has been described in detail in previous works about string solving [28].

Recognising self-dependencies in string variables, especially via explicit powers, is studied in [16,19]. While the languages definable by such equations are generally EDT0L [11], the explicit powers we consider fall within more tractable subclasses. Theoretical results about word equations expressing certain power terms date back even longer [21]. In terms of solver support, the approach of [20] can handle some self-dependencies using regular expressions. In contrast, we identify a broader class of self-dependencies and introduce explicit powers, enabling nesting powers for efficiency (see `track 02` in Sect. 7).

Parikh images improve string reasoning via regular expression membership [9, 31]. Most techniques use single-character Parikh images to detect trivial unsatisfiability. We generalize this by considering Parikh information over multiple character sequences simultaneously. Related notions such as Parikh matrices [32] and their role in decidability have been explored in [13]. Our approach of analysing potential crossing occurrences is strongly related to recompression [18], where such occurrences are stepwise eliminated using case distinctions.

9 Conclusions

We introduced a string solving approach, based on Nielsen transformation rules extended with equality decomposition, power introduction, and Parikh images. Our initial results show the practical potential of our work. As further work, we plan to improve the analysis via Parikh images by using tighter error approximations and relating the Parikh information from multiple equations rather than considering each equation in isolation. Other steps include heuristically splitting equations using auxiliary variables based on dependency analysis and the introduction of non-ground power terms in order to express dependencies that cannot

be represented in our calculus currently. Further, supporting other string-related SMT-LIB functions, including regular expressions, as defined in the SMT-LIB standard [4], is an essential next step.

Acknowledgements. This research was funded in whole or in part by the ERC Consolidator Grant ARTIST 101002685, the ERC Proof of Concept Grant LEARN 101213411, the TU Wien Doctoral College SecInt, the FWF SpyCoDe Grant 10.55776/F85, the WWTF grant ForSmart 10.47379/ICT22007, and the Amazon Research Award 2023 QuAT.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abdulla, P.A., et al.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_10
2. Amadini, R.: A survey on string constraint solving. ACM Comput. Surv. **55**(2), 16:1–16:38 (2023). https://doi.org/10.1145/3484198
3. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: version 2.7. Technical report, Department of Computer Science, The University of Iowa (2025)
5. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021). https://doi.org/10.3233/FAIA201017
6. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: FMCAD, pp. 55–59. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102241
7. Berzish, M., et al.: An SMT solver for regular expressions and linear arithmetic over string length. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 289–312. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_14
8. Bjørner, N.S., Eisenhofer, C., Kovács, L.: Satisfiability modulo custom theories in Z3. In: VMCAI. LNCS, vol. 13881, pp. 91–105. Springer (2023). https://doi.org/10.1007/978-3-031-24950-1_5
9. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019). https://doi.org/10.1145/3290362
10. Chen, Y., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síc, J.: Z3-noodler: an automata-based string solver. In: TACAS. LNCS, vol. 14570, pp. 24–33. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_2
11. Ciobanu, L., Diekert, V., Elder, M.: Solution sets for equations over free groups are EDT0L languages. Int. J. Algebra Comput. **26**(5), 843–886 (2016). https://doi.org/10.1142/S0218196716500363

12. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In: Filiot, E., Jungers, R., Potapov, I. (eds.) RP 2019. LNCS, vol. 11674, pp. 93–106. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30806-3_8
13. Day, J.D., Ganesh, V., He, P., Manea, F., Nowotka, D.: The satisfiability of word equations: decidable and undecidable theories. In: Potapov, I., Reynier, P.-A. (eds.) RP 2018. LNCS, vol. 11123, pp. 15–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00250-3_2
14. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: ISSTA, pp. 151–162. ACM (2007). https://doi.org/10.1145/1273463.1273484
15. Eriksson, B., Stjerna, A., Masellis, R.D., Rümmer, P., Sabelfeld, A.: Black ostrich: web application scanning with string solvers. In: SIGSAC, pp. 549–563. ACM (2023). https://doi.org/10.1145/3576915.3616582
16. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 209–226. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_21
17. Hague, M.: Strings at MOSCA. ACM SIGLOG News **6**(4), 4–22 (2019). https://doi.org/10.1145/3373394.3373396
18. Jez, A.: Recompression: a simple and powerful technique for word equations. J. ACM **63**(1), 4:1–4:51 (2016). https://doi.org/10.1145/2743014
19. Karhumäki, J., Mignosi, F., Plandowski, W.: The expressibility of languages and relations by word equations. J. ACM **47**(3), 483–505 (2000)
20. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C.W., Deters, M.: An efficient SMT solver for string constraints. Formal Methods Syst. Des. **48**(3), 206–234 (2016). https://doi.org/10.1007/S10703-016-0247-6
21. Lyndon, R.C., Schützenberger, M.P., et al.: The equation $a^M = b^N c^P$ in a free group. Michigan Math. J **9**(4), 289–298 (1962)
22. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **145**(2), 147–236 (1977)
23. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: a multi-armed string solver. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 389–406. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_21
24. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
25. Parikh, R.: On context-free languages. J. ACM **13**(4), 570–581 (1966). https://doi.org/10.1145/321356.321364
26. Przybocki, B., Barrett, C.W.: The termination of Nielsen transformations applied to word equations with length constraints. CoRR (2025). https://doi.org/10.48550/ARXIV.2501.11789
27. Reynolds, A., Nötzli, A., Barrett, C., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 23–42. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_2
28. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: Reductions for strings and regular expressions revisited. In: FMCAD, pp. 225–235. IEEE (2020). https://doi.org/10.34727/2020/ISBN.978-3-85448-042-6_30

29. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_20
30. Stanford, C., Veanes, M., Bjørner, N.S.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: PLDI, pp. 620–635. ACM (2021). https://doi.org/10.1145/3453483.3454066
31. Stjerna, A., Rümmer, P.: A constraint solving approach to parikh images of regular languages. Proc. ACM Program. Lang. **8**(OOPSLA1), 1235–1263 (2024). https://doi.org/10.1145/3649855
32. Subramanian, K.G., Huey, A.M., Nagar, A.K.: On parikh matrices. Int. J. Found. Comput. Sci. **20**(2), 211–219 (2009). https://doi.org/10.1142/S0129054109006528
33. Trinh, M., Chu, D., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: SIGSAC, pp. 1232–1243. ACM (2014). https://doi.org/10.1145/2660267.2660372
34. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: SIGPLAN, pp. 32–41. ACM (2007). https://doi.org/10.1145/1250734.1250739