Word Equations in Synergy with Regular Constraints

František Blahoudek[1], Yu-Fang Chen[2], David Chocholatý[1],
Vojtěch Havlena[1], Lukáš Holík[1], Ondřej Lengál[1], and Juraj Síč[1]

[1] Faculty of Information Technology, Brno University of Technology, Czech Republic
[2] Institute of Information Science, Academia Sinica, Taiwan

Abstract. We argue that in string solving, word equations and regular constraints are better mixed together than approached separately as in most current string solvers. We propose a fast algorithm, complete for the fragment of chain-free constraints, in which word equations and regular constraints are tightly integrated and exchange information, efficiently pruning the cases generated by each other and limiting possible combinatorial explosion. The algorithm is based on a novel language-based characterisation of satisfiability of word equations with regular constraints. We experimentally show that our prototype implementation is competitive with the best string solvers and even superior in that it is the fastest on difficult examples and has the least number of timeouts.

1 Introduction

Solving of string constraints (string solving) has gained a significant traction in the last two decades, drawing motivation from verification of programs that manipulate strings. String manipulation is indeed ubiquitous, tricky, and error-prone. It has been a source of security vulnerabilities, such as cross-site scripting or SQL injection, that have been occupying top spots in the lists of software security issues [1–3]; moreover, widely used scripting languages like Python and PHP rely heavily on strings. Interesting new examples of an intensive use of critical string operations can also be found, e.g., in reasoning over configuration files of cloud services [4] or smart contracts [5]. Emergent approaches and tools for string solving are already numerous, for instance [6–54].

A practical solver must handle a wide range of string operations, ranging from regular constraints and word equations across string length constraints to complex functions such as ReplaceAll or integer-string conversions. The solvers translate most kinds of constraints to a few types of basic string constraints. The base algorithm then determines the architecture of the string solver and is the component with the largest impact on its efficiency. The second ingredient of the efficiency are layers of opportunistic heuristics that are effective on established benchmarks. Outside the boundaries where the heuristics apply and the core algorithm must do a heavy lifting, the efficiency may deteriorate.

The most essential string constraints, word equations and regular constraints, are the primary source of difficulty. Their combination is PSPACE-complete [55, 56], decidable by the algorithm of Makanin [57] and Jeż's recompression [56]. Since it is not known how these general algorithms may be implemented efficiently, string solvers use incomplete algorithms or work only with restricted fragments (e.g. straight-line of [21] or chain-free [21, 26], which cover most of existing practical benchmarks), but even

these are still PSPACE-complete (immediately due to Boolean combinations of regular constraints) and practically hard. Most of string solvers use base algorithms that resemble Makanin [57] or Nielsen's [58] algorithm in which word equations and regular constraints each generate one level of disjunctive branching, and the two levels multiply. Regular constraints particularly are considered complex and expensive, and reasoning with them is sometimes postponed and done only as the last step.

In this work, we propose an algorithm in which regular constraints are not avoided but tightly integrated with equations, enabling an exchange of information between equations and regular constraints that leads to a mutual pruning of generated disjunctive choices.

For instance, in cases such as $zyx = xxz \wedge x \in a^* \wedge y \in a^+b^+ \wedge z \in b^*$, attempting to eliminate the equation results in an infinite case split (using, e.g., Nielsen's algorithm [58] or the algorithm of [31]) and it indeed leads to failure for all solvers we have tried. The regular constraints enforce `UNSAT`: since the y on the left contains at least one b, the z on the right must answer with at least one b (x has only a's). Then, since the first letter on the left is the b of z, the first x on the right must be ϵ. Since $x = \epsilon$, we are left with $zy = z$, but the a's within the y cannot be matched by the z on the right as z has only b's.

The ability to infer this kind of information from the regular constraints systematically is in the core of our algorithm. The algorithm gradually refines the regular constraints to fit the equation, until an infeasible constraint is generated (with an empty language) or until a solution is detected. Detecting the existence of a solution is based on our novel characterisation of satisfiability of a string constraint: a constraint $x_1 \dots x_m = x_{m+1} \dots x_n \wedge \bigwedge_{x \in \mathbb{X}} x \in \mathsf{Lang}(x)$, where Lang assigns regular languages to variables in $\mathbb{X}$, has a solution if the constraint is *stable*, that is, the languages of the two sides are equal, $\mathsf{Lang}(x_1) \cdots \mathsf{Lang}(x_m) = \mathsf{Lang}(x_{m+1}) \cdots \mathsf{Lang}(x_n)$. A refinement of the variable languages is derived from a special product of the automata for concatenations of the languages on the left-hand and right-hand sides of the equation. For the case with $zyx = xxz$ above, the algorithm terminates after 2-refinements (as discussed above, inferring that (1) $z \in b^+$ and $x = \epsilon$, (2) there is no a on the right to match the a's in y on the left). The wealth of information in the regular constraints increases with refinements and prunes branches that would be explored otherwise if the equation was considered alone. The algorithm is hence effective even for pure equations, as we show experimentally.

Although our algorithm is complete for `SAT` formulae, in `UNSAT` cases the refinement steps may go on forever. We prove that it is, however, guaranteed to terminate and hence complete for the *chain-free* fragment [26] (and its subset the *straight-line* fragment [21, 22]), the largest known decidable fragment of string constraints that combines equations, regular and transducer constraints, and length constraints. For this fragment, the equality in the definition of stability may be replaced by a single inclusion and only one refinement step is sufficient (the case of single equation generalises to multiple equations where the inclusions must be chosen according to certain criteria).

We have experimentally shown that on established benchmarks featuring hard combinations of word equations and regular constraints, our prototype implementation is competitive with a representative selection of string solvers (CVC5, Z3, Z3str4, Z3str3RE, Z3-Trau, OSTRICH, Sloth, Retro). Besides being generally quite fast, it seems to be superior especially on difficult instances and has the smallest number of timeouts.

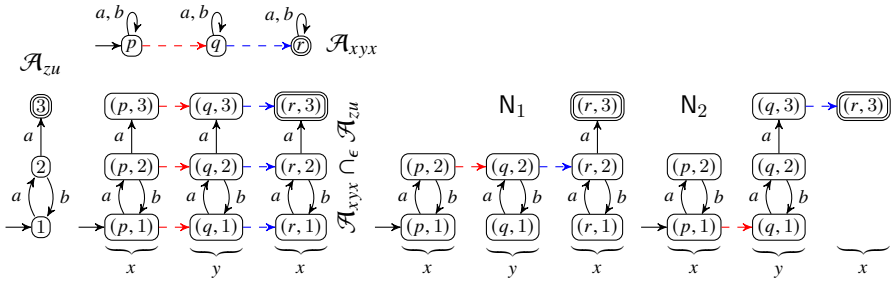


Fig. 1: Automata constructions within the refinement. Dashed lines represent ϵ.

2 Overview

We will first give an informal overview of our algorithm on the following example

$$xyx = zu \quad \wedge \quad ww = xa \quad \wedge \quad u \in (baba)^*a \quad \wedge \quad z \in a(ba)^* \tag{1}$$

with variables u, w, x, y, z over the alphabet $\Sigma = \{a, b\}$.

Our algorithm works by iteratively refining/pruning the languages in the regular membership constraints from words that cannot be present in any solution. We denote the regular constraint for a variable x by $\mathsf{Lang}(x)$. In the example, we have $\mathsf{Lang}(u) = (baba)^*a$, $\mathsf{Lang}(z) = a(ba)^*$ and, implicitly, $\mathsf{Lang}(x) = \mathsf{Lang}(y) = \mathsf{Lang}(w) = \Sigma^*$.

The equation $xyx = zu$ enforces that any solution, an assignment ν of strings to variables, satisfies that the string $s = \nu(x) \cdot \nu(y) \cdot \nu(x) = \nu(z) \cdot \nu(u)$ belongs to the intersection of the concatenations of languages on the left and the right-hand side of the equation, $\mathsf{Lang}(x) \cdot \mathsf{Lang}(y) \cdot \mathsf{Lang}(x) \cap \mathsf{Lang}(z) \cdot \mathsf{Lang}(u)$, as in Eq. (2) below:

$$s \in \overbrace{\Sigma^*}^{x}\overbrace{\Sigma^*}^{y}\overbrace{\Sigma^*}^{x} \cap \overbrace{a(ba)^*}^{z}\overbrace{(baba)^*a}^{u}. \tag{2}$$

We may thus refine the languages of x and y by removing those words that cannot be a part of any string s in the intersection. The refinement is implemented over finite automata representation of languages, assuming that every $\mathsf{Lang}(x_i)$ is represented by the automaton $\mathsf{Aut}(x_i)$. The main steps of the refinement are shown in Fig. 1. First, we construct automata for the two sides of the equation:

- $\mathcal{A}_{xyx}$ is obtained by concatenating $\mathsf{Aut}(x)$, $\mathsf{Aut}(y)$, and $\mathsf{Aut}(x)$ again. It has ϵ-transitions that delimit the borders of occurrences of x and y.
- $\mathcal{A}_{zu}$ is obtained by concatenating $\mathsf{Aut}(z)$ and $\mathsf{Aut}(u)$.

We then combine $\mathcal{A}_{xyx}$ with $\mathcal{A}_{zu}$ through a synchronous product construction that preserves ϵ-transitions into an automaton $\mathcal{A}_{xyx} \cap_\epsilon \mathcal{A}_{zu}$. Seeing ϵ as a letter that delimits variable occurrences, $\mathcal{A}_{xyx} \cap_\epsilon \mathcal{A}_{zu}$ accepts strings $\alpha_1^x \epsilon \alpha^y \epsilon \alpha_2^x$ such that $\alpha_1^x \alpha^y \alpha_2^x \in \mathsf{Lang}(z) \cdot \mathsf{Lang}(u)$, $\alpha_1^x \in \mathsf{Lang}(x)$, $\alpha^y \in \mathsf{Lang}(y)$, and $\alpha_2^x \in \mathsf{Lang}(x)$.

Note that for refining the languages x, y on the left, we do not need to see the borders between z and u on the right. The ϵ-transitions can hence be eliminated from $\mathcal{A}_{zu}$ and it can be minimised. In our particular case, this gives much smaller automaton than the

one obtained by connecting $\mathsf{Aut}(z)$ and $\mathsf{Aut}(u)$ (representing $a(ba)^*$ and $(baba)^*a$, respectively). This is a significant advantage against algorithms that enumerate alignments of borders of the left and the right-hand side variables/solved forms [59].

To extract from $\mathcal{A}_{xyx} \cap_\epsilon \mathcal{A}_{zu}$ the new languages for x and y, we decompose the automata to a disjunction of several automata, which we call *noodles*. Each noodle represents a concatenation of languages $L_1^x \epsilon L^y \epsilon L_2^x$, and is obtained by choosing one ϵ-transition separating the first occurrence of x from y (the left column of red ϵ-transitions in Fig. 1), one ϵ-transition separating y from the second occurrence of x (the right column of blue ϵ-transitions), removing the other ϵ-transitions, and trimming the automaton. We have to split the product into noodles because some values of x can appear together only with some values of y, and this relation must be preserved after extracting their languages from the product (for instance, in $\mathcal{A}_{xyx} \cap_\epsilon \mathcal{A}_{zu}$ in Fig. 1, both first occurences of x and y can have, among others, values aa and ϵ, but if $x = aa$ then y must be ϵ).

Fig. 1 shows two noodles, N_1 and N_2, out of 9 possible noodles from $\mathcal{A}_{xyx} \cap_\epsilon \mathcal{A}_{zu}$. We extract the automata for languages L_1^x, L^y, and L_2^x (their initial and final states are the states with incoming and outgoing ϵ-transitions in the noodle). The refined language for y is then $\mathsf{Lang}(y) = L^y$. The refined language for x is obtained by unifying the languages of the first and the second occurrence of x, $\mathsf{Lang}(x) = L_1^x \cap L_2^x$ (by constructing a standard product of the two automata):

- For N_1, the refinement is $y \in (ba)^*$ and $x \in a$ (computed as $a(ba)^* \cap (ba)^*a$).
- For N_2, the refinement is $y \in a(ba)^*a$ and $x \in \epsilon$ (computed as $(ab)^* \cap \epsilon$).

The 7 remaining noodles generated from $\mathcal{A}_{xyx} \cap_\epsilon \mathcal{A}_{zu}$ yield $x \in \emptyset$ and are discarded. Noodles N_1 and N_2 spawn two disjunctive branches of the computation.

For the branch of N_1, we use the equation $ww = xa$ for the next refinement. Using the newly derived constraint $x \in a$, we obtain Eq. (3) on the right:

$$s \in \overbrace{\Sigma^*}^{w}\ \overbrace{\Sigma^*}^{w} \cap \overbrace{a}^{x}\ a. \quad (3)$$

Similarly as in the previous step, the refinement deduces that $w \in a$. At this point, the languages on both sides of all equations match, and so no more refinement is possible:

$$\overbrace{a}^{x}\ \overbrace{(ba)^*}^{y}\ \overbrace{a}^{x} = \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u} \quad \text{and} \quad \overbrace{a}^{w}\ \overbrace{a}^{w} = \overbrace{a}^{x}\ a. \quad (4)$$

One of the main contributions of this paper, and a cornerstone of our algorithm, is a theorem stating that in this state, when language equality holds for all equations, a solution is guaranteed to exist (see Theorem 1). We can thus conclude with SAT.

3 Preliminaries

Sets and strings. We use $\mathbb{N}$ to denote the set of natural numbers (including 0). We fix a finite *alphabet* Σ of *symbols/letters* (usually denoted $a, b, c, \ldots$) for the rest of the paper. A sequence of symbols $w = a_1 \cdots a_n$ from Σ is a *word* or a *string* over Σ, with its *length* n denoted by $|w|$. The set of all words over Σ is denoted as Σ^*. The *empty word* is denoted by ϵ ($\epsilon \notin \Sigma$), with $|\epsilon| = 0$. The *concatenation* of words u and v is denoted $u \cdot v$, uv for short (ϵ is a neutral element of concatenation). A set of words over Σ is a

language, the concatenation of languages is $L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1 \wedge v \in L_2\}$, L_1L_2 for short. *Bounded iteration* x^i, $i \in \mathbb{N}$, of a word or a language x is defined by $x^0 = \epsilon$ for a word, $x^0 = \{\epsilon\}$ for a language, and $x^{i+1} = x^i \cdot x$. Then $x^* = \bigcup_{i \in \mathbb{N}} x^i$. We often denote regular languages using regular expressions with the standard notation.

Automata. A *(nondeterministic) finite automaton (NFA)* over Σ is a tuple $\mathcal{A} = (Q, \Delta, I, F)$ where Q is a finite set of *states*, Δ is a set of *transitions* of the form $q \text{-}\{a\}\!\!\rightarrow r$ with $q, r \in Q$ and $a \in \Sigma \cup \{\epsilon\}$, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A run of $\mathcal{A}$ over a word $w \in \Sigma^*$ is a sequence $p_0 \text{-}\{a_1\}\!\!\rightarrow p_1 \text{-}\{a_2\}\!\!\rightarrow \ldots \text{-}\{a_n\}\!\!\rightarrow p_n$ where for all $1 \leq i \leq n$ it holds that $a_i \in \Sigma \cup \{\epsilon\}$, $p_{i-1} \text{-}\{a_i\}\!\!\rightarrow p_i \in \Delta$, and $w = a_1 \cdot a_2 \cdots a_n$. The run is *accepting* if $p_0 \in I$ and $p_n \in F$, and the language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of all words for which $\mathcal{A}$ has an accepting run. A language L is called *regular* if it is accepted by some NFA. Two NFAs with the same language are called *equivalent*. An automaton without ϵ-transitions is called *ϵ-free*. An automaton with each state belonging to some accepting run is *trimmed*. To concatenate languages of two NFAs $\mathcal{A} = (Q, \Delta, I, F)$ and $\mathcal{A}' = (Q', \Delta', I', F')$, we construct their *$\epsilon$-concatenation* $\mathcal{A} \circ_\epsilon \mathcal{A}' = (Q \uplus Q', \Delta \uplus \Delta' \uplus \{p \text{-}\{\epsilon\}\!\!\rightarrow q \mid p \in F, q \in I'\}, I, F')$. To intersect their languages, we construct their *ϵ-preserving product* $\mathcal{A} \cap_\epsilon \mathcal{A}' = (Q \times Q', \Delta^\times, I \times I', F \times F')$ where $(q, q') \text{-}\{a\}\!\!\rightarrow (r, r') \in \Delta^\times$ iff either (1) $a \in \Sigma$ and $q \text{-}\{a\}\!\!\rightarrow r \in \Delta, q' \text{-}\{a\}\!\!\rightarrow r' \in \Delta'$, or (2) $a = \epsilon$ and either $q' = r'$, $q \text{-}\{\epsilon\}\!\!\rightarrow r \in \Delta$ or $q = r$, $q' \text{-}\{\epsilon\}\!\!\rightarrow r' \in \Delta'$.

String constraints. We focus on the most essential string constraints, Boolean combinations of atomic string constraints of two types: word equations and regular constraints. Let $\mathbb{X}$ be a set of *string variables* (denoted $u, v, \ldots, z$), fixed for the rest of the paper. A *word equation* is an equation of the form $s = t$ where s and t are (different) *string terms*, i.e., words from $\mathbb{X}^*$.[3] We do not distinguish between $s = t$ and $t = s$. A *regular constraint* is of the form $x \in L$, where $x \in \mathbb{X}$ and L is a regular language. A *string assignment* is a map $\nu \colon \mathbb{X} \to \Sigma^*$. The assignment is a solution for a word equation $s = t$ if $\nu(s) = \nu(t)$ where $\nu(t')$ for a term $t' = x_1 \ldots x_n$ is defined as $\nu(x_1) \cdots \nu(x_n)$, and it is a solution for a regular constraint $x \in L$ if $\nu(x) \in L$. A solution for a *Boolean combination* of atomic constraint is then defined as usual.

4 Stability of String Constraints

The core ingredient of our algorithm, which allows to tightly integrate equations with regular constraints, is the notion of *stability* of a string constraint. Stability of a string constraint is used by our algorithm to indiciate satisfiability.

4.1 Stability of Single-Equation Systems

We will first discuss stability of a *single-equation system* $\Phi \colon s = t \wedge \bigwedge_{x \in \mathbb{X}} x \in \mathsf{Lang}_\Phi(x)$ where $\mathsf{Lang}_\Phi \colon \mathbb{X} \to \mathcal{P}(\Sigma^*)$ is a *language assignment*, an assignment of regular languages to variables. We say that a language assignment Lang *refines* Lang_Φ if

[3] Note that terms with letters from Σ, sometimes used in our examples, can be encoded by replacing each occurrence o of a letter a by a *fresh* variable x_o and a regular constraint $x_o \in \{a\}$.

$\mathsf{Lang}(x) \subseteq \mathsf{Lang}_\Phi(x)$ for all $x \in \mathbb{X}$. If $\mathsf{Lang}(x) = \emptyset$ for some $x \in \mathbb{X}$, it is *infeasible*, otherwise it is *feasible*. For a term $u = x_1 \dots x_n$, we define $\mathsf{Lang}(u) = \mathsf{Lang}(x_1) \cdots \mathsf{Lang}(x_n)$. We say that Lang is *strongly stable for* Φ if $\mathsf{Lang}(s) = \mathsf{Lang}(t)$.

The core result of this work is that the existence of a stable language assignment for Φ implies the existence of a solution, which is formalised below.

Theorem 1. *A single-equation system Φ has a feasible strongly stable language assignment that refines Lang_Φ, iff it has a solution.*

Proof (Sketch of $\Rightarrow$, the other direction is trivial). Let $s = y_1 \dots y_m$ and $t = y_{m+1} \dots y_n$. Note that a solution cannot be found easily by just taking any words $w_i \in \mathsf{Lang}(x_i)$, for $1 \leq i \leq n$, such that $w_1 \cdots w_m = w_{m+1} \cdots w_n$. The reason is that multiple occurrences of the same variable must have the same value. To construct a solution, we first notice that it is enough to use the shortest words in the languages of the variables. We can then assume the lengths of the strings valuating each variable fixed (the smallest lengths in the languages), which in turn fixes the positions of variables' occurrences within the sides of the equation. We then construct the strings in the solution by selecting letters and propagating them through equalities of opposite positions of the equation sides and also between different occurrences of the same position in the same variable. Showing that the process of selecting and propagating letters terminates requires to show that the sequence of constructed partial solutions is decreasing w.r.t. a complex well-founded ordering of partial solutions. The full proof may be found in [60]. □

Additionally, in the special case of *weak* equations—i.e., equations $s = t$ where one of the sides, say t, satisfies the condition that all variables occurring in t occur in $s = t$ exactly once—the stability condition in Theorem 1 can be weakened to one-sided language inclusion only: In case t is the term satisfying the condition, we say that Lang is *weakly stable for* Φ if $\mathsf{Lang}(s) \subseteq \mathsf{Lang}(t)$.

Theorem 2. *Φ with a weak equation has a feasible weakly stable language assignment that refines Lang_Φ iff Φ has a solution.*

Note that weak stability allows multiple occurrences of a variable on the left-hand side of $s = t$. Intuitively, the multiple occurrences must have the same value, and having them on the left-hand side of the inclusion forces their synchronisation. For instance, for $\Phi\colon xx = y \wedge x \in \{a, b\} \wedge y \in \{ab\}$, the inclusion $\mathsf{Lang}(xx) \subseteq \mathsf{Lang}(y)$ is satisfied by no feasible refinement Lang of Lang_Φ, revealing that Φ has no solution, while $\mathsf{Lang}(xx) \supseteq \mathsf{Lang}(y)$ is satisfied already by Lang_Φ itself.

4.2 Stability of Multi-Equation Systems

Next, we extend the definition of stability to *multi-equation systems*, conjunctions of the form $\Phi\colon \mathcal{E} \wedge \bigwedge_{x \in \mathbb{X}} x \in \mathsf{Lang}_\Phi(x)$ where $\mathcal{E}\colon \bigwedge_{i=1}^{m} s_i = t_i$ for $m \in \mathbb{N}$. We assume that every two equations are different, i.e., $\{s_i, t_i\} \neq \{s_j, t_j\}$ if $i \neq j$.

We generalise stability in a way that utilises both strong and weak stability of single equation systems and both Theorem 1 and Theorem 2. Again, we interpret every equation $s_i = t_i$ as a pair of inclusions and show that it suffices to satisfy a certain subset of these

inclusions in order to obtain a solution. A sufficient subset of inclusions is defined through the notion of an *inclusion graph* of $\mathcal{E}$. It is a directed graph $G = (V, E)$ where vertices V are inclusion constraints of the form $\mathsf{s_i} \subseteq \mathsf{t_i}$ or $\mathsf{t_i} \subseteq \mathsf{s_i}$, for $1 \leq i \leq m$, and $E \subseteq V \times V$. An inclusion graph must satisfy the following conditions:

(IG1) For each $s_i = t_i$ in $\mathcal{E}$, at least one of the nodes $\mathsf{s_i} \subseteq \mathsf{t_i}$, $\mathsf{t_i} \subseteq \mathsf{s_i}$ is in V.
(IG2) If $\mathsf{s_i} \subseteq \mathsf{t_i} \in V$ and t_i has a variable with multiple occurrences in right-hand sides of vertices of V, then also $\mathsf{t_i} \subseteq \mathsf{s_i} \in V$.
(IG3) $(\mathsf{s_i} \subseteq \mathsf{t_i}, \mathsf{s_j} \subseteq \mathsf{t_j}) \in E$ iff $\mathsf{s_i} \subseteq \mathsf{t_i}, \mathsf{s_j} \subseteq \mathsf{t_j} \in V$ and s_i and t_j share a variable.
(IG4) If $\mathsf{s_i} \subseteq \mathsf{t_i} \in V$ lies on a cycle, then also $\mathsf{t_i} \subseteq \mathsf{s_i} \in V$.

Note that by (IG3), E is uniquely determined by V. We define that a language assignment Lang is *stable for an inclusion graph* $G = (V, E)$ of $\mathcal{E}$ if it satisfies every inclusion in V.

Theorem 3. *Let G be an inclusion graph of $\mathcal{E}$. Then there is a feasible language assignment that refines Lang_Φ and is stable for G iff Φ has a solution.*

The proof of Theorem 3 is in [60]. Intuitively, the set of inclusions needed to guarantee a solution is specified by the vertices of an inclusion graph. All equations must contribute with at least one inclusion, by Condition (IG1). Including only one inclusion corresponds to using weak stability. Including both inclusions corresponds to using strong stability. We will use inclusion graphs in our algorithm to direct propagation of refinements of language assignments. We will wish to avoid using strong stability when possible since cycles that it creates in the graph may cause the algorithm to diverge.

Conditions (IG2)–(IG4) specify where weak stability is not enough. Namely, Condition (IG2) enforces that to use weak stability, multiple occurrences of a variable can only occur on the left-hand side of an inclusion (as in the definition of weak stability), otherwise strong stability must be used. The edges defined by Condition (IG3) are used in Condition (IG4). An edge means that a refinement of the language assignment made to satisfy the inclusion in the source node may invalidate the inclusion in the target node. Condition (IG4) covers the case of a cyclic dependency of a variable on itself. A self-loop indicates that a variable occurs on both sides of an equation (breaking the definition of weak stability). A longer cycle indicates such a cyclic dependency caused by transitively propagating the inclusion relation.

4.3 Constructing Inclusion Graphs and Chain-Freeness

We now discuss a construction of a suitable inclusion graph. Our algorithm for solving string constraints will use the graph nodes to gradually refine the language assignment, propagating information along the graph edges. It is guaranteed to terminate when the graph is acyclic. Below, we give an algorithm that generates an inclusion graph that contains as few inclusions as possible and is acyclic whenever possible.

The graph is obtained from a simplified version $SG_\mathcal{E}$ of the splitting graph of [26], which is the basis of the definition of the chain-free fragment, for which our algorithm is complete. The nodes of $SG_\mathcal{E}$ are all inclusions $\mathsf{s_i} \subseteq \mathsf{t_i}, \mathsf{t_i} \subseteq \mathsf{s_i}$, for $1 \leq i \leq m$, and it has an edge from $\mathsf{s} \subseteq \mathsf{t}$ to $\mathsf{s}' \subseteq \mathsf{t}'$ if s and t' each have a different occurrence of the same variable (the "*different*" here meaning not the same position in the same term in the same equation, e.g., for inclusions induced by the equation $u = v$, for $u, v \in \mathbb{X}$, there will be no edge between $\mathsf{u} \subseteq \mathsf{v}$ and $\mathsf{v} \subseteq \mathsf{u}$).

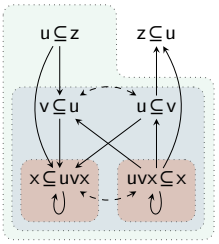
Algorithm 1: $\texttt{incl}(\mathcal{E})$

Input: Conjunction of string equations $\mathcal{E}$.
Output: An inclusion graph of $\mathcal{E}$.

```
1 G := SG_E; V' := ∅;
2 while G has a trivial source SCC ({v}, ∅) do
3 |   G := G \ {v, dual(v)};
4 |   V' := V' ∪ {v};
5 V := V' ∪ the remaining nodes of G;
6 return the inclusion graph with nodes V;
```

The algorithm for constructing an inclusion graph from $SG_{\mathcal{E}}$ starts by iteratively removing nodes that are trivial source strongly connected components (SCCs) from $SG_{\mathcal{E}}$ (trivial means a graph $(\{v\}, \emptyset)$ with no edges, source means with no edges coming from outside into the component). With every removed node v, the algorithm removes from $SG_{\mathcal{E}}$ also the dual node $\mathsf{dual}(v)$ (the other inclusion), and it adds v to the inclusion graph. When no trivial source SCCs are left, that is, the remaining nodes are all reachable from non-trivial SCCs, the algorithm adds to the inclusion graph all the remaining nodes.

The pseudocode of the algorithm is shown in Algorithm 1. It uses $\texttt{SCC}(G)$ to denote the set of SCCs of G and $G \setminus V$ to denote the graph obtained from G by removing the vertices in V together with the adjacent edges.



Example 1. In the picture in the right, we show an example of the construction of the inclusion graph G from $SG_{\mathcal{E}}$ for $\mathcal{E}\colon z = u \wedge u = v \wedge uvx = x$. Edges of $SG_{\mathcal{E}}$ are solid lines, the inclusion graph has both solid and dashed edges. The inner red boxes are the non-trivial SCCs of $SG_{\mathcal{E}}$. They are enclosed in the box of nodes that are added on Line 5 of Algorithm 1. The outer-most box encloses the inclusion graph, including one node added on Line 4. □

Theorem 4. *For a conjunction of equations $\mathcal{E}$, $\texttt{incl}(\mathcal{E})$ is an inclusion graph for $\mathcal{E}$ with the smallest number of vertices. Moreover, if there exists an acyclic inclusion graph for $\mathcal{E}$, then $\texttt{incl}(\mathcal{E})$ is acyclic.*

In Section 5, we will show a satisfiability checking algorithm that guarantees termination when given an acyclic inclusion graph. Here we prove that the existence of an acyclic inclusion graph coincides with the chain-free fragment of string constraints [26], which is the largest known decidable fragment of string constraints with equations, regular and transducer constraints, and length constraints (up to the incomparable fragment of quadratic equations). *Chain-free* constraints are defined as those where the simplified splitting graph $SG_{\mathcal{E}}$ has no cycle. The following theorem is proven in [60].

Theorem 5. *A multi-equation system Φ is chain-free iff there exists an acyclic inclusion graph for Φ.*

5 Algorithm for Satisfiability Checking

Our algorithm for testing satisfiability of a multi-equation system Φ is based on Theorem 3. The algorithm first constructs a suitable inclusion graph of $\mathcal{E}$ using Algorithm 1 and then it gradually refines the original language assignment Lang_{Φ} according to the dependencies in the inclusion graph until it either finds a stable feasible language assignment or concludes that no such language assignment exists.

A language assignment Lang is in the algorithm represented by an *automata assignment* Aut, which assigns to every variable x an ϵ-free NFA $\mathsf{Aut}(x)$ with $L(\mathsf{Aut}(x)) = \mathsf{Lang}(x)$. We use $\mathsf{Aut}(t)$ for a term $t = x_1 \dots x_n$ to denote the NFA $\mathsf{Aut}(x_1) \circ_\epsilon \cdots \circ_\epsilon \mathsf{Aut}(x_n)$. In the following text, we identify a language assignment with the corresponding automata assignment and vice versa.

5.1 Refining Language Assignments by Noodlification

The task of a refinement step is to create a new language assignment that refines the old one, Lang, and satisfies one of the inclusions previously not satisfied, say $\mathtt{s} \subseteq \mathtt{t}$. In order for the algorithm to be sound when returning UNSAT, a refinement step must preserve all existing solutions. It will therefore return a set $\mathcal{T}$ of refinements of Lang that is *tight w.r.t.* $\mathtt{s} \subseteq \mathtt{t}$, that is, every solution of $s = t$ under Lang is also a solution of $s = t$ under some of its refinements in $\mathcal{T}$.

Algorithm 2 computes such a tight set. Line 1 computes the automaton $Product$, which accepts $\mathsf{Lang}(s) \cap \mathsf{Lang}(t)$. In order to be able to extract new languages for the variables of s from it, $Product$ marks borders between the variables of s with ϵ-transitions. That is, when ϵ is understood as a special letter, $Product$ accepts the *delimited language* $L^\epsilon(Product)$ of words $w_1\epsilon\cdots\epsilon w_n$ with $w_i \in \mathsf{Lang}(x_i)$ for $1 \le i \le n$ and $w_1 \cdots w_n \in \mathsf{Lang}(t)$. Notice that $\mathsf{Aut}(t)$ is on Line 1 minimised. This means removal of ϵ-transitions marking the borders of variables' occurrences, and then minimisation by any automata size reduction method (we use simulation quotient [61, 62]). Since the product is then representing only the borders of the variables on the left (because $\mathsf{Aut}(s)$ keeps the ϵ-transitions generated from the concatenation with $\circ_\epsilon$), but not the borders of variables in t, it does not actually generate an explicit representation of possible alignments of borders of variables' occurrences.

Algorithm 2: `refine`(v, Aut)

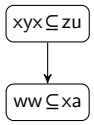
Input: A vertex $v = \mathtt{s} \subseteq \mathtt{t}$ with $s = x_1 \cdots x_n$ and $t = y_1 \cdots y_m$, an automata assignment Aut
Output: A tight refinement of Aut w.r.t. v

1 $Product := \mathsf{Aut}(s) \cap_\epsilon minimise(\mathsf{Aut}(t))$;
2 $Noodles :=$ `noodlify`$(Product)$;
3 $\mathcal{T} := \emptyset$;
4 **for** $\mathsf{N} \in Noodles$ **do**
5 $\quad \mathsf{Aut}' := \mathsf{Aut}$;
6 $\quad$ **for** $1 \le i \le n$ **do**
7 $\quad\quad \mathsf{Aut}'(x_i) := \bigcap\{\mathsf{N}(j) \mid 1 \le j \le n, x_i = x_j\}$;
8 $\quad$ **if** $L(\mathsf{Aut}'(s)) = \emptyset$ **then continue**;
9 $\quad \mathcal{T} := \mathcal{T} \cup \{\mathsf{Aut}'\}$;
10 **return** $\mathcal{T}$;

We then extract from $Product$ a language for each *occurrence* of a variable in s. Line 2 divides $Product$ into a set of the so-called *noodles*, which are sequences of automata $\mathsf{N} = \mathsf{N}(1), \dots, \mathsf{N}(n)$ that preserve the delimited language in the sense that $\bigcup_{\mathsf{N} \in Noodles} L^\epsilon(\mathsf{N}(1) \circ_\epsilon \cdots \circ_\epsilon \mathsf{N}(n)) = L^\epsilon(Product)$.

Technically, assuming w.l.o.g that $Product$ has a single initial state r_0 and a single final state q_n, `noodlify`$(Product)$ generates one noodle N for each $(n-1)$-tuple $q_1 \dashv\epsilon\mapsto r_1, \dots, q_{n-1} \dashv\epsilon\mapsto r_{n-1}$ of transitions that appear, in that order, in an accepting run of $Product$ (note that every accepting run has $n-1$ ϵ-transitions by construction of $Product$, since $\mathsf{Aut}(s)$ also had $n-1$ ϵ-transitions in each accepting run and $minimise(\mathsf{Aut}(t))$ is ϵ-free): for each $1 \le i \le n$, $\mathsf{N}(i)$ arises by trimming $Product$ after its initial states were replaced by $\{r_{i-1}\}$ and final states by $\{q_i\}$.

The **for** loop on Line 4 then turns each noodle N into a refined automata assignment Aut' in $\mathcal{T}$ by unifying/intersecting languages of different occurrences of the same variable: for each $x \in \mathbb{X}$, $\mathsf{Aut}'(x)$ is the automata intersection of all automata $\mathsf{N}(i)$ with $x_i = x$. The fact that $\mathcal{T}$ is a tight set of refinements (i.e., that it preserves all solutions of Aut) follows from that every path of $Product$ can be found in $Noodles$ and that the use of ϵ-transitions allows to reconstruct the NFAs corresponding to the variables.



Example 2. Consider the multi-equation system Φ from Section 2 and the vertex $\mathsf{xyx} \subseteq \mathsf{zu}$ of its inclusion graph given in the right. The construction of the product automaton $Product$ from Algorithm 2 is shown in Fig. 1. The set of noodles $\mathtt{noodlify}(Product) = \{\mathsf{N}_1, \ldots, \mathsf{N}_7\}$ is given in [60] (N_1 and N_2 are in Fig. 1). On Line 6, we need to compute intersections of $\mathsf{N}_i(1) \cap \mathsf{N}_i(3)$ for each noodle N_i. These parts of the noodle correspond to the two occurrences of the same variable x. The only noodles yielding nonempty languages for x are N_1 and N_2. The noodle N_1 leads to a refinement Aut_1 of Aut where $L(\mathsf{Aut}_1(x)) = a$ (computed as the intersection of languages $\mathsf{N}_1(1) = a(ba)^*$ and $\mathsf{N}_1(3) = (ba)^*a$) and $L(\mathsf{Aut}_1(y)) = (ba)^*$. The noodle N_2 leads to a refinement Aut_2 of Aut where $L(\mathsf{Aut}_2(x)) = \epsilon$ (computed as the intersection of languages $\mathsf{N}_2(1) = (ab)^*$ and $\mathsf{N}_2(3) = \epsilon$) and $L(\mathsf{Aut}_2(y)) = a(ba)^*a$. □

Example 3. An example with a non-terminating sequence of refinement steps is $xa = x \wedge x \in a^+$, explained in detail in [60]. Every i-th step refines $\mathsf{Lang}(x)$ to $x \in a^{i+1}a^*$. Note that many similar examples could be handled by simple heuristics that take into account lengths of strings, already used in other solvers. □

5.2 Satisfiability Checking by Refinement Propagation

The pseudocode of the satisfiability check of Φ is given in Algorithm 3. It starts with the automaton assignment Aut_Φ corresponding to Lang_Φ, and it uses graph nodes $\mathsf{s} \subseteq \mathsf{t}$ not satisfied in the current Aut to refine it, that is, to replace Aut by some automaton assignment returned by $\mathtt{refine}(\mathsf{s} \subseteq \mathsf{t}, \mathsf{Aut})$.

The algorithm maintains the current value of Aut and a worklist W of nodes for which the weak-stability condition might be invalidated, either initially or since they were affected by some previous refinement. Nodes are picked from the worklist, and if the inclusion at a node is found not satisfied in the current automata assignment Aut, the node is used to refine it. Stability is detected when W is empty—there in no potentially unsatisfied inclusion.

Algorithm 3: $\mathtt{propagate}(G_\mathcal{E}, \mathsf{Aut}_\Phi)$

Input: Inclusion graph $G_\mathcal{E} = (V, E)$, initial automata assignment Aut_Φ.
Output: `SAT` if Φ is satisfiable, `UNSAT` if Φ is unsatisfiable

```
1  Branches := ⟨(Aut_Φ, V)⟩;
2  while Branches ≠ ∅ do
3      (Aut, W) := Branches.dequeue();
4      if W = ∅ then return SAT ;
5      v = s⊆t := W.dequeue();
6      if L(Aut(s)) ⊆ L(Aut(t)) then
7          Branches.enqueue((Aut, W));
8          continue;
9      T := refine(v, Aut);
10     W' := W;
11     foreach (v, u) ∈ E s.t. u ∉ W do
12         W'.enqueue(u);
13     foreach Aut' ∈ T do
14         Branches.enqueue(Aut', W');
15 return UNSAT;
```

Since $\texttt{refine}(\mathsf{s} \subseteq \mathsf{t}, \mathsf{Aut})$ does not return a single language assignment but a set of language assignments that refine Aut, the computation spawns an independent branch for each of them. Algorithm 3 adds the branches for processing in the queue $Branches$. The branching is disjunctive, meaning stability is returned when a single branch detects stability. If all branches terminate with an infeasible assignment, then the algorithm concludes that the constraint is unsatisfiable.

The worklist and the queue of branches are first-in first-out (this is important for showing termination in Theorem 8). To minimise the number of refinement steps, the nodes are initially inserted in W in an order compatible with a topological order of the SCCs.

Example 4. Consider again the multi-equation system Φ from Section 2 and the inclusion graph in Example 2. The initial automata assignment Aut_Φ is then given as $L(\mathsf{Aut}_\Phi(a)) = \{a\}$, $L(\mathsf{Aut}_\Phi(z)) = a(ba)^*$, $L(\mathsf{Aut}_\Phi(u)) = (baba)^*a$, and $L(\mathsf{Aut}_\Phi(x)) = L(\mathsf{Aut}_\Phi(y)) = L(\mathsf{Aut}_\Phi(w)) = \Sigma^*$. The queue $Branches$ on Line 1 of Algorithm 3 is hence initialised as $\mathsf{Branches} = \langle(\mathsf{Aut}_\Phi, \langle \mathsf{xyx} \subseteq \mathsf{zu}, \mathsf{ww} \subseteq \mathsf{xa}\rangle)\rangle$. The computation of the main loop of Algorithm 3 then proceeds as follows.

1st iteration. The dequeued element is $(\mathsf{Aut}_\Phi, \langle \mathsf{xyx} \subseteq \mathsf{zu}, \mathsf{ww} \subseteq \mathsf{xa}\rangle)$ and v (dequeued from W) is $\mathsf{xyx} \subseteq \mathsf{zu}$. The condition on Line 6 is not satisfied, hence the algorithm calls $\texttt{refine}(\mathsf{xyx} \subseteq \mathsf{zu}, \mathsf{Aut}_\Phi)$. The refinement yields two new automata assignments, $\mathsf{Aut}_1, \mathsf{Aut}_2$, which are defined in Example 2. The queue $Branches$ is hence extended to $\langle(\mathsf{Aut}_1, \langle \mathsf{ww} \subseteq \mathsf{xa}\rangle), (\mathsf{Aut}_2, \langle \mathsf{ww} \subseteq \mathsf{xa}\rangle)\rangle$.

2nd iteration. The dequeued element is $(\mathsf{Aut}_2, \langle \mathsf{ww} \subseteq \mathsf{xa}\rangle)$. The condition on Line 6 is not satisfied since $L(\mathsf{Aut}_2(x)) = \{\epsilon\}$ and $L(\mathsf{Aut}_2(w)) = \Sigma^*$. In this case, $\texttt{refine}(\mathsf{ww} \subseteq \mathsf{xa}, \mathsf{Aut}_2) = \emptyset$, hence nothing is added to $Branches$, i.e., $\mathsf{Branches} = \langle(\mathsf{Aut}_1, \langle \mathsf{ww} \subseteq \mathsf{xa}\rangle)\rangle$.

3rd iteration. The dequeued element is $(\mathsf{Aut}_1, \langle \mathsf{ww} \subseteq \mathsf{xa}\rangle)$. The condition on Line 6 is not satisfied ($\Sigma^* \cdot \Sigma^* \not\subseteq a \cdot a$) and $\texttt{refine}(\mathsf{ww} \subseteq \mathsf{xa}, \mathsf{Aut}_1) = \{\mathsf{Aut}_3\}$ where Aut_3 is as Aut_1 except that $\mathsf{Aut}_3(w)$ accepts only a. $Branches$ is then updated to $\langle(\mathsf{Aut}_3, \emptyset)\rangle$.

4th iteration. The condition on Line 4 is satisfied and the algorithm returns SAT. □

As stated by Theorem 6 below, the algorithm is sound in the general case (an answer is always correct). Moreover, Theorems 5 and 7 imply that when Algorithm 1 is used to construct the inclusion graph, we have a complete algorithm for chain-free constraints.

Theorem 6 (Soundness). *If* $\texttt{propagate}(G_\mathcal{E}, \mathsf{Aut}_\Phi)$ *returns* SAT*, then* Φ *is satisfiable, and if* $\texttt{propagate}(G_\mathcal{E}, \mathsf{Aut}_\Phi)$ *returns* UNSAT*,* Φ *is unsatisfiable.*

Theorem 7. *If* $G_\mathcal{E}$ *is acyclic, then* $\texttt{propagate}(G_\mathcal{E}, \mathsf{Aut}_\Phi)$ *terminates.*

5.3 Working with Shortest Words

Algorithm 3 can be improved with a weaker termination condition that takes into account only shortest words in the languages assigned to variables. Importantly, this gives us completeness in the SAT case for general constraints, i.e., the algorithm is always guaranteed to return SAT if a solution exists.

Let $\mathsf{Lang}^{\min}$ be the language assignment obtained from Lang by assigning to every $x \in \mathbb{X}$ the set of shortest words from $\mathsf{Lang}(x)$ i.e., $\mathsf{Lang}^{\min}(x) = \{w \in \mathsf{Lang}(x) \mid \forall u \in \mathsf{Lang}(x) : |w| \leq |u|\}$. Then, for $\Phi\colon s = t \wedge \bigwedge_{x \in \mathbb{X}} x \in \mathsf{Lang}_\Phi(x)$, we say that Lang is *strongly min-stable for* Φ if $\mathsf{Lang}^{\min}$ is stable for Φ. Similarly, Lang is *weakly min-stable for* Φ if $s = t$ is weak and $\mathsf{Lang}^{\min}(s) \subseteq \mathsf{Lang}(t)$. Note that for weak min-stability, it is enough to have the min-language only on the left, which gives a weaker condition. Theorems 1 and 2 hold for min-stability and weak min-stability, respectively, as well (the proof of the min-versions are in fact a part of the proof of the Theorems 1 and 2). The min-stability of a multi-equation system is then defined in the same way as before, different only in that it uses the min-stability at the nodes instead of stability. Namely, in Algorithm 3, the test $L(\mathsf{Aut}(s)) \subseteq L(\mathsf{Aut}(t))$ on Line 6 is replaced by $L^{\min}(\mathsf{Aut}(s)) \subseteq L(\mathsf{Aut}(t))$. We call this variant of the algorithm $\texttt{propagate}_{min}$. Not only that this new algorithm is still partially correct, may terminate after less refinements, and uses a cheaper test to detect termination, but, mainly, it is complete in the SAT case: if there is a solution, then it is guaranteed to terminate for any system, no matter whether chain-free or not. Intuitively, the algorithm in a sense explores the words in the languages of the variables systematically, taking the words ordered by length, the shortest ones first. Hence, besides that variants of Theorems 6 and 7 with $\texttt{propagate}_{min}$ still hold, we also have Theorem 8:

Theorem 8. *If* Φ *is satisfiable, then* $\texttt{propagate}_{min}(\texttt{incl}(\mathcal{E}), \mathsf{Aut}_\Phi)$ *terminates.*

6 Experimental Evaluation

We implemented our algorithm in a prototype string solver called Noodler [63] using Python and a homemade C++ automata library for manipulating NFAs. We compared the performance of Noodler with a comprehensive selection of other tools, namely, CVC5 [13] (version 1.0.1), Z3 [15] (version 4.8.14), Z3str3RE [20], Z3str4 [64], Z3-Trau [34], OSTRICH [23], Sloth [65], and Retro [48]. In order to have a meaningful comparison with compiled tools (CVC5, Z3, Z3str3RE, Z3str4, Z3-Trau), the reported time for Noodler does not contain the startup time of the Python interpreter and the time taken by loading libraries (this is a constant of around 1.5 s). To be fair, one should take this into account when considering the time of other interpreted tools, such as OSTRICH, Sloth (both Java), and Retro (Python). As can be seen from the results, it would, however, not significantly impact the overall outcome. The experiments were executed on a workstation with an Intel Core i5 661 CPU at 3.33 GHz with 16 GiB of RAM running Debian GNU/Linux. The timeout was set to 60 s.

Benchmarks. We consider the following benchmarks, having removed unsupported formulae (i.e., formulae with length constraints or transducer operations).

- PyEx-Hard ([48], 20,023 formulae): it comes from the PyEx benchmark [10], in particular, it is obtained from 967 difficult instances that neither CVC4 nor Z3 could solve in 10 s. PyEx-Hard then contains 20,023 conjunctions of word equations that Z3's DPLL(T) algorithm sent to its string theory solver when trying to solve them.

Table 1: Results of experiments. For each benchmark and tool, we give the number of timeouts ("T/Os"), the total run time (in seconds), and the run time without timeouts ("time−T/O"). Best values are in **bold**.

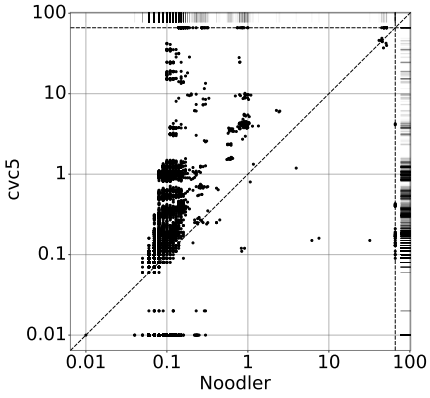
	PyEx-Hard (20,023)			Kaluza-Hard (897)			Str 2 (293)			Slog (1,896)		
	T/Os	time	time−T/O	T/Os	time	time−T/O	T/Os	time	time−T/O	T/Os	time	time−T/O
Noodler	**39**	**5,266**	2,926	**0**	**46**	46	3	**198**	18	**0**	165	165
Z3	2,802	178,078	9,958	207	15,360	2,940	149	8,955	15	2	332	212
CVC5	112	12,523	5,803	**0**	55	55	92	5,525	**5**	**0**	**14**	**14**
Z3str3RE	814	49,744	904	10	622	22	149	8,972	32	55	4,247	947
Z3str4	461	28,114	**454**	17	1,039	**19**	154	9,267	27	208	16,508	4,028
Z3-Trau	108	33,551	27,071	**0**	201	201	10	724	124	5	970	670
OSTRICH	2,979	214,846	36,106	111	14,912	8,252	238	14,497	217	2	13,601	13,481
Sloth	463	371,373	343,593	**0**	3,195	3,195		N/A		202	24,940	12,820
Retro	3,004	199,107	18,867	148	16,404	7,524	**1**	299	239		N/A	

– Kaluza-Hard (897 formulae): it is obtained from the Kaluza benchmark [46] by taking hard formulae from its solution in a similar way for PyEx-Hard.
– Str 2 ([33], 293 formulae) the original benchmark from [33] contains 600 handcrafted formulae including word equations and length constraints; the 307 formulae containing length constraints are removed.
– Slog ([35], 1,896 formulae) contains 1,976 formulae obtained from real web applications using static analysis tools JSA [66] and Stranger [39]. 80 of these formulae contain transducer operations (e.g., ReplaceAll).

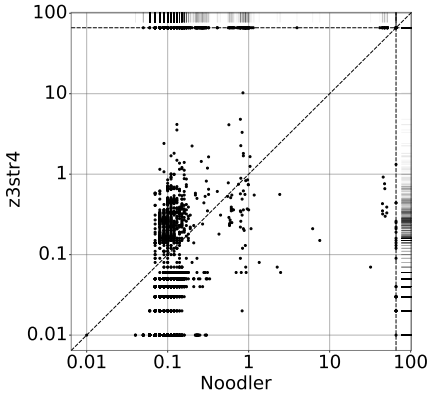
From the benchmarks, only Slog initially contains regular constraints. Note that an interplay between equations and regular constraints happens in our algorithm even with pure equations on the input. Refinement of regular constraints is indeed the only means in which our algorithm accumulates information. Complex regular constraints are generated by refinement steps from an initial assignment of Σ^* for every variable. We also include useful constraints in preprocessing steps, for instance, the equation $z = xay$ where x and y do not occur elsewhere is substituted by $z \in \Sigma^* a \Sigma^*$.

Results. The results of experiments are given in Table 1. For each benchmark, we list the number of timeouts (i.e., unsolved formulae), the total run time (including timeouts), and also the run time on the successfully decided formulae. The results show that from all tools, Noodler has the lowest number of timeouts on the aggregation of all benchmarks (42 timeouts in total) and also on each individual benchmark (except Str 2 where it is the second lowest, 3 against 1). Furthermore, in all benchmarks except Slog, Noodler is faster than other tools (and for Slog it is the second). The results for Sloth on Str 2 are omitted because Sloth was incorrect on this benchmark (the benchmark is not straight-line) and the results for Retro on Slog are omitted because Retro does not support regular constraints.

In Fig. 3, we provide scatter plots comparing the run times of Noodler with the best competitors, CVC5 and Z3str4, on the PyEx-Hard benchmark (scatter plots for the other benchmarks are less interesting and can be found in [60]). We can see that there is indeed a large number of benchmarks where Noodler is faster than both competitors



(a) NOODLER vs. CVC5.



(b) NOODLER vs. Z3STR4.

Fig. 3: The performance of NOODLER and other tools on PYEX-HARD. Times are given in seconds, axes are logarithmic. Dashed lines represent timeouts (60 s).

(and that the performance of NOODLER is more stable, which may be caused by the heuristics in the other tools not always working well). Notice that NOODLER and CVC5 are on this benchmark complementary: they have both some timeouts, but each formula is solved by at least one of the tools.

Moreover, in Fig. 2, we provide a graph showing times needed to solve 1,023 most difficult formulae for the tools on the PYEX-HARD benchmark.

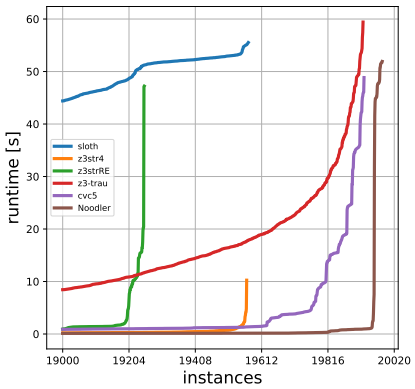


Fig. 2: Times for solving the hardest 1,023 formulae for the tools on PYEX-HARD

Discussion. The results of the experiments show that our algorithm (even in its prototype implementation in Python) can beat well established solvers such as CVC5, Z3, and Z3STR4. In particular, it can solve more benchmarks, and also the average time for (successfully) solving a benchmark is low (as witnessed by the "time−T/O" column in Table 1). The scatter plots also show that it is often complementary to other solvers.

7 Related Work

Our algorithm is an improvement of the automata-based algorithm first proposed in [30], which is, at least in part, used as the basis of several string solvers, namely, NORN [30, 31, 26], TRAU [34, 27–29], OSTRICH [21–23], and Z3STR3RE [20]. The original algorithm first transforms equations to the disjunction of their solved forms [59] through generating alignments of variable boundaries on the equation sides (essentially an incomplete version of Makanin's algorithm). Second, it eliminates concatenation from

regular constraints by *automata splitting*. The algorithm replaces $x \cdot y \in L$ by a disjunction of cases $x \in L_x \wedge y \in L_y$, one case for each state of L's automaton. Each disjunct later entails testing emptiness of $L_x \cap \mathsf{Lang}(x)$ and $L_y \cap \mathsf{Lang}(y)$ by the automata product construction. Trau uses this algorithm within an unsatisfiability check. Trau's main solution finding algorithm also performs a step similar to our refinement, though with languages underapproximated as arithmetic formulae (representing their Parikh images). Sloth [34] implements a compact version of automata splitting through alternating automata. OSTRICH has a way of avoiding the variable boundary alignment for the straight-line formulae, although still uses it outside of it. Z3str3RE optimises the algorithm of [30] heavily by the use of length-aware heuristics.

The two levels of disjunctive branching (transformation into solved form and automata splitting) are costly. For instance, for $xyx = zu \wedge z \in a(ba)^* \wedge u \in (baba)^*a$ (a subformula of the example in Section 2), there would be 14 alignments/solved forms, e.g. those characterised using lengths as follows: (1) $|zu| = 0$; (2) $|y| = |zu|$; (3) $|x| < |z|, |y| = 0$; (4) $|xy| < z, |y| > 0$; (5) $|x| < |z|, |xy| > z$; ... In the case (5) alone—corresponding to the solved form $z = z_1z_2, u = u_1z_1, x = z_1, y = z_2u_1$—automata splitting would generate 15 cases from $z_1z_2 \in \mathsf{Lang}(z)$ and $u_1u_2 \in \mathsf{Lang}(u)$, each entailing one intersection emptiness check (the NFAs for z and u have 3 and 5 states respectively). There would be about a hundred of such cases overall. On the contrary, our algorithm generates only 9 of equivalent cases, 7 if optimised (see Section 2).

Our algorithm has an advantage also over pure automata splitting, irrespective of aligning equations. For instance, consider the constraint $xyx \in L \wedge x \in \mathsf{Lang}(x) \wedge y \in \mathsf{Lang}(y)$. Automata splitting generates a disjunction of n^2 constraints $x \in L_x \wedge y \in L_y$, with n being the number of states of the automaton for L, each constraint with emptiness checks for $\mathsf{Lang}(x) \cap L_x$ and $\mathsf{Lang}(y) \cap L_y$. Our algorithm avoids generating much of these cases by intersecting with the languages of $\mathsf{Lang}(x)$ and $\mathsf{Lang}(y)$ early—the construction of $\mathsf{Lang}(x) \cdot \mathsf{Lang}(y) \cdot \mathsf{Lang}(x)$ prunes much of L's automaton immediately. For instance, if $L = (ab)^*a^+(abcd)^*$ (its NFA has 7 states) and $\mathsf{Lang}(x) = (a + b)^*$, automata splitting explores $7^2 = 49$ cases while our algorithm explores 9 (7 when optimised) of these cases—it would compute the same product and noodles as in Section 2, essentially ignoring the disjunct $(abcd)^*$ of L.

Approaches and tools for string solving are numerous and diverse, with various representations of constraints, algorithms, or sorts of inputs. Many approaches use automata, e.g., Stranger [39–41], Norn [30, 31], OSTRICH [21–25], Trau [26–29], Sloth [34], Slog [35], Slent [36], Z3str3RE [20], Retro [48], ABC [42, 43], Qzy [47], or BEK [51]. Around word equations are centered tools such as CVC4/5 [6–12], Z3 [14, 15], S3 [32], $\texttt{Kepler}_{22}$ [33], StrSolve [37], Woorpje [49]; bit vectors are (among other things) used in Z3Str/2/3/4 [16–19], HAMPI [45]; PASS uses arrays [50]; G-strings [38] and GECODE+S [44] use SAT-solving. Most of these tools and methods handle much wider range of string constraints than equations and regular constraints. Our algorithm is not a complete alternative but a promising basis that could improve some of the existing solvers and become a core of a new one. With regard to equations and regular constraints, the fragment of chain-free constraints [26] that we handle, handled also by Trau, is the largest for which any string solvers offers formal completeness guarantees, with the exception of quadratic equations, handled, e.g., by [48, 33], which are incom-

parable but of a smaller practical relevance (although some tools actually implement Nielsen's algorithm [58] to handle simple quadratic cases). The other solvers guarantee completeness on smaller fragments, notably that of OSTRICH (straight-line), Norn, and Z3str3RE; or use incomplete heuristics that work in practice (giving up guarantees of termination, over or under-approximating by various means). Most string solvers tend to avoid handling regular expressions, by means of postponing them as much as possible or abstracting them into arithmetic/length and other constraints (e.g. Trau, Z3str3RE, Z3str4, CVC4/5, S3). A major point of our work is that taking the opposite approach may work even better when automata are approached from the right angle and implemented carefully, though, heuristics that utilise length information or Parikh images would most probably speed up our algorithm as well. The main selling point of our approach is its efficiency compared to the others, demonstrated on benchmark sets used in other works.

8 Conclusion and Future Work

We have presented a new algorithm for solving a fragment of word equations with regular constraints, complete in SAT cases and for the chain-free fragment. It is based on a tight interconnection of equations with regular constraints and built around a novel characterisation of satisfiability of a string constraint through the notion of stability. We have experimentally shown that the algorithm is very competitive with existing solutions, better especially on difficult examples.

We plan to continue from here towards a complete string solver. This involves including other types of constraints and coming up with a mature and optimised implementation. The core algorithm might also be optimised by using a more compact automata representation of noodles that would eliminate redundancies.

Acknowledgements. This work was supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, the Czech Science Foundation project GA20-07487S, the FIT BUT internal project FIT-S-20-6427, and the project of Ministry of Science and Technology, Taiwan (grant no. 109-2628-E-001-001-MY3).

References

1. OWASP: Top 10. https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf (2013)
2. OWASP: Top 10. https://owasp.org/www-project-top-ten/2017/ (2017)
3. OWASP: Top 10. https://owasp.org/Top10/ (2021)
4. Liana Hadarean: String solving at Amazon. https://mosca19.github.io/program/index.html (2019) Presented at MOSCA'19.
5. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity compiler's model checker. In Shoham, S., Vizel, Y., eds.: Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Volume 13371 of Lecture Notes in Computer Science., Springer (2022) 325–338

6. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In Biere, A., Bloem, R., eds.: Computer Aided Verification, Cham, Springer International Publishing (2014) 646–662
7. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. Formal Methods in System Design **48**(3) (2016) 206–234
8. Barrett, C.W., Tinelli, C., Deters, M., Liang, T., Reynolds, A., Tsiskaridze, N.: Efficient solving of string constraints for security analysis. In: HotSoS'16, ACM Trans. Comput. Log. (2016) 4–6
9. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: FroCoS'15. Volume 9322 of LNCS., Springer (2015) 135–150
10. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In Majumdar, R., Kunčak, V., eds.: Computer Aided Verification, Cham, Springer International Publishing (2017) 453–474
11. Nötzli, A., Reynolds, A., Barbosa, H., Barrett, C., Tinelli, C.: Even faster conflicts and lazier reductions for string solvers. In Shoham, S., Vizel, Y., eds.: Computer Aided Verification, Cham, Springer International Publishing (2022) 205–226
12. Reynolds, A., Notzlit, A., Barrett, C., Tinelli, C.: Reductions for strings and regular expressions revisited. In: 2020 Formal Methods in Computer Aided Design (FMCAD). (2020) 225–235
13. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength smt solver. In Fisman, D., Rosu, G., eds.: Tools and Algorithms for the Construction and Analysis of Systems, Cham, Springer International Publishing (2022) 415–442
14. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: TACAS'09. Volume 5505 of LNCS., Springer (2009) 307–321
15. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In Ramakrishnan, C.R., Rehof, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, Springer Berlin Heidelberg (2008) 337–340
16. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A Z3-based string solver for web application analysis. In: ESEC/FSE'13, ACM Trans. Comput. Log. (2013) 114–124
17. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD). (2017) 55–59
18. Berzish, Murphy: Z3str4: A Solver for Theories over Strings. PhD thesis (2021)
19. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In Kroening, D., Păsăreanu, C.S., eds.: Computer Aided Verification, Cham, Springer International Publishing (2015) 235–254
20. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An SMT solver for regular expressions and linear arithmetic over string length. In Silva, A., Leino, K.R.M., eds.: Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Volume 12760 of Lecture Notes in Computer Science., Springer (2021) 289–312
21. Lin, A.W., Barceló, P.: String solving with word equations and transducers: Towards a logic for analysing mutation XSS. In: POPL'16, ACM Trans. Comput. Log. (2016) 123–136
22. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proc. ACM Program. Lang. **2**(POPL) (2018) 3:1–3:29
23. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL) (2019) 49:1–49:30

24. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proc. ACM Program. Lang. **6**(POPL) (2022) 1–31
25. Chen, T., Hague, M., He, J., Hu, D., Lin, A.W., Rümmer, P., Wu, Z.: A decision procedure for path feasibility of string manipulating programs with integer data type. In Hung, D.V., Sokolsky, O., eds.: Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings. Volume 12302 of Lecture Notes in Computer Science., Springer (2020) 325–342
26. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janku, P.: Chain-free string constraints. In Chen, Y., Cheng, C., Esparza, J., eds.: Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Volume 11781 of Lecture Notes in Computer Science., Springer (2019) 277–293
27. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In Bjørner, N.S., Gurfinkel, A., eds.: 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, IEEE (2018) 1–5
28. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In Cohen, A., Vechev, M.T., eds.: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, ACM (2017) 602–617
29. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Hu, D., Tsai, W., Wu, Z., Yen, D.: Solving not-substring constraint with flat abstraction. In Oh, H., ed.: Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings. Volume 13008 of Lecture Notes in Computer Science., Springer (2021) 305–320
30. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In Biere, A., Bloem, R., eds.: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Volume 8559 of Lecture Notes in Computer Science., Springer (2014) 150–166
31. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In Kroening, D., Pasareanu, C.S., eds.: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Volume 9206 of Lecture Notes in Computer Science., Springer (2015) 462–469
32. Trinh, M., Chu, D., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: CCS, ACM Trans. Comput. Log. (2014) 1232–1243
33. Le, Q.L., He, M.: A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In Ryu, S., ed.: Programming Languages and Systems, Cham, Springer International Publishing (2018) 350–372
34. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Dolby, J., Janku, P., Lin, H., Holík, L., Wu, W.: Efficient handling of string-number conversion. In: Proc. of PLDI'20, ACM (2020) 943–957
35. Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R.: String analysis via automata manipulation with logic circuit representation. In: CAV'16. Volume 9779 of LNCS., Springer (2016) 241–260
36. Wang, H.E., Chen, S.Y., Yu, F., Jiang, J.H.R.: A symbolic model checking approach to the analysis of string and length constraints. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ASE 2018, New York, NY, USA, Association for Computing Machinery (2018) 623–633

37. Hooimeijer, P., Weimer, W.: StrSolve: Solving string constraints lazily. Autom. Softw. Eng. **19**(4) (2012) 531–559
38. Amadini, R., Gange, G., Stuckey, P.J., Tack, G.: A novel approach to string constraint solving. In Beck, J.C., ed.: Principles and Practice of Constraint Programming, Cham, Springer International Publishing (2017) 3–20
39. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for PHP. In: TACAS'10. Volume 6015 of LNCS., Springer (2010) 154–157
40. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods in System Design **44**(1) (2014) 44–70
41. Yu, F., Bultan, T., Ibarra, O.H.: Relational string verification using multi-track automata. Int. J. Found. Comput. Sci. **22**(8) (2011) 1909–1924
42. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In Kroening, D., Păsăreanu, C.S., eds.: Computer Aided Verification, Cham, Springer International Publishing (2015) 255–272
43. Bultan, T., contributors: ABC string solver
44. Scott, J.D., Flener, P., Pearson, J., Schulte, C.: Design and implementation of bounded-length sequence variables. In Salvagnin, D., Lombardi, M., eds.: Integration of AI and OR Techniques in Constraint Programming, Cham, Springer International Publishing (2017) 51–67
45. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Trans. Comput. Log. **21**(4) (2012) 25:1–25:28
46. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: SP'10, IEEE Computer Society (2010) 513–528
47. Cox, A., Leasure, J.: Model checking regular language constraints. CoRR **abs/1708.09073** (2017)
48. Chen, Y., Havlena, V., Lengál, O., Turrini, A.: A symbolic algorithm for the case-split rule in string constraint solving. In d. S. Oliveira, B.C., ed.: Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings. Volume 12470 of Lecture Notes in Computer Science., Springer (2020) 343–363
49. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In Filiot, E., Jungers, R.M., Potapov, I., eds.: Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings. Volume 11674 of Lecture Notes in Computer Science., Springer (2019) 93–106
50. Li, G., Ghosh, I.: Pass: String solving with parameterized array and interval automaton. In Bertacco, V., Legay, A., eds.: Hardware and Software: Verification and Testing, Cham, Springer International Publishing (2013) 15–31
51. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with BEK. In: USENIX Security Symposium 2011, USENIX Association (2011)
52. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic finite state transducers: Algorithms and applications. In: POPL'12, ACM Trans. Comput. Log. (2012) 137–150
53. Fu, X., Li, C.: Modeling regular replacement for string constraint solving. In: NFM'10. Volume NASA/CP-2010-216215 of NASA. (2010) 67–76
54. Trinh, M., Chu, D., Jaffar, J.: progressive reasoning over recursively-defined strings. In: CAV'16. Volume 9779 of LNCS., Springer (2016) 218–240
55. Plandowski, W.: Satisfiability of word equations with constants is in NEXPTIME. In: Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing. STOC '99, New York, NY, USA, Association for Computing Machinery (1999) 721–725

56. Jeż, A.: Recompression: A simple and powerful technique for word equations. J. ACM **63**(1) (feb 2016)
57. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **32**(2) (1977) 147–236 (in Russian).
58. Nielsen, J.: Die isomorphismen der allgemeinen, unendlichen gruppe mit zwei erzeugenden. Mathematische Annalen **78**(1) (1917) 385–397
59. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.C.: Word equations with length constraints: What's decidable? In Biere, A., Nahir, A., Vos, T.E.J., eds.: Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers. Volume 7857 of Lecture Notes in Computer Science., Springer (2012) 209–226
60. Blahoudek, F., Chen, Y., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Word equations in synergy with regular constraints (technical report). `https://arxiv.org/abs/2212.02317` (2022).
61. Aziz, A., Singhal, V., Swamy, G., Brayton, R.K.: Minimizing interacting finite state machines. Technical Report UCB/ERL M93/68, EECS Department, University of California, Berkeley (Sep 1993)
62. Henzinger, M., Henzinger, T., Kopke, P.: Computing simulations on finite and infinite graphs. In: Proceedings of IEEE 36th Annual Foundations of Computer Science. (1995) 453–462
63. Blahoudek, F., Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Noodler. `https://github.com/vhavlena/Noodler` (2022)
64. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A multi-armed string solver. In Huisman, M., Pasareanu, C.S., Zhan, N., eds.: Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings. Volume 13047 of Lecture Notes in Computer Science., Springer (2021) 389–406
65. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proc. ACM Program. Lang. **2**(POPL) (2018) 4:1–4:32
66. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In Cousot, R., ed.: Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings. Volume 2694 of Lecture Notes in Computer Science., Springer (2003) 1–18