



# Word equations in synergy with regular constraints (extended version)

František Blahoudek<sup>1</sup> · Yu-Fang Chen<sup>2</sup> · David Chocholatý<sup>1</sup> ·  
Vojtěch Havlena<sup>1</sup> · Lukáš Holík<sup>1</sup> · Ondřej Lengál<sup>1</sup> · Juraj Síc<sup>1</sup>

Accepted: 22 April 2025 / Published online: 21 May 2025  
© The Author(s) 2025

## Abstract

We propose a new automata-based algorithm for solving string constraints that tightly integrates reasoning about equations and regular constraints. Exchanging information between the two allows an efficient pruning of generated combinatorial cases. The algorithm is based on a novel language-based characterization of satisfiability of word equations with regular constraints. Namely, satisfiability of an equation is implied by its *stability*: the concatenation of the regular languages constraining variables on the left-hand side equals the concatenation of the languages on the right-hand side. It is complete for the chain-free string constraints. We experimentally show that our prototype implementation is competitive with the best string solvers and even superior on difficult examples.

**Keywords** String solving · SMT · Automata · Noodlification

## 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

---

✉ Juraj Síc  
sicjuraj@vut.cz

Vojtěch Havlena  
ihavlena@fit.vut.cz

Lukáš Holík  
holik@fit.vut.cz

Ondřej Lengál  
lengal@fit.vut.cz

<sup>1</sup> Faculty of Information Technology, Brno University of Technology, Božetěchova 2/1, 612 00 Brno, Czechia

<sup>2</sup> Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Nankang, Taipei, Taiwan

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 (which might not always be possible [55]). 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 [56, 57], decidable by the algorithm of Makanin [58] and Jez’s recompression [57]. 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 the string solvers use base algorithms that resemble Makanin [58] or Nielsen’s [59] algorithm in which word equations and regular constraints each generate one level of disjunctive branching, and the two levels multiply. Reasoning about regular constraints particularly is considered complex and expensive, and often turned to only as a last resort.

In this work, we propose an algorithm in which regular constraints are not avoided but instead, they are 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 [59] 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  $\epsilon$ . 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.

Our algorithm systematizes this kind of inference from equations and regular constraints. It 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 characterization of satisfiability of a string constraint: a constraint  $x_1 \dots x_m = x_{m+1} \dots x_n \wedge \bigwedge_{i=1}^m x_i \in \text{Lang}(x_i)$ , where all  $x_i$  are (not necessarily pairwise distinct) variables and  $\text{Lang}$  assigns regular languages to these variables, has a solution if the constraint is *stable*, that is, the languages of the two sides are equal,  $\text{Lang}(x_1) \dots \text{Lang}(x_m) = \text{Lang}(x_{m+1}) \dots \text{Lang}(x_n)$ . 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). Refinements keep accumulating information in regular constraint that in turn allows us to prune branches that would be explored if the equation was considered alone (the algorithm handles pure equations efficiently as well, by deriving regular constraints).

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] which is the most general decidable combination of equations, regular and transducer constraints, and length constraints. For this fragment, the language equality in the definition of stability may be replaced by a single language inclusion. Only one refinement step is then sufficient.

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, Z3STR3RE, Z3- TRAU, Z3- ALPHA, OSTRICH). Besides being generally quite fast, it seems to be superior especially on difficult instances and has the smallest number of timeouts.

This paper is an extended version of [60] presented at FM’23. It contains full proofs, examples and text revisions, new experimental evaluation with updated tools, and it also fixes several imprecisions.

## 2 Preliminaries

**Sets, strings, languages** We use  $\mathbb{N}$  to denote the set of natural numbers (including 0). We fix a finite *alphabet*  $\Sigma$  of *symbols* (usually denoted  $a, b, c, \dots$ ) for the rest of the paper. A sequence of symbols  $w = a_1 \cdots a_n$  from  $\Sigma$  is a *word* or a *string* over  $\Sigma$ , with its *length*  $n$  denoted by  $|w|$ . The set of all words over  $\Sigma$  is denoted as  $\Sigma^*$ . The *empty word* is denoted by  $\epsilon$  ( $\epsilon \notin \Sigma$ ), with  $|\epsilon| = 0$ . The *concatenation* of words  $u$  and  $v$  is denoted  $u \cdot v, uv$  for short ( $\epsilon$  is a neutral element of concatenation). A set of words over  $\Sigma$  is a *language*, the concatenation of languages is  $L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1 \wedge v \in L_2\}, L_1 L_2$  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.

**Graphs** A *directed graph*  $G$  is a pair  $(V, E)$  where  $V$  is a set of vertices and  $E \subseteq V \times V$  is a set of edges. A *strongly connected component* (SCC) of  $G$  is a maximal subgraph of  $G$  where each two vertices are reachable. An SCC is *trivial* if it has exactly one vertex (which does not contain a self-loop) and *non-trivial* otherwise. An SCC  $C$  is *terminal* if the set of vertices reachable from  $C$  equals  $C$  and *source* if there are no edges coming from outside in  $C$ .

**Automata** A (*nondeterministic*) *finite automaton* (NFA) over  $\Sigma$  is a tuple  $\mathcal{A} = (Q, \Delta, I, F)$  where  $Q$  is a finite set of *states*,  $\Delta$  is a set of *transitions* of the form  $q \xrightarrow{a} r$  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  $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} p_n$  where for all  $1 \leq i \leq n$  it holds that  $a_i \in \Sigma \cup \{\epsilon\}$ ,  $p_{i-1} \xrightarrow{a_i} p_i \in \Delta$ , 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  $\epsilon$ -transitions is called  *$\epsilon$ -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*  $\mathcal{A} \circ_\epsilon \mathcal{A}' = (Q \uplus Q', \Delta \uplus \Delta' \uplus \{p \xrightarrow{\epsilon} q \mid p \in F, q \in I'\}, I, F')$ . To intersect their languages, we construct their  *$\epsilon$ -preserving product*  $\mathcal{A} \cap_\epsilon \mathcal{A}' = (Q \times Q', \Delta^\times, I \times I', F \times$

$F'$ ) where  $(q, q') \dashv a \triangleright (r, r') \in \Delta^\times$  if and only if either (1)  $a \in \Sigma$  and  $q \dashv a \triangleright r \in \Delta$ ,  $q' \dashv a \triangleright r' \in \Delta'$ , or (2)  $a = \epsilon$  and either  $q' = r'$ ,  $q \dashv \epsilon \triangleright r \in \Delta$  or  $q = r$ ,  $q' \dashv \epsilon \triangleright r' \in \Delta'$ .

**Noodles** A *noodle* is an automaton which delimits  $n$  subautomata using  $\epsilon$ -transitions. More precisely, an automaton  $N = (Q, \Delta, \{s\}, \{f\})$  is a noodle with  $n$  segments, if for each  $i$ ,  $1 \leq i \leq n$ , there is an automaton  $N(i) = (Q_i, \Delta_i, s_i, f_i)$  such that (1)  $Q = \bigcup_{i=1}^n Q_i$ , (2)  $\Delta = \bigcup_{i=1}^n \Delta_i \cup \Delta_C$ , where each  $\Delta_i$  does not contain  $\epsilon$ -transitions and  $\Delta_C = \{f_i \dashv \epsilon \triangleright s_{i+1} \mid 1 \leq i < n\}$  are connecting  $\epsilon$ -transitions, (3)  $s = s_1$ , and (4)  $f = f_n$ . The language  $L(N)$  can then be seen as a concatenation of languages  $L(N(1))\epsilon L(N(2))\epsilon \dots \epsilon L(N(n))$ .

**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  $u, v, \dots, z$ ), 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  $\mathbb{X}^*$ .<sup>1</sup> 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  $\nu: \mathbb{X} \rightarrow \Sigma^*$ . The assignment is a solution for a word equation  $s = t$  if  $\nu(s) = \nu(t)$  where  $\nu(t')$  for a term  $t' = x_1 \dots x_n$  is defined as  $\nu(x_1) \dots \nu(x_n)$ , 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.

### 3 Stability of string constraints

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

#### 3.1 Stability of single-equation systems

We will first discuss stability of a *single-equation system*

$$\Phi: e_\Phi \wedge \bigwedge_{x \in \mathbb{X}} x \in \text{Lang}_\Phi(x)$$

where  $e_\Phi$  is an equation  $s = t$ ,  $\text{Lang}_\Phi: \mathbb{X} \rightarrow \mathcal{P}(\Sigma^*)$  is a *language assignment*, an assignment of regular languages to variables. We say that a language assignment  $\text{Lang}$  *refines*  $\text{Lang}_\Phi$  if  $\text{Lang}(x) \subseteq \text{Lang}_\Phi(x)$  for all  $x \in \mathbb{X}$ . If  $\text{Lang}(x) = \emptyset$  for some  $x \in \mathbb{X}$ , it is *infeasible*, otherwise it is *feasible*. For a term  $u = x_1 \dots x_n$ , we define  $\text{Lang}(u) = \text{Lang}(x_1) \dots \text{Lang}(x_n)$ . We say that  $\text{Lang}$  is *strongly stable for*  $\Phi$  if  $\text{Lang}(s) = \text{Lang}(t)$ .

The core result of this work is that the existence of a stable language assignment for  $\Phi$  implies the existence of a solution which is formalized below.

**Theorem 1** (Strong stability) *A single-equation system  $\Phi$  has a feasible strongly stable language assignment that refines  $\text{Lang}_\Phi$  if and only if it has a solution.*

To prove Theorem 1 we notice that a weaker, even though more technical, condition works as well. It is *min-stability*, defined as follows. Let  $L^{\min}$  denote the shortest words

<sup>1</sup> Note that terms with letters from  $\Sigma$ , 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  $x_o \in \{a\}$ .

of language  $L$ , i.e.,  $L^{\min} = \{w \mid w \in L \text{ and } |w| \leq |w'| \text{ for all } w' \in L\}$  and let  $\text{Lang}^{\min}$  denote the language assignment that uses only shortest words from  $\text{Lang}$ , i.e., for each  $x \in \mathbb{X}$ ,  $\text{Lang}^{\min}(x) = (\text{Lang}(x))^{\min}$ . For a term  $u = x_1 \dots x_n$ , we define  $\text{Lang}^{\min}(u) = \text{Lang}^{\min}(x_1) \dots \text{Lang}^{\min}(x_n)$ . We say that  $\text{Lang}$  is *strongly min-stable* for  $\Phi$  if  $\text{Lang}^{\min}(s) = \text{Lang}^{\min}(t)$ .

**Theorem 2** (Strong min-stability) *A single-equation system  $\Phi$  has a feasible strongly min-stable language assignment that refines  $\text{Lang}_\Phi$  if and only if it has a solution.*

The proof of min-stability is based on the proof technique commonly used in word equations where, given an equation  $s = t$ , we find its solution by “filling the positions” of the word represented by both sides of the equation so that it stays consistent. Such a technique was used to give a proof for the periodicity theorem of Fine and Wilf [61, 62] or to show that certain properties of words are not expressible as components of solutions of word equations [63], and in many other works [55, 64, 65].

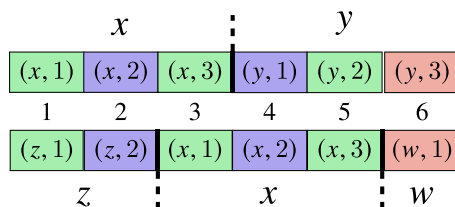
To be more precise, let  $e: x_1 \dots x_m = x_{m+1} \dots x_n$  be a word equation where we are trying to find its solution  $v$  knowing that each variable  $x_i$ ,  $1 \leq i \leq n$  has a fixed length  $\ell_{x_i}$ . We then define  $\ell = \sum_{i=1}^m \ell_{x_i} = \sum_{i=m+1}^n \ell_{x_i}$  as the length of the solution for the equation sides.

The numbers between 1 and  $\ell$  then represent the *positions* in the equation. Each such position is connected with positions in the corresponding left-hand side and right-hand side variables which we call *atoms*. Formally, the set of atoms for variable  $x_i$  is  $\text{Atoms}(x_i) = \{(x_i, j) \mid 1 \leq j \leq \ell_{x_i}\}$  and we collect all atoms in the set  $\text{Atoms} = \bigcup_{i=1}^n \text{Atoms}(x_i)$ . Each position  $p$  in the equation is then connected with its *left atom*  $(x_i, k) \in \text{Atoms}(x_i)$ ,  $1 \leq i \leq m$ , and its *right atom*  $(x_j, k') \in \text{Atoms}(x_j)$ ,  $m + 1 \leq j \leq n$ , where  $p = \sum_{h=1}^{i-1} \ell_{x_h} + k = \sum_{h=m+1}^{j-1} \ell_{x_h} + k'$ . We define  $\text{Atoms}(p) = \{(x_i, k), (x_j, k')\}$  and say that atoms  $(x_i, k)$  and  $(x_j, k')$  are *opposite at the position  $p$* , or just *opposite* (at some position). Last, we define the *value* of an atom  $(x, i)$  in a string assignment  $v$  as the  $i$ th symbol of the word  $v(x)$  and we denote it by  $v(x, i)$ .

**Example 1** Let us explain the notation on an example. Figure 1 shows an equation  $xy = zxw$  where we fix the lengths  $\ell_x = 3$ ,  $\ell_y = 3$ ,  $\ell_z = 2$ , and  $\ell_w = 1$ . The left-hand side of the equation is shown in the upper part, while the right-hand side is shown in the lower part. We also have  $\ell = 3 + 3 = 2 + 3 + 1 = 6$  positions shown in the middle, where for example  $\text{Atoms}(2) = \{(x, 2), (z, 2)\}$  are the atoms opposite at position 2.

Notice, that for the two sides of the equation to be equal, we need to have the same value for each pair of the opposite atoms. In the example above, the opposite atoms  $(x, 2)$  and  $(z, 2)$  must have the same value, but also the opposite atoms  $(y, 1)$  and  $(x, 2)$  must have the same value, which also means that the atoms  $(z, 2)$  and  $(y, 1)$  must have the same value.

This leads to a definition of an equivalence  $\sim$  of atoms as a transitive and reflexive closure of the relation of being opposite. An *atom class* is then an equivalence class of  $\sim$ , and we



**Fig. 1** Equation  $xy = zxw$

denote by  $[\alpha]_{\sim}$  the atom class containing the atom  $\alpha$ . In the example, we have three such equivalence classes differentiated by colors.

In any solution, atoms from each atom class must have the same value. We say that a string assignment  $v$  is *consistent at a set*  $T \subseteq \text{Atoms}$  if  $v$  assigns the same value to every two  $\sim$ -equivalent atoms  $(x, j), (x', j') \in T$ . Obviously,  $v$  is a solution if and only if it is *fully consistent*, i.e. consistent for the entire set  $\text{Atoms}$ .

A central element in the proof is the notion of a *half-full position* in  $T \subseteq \text{Atoms}$ . It is a position with one of its atoms in  $T$  and the other one outside, a *missing atom*. A position can also be *empty*, with no atoms inside  $T$  (all its atoms missing), or *full*, with all atoms inside  $T$  (no atom missing). Note that consistency of a string assignment  $v$  for  $T$  means that for every full position,  $v$  gives the same value to all its atoms.

**Lemma 3** *Let  $T \subsetneq \text{Atoms}$  be a proper subset of  $\text{Atoms}$  without half-full positions and  $v$  a string assignment consistent at  $T$ . Then  $v$  is also consistent at  $T' = T \cup \{(x, i)\}$  where  $(x, i) \in \text{Atoms} \setminus T$  is some atom not occurring in  $T$ .*

**Proof** Because  $T$  is without half-full positions,  $(x, i)$  is an atom of some empty position. Therefore, the consistency of  $v$  at  $T'$  cannot be broken by having different symbol for the opposite atom. Furthermore, no atom of  $[(x, i)]_{\sim}$  could be in  $T$ , otherwise, by the definition of  $\sim$ , there would have to be some half-full position in  $T$ . This means that  $v$  must be consistent at  $T'$ .  $\square$

**Lemma 4** *Let  $\text{Lang}$  be a feasible strongly min-stable language assignment for the equation  $e: x_1 \cdots x_m = x_{m+1} \cdots x_n$  such that for each  $x_i$ ,  $\ell_{x_i}$  is the length of the words from  $\text{Lang}^{\min}(x_i)$ . Let  $T \subsetneq \text{Atoms}$  be a proper subset of  $\text{Atoms}$  for  $e$  with some half-full position and  $v$  a string assignment consistent at  $T$ , such that it assigns to each variable  $x_i$  a string assignment from  $\text{Lang}^{\min}(x_i)$ . Let  $p$  be the left-most half-full position of  $T$  with  $(x, j)$  the atom of  $p$  that is missing in  $T$ . Then there is a string assignment  $v'$  which assigns to each variable  $x_j$  a string assignment from  $\text{Lang}^{\min}(x_j)$  and is consistent at*

$$T' = (T \cup \{(x, j)\}) \setminus \{(x, j') \mid j < j' \leq \ell_x\}.$$

**Proof** We assume that the missing atom  $(x, j)$  is the left atom of  $p$  (the case with the right atom is analogous). We construct  $v'$  as follows. Let  $w = v(x_{m+1}) \cdots v(x_n)$  be the word obtained using the  $v$ -values on the right-hand side of the equation. From min-stability of  $\text{Lang}$ , there must be words  $u_1, \dots, u_m$  from the left languages  $\text{Lang}^{\min}(x_1), \dots, \text{Lang}^{\min}(x_m)$ , respectively, such that  $w = u_1 \cdots u_m$  (note that this sequence of words does not correspond to an assignment since it may associate different occurrences of a variable with different words). Let  $p$  appear withing the  $h$ -th word  $u_h$  in the sequence  $u_1, \dots, u_m$ , i.e.,  $x = x_h$ . We create  $v'$  from  $v$  by replacing  $v(x)$  with  $u_h$ , i.e., we let

$$v' = (v \setminus \{x \mapsto v(x)\}) \cup \{x \mapsto u_h\}.$$

We need to prove that  $v'$  is consistent for  $T'$ . We first show that

$$v'(x', j') = v(x', j') \text{ for every } (x', j') \in T' \setminus \{(x, j)\} \quad (1)$$

This is obvious for any  $x'$  different from  $x$ , as nothing has changed for it:  $v'(x') = v(x')$  and  $\text{Atoms}(x') \cap T' = \text{Atoms}(x') \cap T$ . If on the other hand  $x' = x$ , then we argue as follows. Let  $p' = p - j + j'$  (the position within the same occurrence of  $x$  as  $p$  with  $(x, j') \in \text{Atoms}(p')$ ). Let atom  $(x'', j'')$  be the opposite atom of  $p'$  (the right atom). From the way  $v'$  is constructed, copying the right atoms to the left, we know that  $v'(x, j')$  is the

copy of  $v(x'', j'')$ . We need to show that the copying did not change its value, i.e., that it had the same value before, that is  $v(x, j') = v(x'', j'')$ . From the definition of  $T'$  we know that  $j' < j$  therefore  $p' < p$ . Since  $p$  is the left-most half-full position,  $p'$  must not be half-full in  $T$ . Hence, having an atom in  $T$ ,  $p'$  is full in  $T$  ( $p'$  has an atom in  $T$  because  $T \supset T' \setminus \{(x, j)\}$  and  $(x, j') \in T$  *setminus*  $\{(x, j)\}$ ). Therefore, since  $v$  is consistent for  $T$ , both atoms of  $p'$  had the same value in  $v$ , hence  $v(x, j') = v(x'', j'')$ , so (1) holds. From this and the facts that  $T \supset T' \setminus \{(x, j)\}$  and  $v$  is consistent in  $T$ , it holds that  $v$  is consistent for  $T' \setminus \{(x, j)\}$ .

Finally, we need to show  $v'$  stays consistent for  $T'$ , even with  $(x, j)$ . That is, either  $p$  is not full in  $T'$ , or, given  $(y, k)$ , the other (right) atom of  $p$ ,  $v'(x, j) = v'(y, k)$ . We note that  $(y, k)$  is in  $T$  because we  $p$  is a half-full position of  $T$ . Therefore, we have

$$v'(x, j) = v(y, k) \tag{2}$$

by the construction—the right-to left copying of atoms. We will distinguish two cases, either  $x = y$  or not.

The first case,  $x \neq y$ , is easy. We have  $v(y, k) = v'(y, k)$  because the value of  $y$  in  $v'$  is the same as in  $v$ . Then  $v'(x, j) = v'(y, k)$  by (2).

The second case,  $y = x$ , is more complicated. We have three possibilities:

- $(k < j)$  We know that  $(x, k) \in T$ . Hence from  $k < j$  and the definition of  $T'$ , we have that  $(x, k) \in T'$ . Therefore, from (1), we know that  $v'(x, k) = v(x, k)$ . From this and (2), we get  $v'(x, j) = v'(x, k)$ .
- $(k = j)$  This case is not possible since  $p$  would only have one atom and could never be half-full.
- $(k > j)$  Here  $(x, k) \notin T'$  by definition, hence  $p$  is only half-full in  $T'$ . □

We can use Lemmas 3 and 4 to prove Theorem 2:

**Proof of Theorem 2** ( $\Leftarrow$ ): Let  $v$  be a solution of  $\Phi$ . Then the language assignment  $\text{Lang}$  that assigns to each  $x \in \mathbb{X}$  the singleton language  $\{v(x)\}$  is feasible and strongly min-stable for  $\Phi$ .

( $\Rightarrow$ ): Let  $e_\Phi$  be  $x_1 \cdots x_m = x_{m+1} \cdots x_n$  and let  $\text{Lang}$  be a feasible strongly stable language assignment that refines  $\text{Lang}_\Phi$ . We will show that we can find a solution  $v$  that uses the shortest words, i.e., words from  $\text{Lang}^{\text{min}}$ . This fixes the length of the word  $v(x_i)$  for each variable  $x_i$ ,  $1 \leq i \leq n$ , as the length  $\ell_{x_i}$  of the shortest word in  $\text{Lang}(x_i)$ .

We will now, using Lemmas 3 and 4, construct a sequence  $(T_1, v_1), \dots, (T_k, v_k)$  where for each  $i : 1 \leq i \leq k$ ,  $v_i$  is a string assignment that (1) is consistent at  $T_i \subseteq \text{Atoms}$  and (2) assigns to each variable  $x_j$  a string from  $\text{Lang}^{\text{min}}(x_j)$ . The sequence terminates with  $T_k = \text{Atoms}$ , a fully consistent  $v_k$  that is a solution of  $\Phi$ .

First, we start with  $(T_1, v_1)$  where  $T_1 = \emptyset$  and  $v_1$  randomly assigns to each variable  $x_i$ ,  $1 \leq i \leq n$ , some string from  $\text{Lang}^{\text{min}}(x_i)$ . Obviously,  $v_1$  is consistent for  $\emptyset$ .

Then, given  $(T_i, v_i)$  where  $v_i$  is a string assignment consistent at  $T_i \subsetneq \text{Atoms}$  assigning to each variable  $x_j$  a string from  $\text{Lang}^{\text{min}}(x_j)$ , we construct  $(T_{i+1}, v_{i+1})$  in the following way.

If  $T_i$  does not contain a half-full position we set  $T_{i+1} = T_i \cup \{(x, i)\}$  where  $(x, i) \in \text{Atoms} \setminus T_i$  is some atom not occurring in  $T_i$  and  $v_{i+1} = v_i$ . By Lemma 3,  $v_{i+1}$  is consistent at  $T_{i+1}$  and it still assigns each variable  $x_j$  a string from  $\text{Lang}^{\text{min}}(x_j)$ .

For the case that  $T_i$  contains a half-full position, we set

$$T_{i+1} = (T_i \cup \{(x, j)\}) \setminus \{(x, j') \mid j < j' \leq \ell_x\}$$

where  $(x, j)$  is the missing atom of the left-most half-full position of  $T_i$ . Then by Lemma 4, there is a string assignment  $v_{i+1}$  that is consistent at  $T_{i+1}$  and it assigns to each variable  $x_j$  a string from  $\text{Lang}^{\text{min}}(x_j)$ .

It remains to show that the sequence eventually terminates with  $(T_k, v_k)$  where  $T_k = \text{Atoms}$ . The sequence can terminate only when it reaches  $\text{Atoms}$ , so we only need to show that the sequence grows under some (partial) order  $\leq$ . We define, for  $T_1, T_2 \subseteq \text{Atoms}$ , the relation  $\leq$  as  $T_1 \leq T_2$  if and only if  $T_1 = T_2$  or, if they are not equal, given left-most position  $p$  at which  $T_1$  and  $T_2$  differ, we have  $\text{Atoms}(p) \cap T_1 \subsetneq \text{Atoms}(p) \cap T_2$ . It is easy to see that  $\leq$  is reflexive, antisymmetric and transitive, therefore it is a (partial) order and for each  $T_i$ ,  $1 \leq i < k$ , we have  $T_i < T_{i+1}$ . Therefore,  $v_k$  is a solution of  $\Phi$ .  $\square$

We can now return to the proof of strong stability:

**Proof of Theorem 1** ( $\Leftarrow$ ): Let  $v$  be a solution of  $\Phi$ . Then the language assignment  $\text{Lang}$  that assigns to each  $x \in \mathbb{X}$  the singleton language  $\{v(x)\}$  is feasible and strongly stable for  $\Phi$ .

( $\Rightarrow$ ): Let  $e_\Phi$  be  $x_1 \cdots x_m = x_{m+1} \cdots x_n$  and let  $\text{Lang}$  be a feasible strongly stable language assignment that refines  $\text{Lang}_\Phi$ . Because  $\text{Lang}$  is strongly stable for  $\Phi$ , then we can easily show that is also strongly min-stable for  $\Phi$ , i.e. if  $\text{Lang}(x_1 \cdots x_m) = \text{Lang}(x_{m+1} \cdots x_n)$ , then  $\text{Lang}^{\min}(x_1 \cdots x_m) = \text{Lang}^{\min}(x_{m+1} \cdots x_n)$ .

Indeed, from the equality of the languages, every concatenation  $w_1 \cdots w_m$  of minimum length words on the left must have an equivalent counterpart  $w_{m+1} \cdots w_n$  on the right, and vice versa. The words on the right must be minimal too since otherwise one could compose a shorter word on the right, and its counterpart on the left would be shorter than  $w_1 \cdots w_m$ , which contradicts that  $w_1, \dots, w_m$  are minimal.

Then from Theorem 2,  $\Phi$  must have a solution.  $\square$

**Weak stability** In special cases (that are practically relevant), we can use a sufficient condition for satisfiability that is weaker than strong stability. Namely, we say that  $t$  is *loose* in the equation  $e_\Phi : s = t$  if all variables of  $t$  appear in  $e_\Phi$  only once. In this case, the language equality in strong stability can be weakened to one-sided language inclusion. We say that  $\text{Lang}$  is *weakly stable for  $\Phi$*  if  $t$  is loose in  $e_\Phi$  and  $\text{Lang}(s) \subseteq \text{Lang}(t)$ , and we show a version of theorem Theorem 1 with weak stability:

**Theorem 5** (Weak stability) *A single-equation system  $\Phi : s = t \wedge \bigwedge_{x \in \mathbb{X}} x \in \text{Lang}_\Phi(x)$ , where  $t$  is loose, has a feasible weakly stable language assignment that refines  $\text{Lang}_\Phi$  if and only if it has a solution.*

**Proof** ( $\Leftarrow$ ): Let  $v$  be a solution of  $\Phi$ . Then the language assignment  $\text{Lang}$  that assigns to each  $x \in \mathbb{X}$  the singleton language  $\{v(x)\}$  is feasible and weakly stable for  $\Phi$ .

( $\Rightarrow$ ): Let  $\text{Lang}$  be a feasible weakly stable language assignment that refines  $\text{Lang}_\Phi$  and let  $e_\Phi$  be  $x_1 \cdots x_m = y_1 \cdots y_n$  where  $y_1 \cdots y_n$  is loose, i.e. variables  $y_j$ ,  $1 \leq j \leq n$ , occur in the equations exactly once. We construct a solution  $v$  of  $\Phi$  by assigning strings to variables on the left-hand side so that for each  $x_i$ ,  $1 \leq i \leq m$ ,  $v(x_i) \in \text{Lang}(x_i)$ . From the weak stability of  $\text{Lang}$ , we know that  $v(x_1) \dots v(x_m) \in \text{Lang}(x_1 \dots x_m) \subseteq \text{Lang}(y_1 \dots y_n)$ . We can therefore find words  $w_j \in \text{Lang}(y_j)$ ,  $1 \leq j \leq n$  such that  $w_1 \dots w_n = w$ . By the looseness of  $y_1 \cdots y_n$ , each  $y_j$  occurs in the equation exactly once, and so we can simply let  $v(y_j) = w_j$  to obtain a solution of  $\Phi$ .  $\square$

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 synchronization. For instance, for  $\Phi: xx = y \wedge x \in \{a, b\} \wedge y \in \{ab\}$ , the inclusion  $\text{Lang}(xx) \subseteq \text{Lang}(y)$  is satisfied by no feasible refinement  $\text{Lang}$  of  $\text{Lang}_\Phi$ , revealing that  $\Phi$  has no solution, while  $\text{Lang}(xx) \supseteq \text{Lang}(y)$  is satisfied already by  $\text{Lang}_\Phi$  itself.

### 3.2 Stability of multi-equation systems

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

We generalize stability in a way that combines both strong and weak stability of single equation systems (Theorems 1 and 5) in a way that favors weak stability over strong stability. Every equation  $s = t$  is interpreted as the pair of language inclusions  $\text{Lang}_\Phi(s) \subseteq \text{Lang}_\Phi(t)$ ,  $\text{Lang}_\Phi(t) \subseteq \text{Lang}_\Phi(s)$ . The inclusions are represented as *inclusion terms*  $s \subseteq t$  and  $t \subseteq s$ , respectively, that are *satisfied under*  $\text{Lang}_\Phi$  when the corresponding language inclusion hold. Generalizing the notion of looseness from Section 3.1, we say that a term  $t$  is *loose in a set of inclusion terms*  $I$  if each of its variable has only a single occurrence in the right-hand sides of inclusions of  $I$ . The sufficient condition on the set of inclusions is then defined through the notion of an *inclusion graph* for  $\Phi$ . It is a directed graph  $G = (V, E)$  where vertices  $V$  are inclusion constraints of the form  $s_i \subseteq t_i$  or  $t_i \subseteq s_i$ , for  $1 \leq i \leq m$ , and  $E \subseteq V \times V$ , and which satisfies the following conditions:

- (IG1) For each  $s = t$  in  $\mathcal{E}$ , at least one of the nodes  $s \subseteq t$ ,  $t \subseteq s$  is in  $V$ .
- (IG2) If  $s \subseteq t \in V$  and  $t$  is not loose in  $V$ , then also  $t \subseteq s \in V$ .
- (IG3)  $(s_i \subseteq t_i, s_j \subseteq t_j) \in E$  if and only if  $s_i \subseteq t_i, s_j \subseteq t_j \in V$  and  $s_i$  and  $t_j$  share a variable.
- (IG4) If  $s_i \subseteq t_i \in V$  lies on a cycle, then also  $t_i \subseteq s_i \in V$ .

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

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 a cyclic dependency caused by transitively propagating the inclusion relation.

We will now work towards showing that satisfiability of  $\Phi$  is equivalent to existence of an inclusion graph that is stable for a feasible refinement of  $\text{Lang}_\Phi$  (Theorem 7). In the proofs, we use  $\text{Vars}(C)$  to denote the set of all variables that occur in the vertices of  $C \subseteq V$ . Recall that SCC stands for strongly connected component. We start by proving some technical properties of inclusion graphs:

**Lemma 6** *Let  $G = (V, E)$  be an inclusion graph of  $\Phi$ . Then the following holds:*

1. Every non-trivial SCC of  $G$  is terminal.
2. For any two different non-trivial SCCs  $C_1$  and  $C_2$  of  $G$ ,  $\text{Vars}(C_1) \cap \text{Vars}(C_2) = \emptyset$ .
3. If  $x$  is a variable on the right-hand side of some vertex  $s \subseteq t$  that is a trivial SCC, then  $x$  occurs in  $s \subseteq t$  exactly once.

4. Let  $C_1$  and  $C_2$  be two different SCCs of  $G$  where  $C_1$  is terminal. For every variable  $x \in \text{Vars}(C_1)$ ,  $x$  does not occur on the right-hand side of any vertex of  $C_2$ .

**Proof**

- (1) For the sake of contradiction, assume that there is a non-trivial SCC  $C$  of  $G$ , a vertex  $s \subseteq t \in V$  that is not in  $C$ , and a vertex  $s_C \subseteq t_C \in C$  such that  $(s_C \subseteq t_C, s \subseteq t) \in E$ . From Condition (IG3), it follows that  $s_C$  and  $t$  share a variable, for example  $x$ . As vertex  $s_C \subseteq t_C$  belongs to a non-trivial SCC, it must lie on a cycle, so from Condition (IG4) we have a vertex  $t_C \subseteq s_C \in V$ . The variable  $x$  then occurs on the right-hand side of at least two vertices ( $t_C \subseteq s_C$  and  $s \subseteq t$ ), so by Condition (IG2), there needs to be the vertex  $t \subseteq s \in V$ . From Condition (IG3), we obviously have edges  $(s \subseteq t, t \subseteq s)$  and  $(t_C \subseteq s_C, s_C \subseteq t_C)$  in  $E$ , and also, since  $s_C$  and  $t$  share  $x$ , we have the edge  $(t \subseteq s, t_C \subseteq s_C) \in E$ . But then  $C$  is reachable from  $s \subseteq t$ , which is a contradiction.
- (2) For the sake of contradiction, assume there are two different non-trivial SCCs  $C_1$  and  $C_2$  of  $G$  and a variable  $x \in \text{Vars}(C_1) \cap \text{Vars}(C_2)$ . Then there will be vertices  $s_1 \subseteq t_1, t_1 \subseteq s_1 \in C_1$  and  $s_2 \subseteq t_2, t_2 \subseteq s_2 \in C_2$  with occurrences of  $x$  in  $s_1$  and  $s_2$ . From Condition (IG3), there are edges  $(s_1 \subseteq t_1, t_2 \subseteq s_2) \in E$  and  $(s_2 \subseteq t_2, t_1 \subseteq s_1) \in E$ , which is a contradiction with the assumption that  $C_1$  and  $C_2$  are different.
- (3) We have to show that  $x$  cannot occur in  $s$ , nor can there be two occurrences of  $x$  in  $t$ . If  $x$  occurred in  $s$ , then there would be a self-loop and by Condition (IG4),  $t \subseteq s \in V$ . Similarly, if  $x$  occurred twice in  $t$ , by Condition (IG2),  $t \subseteq s \in V$ . However, by Condition (IG3), there will be edges  $(s \subseteq t, t \subseteq s) \in E$  and  $(t \subseteq s, s \subseteq t) \in E$ , therefore  $s \subseteq t$  is not a trivial SCC, which is a contradiction.
- (4) For the sake of contradiction, assume that  $x$  occurs in some vertex  $s_1 \subseteq t_1$  of  $C_1$  while it also occurs on the right-hand side of some  $s_2 \subseteq t_2$  of  $C_2$ . If  $C_1$  is non-trivial, then, from Condition (IG4), both  $s_1 \subseteq t_1$  and  $t_1 \subseteq s_1$  must belong to  $C_1$ , therefore  $x$  occurs on the left-hand side of one of these vertices of  $C_1$ . We then have either edge  $(s_1 \subseteq t_1, s_2 \subseteq t_2)$  or  $(s_1 \subseteq s_1, s_2 \subseteq t_2)$  from SCC  $C_1$  to SCC  $C_2$ , which is a contradiction with  $C_1$  being terminal. If  $C_1$  is trivial, then it contains only the vertex  $s_1 \subseteq t_1$ . If  $x$  occurs in  $s_1$ , then there is an edge  $(s_1 \subseteq t_1, s_2 \subseteq t_2) \in E$ , which is again a contradiction with  $C_1$  being terminal. Therefore,  $x$  must occur in  $t_1$ , i.e., it occurs on two right-hand sides in  $G$ . However, from Condition (IG2), there will also be a vertex  $t_1 \subseteq s_1 \in V$  and by Condition (IG3), there will be edges  $(s_1 \subseteq t_1, t_1 \subseteq s_1) \in E$  and  $(t_1 \subseteq s_1, s_1 \subseteq t_1) \in E$ , therefore  $s_1 \subseteq t_1$  is not a trivial SCC, which is a contradiction.  $\square$

The following theorem then puts a relation between the satisfiability of a string constraint and the stability of the corresponding inclusion graph. 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.

**Theorem 7** (Inclusion graph stability) *Let  $G$  be an inclusion graph for  $\Phi$ . There is a feasible language assignment that refines  $\text{Lang}_\Phi$  and is stable for  $G$  if and only if  $\Phi$  has a solution.*

**Proof** ( $\Leftarrow$ ): Let  $\nu$  be a solution of  $\Phi$ . Then the language assignment  $\text{Lang}$  that assigns to each  $x \in \mathbb{X}$  the singleton language  $\{\nu(x)\}$  is feasible and stable for  $G$ .

( $\Rightarrow$ ): We will show how to construct a solution of  $\Phi$  given a feasible language assignment  $\text{Lang}$  that refines  $\text{Lang}_\Phi$  and is stable for  $G$ . Intuitively, the construction proceeds by refining  $\text{Lang}$  with the solutions of equations from  $\Phi$ , starting with those whose inclusions occur in non-trivial SCCs (using Theorem 1) and then proceeding upward on the structure of  $G$ , with equations with only one inclusion in  $G$  (using Theorem 5),

Formally, let us consider a non-trivial SCC  $C$  of  $G$ , which, by Lemma 6(1), must be terminal. Since each vertex  $s \subseteq t$  of  $C$  lies on cycle, from Condition (IG4), the vertex  $t \subseteq s$  must be also in  $C$ . Therefore, since  $\text{Lang}$  is feasible and stable for  $G$  (and so also for  $C$ ), it is also a feasible strongly stable language assignment for  $s = t$ . We would now want to apply Theorem 1 to find a solution  $v$  for  $s = t$ , which we would then use to refine  $\text{Lang}$  to create new language assignment  $\text{Lang}'$ , in which each variable  $x$  from  $s = t$  is mapped to the singleton language  $\{v(x)\}$ .

However, it is possible that  $\text{Lang}'$  would then not be stable for other vertices of  $C$  (it would be still stable for vertices outside  $C$ , as according to Lemma 6(4), variables from  $C$  can only occur on the left-hand sides of vertices of other SCCs, and refining languages on the left-hand side has no impact on the stability). Therefore, we need to find the solution of the system  $\Phi_C : \mathcal{E}_C \wedge \bigwedge_{x \in \text{Vars}(C)} x \in \text{Lang}(x)$  where  $\mathcal{E}_C : \bigwedge_{i=1}^k s_i = t_i$  contains only equations of  $\Phi$  whose inclusions appear in  $C$ . This is still multi-equation system, but we can transform it into single-equation system using the following trick. Let  $\sharp$  be a fresh symbol. We create a single-equation system

$$s_1 \sharp s_2 \sharp \dots \sharp s_k = t_1 \sharp t_2 \sharp \dots \sharp t_k \wedge \bigwedge_{x \in \text{Vars}(C)} x \in \text{Lang}(x)$$

whose solutions are exactly the solutions of  $\Phi_C$  and  $\text{Lang}$  is stable for it. According to Theorem 1, this system has a solution  $v_C$ , which is also a solution of  $\Phi_C$ .

We can now take such a solution  $v_C$  for each non-trivial SCC  $C$  (note again, that by Lemma 6(2), they do not share variables) and use them to refine  $\text{Lang}$  in the following way:

$$\text{Lang}'(x) = \begin{cases} \{w\} & \text{if } (x \mapsto w) \in v_C \text{ for some non-trivial SCC } C, \\ \text{Lang}(x) & \text{otherwise.} \end{cases}$$

Because we changed assignment only of the variables occurring in non-trivial SCCs, the language assignment  $\text{Lang}'$  is still stable for  $G$ .

We now proceed by creating a graph  $G'$  by removing all non-trivial SCCs from  $G$  (they are not needed any more since we have their solutions in  $\text{Lang}'$ ) and iteratively applying following step until  $G'$  is empty. Take and remove any terminal vertex  $u = s \subseteq t$  from  $G'$  ( $G'$  contains only trivial SCCs, so it must be acyclic, i.e., it contains a terminal vertex). As  $\{u\}$  is a trivial SCC of  $G$ , by Lemma 6(3), every variable on the right-hand side  $t$  occurs in the corresponding equation  $s = t$  exactly once. Therefore,  $t$  is loose in  $s = t$  and we can use Theorem 5 with the single-equation system  $\Phi_u := s = t \wedge \bigwedge_{x \in \text{Vars}(\{u\})} x \in \text{Lang}'(x)$  to get a string assignment  $v_u$ , which assigns strings to the variables in  $\text{Vars}(\{u\})$  and is a solution of  $\Phi_u$ . We now refine  $\text{Lang}'$  by assigning to each variable  $x \in \text{Vars}(\{u\})$  the singleton  $\{v_u(x)\}$  its solution. Note that by Lemma 6(4),  $\text{Lang}'$  stays stable for  $G'$ , as  $u$  does not share any variable with the right-hand side of any remaining vertex (and again, refining languages for variables on the left-hand side only has no impact on the stability of inclusions).

At the end, we are left with a language assignment  $\text{Lang}'$ , which assigns to each variable a string. We can therefore take the corresponding string assignment  $v$ , where for each variable

$x$ , we have  $v(x) = w$  if and only if  $\text{Lang}'(x) = \{w\}$ . This assignment is a solution for  $\Phi$  as we have shown that it is a solution for every equation of  $\Phi$  and  $\text{Lang}'$  is a refinement of  $\text{Lang}$ .  $\square$

**Min-stability** In our algorithm, we will use inclusions in the graph as a test for termination. However, inclusion testing is generally an expensive operation. To make it simpler, we can, analogously as for single-equation system, define the notion of min-stability for inclusion graphs. We say that the language assignment  $\text{Lang}$  is *min-stable for an inclusion graph*  $G = (V, E)$  if for every inclusion  $v = s \subseteq t \in V$ , it holds that

1. if  $v$  forms a trivial SCC, then normal (not minimal) stability holds for it, i.e.,  $\text{Lang}(s) \subseteq \text{Lang}(t)$ ,
2. if  $v$  is in non-trivial SCC, then  $\text{Lang}^{\min}(s) \subseteq \text{Lang}(t)$ .

We can prove that min-stability is guarantees a solution:

**Theorem 8** (Inclusion graph min-stability) *Let  $G$  be an inclusion graph for  $\Phi$ . There is a feasible language assignment that refines  $\text{Lang}_{\Phi}$  and is min-stable for  $G$  if and only if  $\Phi$  has a solution.*

**Proof** The proof is nearly identical to the proof of Theorem 7 but, to construct the solution for non-trivial SCCs  $C$ , we need to use Theorem 2 instead of Theorem 1. However, to be able to use Theorem 2, we have to prove that if  $\text{Lang}^{\min}(s) \subseteq \text{Lang}(t)$  and  $\text{Lang}^{\min}(t) \subseteq \text{Lang}(s)$  both hold, then  $\text{Lang}^{\min}(s) = \text{Lang}^{\min}(t)$ .

Indeed, every concatenation  $w_s$  of minimum length words of  $s$  must have an equivalent counterpart  $w_t$  for  $t$ . But,  $w_t$  must also be a concatenation of minimum length words, otherwise one could compose a shorter concatenation, and its counterpart in  $s$  would be shorter than  $w_s$ , which contradicts that  $w_s$  consists of minimum length words.

Furthermore, from Lemma 6(2) and 6(4), we have that variables of  $C$  can occur outside  $C$  only on the left-hand side of trivial SCCs, where we use normal stability. Hence, we cannot break their stability by using the solution of  $C$  for refining the language assignment.  $\square$

It would seem that min-stability for inclusion graphs could be defined with the condition  $\text{Lang}^{\min}(s) \subseteq \text{Lang}(t)$  holding for all inclusions of the graph, even those that form trivial SCCs. We can easily prove that Theorem 5 holds even in the case where we use *weak min-stability*, i.e., when for an equation  $s = t$  with loose  $t$ , we have language assignment  $\text{Lang}$  with  $\text{Lang}^{\min}(s) \subseteq \text{Lang}(t)$ . We would then replace Theorem 5 in the proof of Theorem 7, with the one using weak min-stability to construct the solution of such equation. However, this is not correct, as this solution does not necessarily use the shortest words in the right-hand side. For example, given a multi-equation system  $x = y \wedge y = z \wedge x \in \{a\} \wedge y \in \{a, aa\} \wedge z \in \{aa\}$  and an inclusion graph  $y \subseteq z \rightarrow x \subseteq y$ , we can see that for both inclusions, the shortest word on the left-hand side is included in the language of the right-hand side, but the system does not have a solution.

### 3.3 Constructing inclusion graphs and chain-freeness

We now discuss an algorithm for constructing a suitable inclusion graph, i.e. one that contains as few inclusions as possible and is acyclic whenever possible.

The graph is obtained from a simplified version of the splitting graph of [26], which is the basis of the definition of the chain-free fragment, for which our algorithm is complete. More formally, a *simplified splitting graph*  $SG_\Phi$  for a multi-equation system  $\Phi: \bigwedge_{i=1}^m s_i = t_i$  is a directed graph whose nodes are all inclusions  $s_i \subseteq t_i, t_i \subseteq s_i$ , for  $1 \leq i \leq m$ , and it has an edge from  $s \subseteq t$  to  $s' \subseteq t'$  if and only 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  $x = y$ , for  $x, y \in \mathbb{X}$ , there will be no edge between  $x \subseteq y$  and  $y \subseteq x$ , while for the equation  $xx = y$ , there will be an edge from  $xx \subseteq y$  to  $y \subseteq xx$ , as there are two different occurrences of  $x$  in  $xx = y$ ).

The algorithm will be designed based on the following observations, that follow from Lemma 6(1) and are reflected in the lemmas below.

1. Nodes on cycles in a minimal inclusion graph are exactly nodes on cycles in the simplified splitting graph.
2. Only a terminal SCC of an inclusion graph can be non-trivial (Lemma 6(1)).

This means that a minimal inclusion graph consists of non-trivial SCCs that correspond to non-trivial SCCs in the splitting graph, and from acyclic paths leading to them. The acyclic paths are constituted by nodes that do not have their duals in the inclusion graph, since otherwise they would form a cycle. The algorithm therefore constructs the graph by first unfolding the acyclic paths, after which it adds the terminal SCCs.

The algorithm for constructing an inclusion graph from  $SG_\Phi$  starts by iteratively removing nodes that are trivial source strongly-connected components (SCCs) from  $SG_\Phi$ . With every removed node  $v = s \subseteq t$ , the algorithm removes from  $SG_\Phi$  also the dual node  $\text{dual}(v) = t \subseteq s$ , and it adds  $v$  to the inclusion graph. When no trivial source SCCs are left, the algorithm adds to the inclusion graph all the remaining nodes.

The pseudocode of the algorithm is shown in Algorithm 1. It uses  $\text{SCC}(G)$  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 2** In Fig. 2, we show an example of the construction of the inclusion graph  $G$  from  $SG_\Phi$  for  $\mathcal{E}: z = u \wedge u = v \wedge uvx = x$ . Edges of  $SG_\Phi$  are solid lines, the inclusion graph has both solid and dashed edges. The inner red boxes are the non-trivial SCCs of  $SG_\Phi$ . They are enclosed in the box of nodes that are added on Line 5 of Algorithm 1. The outermost box encloses the inclusion graph, including one node added on Line 4.

---

**Algorithm 1**  $\text{incl}(\Phi)$

---

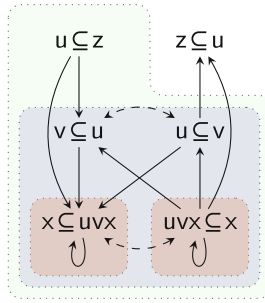
**Input:** A multi-equation system  $\Phi$ .

**Output:** An inclusion graph for  $\Phi$ .

```

1  $G := SG_\Phi; V' := \emptyset$ 
2 while  $G$  has a trivial source SCC  $(\{v\}, \emptyset)$  do
3   |  $G := G \setminus \{v, \text{dual}(v)\}$ 
4   |  $V' := V' \cup \{v\}$ 
5  $V := V' \cup$  the remaining nodes of  $G$ 
6  $E :=$  edges induced by (IG3) for  $V$ 
7 return  $(V, E)$ 
```

---



**Fig. 2** The inclusion graph for  $z = u \wedge u = v \wedge uvx = x$

**Lemma 9** Let  $SG_\Phi = (V, E)$  be the simplified splitting graph for a multi-equation system  $\Phi$  and  $V_c \subseteq V$  the nodes that are on some cycle of  $SG_\Phi$ . Let  $v \in V$  be a node with a path from  $V_c$  to  $\text{dual}(v)$ . Then every inclusion graph for  $\Phi$  contains node  $v$ .

**Proof** Let  $\pi = u_1, u_2, \dots, u_n$ , where  $u_n = \text{dual}(v)$ , be the path from  $V_c$  to  $\text{dual}(v)$ . We show by induction that  $\text{dual}(u_i)$ ,  $1 \leq i \leq n$ , must be in every inclusion graph for  $\Phi$  which immediately proves the lemma, as  $\text{dual}(u_n) = v$ .

Base case ( $i = 1$ ): We have  $u_1 \in V_c$ , i.e., there is a cycle  $\pi_c = u_1, v_1, v_2, \dots, v_m, u_1$  in  $SG_\Phi$ . By contradiction, assume that there exists an inclusion graph  $G = (V_G, E_G)$  that does not contain  $\text{dual}(u_1)$ . By Condition (IG1),  $u_1 \in V_G$ . Furthermore, there must be a  $v_k = s_k \subseteq t_k \in \pi_c$ , s.t.  $v_k \notin V_G$ , otherwise the cycle  $\pi_c$  would be fully in  $G$  (as the condition for edges of  $SG_\Phi$  is a stricter version of (IG3)) and by (IG4),  $\text{dual}(u_1)$  would have to be a node of  $G$ . By Condition (IG1), we again have  $\text{dual}(v_k) \in V_G$ . Let  $k$  be the largest possible, i.e.  $v_{k+1} = s_{k+1} \subseteq t_{k+1} \in V_G$  (for  $k = m$  we have  $v_{k+1} = u_1$ ). Furthermore, from the fact that  $(v_k, v_{k+1}) \in E$ , there must be different occurrences of some variable  $x$  in  $s_k$  and  $t_{k+1}$ . But  $s_k$  is on the right-hand side of  $\text{dual}(v_k)$ ,  $t_{k+1}$  is on the right-hand side of  $v_{k+1}$ , and both these nodes are in  $G$ , therefore, from Condition (IG2),  $v_k \in V_G$ , which is a contradiction.

Induction step ( $i \geq 1$ ): We know that  $\text{dual}(u_{i-1})$  is in every inclusion graph for  $\Phi$ . By contradiction, assume that there exists an inclusion graph  $G = (V_G, E_G)$ , where  $\text{dual}(u_i) \notin V_G$ , so by y Condition (IG1),  $u_i \in V_G$ . From  $(u_{i-1}, u_i) \in E$ , we know that there are different occurrences of some variable  $x$  on the left-hand side of  $u_{i-1}$  and the right-hand side of  $u_i$ . But then  $x$  occurs both on the right-hand side of  $u_i$  and right-hand side of  $\text{dual}(u_{i-1})$ , both these nodes are in  $G$ , therefore, from Condition (IG2),  $\text{dual}(u_i) \in V_G$ , which is a contradiction.  $\square$

**Theorem 10** For a multi-equation system  $\Phi$ ,  $\text{incl}(\Phi)$  is an inclusion graph for  $\Phi$  with the smallest possible number of vertices. Moreover, if the simplified splitting graph  $SG_\Phi$  is acyclic, then  $\text{incl}(\Phi)$  is also acyclic.

**Proof** Let  $G_\Phi = (V, E)$  be a graph obtained by  $\text{incl}(\Phi)$  and  $V'$  a set of vertices at the end of the algorithm, i.e., on Line 5. We prove that  $G_\Phi$  meets Conditions (IG1)–(IG4).

**Condition (IG1)** Follows trivially from Algorithm 1.

**Condition (IG2)** Consider some  $u = s \subseteq t \in V$  for which Condition (IG2) does not hold, i.e.,  $\text{dual}(u) \notin V$  (which also means that  $u \in V'$ ) and the term  $t$  contains a variable  $x$ , which either occurs twice in  $t$  or there is some other vertex  $u' = s' \subseteq w_1 x w_2 \in V$ . If  $x$  occurs twice in  $t$ , then there is an edge from  $\text{dual}(u)$  to  $u$  in  $SG_\Phi$ . Since  $u \in V'$ , we have that  $u$  had to be a source at the time of his adding to  $V'$ , but that is not possible, as  $\text{dual}(u)$  can be only removed while we add  $u$  to  $V'$ . We can therefore assume we have some other vertex  $u' = s' \subseteq w_1 x w_2 \in V$ . Because of the variable  $x$ , there is an edge in  $SG_\Phi$  from  $\text{dual}(u') = w_1 x w_2 \subseteq s'$  to  $u$  and from  $\text{dual}(u)$  to  $u'$ . Again,  $u$  had to be a source at the time of his adding to  $V'$ , so either  $u'$  or  $\text{dual}(u')$  had to be added to  $V'$  first. As there is an edge from  $\text{dual}(u)$  to  $u'$ ,  $u'$  cannot be added before  $u$ . However, adding  $\text{dual}(u')$  to  $V'$  would mean that  $u' \notin V$ , which is a contradiction.

**Condition (IG3)** Given directly from Line 6 of Algorithm 1.

**Condition (IG4)** Let  $\pi = v_1, v_2, \dots, v_k, v_1$  be a cycle in  $G_\Phi$ . We need to show that no node of  $\pi$  could have been added to  $V'$ . If that holds, all duals of nodes of  $\pi$  must be in  $G_\Phi$ . We can show this by showing a cyclical dependency between the nodes, i.e., to add  $v_i$  to  $V'$ , we had to add  $v_{i-1}$  to  $V'$  before (for  $i = 1$ , by  $i - 1$  we mean  $k$ ). If there is an edge  $(v_{i-1}, v_i)$  in  $SG_\Phi$ , then obviously,  $v_i$  can become source (and therefore be able to be added to  $V'$ ) only after  $v_{i-1}$  had been added to  $V'$ . We can therefore focus only on the case where  $(v_{i-1}, v_i)$  is in  $G_\Phi$  but not in  $SG_\Phi$ . This can only happen when the left-hand side of  $v_{i-1}$  and the right-hand side of  $v_i$  share variables, but only the *same* occurrences, i.e.,  $v_{i-1} = \text{dual}(v_i)$ . However, this means that by adding  $v_{i-1}$  or  $v_i$  to  $V'$ , we would remove the other one from  $G_\Phi$ , which is a contradiction with  $\pi$  being a cycle of  $G_\Phi$ .

To show that  $G_\Phi$  is the inclusion graph with the smallest possible number of vertices, we need to realize that for each node  $v$  of  $SG_\Phi$ , each inclusion graph must contain at least one of  $v$  and  $\text{dual}(v)$ . We can now divide nodes of  $SG_\Phi$  according to Lemma 9: those whose dual is reachable from some cycle and those whose dual is not. For the first group, all these nodes will be in every inclusion graph. For the second group, either  $v$  is added to  $V'$  during the run of the algorithm (which removes  $\text{dual}(v)$  from the graph) or  $\text{dual}(v)$  will eventually become trivial source SCC (and will be added to  $V'$ , removing  $v$ ). Either way,  $G_\Phi$  contains only one of  $v$  and  $\text{dual}(v)$ , which means it has the smallest possible number of vertices.

We now prove the second part of the theorem. Assume that  $SG_\Phi$  is acyclic. During the computation, we only *remove* vertices from  $SG_\Phi$ , which means that it always stays acyclic. Furthermore, acyclicity implies that we can always find some trivial source SCC on Line 2. Therefore, no inclusion of the resulting inclusion graph  $G_\Phi$  has its dual in  $G_\Phi$ , and hence, from Condition (IG4), we know that it must be acyclic. □

In Section 4, 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-freeness of string constraints [26] (although looking from the equation point of view, chain-free equations are incomparable with the fragment of quadratic equations, the chain-free fragment of equations, regular, length, and transducer constraints is the largest known decidable fragment involving all these extended string constraints). *Chain-free* constraints are defined as those where the splitting graph of [26] is acyclic. As

the following lemma shows, chain-free constraints without transducers can also be defined using simplified splitting graph  $SG_\Phi$ .

**Lemma 11** *A multi-equation system  $\Phi$  is chain-free if and only if  $SG_\Phi$  is acyclic.*

**Proof** Let  $G = (P, E, \text{var}, \text{con})$  be a splitting graph for  $\Phi$  as defined in [26]. For the following proof, it is enough to know that  $P$  contains a unique node (called position) for each occurrence of a variable in each equation (the equation  $x_1 \dots x_n = y_1 \dots y_m$  adds  $n+m$  nodes to  $P$ ) and that there is an edge from position  $p$  to  $p'$  if and only if there is an intermediate position  $p''$  where  $p'$  and  $p''$  are different positions of the same variable and  $p$  and  $p''$  are positions in the same equation but on opposite sides. We need to show that  $G$  contains cycle if and only if  $SG_\Phi$  contains cycle.

( $\Rightarrow$ ) Let  $\pi = p_1 p_2 \dots p_n$ , where  $p_1 = p_n$ , be a cycle of  $G$ . We define a mapping  $M$  from the positions of the cycle  $\pi$  to nodes of  $SG_\Phi$ , where position  $p$  of equation  $s = t$  is mapped in  $M$  to either  $s \subseteq t$  (if  $p$  is a position of  $t$ ) or to  $t \subseteq s$  (if  $p$  is a position in  $s$ ). We now prove that for each  $i$ ,  $1 \leq i < n$ , there is an edge from  $M(p_i)$  to  $M(p_{i+1})$  in  $SG_\Phi$  (which means that  $M(\pi) = M(p_1) \dots M(p_n)$  is a cycle of  $SG_\Phi$ ). Let  $M(p_i) = s_i \subseteq t_i$  ( $p_i$  is a position of  $t_i$ ) and  $M(p_{i+1}) = s_{i+1} \subseteq t_{i+1}$  ( $p_{i+1}$  is a position of  $t_{i+1}$ ). Because there is an edge from  $p_i$  to  $p_{i+1}$ , there must be a position  $q$  opposite of  $p_i$  ( $q$  is a position of  $s_i$ ) and  $p_{i+1}$  and  $q$  represent different occurrences of the same variable. From the definition of  $SG_\Phi$ , it immediately follows that there is an edge from  $M(p_i)$  to  $M(p_{i+1})$ .

( $\Leftarrow$ ) Let  $\pi = s_1 \subseteq t_1 \dots s_n \subseteq t_n$  be a cycle of  $SG_\Phi$  (the first and the last vertex of  $\pi$  are the same). We take  $\pi' = p_1 p_2 \dots p_n$ , where  $p_1 = p_n$  and for each  $i$ ,  $2 < i \leq n$ ,  $p_i$  is defined in the following way. From the definition of  $SG_\Phi$  and because there is a transition from  $s_{i-1} \subseteq t_{i-1}$  to  $s_i \subseteq t_i$ , the same variable  $x$  must occur both in  $s_{i-1}$  and  $t_i$  (and the occurrences are different). We take  $p_i$  as the position of the occurrence of  $x$  in  $t_i$  and let  $q$  be the position of the occurrence of  $x$  in  $s_{i-1}$ . We now have that  $q$  and  $p_i$  are different positions of the same variable  $x$ ,  $p_{i-1}$  is opposite of  $q$  ( $q$  is position of  $t_{i-1}$ ), hence there is a transition from  $p_{i-1}$  to  $p_i$  in  $G$ . Therefore,  $\pi'$  is a cycle of  $G$ .  $\square$

**Theorem 12** *A multi-equation system  $\Phi$  is chain-free if and only if there exists an acyclic inclusion graph for  $\Phi$ .*

**Proof** ( $\Rightarrow$ ): If  $\Phi$  is chain-free, then by Lemma 11, the corresponding simplified splitting graph  $SG_\Phi$  is acyclic. From Theorem 10,  $\text{incl}(\mathcal{E})$  is then acyclic inclusion graph.

( $\Leftarrow$ ): If  $\Phi$  is not chain-free, then the corresponding simplified splitting graph  $SG_\Phi$  contains a cycle  $\pi = v_1, v_2, \dots, v_n, v_1$ . It is easy to see that if there is an edge  $(v, v')$  in  $SG_\Phi$ , then there is also an edge  $(\text{dual}(v'), \text{dual}(v))$  in  $SG_\Phi$ . This means that there is also a cycle  $\text{dual}(\pi) = \text{dual}(v_1), \text{dual}(v_n), \dots, \text{dual}(v_1)$  in  $SG_\Phi$ . From Lemma 11 we have that  $v_1, \dots, v_n$  are nodes of every inclusion graph for  $\Phi$  and because the condition for edges of  $SG_\Phi$  is a stricter version of Condition (IG3), every inclusion graph must contain the cycle  $\pi$ .  $\square$

**Corollary 13** *A multi-equation system  $\Phi$  is chain-free if and only if  $\text{incl}(\Phi)$  is acyclic.*

**Proof** From Theorems 10 and 12 and Lemma 11.  $\square$

## 4 Algorithm for satisfiability checking

Our algorithm for testing satisfiability of a multi-equation system  $\Phi$  is based on Theorem 7. The algorithm first constructs a suitable inclusion graph of  $\mathcal{E}$  using Algorithm 1 and then it

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

### 4.1 Overview

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

$$xyx = zu \quad \wedge \quad ww = xa \quad \wedge \quad u \in (baba)^*a \quad \wedge \quad z \in a(ba)^* \tag{3}$$

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  $\text{Lang}(x)$ . In the example, we have  $\text{Lang}(u) = (baba)^*a$ ,  $\text{Lang}(z) = a(ba)^*$  and, implicitly,  $\text{Lang}(x) = \text{Lang}(y) = \text{Lang}(w) = \Sigma^*$ .

The equation  $xyx = zu$  enforces that any solution, an assignment  $v$  of strings to variables, satisfies that the string  $s = v(x) \cdot v(y) \cdot v(x) = v(z) \cdot v(u)$  belongs to the intersection of the concatenations of languages on the left and the right-hand side of the equation,  $\text{Lang}(x) \cdot \text{Lang}(y) \cdot \text{Lang}(x) \cap \text{Lang}(z) \cdot \text{Lang}(u)$ , as in (4) below:

$$s \in \overbrace{\Sigma^*}^x \overbrace{\Sigma^*}^y \overbrace{\Sigma^*}^x = \overbrace{a(ba)^*}^z \overbrace{(baba)^*a}^u \tag{4}$$

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  $\text{Lang}(x_i)$  is represented by the automaton  $\text{Aut}(x_i)$ . The main steps of the refinement are shown in Fig. 3. First, we construct automata for the two sides of the equation:

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

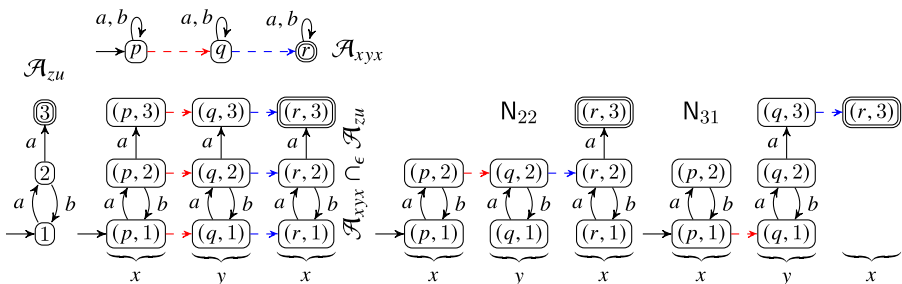


Fig. 3 Automata constructions within the refinement. Dashed lines represent  $\epsilon$

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

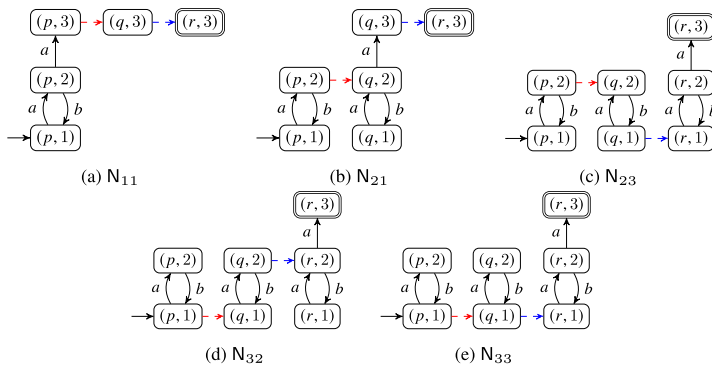
Note that for refining the languages of  $x$  and  $y$  on the left, we do not need to see the borders between  $z$  and  $u$  on the right. The  $\epsilon$ -transitions can hence be eliminated from  $\mathcal{A}_{zu}$  and it can be minimized. In our particular case, this gives much smaller automaton than the one obtained by connecting  $\text{Aut}(z)$  and  $\text{Aut}(u)$  (representing  $a(ba)^*$  and  $(baba)^*a$ , respectively).

To extract the new languages for  $x$  and  $y$  from  $\mathcal{A}_{xyx} \cap_{\epsilon} \mathcal{A}_{zu}$ , we decompose the automata to a disjunction of several *noodles* of 3 segments. Each noodle represents a concatenation of languages  $L_1^x \epsilon L^y \epsilon L_2^x$ , and is obtained by choosing one  $\epsilon$ -transition separating the first occurrence of  $x$  from  $y$  (the left column of red  $\epsilon$ -transitions in Fig. 3), one  $\epsilon$ -transition separating  $y$  from the second occurrence of  $x$  (the right column of blue  $\epsilon$ -transitions), removing the other  $\epsilon$ -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. 3, both first occurrences of  $x$  and  $y$  can have, among others, values  $aa$  and  $\epsilon$ , but if  $x = aa$  then  $y$  must be  $\epsilon$ ).

Figure 3 shows two noodles,  $N_{22}$  and  $N_{31}$ , out of 9 noodles that would be generated from  $\mathcal{A}_{xyx} \cap_{\epsilon} \mathcal{A}_{zu}$  (the notation  $N_{ij}$  indicates the chosen red and blue epsilon transition, respectively). For each of the 9 noodles, 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  $\epsilon$ -transitions in the noodle). The refined language for  $y$  is then  $\text{Lang}(y) = L^y$ . The refined language for  $x$  is obtained by unifying the languages of the first and the second occurrence of  $x$ ,  $\text{Lang}(x) = L_1^x \cap L_2^x$  (by constructing a standard product of the two automata):

- For  $N_{22}$ , the refinement is  $y \in (ba)^*$  and  $x \in a$  (computed as  $a(ba)^* \cap (ba)^*a$ ).
- For  $N_{31}$ , the refinement is  $y \in a(ba)^*a$  and  $x \in \epsilon$  (computed as  $(ab)^* \cap \epsilon$ ).

The 7 remaining noodles generated from  $\mathcal{A}_{xyx} \cap_{\epsilon} \mathcal{A}_{zu}$  shown in Fig. 4 yield  $x \in \emptyset$  and are discarded. Noodles  $N_{22}$  and  $N_{31}$  spawn two disjunctive branches of the computation.



**Fig. 4** Remaining noodles, leading to empty language for  $x$ . Useless states are trimmed, so noodles  $N_{12}$  and  $N_{13}$ , that do not have useful states, are not shown

**Algorithm 2**  $\text{refine}(v, \text{Aut})$

```

Input: A vertex  $v = s \sqsubseteq t$  with  $s = x_1 \cdots x_n$  and  $t = y_1 \cdots y_m$ 
          An automata assignment  $\text{Aut}$ 
Output: A tight refinement of  $\text{Aut}$  w.r.t.  $v$ 
1 Product :=  $\text{Aut}(s) \cap_{\epsilon} \text{minimize}(\text{Aut}(t))$ 
2 Noodles :=  $\text{noodlify}(\text{Product})$ 
3  $T := \emptyset$ 
4 for  $N \in \text{Noodles}$  do
5    $\text{Aut}' := \text{Aut}$ 
6   for  $1 \leq i \leq n$  do
7      $\text{Aut}'(x_i) := \bigcap \{N(j) \mid 1 \leq j \leq n, x_i = x_j\}$ 
8     if  $L(\text{Aut}'(s)) = \emptyset$  then continue
9      $T := T \cup \{\text{Aut}'\}$ 
10 return  $T$ 

```

For the branch of  $N_{22}$ , we use the equation  $ww = xa$  for the next refinement. Using the newly derived constraint  $x \in a$ , we obtain:

$$s \in \overbrace{\Sigma^*}^w \overbrace{\Sigma^*}^w \cap \overbrace{a}^x a. \tag{5}$$

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:

$$\overbrace{a}^x \overbrace{(ba)^*}^y \overbrace{a}^x = \overbrace{a(ba)^*}^z \overbrace{(baba)^*a}^u \quad \text{and} \quad \overbrace{a}^w \overbrace{a}^w = \overbrace{a}^x a. \tag{6}$$

We therefore found a stable language assignment, therefore, a solution is guaranteed to exist (see Theorems 1 and 7). We can thus conclude with SAT.

**4.2 Refining language assignments by noodlification**

The task of a refinement step is to create a new language assignment that refines the old one,  $\text{Lang}$ , and satisfies one of the inclusions previously not satisfied, say  $s \sqsubseteq t$ . 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  $T$  of refinements of  $\text{Lang}$  that is *tight w.r.t.*  $s \sqsubseteq t$ , that is, every solution of  $s = t$  under  $\text{Lang}$  is also a solution of  $s = t$  under some of its refinements in  $T$ .

Algorithm 2 computes such a tight set. Line 1 computes the automaton *Product*, which accepts  $\text{Lang}(s) \cap \text{Lang}(t)$ . 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  $\epsilon$ -transitions. That is, when  $\epsilon$  is understood as a special letter, *Product* accepts the *delimited language*  $L^\epsilon(\text{Product})$  of words  $w_1 \epsilon \cdots \epsilon w_n$  with  $w_i \in \text{Lang}(x_i)$  for  $1 \leq i \leq n$  and  $w_1 \cdots w_n \in \text{Lang}(t)$ . Notice that  $\text{Aut}(t)$  is minimized on Line 1. This means removal of  $\epsilon$ -transitions marking the borders of variables' occurrences, and then minimization by any automata size reduction method (we use simulation quotient [66, 67]). Since the product is then representing only the borders of the variables on the left (because  $\text{Aut}(s)$  keeps the  $\epsilon$ -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.

We then extract from *Product* a language for each occurrence of a variable in *s*. Line 2 divides *Product* into a set of *noodles* of *n* segments that preserve the delimited language in the sense that  $\bigcup_{N \in \text{Noodles}} L^\epsilon(N(1) \circ_\epsilon \dots \circ_\epsilon N(n)) = L^\epsilon(\text{Product})$ .

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  $q_1 \dashv\epsilon \triangleright r_1, \dots, q_{n-1} \dashv\epsilon \triangleright r_{n-1}$  of transitions that appear, in that order, in an accepting run of *Product* (note that every accepting run has  $n - 1$   $\epsilon$ -transitions by construction of *Product*, since  $\text{Aut}(s)$  also had  $n - 1$   $\epsilon$ -transitions in each accepting run and *minimize(Aut(*t*))* is  $\epsilon$ -free): for each  $1 \leq i \leq n$ ,  $N(i)$  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  $\text{Aut}'$  in  $\mathcal{T}$  by unifying/intersecting languages of different occurrences of the same variable: for each  $x \in \mathbb{X}$ ,  $\text{Aut}'(x)$  is the automata intersection of all automata  $N(i)$  with  $x_i = x$ . The fact that  $\mathcal{T}$  is a tight set of refinements (i.e., that it preserves all solutions of  $\text{Aut}$ ) follows from that every path of *Product* can be found in *Noodles* and that the use of  $\epsilon$ -transitions allows us to reconstruct the NFAs corresponding to the variables.

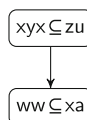
**Example 3** Consider the multi-equation system  $\Phi : xyx = zu \wedge ww = xa \wedge u \in (baba)^*a \wedge z \in a(ba)^*$  from Section 4.1 and the vertex  $xyx \subseteq zu$  of its inclusion graph given in Fig. 5. The construction of the product automaton *Product* from Algorithm 2 and the set of noodles `noodlify(Product)` =  $\{N_{11}, \dots, N_{33}\}$  are shown in Figs. 3 and 4. On Line 6, we need to compute intersections of  $N_{ij}(1) \cap N_{ij}(3)$  for each noodle  $N_{ij}$ . These parts of the noodle correspond to the two occurrences of the same variable *x*. The only noodles yielding nonempty languages for *x* are  $N_{22}$  and  $N_{31}$ . The noodle  $N_{22}$  leads to a refinement  $\text{Aut}_{22}$  of  $\text{Aut}$  where  $L(\text{Aut}_{22}(x)) = N_{22}(1) \cap N_{22}(3) = a(ba)^* \cap (ba)^*a = a$ . The noodle  $N_{31}$  leads to a refinement  $\text{Aut}_{31}$  of  $\text{Aut}$  where  $L(\text{Aut}_{31}(x)) = N_{31}(1) \cap N_{31}(3) = (ab)^* \cap \epsilon = \epsilon$ .

### 4.3 Satisfiability checking by refinement propagation

We introduce two versions of an algorithm for checking satisfiability of multi-equation system  $\Phi$ . The first one, `propagate(Φ)`, uses normal stability of inclusion graphs (Theorem 7) while the second one, `propagate_min(Φ)`, uses min-stability (Theorem 8). The pseudocode of `propagate(Φ)` is given in Algorithm 3, while the version `propagate_min(Φ)` is obtained from it by replacing Line 9 by the comment on Line 8. In this section we focus on `propagate(Φ)`.

The algorithm starts with the inclusion graph  $G_\Phi = (V, E)$  computed using Algorithm 1 and the automaton assignment  $\text{Aut}_\Phi$  corresponding to  $\text{Lang}_\Phi$ . It then uses graph nodes  $s \subseteq t$  not satisfied in the current  $\text{Aut}$  to refine it, that is, to replace  $\text{Aut}$  by some automaton assignment returned by `refine(s ⊆ t, Aut)`.

The algorithm maintains the current value of  $\text{Aut}$  and a worklist *W* of nodes for which the 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



**Fig. 5** The inclusion graph for  $xyx = zu \wedge ww = xa \wedge u \in (baba)^*a \wedge z \in a(ba)^*$

**Algorithm 3**  $\text{propagate}(\Phi) / \text{propagate}_{\min}(\Phi)$

```

Input: A multi-equation system  $\Phi$ 
Output: SAT if  $\Phi$  is satisfiable
           UNSAT if  $\Phi$  is unsatisfiable
1  $G_\Phi := \text{incl}(\Phi)$  with  $G_\Phi = (V, E)$ 
2  $\text{Aut}_\Phi := \{x \mapsto \text{NFA accepting } \text{Lang}_\Phi(x) \mid x \in \mathbb{X}\}$ 
   //  $V$  are ordered compatible with a topological order of the SCCs
3  $\text{Branches} := \langle (\text{Aut}_\Phi, \text{toposort}(V)) \rangle$ 
4 while  $\text{Branches} \neq \emptyset$  do
5    $(\text{Aut}, W) := \text{Branches.dequeue}()$ 
6   if  $W = \emptyset$  then return SAT
7    $v = s \subseteq t := W.dequeue()$ 
8   // if  $L^{\min} \text{Aut}(s) \subseteq L(\text{Aut}(t))$  and  $v$  is in non-trivial SCC of  $G_\Phi$  then
9   if  $L(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$  then
10    |  $\text{Branches.enqueue}((\text{Aut}, W))$ 
11  else
12    |  $\mathcal{T} := \text{refine}(v, \text{Aut})$ 
13    |  $W' := W$ 
14    | foreach  $(v, u) \in E$  s.t.  $u \notin W$  do
15    | |  $W'.enqueue(u)$ 
16    | foreach  $\text{Aut}' \in \mathcal{T}$  do
17    | |  $\text{Branches.enqueue}((\text{Aut}', W'))$ 
18 return UNSAT

```

is found not satisfied in the current automata assignment  $\text{Aut}$ , the node is used to refine it. Stability is detected when  $W$  is empty—there in no potentially unsatisfied inclusion.

Since  $\text{refine}(s \subseteq t, \text{Aut})$  does not return a single language assignment that refine  $\text{Aut}$  but a set of language assignments, the computation spawns an independent branch for each of them. Algorithm 3 schedules the branches for processing in the queue  $\text{Branches}$ . The branching is disjunctive, meaning SAT 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 both first-in first-out (this is important for showing termination of  $\text{propagate}_{\min}(\Phi)$  in Theorem 17). To minimize the number of refinement steps, the nodes are initially inserted in  $W$  in an order compatible with a topological order of the SCCs (this is also important for showing the soundness of  $\text{propagate}_{\min}(\Phi)$  in Theorem 16).

**Example 4** Consider again the multi-equation system  $\Phi$  from Section 4.1 and the inclusion graph in Example 3. The initial automata assignment  $\text{Aut}_\Phi$  is then given as  $L(\text{Aut}_\Phi(a)) = \{a\}$ ,  $L(\text{Aut}_\Phi(z)) = a(ba)^*$ ,  $L(\text{Aut}_\Phi(u)) = (baba)^*a$ , and, for the rest,  $L(\text{Aut}_\Phi(x)) = L(\text{Aut}_\Phi(y)) = L(\text{Aut}_\Phi(w)) = \Sigma^*$ . The queue  $\text{Branches}$  on Line 3 of Algorithm 3 is hence initialized as  $\text{Branches} = \langle (\text{Aut}_\Phi, \langle xyx \subseteq zu, ww \subseteq xa \rangle) \rangle$ . The computation of the main loop of Algorithm 3 then proceeds as follows.

**1st iteration.** The dequeued element is  $(\text{Aut}_\Phi, \langle xyx \subseteq zu, ww \subseteq xa \rangle)$  and  $v$  (dequeued from  $W$ ) is  $xyx \subseteq zu$ . The condition on Line 9 is not satisfied ( $\Sigma^* \cdot \Sigma^* \cdot \Sigma^* \not\subseteq a(ba)^* \cdot (baba)^*a$ ), hence the algorithm calls  $\text{refine}(xyx \subseteq zu, \text{Aut}_\Phi)$ . The refinement yields two new automata assignments,  $\text{Aut}_{22}, \text{Aut}_{31}$  which are defined in Example 3. The queue  $\text{Branches}$  is hence extended to  $\langle (\text{Aut}_{22}, \langle ww \subseteq xa \rangle), (\text{Aut}_{31}, \langle ww \subseteq xa \rangle) \rangle$ .

- 2nd iteration.** The dequeued element is  $(\text{Aut}_{31}, \langle \text{ww} \subseteq \text{xa} \rangle)$ . The condition on Line 9 is not satisfied since  $L(\text{Aut}_{31}(x)) = \{\epsilon\}$  and  $L(\text{Aut}_{31}(w)) = \Sigma^*$ . In this case,  $\text{refine}(\text{ww} \subseteq \text{xa}, \text{Aut}_{31}) = \emptyset$  and  $\text{Branches} = \langle (\text{Aut}_{22}, \langle \text{ww} \subseteq \text{xa} \rangle) \rangle$ .
- 3rd iteration.** The dequeued element is  $(\text{Aut}_{22}, \langle \text{ww} \subseteq \text{xa} \rangle)$ . The condition on Line 9 is not satisfied ( $\Sigma^* \cdot \Sigma^* \not\subseteq a \cdot a$ ) and  $\text{refine}(\text{ww} \subseteq \text{xa}, \text{Aut}_{22}) = \{\text{Aut}_{221}\}$  where  $\text{Aut}_{221}$  is as  $\text{Aut}_{22}$  except that  $\text{Aut}_{221}(w)$  accepts only  $a$ .  $\text{Branches}$  is then updated to  $\langle (\text{Aut}_{221}, \emptyset) \rangle$ .
- 4th iteration.** The condition on Line 6 is satisfied and the algorithm returns SAT.

**Example 5** Consider for instance the system  $xy = x \wedge x \in a^+ \wedge y \in a$ . The inclusion graph (actually the only one possible) is shown in Fig. 6. In the initial automata assignment  $\text{Aut}_\Phi$  we have  $L(\text{Aut}_\Phi(x)) = a^+$  and  $L(\text{Aut}_\Phi(y)) = \{a\}$ . The queue  $\text{Branches}$  on Line 3 of Algorithm 3 is initialized as  $\text{Branches} = \langle (\text{Aut}_\Phi, \langle x \subseteq xy, xy \subseteq x \rangle) \rangle$ . The computation then looks as follows:

- 1st iteration.** The inclusion check on Line 9 is not satisfied, hence the algorithm calls  $\text{refine}(x \subseteq xy, \text{Aut}_\Phi)$ . The refinement yields a new automata assignment  $\text{Aut}_1$  refining  $\text{Aut}_\Phi$  with  $\text{Aut}_1(x) = a^+a$ . The queue  $\text{Branches}$  is updated to  $\langle (\text{Aut}_1, \langle x \subseteq xy, xy \subseteq x \rangle) \rangle$ .
- 2nd iteration.** The inclusion check is satisfied for  $xy \subseteq x$ , hence the queue  $\text{Branches}$  is updated to  $\langle (\text{Aut}_1, \langle x \subseteq xy \rangle) \rangle$ .
- 3rd iteration.** The inclusion check is not satisfied, hence  $\text{refine}(x \subseteq xy, \text{Aut}_1)$  yields a new automata assignment  $\text{Aut}_2$  refining  $\text{Aut}_1$  with  $\text{Aut}_2(x) = a^+a^2$ . The queue  $\text{Branches}$  is then given as  $\langle (\text{Aut}_2, \langle x \subseteq xy, xy \subseteq x \rangle) \rangle$ .
- 4th iteration.** The inclusion check is satisfied for  $xy \subseteq x$ , hence the queue  $\text{Branches}$  is updated to  $\langle (\text{Aut}_2, \langle x \subseteq xy \rangle) \rangle$ .

...

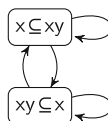
It is evident that Algorithm 3 does not terminate on this case (which is clearly unsatisfiable), since the refined automata assignments for  $x$  reach  $a^+a^n$  for all  $n \in \mathbb{N}$ . Note that many similar examples could be handled by simple heuristics that take into account lengths of strings, already used in other solvers. For example, we could easily deduce from  $xy = x$  that  $y$  is an empty string and immediately return UNSAT as it clashes with  $y \in a$ .

We now prove that the algorithm is sound in the general case (an answer is always correct) and it is complete for the chain-free fragment.

**Theorem 14** (Soundness) *If  $\text{propagate}(\Phi)$  returns SAT, then  $\Phi$  is satisfiable, and if  $\text{propagate}(\Phi)$  returns UNSAT,  $\Phi$  is unsatisfiable.*

**Proof** We prove by induction the following invariant of the algorithm that need to hold in every iteration of the main loop: for each  $(\text{Aut}, W) \in \text{Branches}$  and  $v = s \subseteq t \in V \setminus W$  it holds that  $L(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$ .

Base case: The first element of  $\text{Branches}$  on Line 3 trivially satisfies the invariant.



**Fig. 6** The inclusion graph for  $xy = x \wedge x \in a^+ \wedge y \in a$

Induction step: Assume that the invariant holds for all  $(\text{Aut}, W)$  in *Branches*. We prove that it is still valid after an iteration of the main loop. Consider  $(\text{Aut}, W)$  on Line 5 and  $v = s \sqsubseteq t$  on Line 7. If the condition on Line 9 holds then for  $(\text{Aut}, W \setminus \{v\})$  the invariant clearly holds. We proceed with the case the condition is not fulfilled. First, assume the case that  $s$  and  $t$  do not share a variable. Then, from the property of `refine`, we have that for each  $\text{Aut}' \in \mathcal{T}$ :  $L(\text{Aut}'(s)) \subseteq L(\text{Aut}'(t))$ , therefore,  $v$  need not be included in  $W'$ . From the Condition (IG3) of the inclusion graph, we have that only successors of  $v$  in  $G_\Phi$  might be affected by the refinement of  $s$  (we changed the assignment to variables in  $s$ , so the languages on the right-hand sides of successors of  $v$  might have changed). Hence, the invariant holds for each  $(\text{Aut}', W')$  where  $\text{Aut}' \in \mathcal{T}$ . For the case that  $s$  and  $t$  share a variable, the reasoning is the same as in the previous case except that the inclusion  $L(\text{Aut}'(s)) \subseteq L(\text{Aut}'(t))$  might not be true in general. However, again from Condition (IG3) we get that  $v$  has a self-loop in  $G_\Phi$  and hence it is included to  $W'$  as well.

Based on the invariant, if the algorithm returns SAT, we have that the  $G_\Phi$  is stable w.r.t.  $\text{Aut}$  (Line 6). From Theorem 7 we then obtain that  $\Phi$  is satisfiable. Further, we know that `refine` preserves solutions. Therefore, for each solution  $\nu$  of  $\Phi$ , there is some  $(\text{Aut}'', W'') \in \text{Branches}$  s.t.  $\nu(x) \in L(\text{Aut}''(x))$  for each variable  $x$ . Hence, if the algorithm returns UNSAT,  $\text{Branches} = \emptyset$ , which means that  $\Phi$  is unsatisfiable. □

**Theorem 15** (Termination) *If  $\Phi$  is chain-free, then `propagate`( $\Phi$ ) terminates.*

**Proof** In the following proof, by *successors* of  $(\text{Aut}, W)$  we mean all pairs  $(\text{Aut}', W')$  that were added to *Branches* during processing of  $(\text{Aut}, W)$  in the main loop of `propagate`( $\Phi$ ) (Lines 10 and 17). The computation of the algorithm can then be seen as a (possibly) infinite tree whose vertices are labelled by items from *Branches*. We will show that for acyclic inclusion graph  $G_\Phi = (V, E)$ , this tree is finite, which means that `propagate`( $\Phi$ ) terminates. According to Corollary 13, this is enough to prove the theorem.

We first define a partial order  $\preceq$  on the (finite) set of subsets of  $V$  and we show that for each successor  $(\text{Aut}', W')$  of  $(\text{Aut}, W)$ ,  $W'$  is strictly larger than  $W$  in it, i.e.,  $W \prec W'$ . Because  $G_\Phi$  is acyclic, the ordering `toposort`( $V$ ) on Line 3 is a topological ordering of *vertices* and not just an ordering compatible with the ordering of SCCs. This means that for edge  $(u, v)$  on Line 14,  $v$  is greater than  $u$ . For  $W_1, W_2 \subseteq V$ , we then define  $W_1 \preceq W_2$  if and only if  $W_1 = W_2$  or there is  $v_k$  with  $v_k \in W_1, v_k \notin W_2$  and  $W_1 \cap \{v_1, \dots, v_{k-1}\} = W_2 \cap \{v_1, \dots, v_{k-1}\}$ . For  $W$  and  $W'$ , such  $v_k$  is the vertex  $v$  from Line 7. This is because  $W'$  is equal to  $W \setminus \{v\}$  with some nodes possibly added on Line 15, which are however all greater than  $v$  in the topological ordering. Hence,  $W \prec W'$  and because  $\preceq$  is defined on a finite set, the computation tree must be finite. □

#### 4.4 Working with the shortest words

By using min-stability of inclusion graphs, the algorithm `propagate`( $\Phi$ ) can be improved with a weaker termination condition that takes into account only the shortest words in the languages assigned to variables. Such a condition is potentially cheaper (we work with the smaller automaton containing only the shortest words instead of the full automaton) and more importantly, it terminates sooner, giving us completeness in the SAT case for general constraints, i.e., the algorithm is always guaranteed to return SAT if a solution exists.

Our target is to get an automata assignment  $\text{Aut}$  that fulfills the conditions of min-stability of inclusion graphs, i.e., for  $v = s \subseteq t \in V$

- if  $v$  forms trivial SCC of  $G_\Phi$ , then  $L(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$ ,
- if  $v$  is in non-trivial SCC of  $G_\Phi$  then  $L^{\min}(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$ .

This results in the algorithm  $\text{propagate}_{\min}(\Phi)$ , where the condition  $L(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$  on Line 9 of Algorithm 3 is changed to  $L^{\min}(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$  (we use  $L^{\min}(A)$  as a shorthand for  $(L(A))^{\min}$ ). However, for vertices  $v$  that form trivial SCCs of  $G_\Phi$ , this condition is not sufficient, we need the full stability  $L(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$  to hold. Therefore, the condition is checked only for vertices that are in non-trivial SCCs. For vertices that form trivial SCCs, we continue with the refinement step, which results in automata assignments in which the inclusion  $L(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$  holds.

**Theorem 16** (Soundness) *If  $\text{propagate}_{\min}(\Phi)$  returns SAT, then  $\Phi$  is satisfiable, and if  $\text{propagate}_{\min}(\Phi)$  returns UNSAT,  $\Phi$  is unsatisfiable.*

**Proof** The proof is very similar to the proof of Theorem 14, but we want to show an invariant that for each  $(\text{Aut}, W) \in \text{Branches}$ , the min-stability holds for each  $v \in V \setminus W$ . However, we prove a slightly weaker invariant of the main loop, which can still be used to show that theorem holds: for each  $(\text{Aut}, W) \in \text{Branches}$  and  $v = s \subseteq t \in V \setminus W$ , it holds that

- if  $v$  forms trivial SCC of  $G_\Phi$ , then  $L(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$ ,
- if  $v$  is in non-trivial SCC of  $G_\Phi$ , then either  $L^{\min}(\text{Aut}(s)) \subseteq L(\text{Aut}(t))$ , or both  $\text{dual}(v) = t \subseteq s \in W$  and  $L^{\min}(\text{Aut}(t)) \not\subseteq L(\text{Aut}(s))$ .

Base case: The first element of  $\text{Branches}$  on Line 3 trivially satisfies the invariant.

Induction step: Assume that the invariant holds for all  $(\text{Aut}, W)$  in  $\text{Branches}$ . We prove that it is still valid after an iteration of the main loop. Consider  $(\text{Aut}, W)$  on Line 5 and  $v = s \subseteq t$  on Line 7. If the condition on Line 8 holds, then we know from the invariant that  $L^{\min}(\text{Aut}(t)) \subseteq L(\text{Aut}(s))$  holds for the dual of  $v$ , so the invariant clearly holds for  $(\text{Aut}, W \setminus \{v\})$ . We proceed with the case the condition is not fulfilled. For each  $\text{Aut}' \in \mathcal{T}$ , we know that  $\text{Aut}'$  and  $\text{Aut}$  differ only at the variables from the left-hand side  $s$  (more specifically,  $L(\text{Aut}'(s)) \subset L(\text{Aut}(s))$ ) and  $W'$  contains all vertices whose right-hand side share some variable with  $s$  (which also includes its dual, if it is a vertex of  $G_\Phi$ ). Then the only way that the invariant does not hold for  $(\text{Aut}', W')$  is if there is some different vertex  $v' = s' \subseteq t' \notin W'$  that is in non-trivial SCC and shares with  $s$  only variables from  $s'$  (for trivial SCCs, refining left-hand side has no impact on stability),  $L^{\min}(\text{Aut}'(s')) \not\subseteq L(\text{Aut}'(t'))$ , and  $L^{\min}(\text{Aut}'(t')) \subseteq L(\text{Aut}'(s'))$  ( $\text{dual}(v')$  must belong to  $W'$ , as right-hand side of  $v'$  shares a variable with the left-hand side of  $v$ ). For the sake of contradiction, assume that such  $v'$  exists. We know that  $s$  and  $t'$  do not share a variable, so  $L(\text{Aut}'(t')) = L(\text{Aut}(t'))$ . Furthermore, because  $v' \notin W$ , from the invariant, either  $L^{\min}(\text{Aut}(s')) \subseteq L(\text{Aut}(t'))$  or both  $\text{dual}(v') = t' \subseteq s' \in W$  and  $L^{\min}(\text{Aut}(t')) \not\subseteq L(\text{Aut}(s'))$ . We have  $L^{\min}(\text{Aut}(t')) = L^{\min}(\text{Aut}'(t')) \subseteq L(\text{Aut}'(s')) \subseteq L(\text{Aut}(s'))$ , so the second case is not possible, and it also means for the first case that  $L^{\min}(\text{Aut}(s')) = L^{\min}(\text{Aut}(t'))$  (as shown in the proof of Theorem 8). We have  $L^{\min}(\text{Aut}(s')) \subseteq L(\text{Aut}(t'))$  and  $L^{\min}(\text{Aut}'(s')) \not\subseteq L(\text{Aut}'(t')) = L(\text{Aut}(t'))$ , and as  $\text{Aut}'$  is refinement of  $\text{Aut}$ , this means that the words from  $L^{\min}(\text{Aut}'(s'))$  must be longer

than those in  $L^{\min}(\text{Aut}(s'))$ , so  $L^{\min}(\text{Aut}(s')) \not\subseteq L(\text{Aut}'s')$ . However,  $L^{\min}(\text{Aut}(s')) = L^{\min}(\text{Aut}(t')) = L^{\min}(\text{Aut}'(t')) \subseteq L(\text{Aut}'(s'))$  which is a contradiction.

Based on the invariant, if the algorithm returns SAT, i.e.,  $W = \emptyset$ , we know that the  $G_\Phi$  is min-stable w.r.t. Aut (Line 6). From Theorem 8, we obtain that  $\Phi$  is satisfiable. Further, because refine preserves solutions, we can be sure that if the algorithm returns UNSAT, then  $\Phi$  is unsatisfiable. □

Intuitively, the algorithm explores the words in the languages of the variables systematically, from the shortest to the longest. Hence, it is not hard to see that for SAT cases, the algorithm terminates:

**Theorem 17 (Termination)** *If  $\Phi$  is chain-free or satisfiable, then  $\text{propagate}_{\min}(\Phi)$  terminates.*

**Proof** Let  $\Phi$  be chain-free or satisfiable. For the sake of contradiction, assume that  $\text{propagate}_{\min}(\Phi)$  does not terminate. As *Branches* is FIFO, the computation tree (in the sense used in the proof of Theorem 15) is searched breadth-first, therefore, the non-termination means that there is no terminal node in the tree.

Let therefore  $\pi = (\text{Aut}_1, W_1), (\text{Aut}_2, W_2) \dots$  be one of the infinite paths in this tree. For each  $i \in \mathbb{N}$ ,  $W_i$  must not be empty, otherwise the algorithm would terminate with SAT. Therefore, there must be a vertex  $v = s \subseteq t$ , which, for infinitely many indices  $i$ , belongs to  $W_i$ . Furthermore, because each  $W_i$  is FIFO, there must be infinitely many indices  $i$  where  $v$  is at the front of  $W_i$ , i.e.,  $v$  is the processed vertex on Line 7 during the processing of  $(\text{Aut}_i, W_i)$ . We denote this set of indices as  $I = \{i \in \mathbb{N} \mid v \text{ is at the front of } W_i\}$ . For each  $i \in I$ ,  $v$  is removed from  $W_i$  during the processing of  $(\text{Aut}_i, W_i)$ , so there must be an index  $j, j \leq i$ , s.t. during the processing of  $(\text{Aut}_j, W_j)$ ,  $v$  is added to  $W_{j+1}$  on Line 15 and for all  $k \in I, i < k$ , we have  $j < k$ . This means that there is an edge from  $v_j$ , the vertex that is processed during the processing of  $(\text{Aut}_j, W_j)$ , to  $v$ . Let  $J$  be the set of all such indices  $j$  and  $V_J$  all such vertices  $v_j$ . Obviously,  $J$  is an infinite set, so there must be at least one vertex  $v' \in V_J$  s.t. the set of indices  $j$  where  $v'$  is at the front of  $W_j$  is infinite. We can follow this reasoning and find a vertex  $v''$  with a similar infinite set of indices and an edge to  $v'$ . Continuing this, we find a path of such vertices until we eventually reach  $v$  again, meaning that  $v$  is on a cycle. From this, we know that  $\Phi$  cannot be chain-free, therefore, it must be satisfiable, which means that it must have some solution  $\nu$ .

Because there is no terminal node in the computation tree and because refine preserves solutions,  $\nu$  must be preserved in each assignment of some infinite path of the computation tree. W.l.o.g., we can assume that  $\pi$  is this path. Now, for each  $j \in J$ , during the processing of  $(\text{Aut}_j, W_j)$  and  $v_j = s_j \subseteq t_j$ , we reached Line 15 to add  $v$  to  $W_j$ , therefore,  $L^{\min}(\text{Aut}_j(s_j)) \not\subseteq L(\text{Aut}(t_j))$  and  $\text{refine}(v_j, \text{Aut}_j)$  must have been called. This means that there must be a variable  $x$  in  $s_j$  whose language was refined in such a way that either  $L^{\min}(\text{Aut}_{j+1}(x)) \subset L^{\min}(\text{Aut}_j(x))$  or the length of the shortest words in  $L^{\min}(\text{Aut}_{j+1}(x))$  is larger than in  $L^{\min}(\text{Aut}_j(x))$ . Because  $J$  is infinite, there must be a variable for which this refinement happens infinitely many times. Assume this variable is  $x$ . We know that  $\pi$  preserves the solution  $\nu$ , therefore,  $\nu(x) \in L(\text{Aut}_i(x))$  for all  $i \in \mathbb{N}$ . However, we will eventually reach, for some index  $j \in J$ , the situation where  $L^{\min}(\text{Aut}_j(x)) = \{\nu(x)\}$ . At this point, the length of the shortest words in  $L^{\min}(\text{Aut}_{j+1}(x))$  must be larger than the length of  $\nu(x)$ , but that contradicts the fact that  $\nu(x) \in L(\text{Aut}_{j+1}(x))$ . □

## 5 Experimental evaluation

We implemented our algorithm in a prototype string solver called NOODLER [68] using Python and C++ automata library MATA [69] for manipulating NFAs. We compared the performance of NOODLER with a comprehensive selection of other tools, namely, CVC5 [13] (version 1.2.0), Z3 [15] (version 4.13.4), Z3STR3RE [20], Z3-TRAU [34], Z3-ALPHA [70] (SMT-COMP'24 version), and OSTRICH [23] (version 1.4). Compared to [60], we removed from the evaluation the tools Z3STR4 (because it is not available to download anymore), SLOTH (as during the rerun of the experiments it gave incorrect results for all benchmark sets), and RETRO (as we were unable to run it) from the evaluation. In order to have a meaningful comparison with compiled tools (CVC5, Z3, Z3STR3RE, Z3-TRAU, Z3-ALPHA), 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 the other interpreted tool OSTRICH (Java). 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 AMD Ryzen 5 5600G CPU @ 3.8 GHz with 100 GiB of RAM running Ubuntu 22.04.4. The timeout was set to 120 s, memory limit was 8 GiB (16 GiB for OSTRICH as it otherwise fails).

**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.
- **KALUZA-HARD** (897 formulae): it is obtained from the KALUZA benchmark [46] by taking hard formulae from its solution similarly as in PYEX-HARD.
- **STR 2** ([33], 292 formulae) the original benchmark from [33] contains 600 hand-crafted formulae including word equations and length constraints; the 308 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 [71] and STRANGER [39]. 80 of these formulae were removed as they 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  $\Sigma^*$  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 for PYEX-HARD where it is the second lowest. Furthermore, it is faster than other tools in PYEX-HARD and STR 2 and second fastest (after CVC5) in KALUZA-HARD and SLOG.

In Fig. 7, we provide scatter plots comparing the run times of NOODLER with the other tools on all benchmark. We can see that there is indeed a large number of benchmarks where

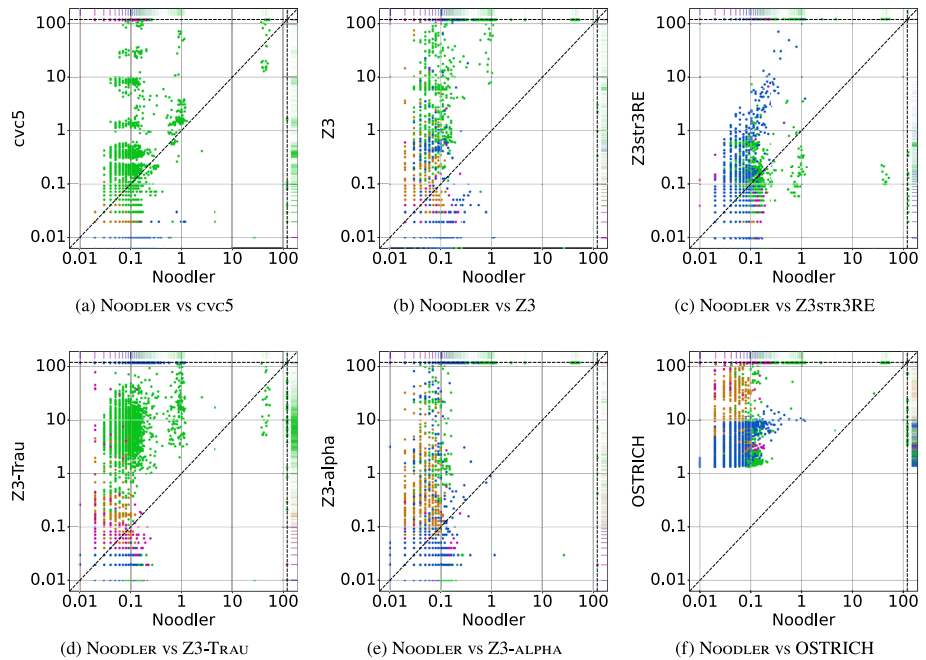
**Table 1** Results of experiments

	PYEX- HARD (20,023)			KALUZA- HARD (897)		
	T/Os	time	time–T/O	T/Os	time	time–T/O
NOODLER	40	<b>7,154</b>	2,354	<b>0</b>	43	43
CVC5	<b>36</b>	11,084	6,764	<b>0</b>	<b>9</b>	<b>9</b>
Z3	2,644	323,202	5,922	62	7,697	257
Z3STR3RE	795	95,755	<b>355</b>	10	1,209	<b>9</b>
Z3- TRAU	10	29,860	28,660	<b>0</b>	120	120
Z3- ALPHA	*3,058	369,702	2,742	231	28,650	930
OSTRICH	2,948	389,629	35,869	25	14,531	11,531

	STR 2 (292)			SLOG (1,896)		
	T/Os	time	time–T/O	T/Os	time	time–T/O
NOODLER	<b>2</b>	<b>254</b>	14	<b>0</b>	88	88
CVC5	92	11,041	<b>1</b>	<b>0</b>	<b>3</b>	<b>3</b>
Z3	121	14,541	21	15	2,137	337
Z3STR3RE	167	20,055	15	49	6,503	623
Z3- TRAU	3	724	363	745	89,424	24
Z3- ALPHA	*126	15,144	24	*79	10,771	1,291
OSTRICH	216	27,596	1,676	<b>0</b>	5,886	5,886

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”). Z3- ALPHA gives incorrect results for some benchmarks, marked with \*. Best values are in **bold**



**Fig. 7** The performance of NOODLER and other tools on all benchmarks: ● PYEX- HARD, ● KALUZA- HARD, ● STR 2, ● SLOG. Times are given in seconds, axes are logarithmic. Dashed lines represent timeouts (120 s)

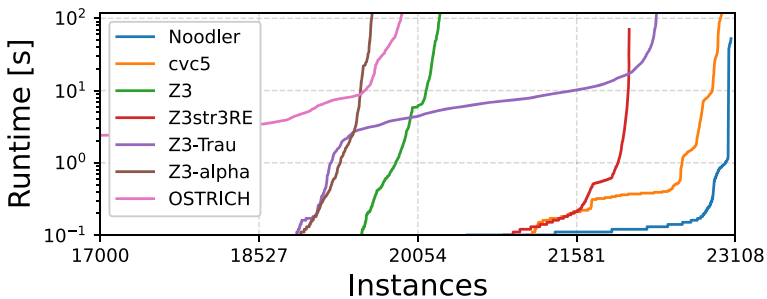
NOODLER is faster than its competitors (and that the performance of NOODLER is more stable, which may be caused by the heuristics in the other tools not always working well). Notice that NOODLER and CVC5 are complementary: they have both some timeouts, but each formula is solved by at least one of the tools. Moreover, in Fig. 8, we provide a cactus plot showing times needed to solve 3,108 most difficult formulae for the tools.

**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 or Z3. 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.

### 6 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 [26, 30, 31], TRAU [27–29, 34], OSTRICH [21–23], and Z3STR3RE [20]. The original algorithm first transforms equations to the disjunction of their solved forms [72] 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  $L_x \cap \text{Lang}(x)$  and  $L_y \cap \text{Lang}(y)$  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 optimizes 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 4.1), there would be 14 alignments/solved forms, e.g. those characterized 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



**Fig. 8** Times for solving the hardest 3,108 formulae for the tools

solved form  $z = z_1z_2$ ,  $u = u_1z_1$ ,  $x = z_1$ ,  $y = z_2u_1$ —automata splitting would generate 15 cases from  $z_1z_2 \in \text{Lang}(z)$  and  $u_1u_2 \in \text{Lang}(u)$ , 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 optimized (see Section 4.1).

Our algorithm has an advantage also over pure automata splitting, irrespective of aligning equations. For instance, consider the constraint  $xyx \in L \wedge x \in \text{Lang}(x) \wedge y \in \text{Lang}(y)$ . 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  $\text{Lang}(x) \cap L_x$  and  $\text{Lang}(y) \cap L_y$ . Our algorithm avoids generating much of these cases by intersecting with the languages of  $\text{Lang}(x)$  and  $\text{Lang}(y)$  early—the construction of  $\text{Lang}(x) \cdot \text{Lang}(y) \cdot \text{Lang}(x)$  prunes much of  $L$ 's automaton immediately. For instance, if  $L = (ab)^*a^+(abcd)^*$  (its NFA has 7 states) and  $\text{Lang}(x) = (a + b)^*$ , automata splitting explores  $7^2 = 49$  cases while our algorithm explores 9 (7 when optimized) of these cases—it would compute the same product and noodles as in Section 4.1, 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], Kepler<sub>22</sub> [33], StrSolve [37], Woopje [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] are extensions to constraint programming and use propagation. Constraint programming solvers can also be used with the MiniZinc modelling language [73]. 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 [33, 48], which are incomparable but of a smaller practical relevance (although some tools actually implement Nielsen's algorithm [59] 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 utilize 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.

## 7 Conclusion

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 characterization 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.

**Author Contributions** All authors contributed to the formulation of the core theoretical framework, including formulating and proving the core theorems and lemmas. F. Blahoudek, D. Chocholatý, V. Havlena, and J. Sîc contributed to the development of the early prototype and the final implementation. All authors participated in discussions, manuscript writing, and iterative revisions, ensuring the clarity and coherence of the final publication.

**Funding** Open access publishing supported by the institutions participating in the CzechELib Transformative Agreement. This work was supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, the Czech Science Foundation projects GA23-07565S, the FIT BUT internal project FIT-S-23-8151, and the project of Ministry of Science and Technology, Taiwan (grant no. 109-2628-E-001-001-MY3).  The work of David Chocholatý, Brno Ph.D. Talent Scholarship Holder, is funded by the Brno City Municipality.

**Data Availability** The source code of NOODLER can be found at <https://github.com/vhavlena/Noodler>. The benchmarks, the scripts, and the results of the evaluation can be found at <https://github.com/VeriFIT/smt-bench/tree/fmjournal> and <https://github.com/VeriFIT/smt-string-bench-results/tree/fmjournal>.

## Declarations

**Competing interests** The authors have no relevant financial or non-financial interests to disclose.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

1. OWASP, Top 10. [https://www.owasp.org/images/f/f8/OWASP\\_Top\\_10\\_-\\_2013.pdf](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. Hadarean, L. (2019). String solving at Amazon. <https://mosca19.github.io/program/index.html>. Presented at MOSCA'19.
5. Alt, L., Blicha, M., Hyvärinen, A. E. J., & Sharygina, N. (2022). SolCMC: Solidity compiler's model checker. In S. Shoham, & Y. Vizel (Eds.), *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I, Vol. 13371 of Lecture Notes in Computer Science* (pp. 325–338). Springer. [https://doi.org/10.1007/978-3-031-13185-1\\_16](https://doi.org/10.1007/978-3-031-13185-1_16)
6. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., & Deters, M. (2014). A DPLL(T) theory solver for a theory of strings and regular expressions. In A. Biere & R. Bloem (Eds.), *Computer Aided Verification* (pp. 646–662). Cham: Springer International Publishing.
7. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., & Deters, M. (2016). An efficient SMT solver for string constraints. *Formal Methods in System Design*, 48(3), 206–234.
8. Barrett, C. W., Tinelli, C., Deters, M., Liang, T., Reynolds, A., & Tsiskaridze, N. (2016). Efficient solving of string constraints for security analysis. In *HotSoS'16, ACM Transactions on Computational Logic* (pp. 4–6).

9. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., & Barrett, C. (2015). A decision procedure for regular membership and length constraints over unbounded strings. In *FroCoS'15, Vol. 9322 of LNCS* (pp. 135–150). Springer.
10. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., & Tinelli, C. (2017). Scaling up DPLL(T) string solvers using context-dependent simplification. In R. Majumdar & V. Kunčák (Eds.), *Computer Aided Verification* (pp. 453–474). Cham: Springer International Publishing.
11. Nötzli, A., Reynolds, A., Barbosa, H., Barrett, C., & Tinelli, C. (2022). Even faster conflicts and lazier reductions for string solvers. In S. Shoham & Y. Vizel (Eds.), *Computer Aided Verification* (pp. 205–226). Cham: Springer International Publishing.
12. Reynolds, A., Notzlit, A., Barrett, C., & Tinelli, C. (2020). Reductions for strings and regular expressions revisited. In *2020 Formal Methods in Computer Aided Design (FMCAD)* (pp. 225–235). [https://doi.org/10.34727/2020/isbn.978-3-85448-042-6\\_30](https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_30)
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. (2022). cvc5: A versatile and industrial-strength smt solver. In D. Fisman & G. Rosu (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 415–442). Cham: Springer International Publishing.
14. Bjørner, N., Tillmann, N., & Voronkov, A. (2009). Path feasibility analysis for string-manipulating programs. In *TACAS'09, Vol. 5505 of LNCS* (pp. 307–321). Springer.
15. de Moura, L., & Bjørner, N. (2008). Z3: An efficient smt solver. In C. R. Ramakrishnan & J. Rehof (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337–340). Berlin Heidelberg, Berlin, Heidelberg: Springer.
16. Zheng, Y., Zhang, X., & Ganesh, V. (2013). Z3-str: A Z3-based string solver for web application analysis. In *ESEC/FSE'13, ACM Transactions on Computational Logic* (pp. 114–124).
17. Berzish, M., Ganesh, V., & Zheng, Y. (2017). Z3str3: A string solver with theory-aware heuristics. In *Formal Methods in Computer Aided Design (FMCAD)* (pp. 55–59). <https://doi.org/10.23919/FMCAD.2017.8102241>
18. Berzish, M. (2021). Z3str4: A solver for theories over strings, Ph.D. thesis. <http://hdl.handle.net/10012/17102>
19. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., & Zhang, X. (2015). Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In D. Kroening & C. S. Păsăreanu (Eds.), *Computer Aided Verification* (pp. 235–254). Cham: Springer International Publishing.
20. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J. D., Nowotka, D., & Ganesh, V. (2021). An SMT solver for regular expressions and linear arithmetic over string length. In A. Silva, & K. R. M. Leino (Eds.), *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II, Vol. 12760 of Lecture Notes in Computer Science* (pp. 289–312). Springer. [https://doi.org/10.1007/978-3-030-81688-9\\_14](https://doi.org/10.1007/978-3-030-81688-9_14)
21. Lin, A. W., & Barceló, P. (2016). String solving with word equations and transducers: Towards a logic for analysing mutation XSS. In *POPL'16, ACM Transactions on Computational Logic*. (pp. 123–136).
22. Chen, T., Chen, Y., Hague, M., Lin, A. W., & Wu, Z. (2018). What is decidable about string constraints with the replaceall function. *Proceedings of ACM Programming Languages*, 2(POPL), 3:1-3:29. <https://doi.org/10.1145/3158091>
23. Chen, T., Hague, M., Lin, A. W., Rümmer, P., & Wu, Z. (2019). Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proceedings of ACM Programming Languages*, 3(POPL), 49:1–49:30. <https://doi.org/10.1145/3290362>
24. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A. W., Rümmer, P., & Wu, Z. (2022). Solving string constraints with regex-dependent functions through transducers with priorities and variables. *Proceedings of ACM Programming Languages*, 6(POPL), 1–31. <https://doi.org/10.1145/3498707>
25. Chen, T., Hague, M., He, J., Hu, D., Lin, A. W., Rümmer, P., & Wu, Z. (2020). A decision procedure for path feasibility of string manipulating programs with integer data type. In D. V. Hung, & O. Sokolsky (Eds.), *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings, Vol. 12302 of Lecture Notes in Computer Science* (pp. 325–342). Springer. [https://doi.org/10.1007/978-3-030-59152-6\\_18](https://doi.org/10.1007/978-3-030-59152-6_18)
26. Abdulla, P. A., Atig, M. F., Diep, B. P., Holík, L., & Janku, P. (2019). Chain-free string constraints. In Y. Chen, C. Cheng, & J. Esparza (Eds.), *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings, Vol. 11781 of Lecture Notes in Computer Science* (pp. 277–293). Springer. [https://doi.org/10.1007/978-3-030-31784-3\\_16](https://doi.org/10.1007/978-3-030-31784-3_16)
27. Abdulla, P. A., Atig, M. F., Chen, Y., Diep, B. P., Holík, L., Rezine, A., & Rümmer, P. (2018). Trau: SMT solver for string constraints. In N. S. Bjørner, & A. Gurfinkel (Eds.), *2018 Formal Methods in Computer*

- Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018* (pp. 1–5). IEEE. <https://doi.org/10.23919/FMCAD.2018.8602997>
28. Abdulla, P. A., Atig, M. F., Chen, Y., Diep, B. P., Holík, L., Rezine, A., & Rümmer, P. (2017). Flatten and conquer: a framework for efficient analysis of string constraints. In A. Cohen, & M. T. Vechev (Eds.), *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (pp. 602–617). ACM. <https://doi.org/10.1145/3062341.3062384>
  29. Abdulla, P. A., Atig, M. F., Chen, Y., Diep, B. P., Holík, L., Hu, D., Tsai, W., Wu, Z., & Yen, D. (2021). Solving not-substring constraint with flat abstraction. In H. Oh (Ed.), *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings, Vol. 13008 of Lecture Notes in Computer Science* (pp. 305–320). Springer. [https://doi.org/10.1007/978-3-030-89051-3\\_17](https://doi.org/10.1007/978-3-030-89051-3_17)
  30. Abdulla, P. A., Atig, M. F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., & Stenman, J. (2014). String constraints for verification. In A. Biere, & R. Bloem (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, Vol. 8559 of Lecture Notes in Computer Science* (pp. 150–166). Springer. [https://doi.org/10.1007/978-3-319-08867-9\\_10](https://doi.org/10.1007/978-3-319-08867-9_10)
  31. Abdulla, P. A., Atig, M. F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., & Stenman, J. (2015). Norn: An SMT solver for string constraints. In D. Kroening, & C. S. Pasareanu (Eds.), *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, Vol. 9206 of Lecture Notes in Computer Science* (pp. 462–469). Springer. [https://doi.org/10.1007/978-3-319-21690-4\\_29](https://doi.org/10.1007/978-3-319-21690-4_29)
  32. Trinh, M., Chu, D., & Jaffar, J. (2014). S3: A symbolic string solver for vulnerability detection in web applications. In *CCS, ACM Transactions on Computational Logic* (pp. 1232–1243).
  33. Le, Q. L., & He, M. (2018). A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In S. Ryu (Ed.), *Programming Languages and Systems* (pp. 350–372). Cham: Springer International Publishing.
  34. Abdulla, P. A., Atig, M. F., Chen, Y., Diep, B. P., Dolby, J., Janku, P., Lin, H., Holík, L., & Wu, W. (2020). Efficient handling of string-number conversion. In *Proceedings of PLDI'20* (pp. 943–957). ACM. <https://doi.org/10.1145/3385412.3386034>
  35. Wang, H., Tsai, T., Lin, C., Yu, F., & Jiang, J. R. (2016). String analysis via automata manipulation with logic circuit representation. In *CAV'16, Vol. 9779 of LNCS* (pp. 241–260). Springer.
  36. Wang, H.-E., Chen, S.-Y., Yu, F., & Jiang, J.-H. R. (2018). A symbolic model checking approach to the analysis of string and length constraints. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Association for Computing Machinery, New York, NY, USA* (pp. 623–633). <https://doi.org/10.1145/3238147.3238189>
  37. Hooimeijer, P., & Weimer, W. (2012). StrSolve: Solving string constraints lazily. *Automated Software Engineering*, 19(4), 531–559.
  38. Amadini, R., Gange, G., Stuckey, P. J., & Tack, G. (2017). A novel approach to string constraint solving. In J. C. Beck (Ed.), *Principles and Practice of Constraint Programming* (pp. 3–20). Cham: Springer International Publishing.
  39. Yu, F., Alkhalaf, M., & Bultan, T. (2010). Stranger: An automata-based string analysis tool for PHP. In *TACAS'10, Vol. 6015 of LNCS* (pp. 154–157). Springer.
  40. Yu, F., Alkhalaf, M., Bultan, T., & Ibarra, O. H. (2014). Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1), 44–70.
  41. Yu, F., Bultan, T., & Ibarra, O. H. (2011). Relational string verification using multi-track automata. *International Journal of Foundations of Computer Science*, 22(8), 1909–1924.
  42. Aydin, A., Bang, L., & Bultan, T. (2015). Automata-based model counting for string constraints. In D. Kroening & C. S. Pasareanu (Eds.), *Computer Aided Verification* (pp. 255–272). Cham: Springer International Publishing.
  43. Bultan, T. (2025). contributors, ABC string solver. <https://github.com/vlab-cs-ucsb/ABC>
  44. Scott, J. D., Flener, P., Pearson, J., & Schulte, C. (2017). Design and implementation of bounded-length sequence variables. In D. Salvagnin & M. Lombardi (Eds.), *Integration of AI and OR Techniques in Constraint Programming* (pp. 51–67). Cham: Springer International Publishing.
  45. Kiezun, A., Ganesh, V., Artzi, S., Guo, P. J., Hooimeijer, P., & Ernst, M. D. (2012). HAMP: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Computational Logic*, 21(4), 25:1-25:28.
  46. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., & Song, D. (2010). A symbolic execution framework for JavaScript. In *SP'10* (pp. 513–528). IEEE Computer Society.
  47. Cox, A., & Leasure, J. (2017). Model checking regular language constraints. CoRR abs/1708.09073.

48. Chen, Y., Havlena, V., Lengál, O., & Turrini, A. (2020). A symbolic algorithm for the case-split rule in string constraint solving. In B. C. d. S. Oliveira (Ed.), *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings, Vol. 12470 of Lecture Notes in Computer Science* (pp. 343–363). Springer. [https://doi.org/10.1007/978-3-030-64437-6\\_18](https://doi.org/10.1007/978-3-030-64437-6_18)
49. Day, J. D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., & Poulsen, D. B. (2019). On solving word equations using SAT. In E. Filiot, R. M. Jungers, & I. Potapov (Eds.), *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings, Vol. 11674 of Lecture Notes in Computer Science* (pp. 93–106). Springer. [https://doi.org/10.1007/978-3-030-30806-3\\_8](https://doi.org/10.1007/978-3-030-30806-3_8)
50. Li, G., & Ghosh, I. (2013). Pass: String solving with parameterized array and interval automaton. In V. Bertacco & A. Legay (Eds.), *Hardware and Software: Verification and Testing* (pp. 15–31). Cham: Springer International Publishing.
51. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., & Veanes, M. (2011). Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium 2011, USENIX Association*.
52. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., & Bjørner, N. (2012). Symbolic finite state transducers: Algorithms and applications. In *POPL'12, ACM Transactions on Computational Logic* (pp. 137–150).
53. Fu, X., & Li, C. (2010). Modeling regular replacement for string constraint solving. In *NFM'10, Vol. NASA/CP-2010-216215 of NASA* (pp. 67–76).
54. Trinh, M., Chu, D., & Jaffar, J. (2016). Progressive reasoning over recursively-defined strings. In *CAV'16, Vol. 9779 of LNCS* (pp. 218–240). Springer.
55. Day, J. D., Ganesh, V., Grewal, N., & Manea, F. (2023). On the expressive power of string constraints. *Proceedings of the ACM on Programming Languages* 7(POPL). <https://doi.org/10.1145/3571203>
56. Plandowski, W. (1999). Satisfiability of word equations with constants is in NEXPTIME. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, STOC '99, Association for Computing Machinery, New York, NY, USA* (pp. 721–725). <https://doi.org/10.1145/301250.301443>
57. Jež, A. (2016). Recompression: A simple and powerful technique for word equations. *Journal of ACM* 63(1). <https://doi.org/10.1145/2743014>
58. Makanin, G. S. (1977). The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 32(2), 147–236. (in Russian).
59. Nielsen, J. (1917). Die isomorphismen der allgemeinen, unendlichen gruppe mit zwei erzeugenden. *Mathematische Annalen*, 78(1), 385–397.
60. Blahoudek, F., Chen, Y., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., & Síc, J. (2023). Word equations in synergy with regular constraints. In M. Chechik, J. Katoen, & M. Leucker (Eds.), *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings, Vol. 14000 of Lecture Notes in Computer Science* (pp. 403–423). Springer. [https://doi.org/10.1007/978-3-031-27481-7\\_23](https://doi.org/10.1007/978-3-031-27481-7_23)
61. Lentin, A. (1970). Équations dans les monoïdes libres. *Mathématiques et sciences humaines* 31, 5–16. [http://www.numdam.org/item/MSH\\_1970\\_\\_31\\_\\_5\\_0/](http://www.numdam.org/item/MSH_1970__31__5_0/)
62. Choffrut, C., & Karhumäki, J. (1997). *Combinatorics of Words*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 329–438. [https://doi.org/10.1007/978-3-642-59136-5\\_6](https://doi.org/10.1007/978-3-642-59136-5_6)
63. Karhumäki, J., Mignosi, F., & Plandowski, W. (2000). The expressibility of languages and relations by word equations. *Journal of the ACM*, 47(3), 483–505. <https://doi.org/10.1145/337244.337255>
64. Plandowski, W., & Rytter, W. (1998). Application of lempel-ziv encodings to the solution of word equations. In K. G. Larsen, S. Skyum, & G. Winskel (Eds.), *Automata, Languages and Programming* (pp. 731–742). Berlin, Heidelberg: Springer Berlin Heidelberg.
65. Day, J. D., Manea, F., & Nowotka, D. (2019). Upper bounds on the length of minimal solutions to certain quadratic word equations. In P. Rossmanith, P. Heggernes, J.-P., & Katoen (Eds.), *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019), Vol. 138 of Leibniz International Proceedings in Informatics (LIPIcs)* (pp. 44:1–44:15). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany. <https://doi.org/10.4230/LIPIcs.MFCS.2019.44>. <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.MFCS.2019.44>
66. Aziz, A., Singhal, V., Swamy, G., & Brayton, R. K. (1993). Minimizing interacting finite state machines. Tech. Rep. UCB/ERL M93/68, EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1993/2425.html>
67. Henzinger, M., Henzinger, T., & Kopke, P. (1995). Computing simulations on finite and infinite graphs. In *Proceedings of IEEE 36th Annual Foundations of Computer Science* (pp. 453–462). <https://doi.org/10.1109/SFCS.1995.492576>
68. Blahoudek, F., Chen, Y.-F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., & Síc, J. (2022). Noodler. <https://github.com/vhavlena/Noodler>

69. Chocholatý, D., Fiedor, T., Havlena, V., Holík, L., Hruška, M., Lengál, O., & Síč, J. (2024). Mata: A fast and simple finite automata library. In B. Finkbeiner & L. Kovács (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 130–151). Cham: Springer Nature Switzerland.
70. Lu, Z., Siemer, S., Jha, P., Day, J., Manea, F., & Ganesh, V. (2024). Layered and staged monte carlo tree search for smt strategy synthesis. In K. Larson (Ed.), *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24* (pp. 1907–1915). International Joint Conferences on Artificial Intelligence Organization. main Track. <https://doi.org/10.24963/ijcai.2024/211>
71. Christensen, A. S., Møller, A., & Schwartzbach, M. I. (2003). Precise analysis of string expressions. In R. Cousot (Ed.), *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings, Vol. 2694 of Lecture Notes in Computer Science* (pp. 1–18). Springer. [https://doi.org/10.1007/3-540-44898-5\\_1](https://doi.org/10.1007/3-540-44898-5_1)
72. Ganesh, V., Minnes, M., Solar-Lezama, A., & Rinard, M. C. (2012). Word equations with length constraints: What's decidable?. In A. Biere, A. Nahir, & T. E. J. Vos (Eds.), *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers, Vol. 7857 of Lecture Notes in Computer Science* (pp. 209–226). Springer. [https://doi.org/10.1007/978-3-642-39611-3\\_21](https://doi.org/10.1007/978-3-642-39611-3_21)
73. Amadini, R., Flener, P., Pearson, J., Scott, J. D., Stuckey, P. J., & Tack, G. (2017). Minizinc with strings. In M. V. Hermenegildo & P. Lopez-Garcia (Eds.), *Logic-Based Program Synthesis and Transformation* (pp. 59–75). Cham: Springer International Publishing.

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.