Formal Methods in Computer-Aided Design 2024

Solving String Constraints with Concatenation Using SAT

Kevin Lotz*, Amit Goel†, Bruno Dutertre‡, Benjamin Kiesl-Reiter§, Soonho Kong‡, and Dirk Nowotka*
*Department of Computer Science, Kiel University, Kiel, Germany, {kel, dn}@informatik.uni-kiel.de
†Amazon Web Services, Portland, OR, USA, amgoel@amazon.com
‡Amazon Web Services, Santa Clara, CA, USA, {dutebrun, soonho}@amazon.com
§Amazon Web Services, Munich, Germany, benkiesl@amazon.com

***Abstract*—We present a decision procedure for solving quantifier-free first-order formulas over the theory of strings, involving equality, regular constraints, and concatenation of string terms. Our approach uses an eager reduction to the Boolean satisfiability problem and extends the NFA2SAT string solver. We describe a novel SAT encoding for word equations that iteratively expands the search space and leverages incremental SAT solving. For unsatisfiable formulas, we estimate the bounds on the smallest solution from arithmetic constraints derived from word equations. An experimental evaluation shows that our approach is competitive with state-of-the-art string solvers and complements existing methods in string solving.**

I. Introduction

Reasoning on string manipulation is a crucial aspect of ensuring software correctness. In recent years, a variety of tools, known as *string solvers*, have been developed to automate decision procedures for various logical theories over strings. Advancements in string solving have been driven by web-application security [16], [27], [29] and model checking [11]. These fields rely on automated reasoning on strings to identify critical security vulnerabilities. More recently, string solving has been used to verify security properties of cloud access policies [1], [26].

The theory of strings draws upon combinatorics on words [25], [12], [4], [22]. Central to this theory are word equations, which are expressions that equate two strings constructed by *concatenating* variables and constant words. Solving word equations amounts to finding substitutions for the variables that make the two sides of the equation identical. For example, we find a solution for $a \cdot \mathsf{x} \doteq \mathsf{y} \cdot a$ by substituting both variables x and y with a. Solving word equations is decidable [22], [8], [28], but the decision procedures resulting from the theoretical results are too expensive to be practical. To overcome this limitation, string solvers employ heuristic approaches and impose restrictions on the constraint languages to achieve scalability in practical use cases.

Most modern string solvers [23], [2], [14], [5], [6], [21] are built upon the *CDCL(T)* paradigm, also called *lazy* solving. This framework operates in two steps: first, a SAT solver searches for a model of the propositional structure of a formula, and second, a theory solver decides whether this model is consistent in a background theory T. An alternative approach is *eager* solving, which encodes the input problem into a single propositional formula. In the context of string solving, eager approaches were first explored by the WOORPJE [7] solver for word equations and its extension to regular constraints [18].

In previous work [20], we presented the eager string solver NFA2SAT that decides the satisfiability of formulas within a restricted logical fragment, which includes regular constraints and equality between strings but excludes concatenation of string terms. The solver is complete on this fragment, but it supports a less expressive logic compared to other string solvers. Here, we bridge this gap by extending NFA2SAT's decision procedure to support word equations.

The NFA2SAT procedure sets bounds on the lengths of all string variables occurring in a formula, encodes the bounded problem into a propositional formula, and tests its satisfiability. If the formula is unsatisfiable, the procedure iterates by incrementally increasing these bounds until either a solution is found, or the bounds exceed the theoretical length of the minimal solution to the formula, at which point the formula is declared unsatisfiable.

To enable support for word equations, we introduce a new method to encode the satisfiability problem of bounded word equations into propositional logic. We also prove an alphabet-reduction result, which we use to obtain a small alphabet that is sufficiently large to preserve satisfiability. This reduction is critical to ensure the practicality of the encoding because it reduces the size of the propositional formula, thereby allowing for more efficient SAT solving. We then propose an incomplete but practical approach to detecting unsatisfiability, by analyzing linear integer equations over the lengths of the string variables that occur in word equations. An experimental evaluation on a large set of benchmarks shows that our approach is competitive with state-of-the-art string solvers and works well as a complement to lazy solvers.

II. Preliminaries

A *word* is a finite sequence $w = w_1 \cdots w_n$ where each w_i is a symbol in a finite alphabet A. We denote by $|w| = n$ the length of word w. The set of all words over A is denoted by A^*. We denote by $w \cdot w'$ the concatenation of two words w and w', and we denote the empty word by ε. A word u is called a *factor* of w if w can be written $v \cdot u \cdot v'$. It is called a *prefix* (*suffix*) if $v = \varepsilon$ ($v' = \varepsilon$). We use $|w|_a$ to denote the number of occurrences of symbol a in word w. We fix

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_9

an alphabet $\Sigma = \{ a, b, c, \dots \}$ of *constants* and an alphabet $\Gamma = \{ \mathsf{x}, \mathsf{y}, \mathsf{z}, \dots \}$ of *variables*. A word $w \in \Sigma^*$ is called a *constant word* and a word $\alpha \in (\Sigma \cup \Gamma)^*$ is called a *pattern*.

A *word equation* $\alpha \doteq \beta$ is a pair of patterns α, β. A *regular constraint* $\alpha \mathrel{\dot\in} R$ consists of a pattern $\alpha \mathrel{\dot\in} (\Sigma \cup \Gamma)^*$ and a regular expression R over the alphabet Σ. Let $h : (\Sigma \cup \Gamma)^* \to \Sigma^*$ be a morphism that is constant for all $c \in \Sigma$, i.e., $h(c) = c$. Then h is a solution of $\alpha \doteq \beta$ (written $h \models \alpha \doteq \beta$) if $h(\alpha) = h(\beta)$, and a solution of $\alpha \mathrel{\dot{\neq}} \beta$ (written $h \models \alpha \mathrel{\dot{\neq}} \beta$) if $h(\alpha) \neq h(\beta)$. Similarly, h is a solution of $\alpha \mathrel{\dot\in} R$ (written $h \models \alpha \mathrel{\dot\in} R$) if $h(\alpha) \in \mathcal{L}(R)$, and a solution of $\alpha \mathrel{\dot\notin} R$ (written $h \models \alpha \mathrel{\dot\notin} R$) if $h(\alpha) \notin \mathcal{L}(R)$, where $\mathcal{L}(R)$ denotes the regular language defined by R. Any substitution of the variables $h : \Gamma \to \Sigma^*$ can be canonically extended to a morphism, and vice versa. We therefore use the terms substitution and morphism interchangeably.

A function $l : \Gamma \to \mathbb{N}$ that assigns a length $l(\mathsf{x})$ to each variable $\mathsf{x} \in \Gamma$ is called a *length assignment*. Given a length assignment l, we use $\vec{\mathsf{x}}^l$ to refer to the sequence $\mathsf{x}[1] \cdots \mathsf{x}[l(\mathsf{x})]$ over the alphabet $\vec{\Gamma} = \{ \mathsf{x}[k] \mid \mathsf{x} \in \Gamma, k \in \mathbb{N} \}$. In this sequence, the $\mathsf{x}[i]$ can be interpreted as variables ranging over Σ, that is, each $\mathsf{x}[i]$ denotes a single character of Σ. We lift this definition to patterns with $\overrightarrow{w\alpha}^l = w \cdot \vec{\alpha}^l$ and $\overrightarrow{\mathsf{x} \cdot \alpha}^l = \vec{\mathsf{x}}^l \cdot \vec{\alpha}^l$ for all $w \in \Sigma^*$. For a word equation $\alpha \doteq \beta$, a length assignment l with $|\vec{\alpha}^l| = n = |\vec{\beta}^l|$ induces an equivalence relation of the positions $1, \dots, n$. Two positions i, j are *equivalent* under l, written as $i \sim_l j$, if $\vec{\alpha}[i] = \vec{\alpha}[j]$, $\vec{\beta}[i] = \vec{\beta}[j]$, or $\vec{\alpha}[i] = \vec{\beta}[j]$. If $i \sim_l j$, then characters at position i and j must be equal in any solution that is consistent with the length assignment. We call h an l-substitution if $|h(\mathsf{x})| = l(\mathsf{x})$ for all $\mathsf{x} \in \Gamma$. If h is an l-substitution, then h is a solution if and only if $h(\alpha)[i] = h(\alpha)[j]$ for all i, j with $i \sim_l j$. In that case, we call h an *l-solution*. Solving word equations by assigning a constant from Σ to every $\mathsf{x}[1] \cdots \mathsf{x}[l(\mathsf{x})]$ for all $\mathsf{x} \in \Gamma$ to find a morphism that satisfies the above condition is also known as *filling the positions* [15], [25].

We consider quantifier-free first-order formulas in which all atoms are word equations or regular constraints, or can be reduced to them. For such a formula ψ, we use $\mathrm{atoms}(\psi)$ to denote the set of atoms that occur in ψ, $\mathrm{vars}(\psi)$ to denote the set of variables in ψ, and $\Sigma(\psi)$ to denote the set of constants occurring in ψ. A substitution $h : \Gamma \to \Sigma^*$ is called a *model* of ψ, written $h \models \psi$, if ψ evaluates to true under h using the standard semantics of Boolean connectives. We assume throughout the paper that ψ is in negative normal form (NNF), that is, negations occur only in front of atoms. The literals of ψ can be of the form $\alpha \doteq \beta$, $\neg(\alpha \doteq \beta)$, $\alpha \mathrel{\dot\in} R$, or $\neg(\alpha \mathrel{\dot\in} R)$. We use $\alpha \mathrel{\dot{\neq}} \beta$ and $\alpha \mathrel{\dot\notin} R$ as short-hand notation for $\neg(\alpha \doteq \beta)$ and $\neg(\alpha \mathrel{\dot\in} R)$, respectively. We call ψ *conjunctive* if it is a conjunction of literals. We say ψ is in *normal form* if all literals have the form $\alpha \doteq \beta$, $\mathsf{x} \in R$, $\mathsf{x} \mathrel{\dot\notin} R$, or $\mathsf{x} \mathrel{\dot{\neq}} \mathsf{y}$. For every formula ψ there exists an equisatisfiable formula ψ' in normal form. We construct it by rewriting literals of the form $\alpha \mathrel{\dot{\neq}} \beta$ to $\alpha = \mathsf{t}_\alpha \wedge \beta = \mathsf{t}_\beta \wedge \mathsf{t}_\alpha \mathrel{\dot{\neq}} \mathsf{t}_\beta$, and literals of the form $\alpha \mathrel{\dot\in} R$ ($\alpha \mathrel{\dot\notin} R$) to $\alpha = \mathsf{t}_\alpha \wedge \mathsf{t}_\alpha \mathrel{\dot\in} R$ ($\alpha = \mathsf{t}_\alpha \wedge \mathsf{t}_\alpha \mathrel{\dot\notin} R$), where t_α and t_β are fresh variables. We have $\Sigma(\psi) = \Sigma(\psi')$ and $\mathrm{vars}(\psi) \subseteq \mathrm{vars}(\psi')$. If $h \models \psi'$, then $h \models \psi$, and if $h \models \psi$, then $h' \models \psi'$, where h' extends h by setting $h'(\mathsf{t}_\alpha) = h(\alpha)$.

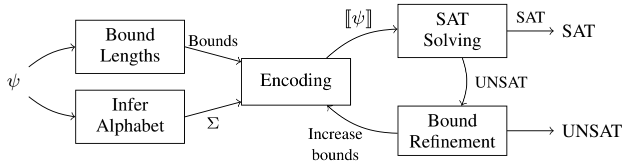


Fig. 1: Overview of the decision procedure.

$$
\begin{aligned}
F &:= F \vee F \mid F \wedge F \mid \neg F \mid \text{Atom} \\
\text{Atom} &:= t_{\text{str}} \mathrel{\dot\in} RE \mid \mathsf{t}_{\text{str}} \doteq \mathsf{t}_{\text{str}} \\
RE &:= RE \cup RE \mid RE \cdot RE \mid RE^* \mid RE \cap RE \mid \,?\, \mid w \\
t_{\text{str}} &:= \mathsf{x} \mid w \mid t_{\text{str}} \cdot t_{\text{str}}
\end{aligned}
$$

Fig. 2: Syntax: x denotes a variables, w denotes a word of Σ^*, and RE denotes a regular expression, where ? is the wildcard character.

III. The Decision Procedure

Our decision procedure accepts formulas in the syntax given in Figure 2. The procedure is based on an eager reduction to the Boolean satisfiability problem. This reduction assumes fixed upper bounds on the lengths of variables and translates the input problem into a propositional formula that is equisatisfiable for these bounds. It builds upon the string solver NFA2SAT, which we will review first.

The NFA2SAT procedure is depicted in Figure 1. It begins by assigning an initially small upper bound $b(\mathsf{x})$ on the length of all string variables x. Subsequently, the solver constrains the search space for substitutions to a small alphabet Σ that preserves satisfiability. If the problem is satisfiable in any superset of Σ, it remains satisfiable in Σ. Therefore, the procedure only needs to consider substitutions that map variables to words in the reduced alphabet Σ^*.

For the given bounds and alphabet, NFA2SAT encodes the first-order formula ψ into a propositional formula $[\![\psi]\!]$. If $[\![\psi]\!]$ is satisfiable, NFA2SAT declares ψ satisfiable. If $[\![\psi]\!]$ is unsatisfiable, then there are no solutions that satisfy ψ within the given upper bounds. In this case, NFA2SAT increases the bounds on the variable lengths and incrementally encodes the problem for these new bounds. This incremental encoding produces new clauses without discarding those from the previous SAT solver invocation, leveraging the benefits of incremental SAT solving under assumptions [10].

This process repeats until either the bounds are sufficiently large for $[\![\psi]\!]$ to be satisfiable, or unsatisfiability can be concluded based on the *small model property* stating that if ψ is satisfiable, then there exists a smallest model. We can compute bounds on the variable lengths in the smallest model and compare them to the bounds that resulted in the unsatisfiable encoding. If the bounds of the unsatisfiable encoding exceed those of the smallest model, then no solution exists.

To improve efficiency, NFA2SAT utilizes the unsat core from the last SAT solver call to refine variable bounds and handle unsatisfiability. It first computes the powerset of the (typically small) set of literals encoded in the unsat core. For each subset in the powerset, the solver calculates the small model bounds of the conjunction of the literals in that subset. Finally, take the maximum among all these computed bounds. The iteration over the powerset is necessary because the core is not necessarily minimal. If the bounds used for the last call to the SAT solver exceed the maximum, then increasing the bounds will not eliminate the unsat core and we can then conclude that ψ is not satisfiable. Otherwise, only the bounds of the variables occurring in a literal encoded in the unsat core are increased.

We extend the existing NFA2SAT procedure to support arbitrary concatenation, which amounts to solving word equations after conversion to normal form. Our new procedure includes three new components:

1) **Alphabet Reduction:** NFA2SAT narrows the search space by determining a small alphabet in which the formula is satisfiable if it is in any larger alphabet. In Section IV we show that adding one character to $\Sigma(\psi)$ for each $\not\doteq$ atom is sufficient to preserve satisfiability.
2) **Encoding of Word Equations:** In Section V, we introduce a new encoding that translates word equations into propositional logic in a way that is compatible with NFA2SAT's incremental framework.
3) **Handling Unsatisfiability:** For cases where no solution exists, we propose a simple but incomplete technique to compute the bounds of the small model if the unsat core contains (encoded) word equations. This approach is detailed in Section VI.

IV. Alphabet Reduction

A model for a string formula is a mapping from variables to constant words in an alphabet Σ, which can depend on the context. For example, in the SMT-LIB standard [3], Σ is a subset of the Unicode alphabet that contains $196,607$ symbols. In most cases, a model for satisfiable formulas can be constructed using only a small subset of Σ. Restricting the search space to such a subset is essential for our propositional encoding to be practical. Let $\Sigma(\psi)$ be the set of all constants occurring in ψ. If ψ does not contain string concatenation, it is satisfiable if and only if it is satisfiable in the alphabet $\Sigma(\psi)$ augmented with one additional character per variable [20]. When string concatenation is allowed, we instead need one additional character per negated equation. We show this result by fixing a solution $h : \Gamma \rightarrow \Sigma^*$ over any alphabet Σ and constructing a new solution $h' : \Gamma \rightarrow (\Sigma(\psi) \cup A)^*$, where A is disjoint from $\Sigma(\psi)$ and has a cardinality equal to the number of inequalities in ψ. The construction is based on the method of filling the positions.

Given a word equation $\alpha \doteq \beta$ and a length assignment l, we lift the equivalence relation $\sim_l$ to the elements of $\overrightarrow{\Gamma} \cup \Sigma$. Two elements $r, s \in \Sigma \cup \overrightarrow{\Gamma}$ are equivalent (under l), written as $r \sim_l s$, if there are i, j with $i \sim_l j$ and $\overrightarrow{\alpha}^l[i] = r \wedge \overrightarrow{\alpha}^l[j] = s$ or $\overrightarrow{\beta}^l[i] = r \wedge \overrightarrow{\beta}^l[j] = s$, or $\overrightarrow{\alpha}^l[i] = r \wedge \overrightarrow{\beta}^l[j] = s$. This defines an equivalence relation on $\Sigma \cup \overrightarrow{\Gamma}$. We use $[r]_{\sim_l}$ to refer to the transitive reflexive closure of $r \in \Sigma \cup \overrightarrow{\Gamma}$ under l. If l is clear from the context, we simply write $r \sim s$ and $[r]_\sim$. An l-solution h maps every class $[r]_\sim$ to exactly one constant c in the sense that $h(\mathsf{x})[j] = c$ for all $\mathsf{x}[j] \in [r]_\sim$. If $[r]_\sim$ contains a constant character then c is that character. For example, let $\mathsf{x} \cdot a \doteq a \cdot \mathsf{y}$ be a word equation. The length assignment $l(\mathsf{x}) = 3$ and $l(\mathsf{y}) = 3$ yields $\mathsf{x}[1]{\cdot}\mathsf{x}[2]{\cdot}\mathsf{x}[3]{\cdot}a \doteq a{\cdot}\mathsf{y}[1]{\cdot}\mathsf{y}[2]{\cdot}\mathsf{y}[3]$. It induces the three equivalence classes $\{\, \mathsf{x}[1], \mathsf{y}[3], a \,\}$, $\{\, \mathsf{x}[2], \mathsf{y}[1] \,\}$, and $\{\, \mathsf{x}[3], \mathsf{y}[2] \,\}$. Every l-substitution h that satisfies $h(\mathsf{x})[1] = h(\mathsf{y})[3] = a$, $h(\mathsf{x})[2] = h(\mathsf{y})[1]$, and $h(\mathsf{x})[3] = h(\mathsf{y})[2]$ is a solution.

It's a well-known result that any satisfiable word equation has a solution in the alphabet $\Sigma(\alpha \doteq \beta)$ [15]. A similar result holds for regular constraints [20] and both results can be combined to show that a word equation with regular constraints on the variables is satisfiable if and only if it has a solution that uses only the constants occurring in the problem (if there is at least one). This no longer holds when negations are allowed. For example, consider the formula $\mathsf{x}{\cdot}a \doteq a{\cdot}\mathsf{y} \wedge \mathsf{x} \not\doteq \mathsf{y}$. Any solution h must satisfy $|h(\mathsf{x})| = |h(\mathsf{y})|$ but constructing an h with $h(\mathsf{x}) \neq h(\mathsf{y})$ is not possible if we use only a, the sole constant.

We first generalize the result to formulas of the form $\psi := \alpha \doteq \beta \wedge \psi_{\neq} \wedge \psi_{\in}$, where $\psi_{\neq}$ and $\psi_{\in}$ are conjunctions of inequalities between variables $\mathsf{x} \not\doteq \mathsf{y}$, and regular constraints $\mathsf{x} \mathrel{\dot\in} R$, respectively. This restriction implicitly includes negated regular constraints, as $\mathsf{x} \mathrel{\dot\notin} R$ can be equivalently formulated as $\mathsf{x} \mathrel{\dot\in} \bar{R}$, where $\bar{R}$ is the regular complement of R. For an l-solution h, we define the graph $\mathcal{G}_\psi(h) = (V, E)$ where V is the set of equivalence classes induced by l on $\alpha \doteq \beta$. The set E includes an edge $\{\, [\mathsf{x}[k]]_\sim, [\mathsf{y}[k]]_\sim \,\}$ iff $\psi_{\neq}$ contains $\mathsf{x} \not\doteq \mathsf{y}$, $|h(\mathsf{x})| = |h(\mathsf{y})|$, $h(\mathsf{x})[k] \neq h(\mathsf{y})[k]$, and $h(\mathsf{x})[k'] = h(\mathsf{y})[k']$ for all $k' < k$, meaning k is the smallest index where $h(\mathsf{x})$ and $h(\mathsf{y})$ disagree.

If $\mathcal{G}_\psi(h)$ can be colored with n colors, then a new model h' can be constructed using no more than n constants in addition to $\Sigma(\psi)$. Here, a color acts as a new constant, with h' mapping each equivalence class to the vertex color if the original model h mapped its members to a symbol that is not in $\Sigma(\psi)$.

Lemma 1. *Let $\psi := \alpha \doteq \beta \wedge \psi_{\neq} \wedge \psi_{\in}$ and h be a solution. If $\mathcal{G}_\psi(h)$ is n-colorable, then ψ has a solution over $\Sigma(\psi) \cup A$ where A is an alphabet disjoint from $\Sigma(\psi)$ with $|A| = n$.*

Thus, the minimal number of characters required in addition to $\Sigma(\psi)$ is the chromatic number of $\mathcal{G}_\psi(h)$. The next lemma gives an upper bound on this number. This follows from the fact that the graph has at most $|\mathrm{atoms}(\psi_{\neq})|$ edges.

Lemma 2. *Let $\psi := \alpha \doteq \beta \wedge \psi_{\neq} \wedge \psi_{\in}$ be a formula and h be a solution. Then $\mathcal{G}_\psi(h)$ is $|\mathrm{atoms}(\psi_{\neq})| + 1$ colorable.*

Combining the above results gives our main theorem.

Theorem 3. *Let ψ be a formula over word equations and regular constraints with n inequalities. Then ψ has a solution if and only if it has a solution in $\Sigma(\psi) \cup A$ where $A \cap \Sigma(\psi) = \emptyset$ and $|A| = n + 1$.*

The theorem can be shown by assuming ψ is in disjunctive normal form and equivalently rewriting each disjunct to match the form $\alpha \doteq \beta \wedge \psi_{\neq} \wedge \psi_{\in}$. Then, each disjunct can have at most n inequalities, and we can apply Lemma 1 and Lemma 2 to obtain the bound.

The bound on $|A|$ is not always tight because Lemma 2 gives only a coarse bound on the chromatic number. For instance, by applying results from [9], it can be lowered to $\mathcal{O}(\sqrt{n})$ for a formula with n inequalities. However, it is small enough to be practical. Additionally, since formulas often contain fewer inequalities than variables, this result improves our previously known bound of $|\mathrm{vars}(\psi)|$ for formulas ψ that do not include concatenation.

V. Encoding Word Equations

For a first-order formula ψ, our decision procedure fixes some bounds b and translates ψ into a propositional formula $\llbracket \psi \rrbracket^{b}$, which is satisfiable if and only if ψ has a solution within bounds b. The formula is constructed by encoding all literals of ψ individually. We present the encoding for literals that are word equations $\alpha \doteq \beta$, denoted $\llbracket \alpha \doteq \beta \rrbracket^{b}$. Intuitively, $\llbracket \alpha \doteq \beta \rrbracket^{b}$ asks the SAT solver to "guess" a word w for which there exists a substitution h such that $h(\alpha) = w$ and $h(\beta) = w$, making w a *solution word*. It is the conjunction of four formulas $\llbracket w \rrbracket^{b} \wedge \llbracket lh \rrbracket^{b} \wedge \llbracket m(\alpha) \rrbracket^{b} \wedge \llbracket m(\beta) \rrbracket^{b}$, modeling the set of all potential solution words, the set of all possible l-substitutions, and the constraint that the encoded substitution must map both patterns to the same word, respectively. The encoding is sound in the following sense.

Theorem 4. *Let $\alpha \doteq \beta$ be a word equation and b be a function assigning an upper bound to every variable in the equation. Then $\alpha \doteq \beta$ is satisfiable under b if and only if $\llbracket \alpha \doteq \beta \rrbracket^{b}$ is satisfiable.*

If ψ, containing $\alpha \doteq \beta$ as a literal, is not satisfiable with bounds b, then NFA2SAT proceeds to check the satisfiability for larger bounds b'. This results in n calls to the SAT solver, with bounds $b_1, \ldots, b_n$. To make this procedure efficient, the encoding is *incremental*. That is, the encoding $\llbracket \psi \rrbracket^{b_k}$ is constructed by only adding more clauses to the formula $\llbracket \psi \rrbracket^{b_{k-1}}$. In the following, we present the encoding $\llbracket \alpha \doteq \beta \rrbracket^{b_k}$ assuming that $\llbracket \alpha \doteq \beta \rrbracket^{b_{k-1}}$ was already encoded. To avoid treating edge cases, we assume $b_0(\mathsf{x}) = 0$ for all $\mathsf{x} \in \Gamma$. Additionally, we assume $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$.

A. Encoding Words

We encode the set of all words that are possible solutions to the equation in $\llbracket w \rrbracket^{b_k}$. This includes all words over Σ with length no longer than $U_k = \min(|\overrightarrow{\alpha}^{b_k}|, |\overrightarrow{\beta}^{b_k}|)$, i.e., the length of the smaller of the longest words that either side of the equation can be mapped to under bounds b_k. No substitution h with $|h(\alpha)| > U_k$ or $|h(\beta)| > U_k$ can be a solution w.r.t. to b_k because at least one side of the equation cannot be mapped to a word of length greater than U_k under b_k.

We first pick a new symbol λ that is not in Σ and set $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. The symbol λ denotes an unused position, i.e., a position that is to be mapped to the empty word. Setting an appropriate set of positions to λ allows us to encode all possible words over Σ with length at most U_k. We encode the set $\{\, w \in \Sigma^* \mid |w| \leq U_k \,\}$ by introducing the Boolean variables w_i^c for each position $1 \leq i \leq U_k$ and character $c \in \Sigma_\lambda$. Boolean variable w_i^c is true if c occurs at position i in w. We enforce that exactly one of the w_i^c is true using the following formula

$$\llbracket w \rrbracket^{b_k} = \bigwedge_{i=U_{k-1}}^{U_k} EO\{w_i^c \mid c \in \Sigma_\lambda\} \wedge \bigwedge_{i=U_{k-1}}^{U_k - 1} w_i^\lambda \to w_{i+1}^\lambda.$$

This takes into account that words of length up to U_{k-1} have been encoded in a previous call. In this formula, EO is an encoding of the *exactly-one* constraint on the variables (see [17]). Because concatenation with λ is neutral, we use the second conjunct of the encoding to break symmetry. This ensures that every Boolean assignment σ with $\sigma \models \llbracket w \rrbracket^{b_k}$ encodes exactly one word and for every word no longer than U_k there is exactly one σ with $\sigma \models \llbracket w \rrbracket^{b_k}$.

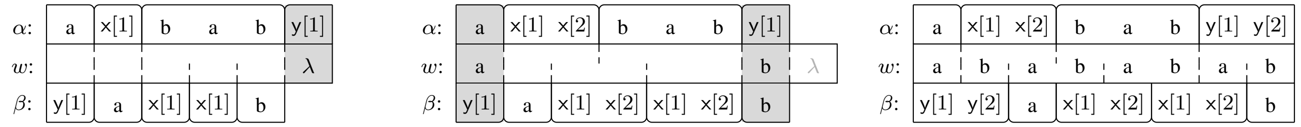
B. Encoding l-Substitutions

For all length assignments l bounded by b_k, i.e., for all l with $l(\mathsf{x}) \leq b_k(\mathsf{x})$ for all $\mathsf{x} \in \Gamma$, we encode the set of all possible l-substitutions. This is achieved by initially encoding all substitutions $\llbracket h \rrbracket^{b_k}$ and all length assignments in $\llbracket l \rrbracket^{b_k}$ (both limited by b_k), and then ensuring that the length of a variable substitution coincides with the length assignment.

The encoding of substitutions is constructed using a set of Boolean variables $\{\, h_{\mathsf{x}[i]}^{a} \mid a \in \Sigma_\lambda \,\}$. We ensure that every satisfying assignment to $\llbracket h \rrbracket^{b_k}$ encodes exactly one constant word for every variable x, i.e., the substitution of x, by employing an exactly-one constraint exactly as done for the encoding of words.

To encode all possible length assignments, we introduce a set of Boolean variables $\{\, L_{\mathsf{x}}^i \mid 0 \leq i \leq b_k(\mathsf{x}) \,\}$ for all $\mathsf{x} \in \Gamma$. We encode that L_{x}^i is true iff $l(\mathsf{x}) = i$, taking into account that the length assignments for bounds b_{k-1} are already encoded. This makes standard *exactly-one* encodings unsuitable and we instead use the following formula

$$\begin{aligned}
\llbracket l(\mathsf{x}) \rrbracket^{b_k} := \quad & (a_{\mathsf{x},k} \to \bigvee_{i=b_{k-1}(\mathsf{x})+1}^{b_k(\mathsf{x})} L_{\mathsf{x}}^i \vee a_{\mathsf{x},k-1}) && (1) \\
& \wedge \bigwedge_{i=0}^{b_{k-1}(\mathsf{x})} \bigwedge_{j=b_{k-1}(\mathsf{x})+1}^{b_k(\mathsf{x})} L_{\mathsf{x}}^i \to \neg L_{\mathsf{x}}^j && (2) \\
& \wedge \bigwedge_{i=b_{k-1}(\mathsf{x})+1}^{b_k(\mathsf{x})} \bigwedge_{j=i+1}^{b_k(\mathsf{x})} L_{\mathsf{x}}^i \to \neg L_{\mathsf{x}}^j && (3)
\end{aligned}$$



(a) We first assume an upper bound of 1 for both variables, i.e. $b_1(\mathsf{x}) = b_1(\mathsf{y}) = 1$. In that case, $[\![w]\!]^{b_1}$ encodes all words up to length 6. When assigning length 1 to all variables, $[\![lh]\!]^{b_1}$ and $[\![m(\beta)]\!]^{b_1}$ are conflicting: $[\![lh]\!]$ requires that $\mathsf{y}[1]$ is not λ, but since the last segment of β ends at position 5, $[\![m(\beta)]\!]$ requires that the 6th position, which aligns with $\mathsf{y}[1]$, is λ. Any other length assignment under b_1 will result in a similar situation. Both, the equation and the encoding are unsatisfiable under b_1.

(b) When assuming an upper bound of 2 for both variables, $b_2(\mathsf{x}) = b_2(\mathsf{y}) = 2$, $[\![w]\!]^{b_2}$ encodes all words up to length 8. Assigning length 2 to x and y maps both patterns to the same length. However, this length assignment still results in a conflict: The first segment of α (a) and the first segment of β (y) both start at position 1, so $[\![m(\alpha)]\!]^{b_2}$ and $[\![m(\beta)]\!]^{b_2}$ entail $w[1] = \mathsf{y}[1] = a$. At the same time, $[\![m(\alpha)]\!]^{b_2}$ and $[\![m(\beta)]\!]^{b_2}$ entail $w[1] = \mathsf{y}[1] = b$ because the respective last segments, b and y, start at position 7.

(c) When instead assigning length 2 to x y, the encoding, and therefore the equation, becomes satisfiable and we find a solution h with $h(\mathsf{x}) = ba$ and $h(\mathsf{y}) = ab$, resulting in the solution word $abababab$.

Fig. 3: Demonstrates of the encoding for the example equation $a \cdot \mathsf{x} \cdot bab \cdot \mathsf{y} \doteq \mathsf{y} \cdot a\mathsf{xx} \cdot b$. The figures illustrate how the encoding operates by fixing length assignments for the variables. This is analogous to the SAT solver assigning truth values to the L^i_{x} variables during the search procedure.

The Boolean variable $a_{\mathsf{x},k}$ is an assumption in the k^{th} call to the SAT solver. Part (1) states that if $a_{\mathsf{x},k}$ is true, then at least one of $\{ L^i_{\mathsf{x}} \mid b_{k-1}(\mathsf{x}) < i \leq b_k(\mathsf{x}) \}$ needs to be true, unless $a_{\mathsf{x},k-1}$ is true (defining $a_{\mathsf{x},0} = \bot$). If $a_{\mathsf{x},k-1}$ is true, then one L^i_{x} with $0 \leq i \leq b_{k-1}(\mathsf{x})$ must be true, establishing that there is at least one $i \leq b_k(\mathsf{x})$ such that L^i_{x} is true. The conjunction (2) and (3) guarantees that at most one L^i_{x} with $i \leq b_k(\mathsf{x})$ is true. Thus, the encoding ensures that exactly one of the variables L^i_{x} with $0 \leq i \leq b_k(\mathsf{x})$ is true.

Finally, $[\![lh]\!]^{b_k}$ combines $[\![h]\!]^{b_k}$ and $[\![l]\!]^{b_k}$ to ensure that the length of each substitution matches the assigned length. This is expressed by ensuring $h(\mathsf{x})[i] \cdots h(\mathsf{x})[b_k(\mathsf{x})] = \varepsilon$ if and only if L^i_{x} is true, using

$$[\![lh]\!]^{b_k} = [\![h]\!]^{b_k} \wedge [\![l]\!]^{b_k} \wedge \bigwedge_{\mathsf{x} \in \Gamma} \bigwedge_{i=b_k(\mathsf{x})}^{b_k(\mathsf{x})-1} (h^{\lambda}_{\mathsf{x}[i+1]} \leftrightarrow L^i_{\mathsf{x}}).$$

Assigning true to $h^{\lambda}_{\mathsf{x}[i+1]}$ encodes that the suffix of the substitution of x starting at $i+1$ is empty. Because exactly one L^i_{x} is true, this asserts that the length of the substitution of x is exactly i if L^i_{x} is true.

C. Matching Patterns To Words

We constrain that any assignment satisfying $[\![lh]\!]^{b_k} \wedge [\![w]\!]^{b_k}$ encodes an l-substitution h and a word w such that $h(\alpha) = w = h(\beta)$, asserting that h is a solution. This is achieved through the formulas $[\![m(\alpha)]\!]^{b_k}$ and $[\![m(\beta)]\!]^{b_k}$, which encode that h maps the i^{th} position of $\vec{\alpha}^{l}$ and $\vec{\beta}^{l}$ to the i^{th} position of w, for any encode length assignment l. Since the encoding is the same for both sides of the equation, we describe it using a generic pattern γ.

The idea of $[\![m(\gamma)]\!]^{b_k}$ is to split γ into consecutive factors of variables and constant words and assert that if a factor *starts* at position p in $\vec{\gamma}^{l}$ and has length k, then its substitution must be equal to the factor of the solution word w from p to $p+k-1$. An example of how this idea is reflected in the encoding is shown in Figure 3.

Formally, we define the segmentation of γ, denoted $\mathrm{seg}(\gamma)$, as the unique factorization $(\gamma_{(1)}, \ldots, \gamma_{(n)})$ of γ with $\gamma_{(i)} \in \Gamma$ or $\gamma_{(i)} \in \Sigma^+$ for all $i \leq n$, and if $\gamma_{(i)} \in \Sigma^+$ then either $\gamma_{(i+1)} \in \Gamma$ or $i = n$. For example, the pattern $\mathsf{x}{\cdot}abc{\cdot}\mathsf{y}{\cdot}\mathsf{x}{\cdot}def$ is factorized into five segments $(\mathsf{x}, abc, \mathsf{y}, \mathsf{x}, def)$. Given a length assignment l, the *start position* of $\gamma_{(i)}$ is given by $\sum_{j=1}^{i-1} |\gamma_{(j)}| + 1$, the sum of the lengths of all preceding segments plus one, where $|\gamma_{(i)}|$ is $l(\mathsf{x})$ if $\gamma_{(i)} = \mathsf{x}$ and $|v|$ if $\gamma_{(i)} = v \in \Sigma^+$. The start position of the first segment is thus always 1.

To ensure the matching between the patterns and the solution word, we first encode set set of all possible start positions for each segment of $\mathrm{seg}(\gamma)$ w.r.t. b_k and condition them on the lengths assigned to the variables using the L^i_{x} variables. The encoding then ensures if a segment starts at position p and has length k, the factor of the solution word from p to $p+k-1$ must be equal to $h(\gamma_{(i)})$, the constant word that the encoded morphism h maps $\gamma_{(i)}$ to. The idea is illustrated in Figure 4.

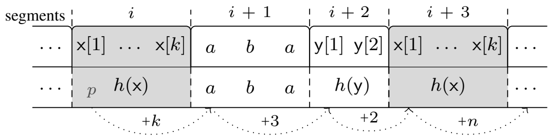


Fig. 4: Matching a pattern γ to a word w. If the i^{th} segment of γ, x, starts at position p, then the factor of w from p to $p+k-1$ must be equal to $h(\mathsf{x})$, and the $i+1^{\text{th}}$ segment must start at position $p+k$.

To encode the start positions, we introduce a set of Boolean variables $\{ S(\gamma)^p_i \mid 0 \leq p \leq U_k \}$ for all $1 \leq i \leq |\mathrm{seg}(\gamma)| + 1$, modeling that $S(\gamma)^p_i$ is true if $\gamma_{(i)}$ starts at position p, where

$S(\gamma)^p_{|\mathrm{seg}(\gamma)|+1}$ marks the end of the pattern. The matching is then encoded using the formula

$$\begin{aligned}
[\![m(\gamma)]\!]^{b_k} := \quad & S(\gamma)^1_1 && (1)\\
\wedge & \bigwedge_{p=U_{k-1}}^{U_k} S(\gamma)^p_{|\mathrm{seg}(\gamma)|+1} \rightarrow w^\lambda_p && (2)\\
\wedge & \bigwedge_{i=1}^{|\mathrm{seg}(\gamma)|-1} \mathrm{match}(\gamma, i). && (3)
\end{aligned}$$

Here, (1) encodes that the first segment starts at the first position. The second part, (2), ensures that the length of the solution word w equals the length of γ under l, by encoding that the first position of w following the last segment is mapped to λ. The last part, (3), establishes the matching between each segment $\gamma_{(i)}$ and the corresponding factor of the solution word, and determines the start position of $\gamma_{(i+1)}$. The encoding depends on whether $\gamma_{(i)}$ is a constant or a variable.

If $\gamma_{(i)} = v$ for some $v \in \Sigma^+$, then $\mathrm{match}(\gamma, i)$ is given by

$$\bigwedge_{p=\max(U_{k-1}-|v|,0)+1}^{U_k-|v|} S(\gamma)^p_i \rightarrow \bigwedge_{j=1}^{|v|} w^{v[j]}_{p+j-1} \wedge S^{p+|v|}_{i+1}.$$

The formula states that if $\gamma_{(i)}$ starts at position p, then the factor of w from p to $p + |v| - 1$ must be equal to v and $\gamma_{(i+1)}$ starts at $p + |v|$. The latest position at which the $\gamma_{(i)}$ can start is $U_k - |v|$, as otherwise, it would exceed U_k.

If $\gamma_{(i)} = \mathsf{x}$ for some variable $\mathsf{x} \in \Gamma$, then $\mathrm{match}(\gamma, i)$ is instead given by

$$\bigwedge_{(p,l)\in M_k \setminus M_{k-1}} S(\gamma)^p_i \wedge L^l_{\mathsf{x}} \rightarrow \bigwedge_{j=0}^{l-1} \bigwedge_{c\in\Sigma} (h^c_{\mathsf{x}[j]} \leftrightarrow w^c_{p+j}) \wedge S^{p+l}_{i+1}.$$

Here, $M_k = \{ (p, l) \mid p < U_k \wedge l \leq b_k(\mathsf{x}) \wedge p + l \leq U_k \}$ is the set of all pairs of positions and length assignments w.r.t b_k, such that $p + l(\mathsf{x}) \leq U_k$. The formula ensures that if $\gamma_{(i)}$ starts at position p and has length l, then the factor of w from p to $p + l - 1$ must be equal to $h(\mathsf{x})$, and that $\gamma_{(i+1)}$ starts at position $p + l$.

To guide the SAT solver, we impose an *at-most-one* constraint on $\{ S(\gamma)^p_i \mid 0 \leq p \leq U_k \}$ for all $1 \leq i \leq |\mathrm{seg}(\gamma)| + 1$. Additionally, we *disable* all infeasible start positions relative to U_k with assumptions. For segments $\gamma_{(i)} = v \in \Sigma^+$, we add $\neg S(\gamma)^p_i$ as an assumption for all p with $p > U_k - |v|$. For segments $\gamma_{(i)} = \mathsf{x} \in \Gamma$, we add the clauses $a \rightarrow \neg(L^l_{\mathsf{x}} \wedge S(\gamma)^p_i)$, with fresh variable a, for all l, p with $l < b(\mathsf{x})$, $p < U_k$, and $l + p > U_k$, and add a as an assumption.

VI. Finding and Refining Bounds

Whenever the SAT solver determines that the formula is unsatisfiable under bounds b, our procedure continues with larger bounds b'. This terminates once the bounds are either large enough to construct a solution or exceed the bounds of the smallest model, as explained in Section III.

Theoretical bounds on the minimal solution to a word equation can be computed, but these bounds can be doubly exponential [24]. Given this complexity, using exact minimal bounds is impractical. Instead, we employ a heuristic to identify tighter bounds by extracting linear constraints on variable lengths from the word equations. We use a known method for bounding the solutions of the resulting linear integer problem. This approach is sound but not complete as it may fail to find finite bounds on the variables.

Algorithm 1 Iterative Bound Refinement

```
Input: Conjunctive formula $\psi$
Output: Bounds for $\psi$ or UNSAT if $\psi$ is unsatisfiable
  $\mathrm{lb}, \mathrm{ub} \leftarrow$ init($\psi$)
  repeat
      for $\alpha \doteq \beta \in \mathrm{atoms}(\psi)$ do
          $\mathrm{lb}', \mathrm{ub}' \leftarrow$ refinement_step($[\alpha \doteq \beta]_L$, lb, ub)
          if conflict(lb′, ub′) then
              return UNSAT
          end if
      end for
  until $\mathrm{lb}' = \mathrm{lb} \wedge \mathrm{ub}' = \mathrm{ub}$
  return $\mathrm{lb}, \mathrm{ub}$
```

For a word equation $\alpha \doteq \beta$ we define the linear integer equation $[\alpha \doteq \beta]_L := \sum_{\mathsf{x}\in\Gamma}(|\alpha|_{\mathsf{x}} - |\beta|_{\mathsf{x}}) \cdot |\mathsf{x}| = \sum_{a\in\Sigma} |\beta|_a - |\alpha|_a$. The equation is satisfied by every substitution h with $|h(\alpha)| = |h(\beta)|$. Especially, if h is a solution for $\alpha \doteq \beta$, then it also satisfies $[\alpha \doteq \beta]_L$. Conversely, if a h does not satisfy $[\alpha \doteq \beta]_L$, it is not a solution for $\alpha \doteq \beta$. For instance, consider the word equation $\mathsf{z}{\cdot}b{\cdot}\mathsf{z}{\cdot}\mathsf{x} \doteq ba{\cdot}\mathsf{y}{\cdot}a{\cdot}\mathsf{y}{\cdot}bb$ which has the corresponding integer equation $2{\cdot}|\mathsf{z}| + |\mathsf{x}| + 1 = 2{\cdot}|\mathsf{y}| + 5$. The substitution $h = \{\mathsf{x} \leftarrow a, \mathsf{y} \leftarrow a, \mathsf{z} \leftarrow baaaa\}$ is not a solution because $2|h(\mathsf{z})| + |h(\mathsf{x})| + 1 = 12 \neq 9 = 2|h(\mathsf{y})| + 5$.

We treat $[\alpha \doteq \beta]_L$ as an equation over variables $\Gamma_L = \{ |\mathsf{x}| \mid \mathsf{x} \in \Gamma \}$. For a conjunction of word equations ψ, we lift $[\psi]_L$ to the conjunction of the corresponding integer equations. Our procedure computes lower and upper bounds $\mathrm{lb}, \mathrm{ub} : \Gamma_L \rightarrow \mathbb{N} \cup \infty$ such that $\mathrm{lb}(|\mathsf{x}|) \leq g(|\mathsf{x}|) \leq \mathrm{ub}(|\mathsf{x}|)$ for all $g : \Gamma_L \rightarrow \mathbb{N}$ satisfying $[\psi]_L$. The algorithm is sketched in Algorithm 1 and an example is shown in Figure 5.

Iterative Bound Refinement: For a conjunctive formula ψ in normal form, we derive bounds on the variables of ψ using an incremental refinement procedure. If variable x is constrained to belong to a regular expression R and R is recognized by a cycle-free n-state automaton then we set the initial bounds on x to $\mathrm{lb}_0(\mathsf{x}) = 0$ and $\mathrm{ub}_0(\mathsf{x}) = n$. Otherwise, we initialize the bounds to $\mathrm{lb}_0(\mathsf{x}) = 0$ and $\mathrm{ub}_0(\mathsf{x}) = \infty$. Thus, $\mathrm{lb}_0(|\mathsf{x}|) \leq g(|\mathsf{x}|) \leq \mathrm{ub}_0(|\mathsf{x}|)$ holds initially. The algorithm then iteratively refines the bounds until a conflict is detected or a fixed point is reached. A conflict occurs if $\mathrm{lb}(|\mathsf{x}|) > \mathrm{ub}(|\mathsf{x}|)$, $\mathrm{ub}(|\mathsf{x}|) < 0$ or $\mathrm{lb}(|\mathsf{x}|) = \infty$ for some variable x, in which case $[\psi]_L$, and therefore ψ, are unsatisfiable. If no conflict is found, then the functions lb and ub provide bounds on the lengths of the variables for the solutions to ψ. Specifically, if $\mathrm{ub}(|\mathsf{x}|) \neq \infty$ for all x, then ub are bounds on the smallest model for ψ. Algorithm 1 may not terminate in general (see [13]). In our implementation, we enforce an upper limit on the number of iterations and we return the best bounds available when this limit is reached.

Bounds: $\mathrm{ub}(\mathsf{x}) = \infty \quad \mathrm{ub}(\mathsf{y}) = \infty \quad \mathrm{ub}(\mathsf{z}) = \infty$
$\mathrm{lb}(\mathsf{x}) = 0 \quad \mathrm{lb}(\mathsf{y}) = 0 \quad \mathrm{lb}(\mathsf{z}) = 0$

1. Refine Equation $|\mathsf{x}| + 2{\cdot}|\mathsf{y}| = |\mathsf{y}| + 2$

$$|\mathsf{x}| = -|\mathsf{y}| + 2 \rightarrow |\mathsf{x}| \leq \mathrm{ub}(-|\mathsf{y}| + 2) \rightarrow |\mathsf{x}| \leq -\,\mathrm{lb}(|\mathsf{y}|) + 2 \rightarrow |\mathsf{x}| \leq 2$$

$$|\mathsf{y}| = -|\mathsf{x}| + 2 \rightarrow |\mathsf{y}| \leq \mathrm{ub}(-|\mathsf{x}| + 2) \rightarrow |\mathsf{y}| \leq -\,\mathrm{lb}(|\mathsf{x}|) + 2 \rightarrow |\mathsf{y}| \leq 2$$

Refined Bounds: $\mathrm{ub}'(\mathsf{x}) = \mathbf{2} \quad \mathrm{ub}'(\mathsf{y}) = \mathbf{2} \quad \mathrm{ub}'(\mathsf{z}) = \infty$
$\mathrm{lb}'(\mathsf{x}) = 0 \quad \mathrm{lb}'(\mathsf{y}) = 0 \quad \mathrm{lb}'(\mathsf{z}) = 0$

2. Refine Equation $2{\cdot}|\mathsf{z}| + |\mathsf{x}| + 1 = 2{\cdot}|\mathsf{y}| + 5$

$$|\mathsf{z}| = (-|\mathsf{x}| + 2|\mathsf{y}| + 4) \cdot \tfrac{1}{2} \rightarrow |\mathsf{z}| \leq -\,\mathrm{lb}(|\mathsf{x}|) + (2 \cdot \mathrm{ub}(|\mathsf{y}|) + 4) \cdot \tfrac{1}{2}$$
$$\rightarrow |\mathsf{z}| \leq 4$$

Refined Bounds: $\mathrm{ub}'(\mathsf{x}) = \mathbf{2} \quad \mathrm{ub}'(\mathsf{y}) = \mathbf{2} \quad \mathrm{ub}'(\mathsf{z}) = \mathbf{4}$
$\mathrm{lb}'(\mathsf{x}) = 0 \quad \mathrm{lb}'(\mathsf{y}) = 0, \quad \mathrm{lb}'(\mathsf{z}) = 0$

Fig. 5: Bound refinement for the system of word equations $\mathsf{x}{\cdot}\mathsf{y}{\cdot}\mathsf{y} \doteq a{\cdot}\mathsf{y}{\cdot}a \wedge \mathsf{z}{\cdot}b{\cdot}\mathsf{z}{\cdot}\mathsf{x} \doteq ba{\cdot}\mathsf{y}{\cdot}a{\cdot}\mathsf{y}{\cdot}bb$. Only upper bounds are shown. The initial bounds on $|\mathsf{x}|$, $|\mathsf{y}|$, and $|\mathsf{z}|$ are $[0, \infty)$. After processing the two equations, the bounds are refined to $0 \leq |\mathsf{x}| \leq 2$, $0 \leq |\mathsf{y}| \leq 2$, and $0 \leq |\mathsf{z}| \leq 4$.

Refinement Steps: The key part in the procedure is the *implied bound refinement* of [13]. This procedure extends the functions lb, ub to arithmetic terms. For constants c, set $\mathrm{lb}(c) = \mathrm{ub}(c) = c$. For terms of the form $c|\mathsf{x}|$, the value depends on whether the constant c is positive. If $c \geq 0$, then $\mathrm{lb}(c|\mathsf{x}|) = c \cdot \mathrm{lb}(|\mathsf{x}|)$ and $\mathrm{ub}(c|\mathsf{x}|) = c \cdot \mathrm{ub}(|\mathsf{x}|)$. If $c < 0$, then $\mathrm{lb}(c|\mathsf{x}|) = c{\cdot}\mathrm{ub}(|\mathsf{x}|)$ and $\mathrm{ub}(c|\mathsf{x}|) = c{\cdot}\mathrm{lb}(|\mathsf{x}|)$ instead. For sums of the form $T_1 + T_2$, we just use $\mathrm{lb}(T_1 + T_2) = \mathrm{lb}(T_1) + \mathrm{lb}(T_2)$ and $\mathrm{ub}(T_1 + T_2) = \mathrm{ub}(T_1) + \mathrm{ub}(T_2)$. Then, for any term T, $\mathrm{lb}(T)$ and $\mathrm{ub}(T)$ are the smallest and largest value T can assume when respecting the bounds that lb and ub impose on the variables:

Lemma 5. *Let* $|\mathsf{x}| = T$ *be* $[\alpha \doteq \beta]_L$ *solved for* x*, and* lb *and* ub *be bounds for* $[\alpha \doteq \beta]_L$*. If* g *is a solution for* $[\alpha \doteq \beta]_L$*, then* $\mathrm{lb}(T) \leq g(|\mathsf{x}|) \leq \mathrm{ub}(T)$ *holds.*

If lb and ub are lower and upper bounds on the solutions for $[\alpha \doteq \beta]_L$, and the linear constraints imply an equation of the form $|\mathsf{x}| = T$ where T does not contain $|\mathsf{x}|$, then $\mathrm{lb}(T)$ and $\mathrm{ub}(T)$ are lower and upper bounds for $|\mathsf{x}|$ in $[\alpha \doteq \beta]_L$. If these bounds improve on $\mathrm{lb}(|\mathsf{x}|)$ and $\mathrm{ub}(|\mathsf{x}|)$ then the procedure updates both and iterates.

VII. Experimental Evaluation

The NFA2SAT solver is written in Rust and uses the SAT solver CaDiCaL-1.5.2. The source code consists of about 18k lines of Rust. Compared with the earlier version described in a previous paper [20], we have made several extensions to support word equations. First, the input formula φ is rewritten into an equivalent formula in normal form in which all literals are either word equations, inequalities between variables, or (negated) regular constraints (see Section II). The propositional encoding of regular constraints and inequalities between variables is explained in [20] and has not changed. Word equations are encoded as explained in Section V and the alphabet reduction is implemented as explained in Section IV. The bound refinement technique explained in Section VI is used between SAT solver invocations to obtain small model bounds and handle unsatisfiable instances, if the UNSAT core contains (encoded) word equations.

Some types of negated constraints cannot be encoded directly because they implicitly introduce universal quantifiers. For example, the literal $\neg\, \mathtt{contains}(a{\cdot}\mathsf{x}{\cdot}b, \mathsf{y})$ with variables x and y, and constants a and b, is equivalent to $\forall \mathsf{z}_1 \mathsf{z}_2.\ \mathsf{z}_1{\cdot}\mathsf{y}{\cdot}\mathsf{z}_2 \neq a{\cdot}\mathsf{x}{\cdot}b$. We handle such constraints lazily using a CEGAR-style approach: NFA2SAT tries to find a solution that ignores these types of constraints. If a solution is found, the solver checks whether it satisfies the negated constraints that were ignored. If so, the original formula is satisfiable. If some of the unhandled negated constraints are not true, then we restart NFA2SAT with a constraint that forces it to search for another solution.

We compare NFA2SAT with CVC5 (version 1.1.1), Z3 (version 4.13.0), NOODLER (commit #e1e46068) and OSTRICH (commit #f7f0aa8c). We also include results from NFA2SAT when the bound refinement is disabled. We run the experiments on an Amazon EC2 M5.24xlarge instance running Amazon Linux 2, equipped with 384 GB RAM and 96 Intel Xeon CPUs running at 2.50 GHz. We ran 48 solvers in parallel, with a 300 second timeout and a 16 GB memory limit per problem.
We have evaluated our approach on the ZaligVinder [19] benchmark set[1]. The set contains 82,632 problems from different sources and includes all string problems from SMT-LIB. Out of these problems, 33,091 are in the logical fragment supported by NFA2SAT. The others include constraints currently unsupported by NFA2SAT, e.g., constraints on string lengths.

Table I summarizes the results. The table shows the number of satisfiable and unsatisfiable problems solved by each solver. It also includes the total runtime of each solver on the problems it successfully solves. On these benchmarks, NFA2SAT is competitive with CVC5, Z3, and OSTRICH. NOODLER is faster overall than the other solvers by a significant margin. The baseline version of NFA2SAT solves more problems in total than CVC5, but fewer than Z3, OSTRICH, and NOODLER. NOODLER solves the most problems overall. The table also shows that the bound refinement heuristic helps performance on both satisfiable and unsatisfiable instances. It increases the number of solved problems by 39 and reduces the total runtime by 6,138 seconds. The table shows that the solvers have different characteristics. CVC5 is faster than the other solvers on satisfiable problems but it is slower on unsatisfiable instances. Conversely, Z3, OSTRICH, and both versions of NFA2SAT are slower overall on satisfiable instances. NOODLER is the fastest solver on unsatisfiable problems. NFA2SAT comes second on unsatisfiable problems, but solves fewer problems than NOODLER. OSTRICH is close to NOODLER in terms of the number of solved unsatisfiable problems but it is slower. The table also shows that NFA2SAT is faster on average than Z3, OSTRICH, and CVC5 on the problems that it can solve, only NOODLER is faster.

[1] Available at https://github.com/zaligvinder/zaligvinder

TABLE I: Results on the ZaligVinder Benchmarks. NFA2SAT is our baseline solver. NFA2SAT (no ref) is the same solver with bound refinement disabled.

	Solved Problems			Runtimes Total (s)			Runtimes Average (s)		
Solver	Sat	Unsat	Total	Sat	Unsat	Total	Sat	Unsat	Total
CVC5-1.1.1	25,443	7,058	32,501	1,250.40	34,285.09	35,535.49	0.05	4.86	1.09
Z3-4.13.0	25,480	7,164	32,644	43,177.51	3,083.34	46,260.85	1.69	0.43	1.42
ostrich	25,439	7,388	32,827	107,786.97	62,183.89	169,970.86	4.24	8.42	5.18
noodler	25,536	7,539	33,075	1,276.80	407.51	1,684.31	0.05	0.05	0.05
NFA2SAT	25,406	7,118	32,524	15,276.99	1,389.70	17,666.69	0.6	0.2	0.54
NFA2SAT (no ref)	25,384	7,101	32,485	21,624.50	2,180.07	23,804.57	0.85	0.31	0.73

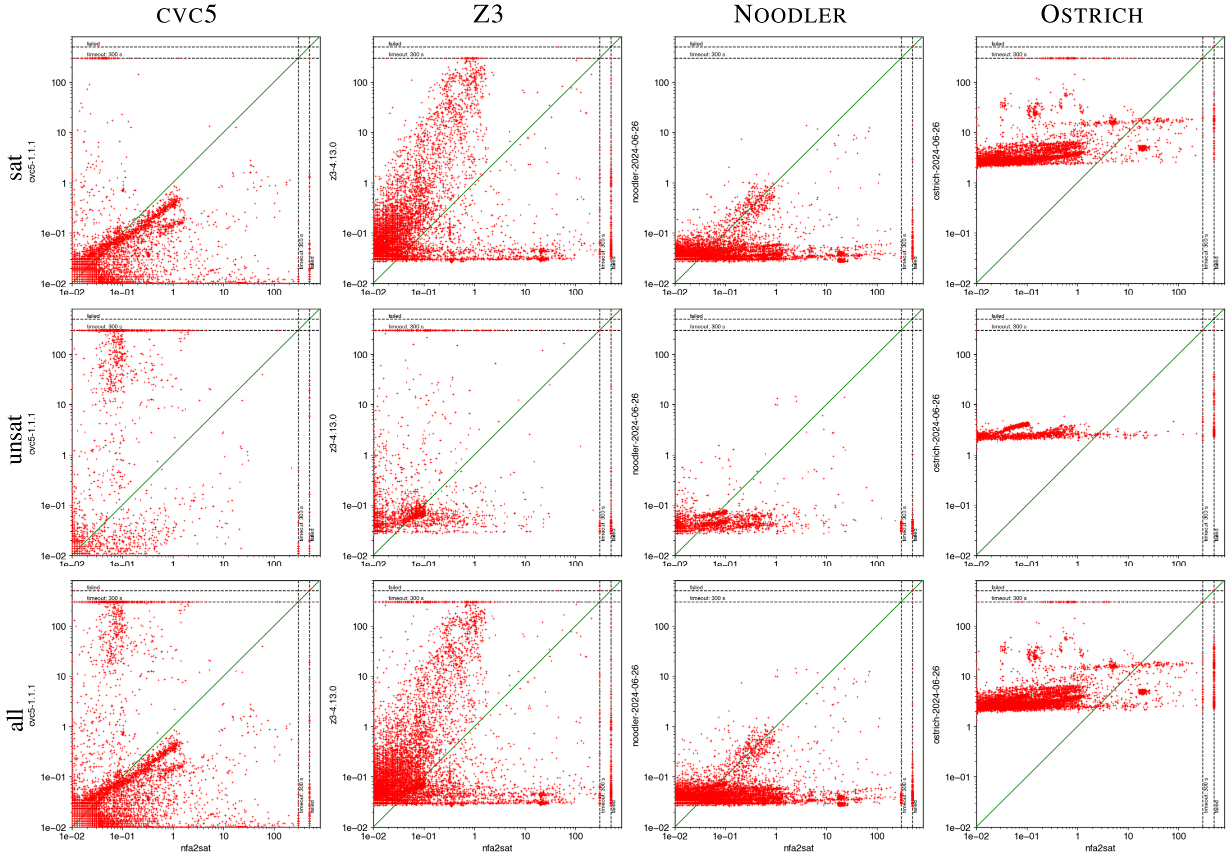


Fig. 6: Scatter plots comparing NFA2SAT (x-axis) with CVC5, Z3, NOODLER, and OSTRICH (y-axis). The first row contains only satisfiable, the second row only unsatisfiable, and the last row all problems. The axes are on a logarithmic scale. The diagonal line represents equal runtime. Points above the diagonal are problems where NFA2SAT is faster. The first dashed line represents timeouts. The second dashed line represents failures (crashes, out-of-memory).

The scatter plots in Figure 6 show that the techniques employed by NFA2SAT and the other solvers are complementary. Every column compares NFA2SAT with a different solver. The leftmost plots show that CVC5 is generally faster on satisfiable examples (points below the diagonal), but not on all problems. The converse happens on unsatisfiable problems (second row). One can also see that CVC5 and NFA2SAT are good on different sets of unsatisfiable benchmarks: the plot for unsatisfiable problems does not have many points close to the diagonal. This behavior is even more pronounced when we compare NFA2SAT and Z3 (second column). The plots show a pattern where some benchmarks are easier for NFA2SAT and others are easier for Z3, with not many points along the diagonal. Many problems solved by NFA2SAT in less than 1 second are harder for Z3, and conversely, many problems solved by Z3 in less than 0.1 seconds are hard for NFA2SAT. We can also see that Z3 has a higher startup cost than NFA2SAT and CVC5 on these problems. The plots comparing NFA2SAT with NOODLER show a pattern similar to the comparison with CVC5, but NOODLER is faster overall. For satisfiable problems, there are many cases along the diagonal, indicating that the solvers perform equally well on these benchmarks. However, there is also a large set of problems where NOODLER is faster, as shown by the concentration

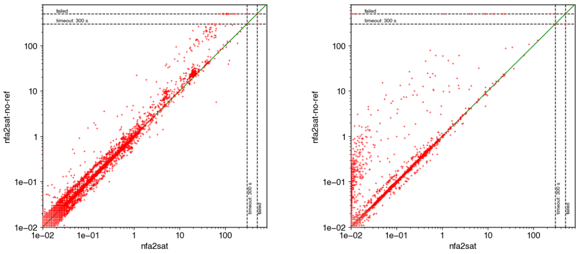


Fig. 7: NFA2SAT with and without bound refinement on SAT and UNSAT instances, respectively.

of points below the diagonal. Despite this, NFA2SAT still outperforms NOODLER on a subset of problems, both satisfiable and unsatisfiable. NOODLER is based on Z3, and thus shares the same startup cost. The plots comparing NFA2SAT with OSTRICH indicate that NFA2SAT is overall faster on most problems, both satisfiable and unsatisfiable, as shown by the majority of points above the diagonal. For many of these problems, NFA2SAT's advantage can be attributed to OSTRICH's high startup time, which is about 2 seconds. There is a significant number of both satisfiable and unsatisfiable problems where OSTRICH is faster, or which OSTRICHsolved but NFA2SAT could not solve. On unsatisfiable problems, OSTRICH solves more problems than NFA2SAT. Only a few instances are close to the diagonal, emphasizing that NFA2SAT and OSTRICH complement each other.

The scatter plots in Figure 7 show the impact of the bound-refinement heuristics. The plot shows many points close to the diagonal, which are problems where bound refinement does not help or hurt. But most of the other points are above the diagonal. These are problems where bound refinement improves runtime.

VIII. Conclusion

We have added support for word equations to the NFA2SAT string solver. Our approach relies on a novel SAT encoding of word equations that is based on enumerating constant words and matching both sides of a word equation to the same constant word. The encoding makes use of incremental SAT solving. To detect unsatisfiable instances, we propose an incomplete but practical technique that derives linear constraints on the length of variables occurring in word equations and uses a bound-refinement algorithm. An empirical evaluation on a large set of benchmarks demonstrates that our approach is competitive with the state-of-the-art solvers CVC5 and Z3. More important, the techniques employed by NFA2SAT are complementary which brings benefits to portfolio-solving strategies. In future work, we plan to support atoms that constrain the lengths of strings. Additionally, we want to explore a more diverse array of approaches to determine unsatisfiability and optimize the SAT encoding in order to improve the solver's efficiency.

References

[1] Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8602994

[2] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24

[3] Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`

[4] Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: String theories involving regular membership predicates: From practice to theory and back. In: Lecroq, T., Puzynina, S. (eds.) Combinatorics on Words. pp. 50–64. Springer International Publishing, Cham (2021)

[5] Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498707, https://doi.org/10.1145/3498707

[6] Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-noodler: An automata-based string solver. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 24–33. Springer Nature Switzerland, Cham (2024)

[7] Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On Solving Word Equations Using SAT, p. 93–106. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-30806-3_8

[8] Diekert, V.: Makanin's algorithm for solving word equations with regular constraints. Tech. Rep. 1998/02, University of Stuttgart (March 1998). https://doi.org/10.18419/opus-2419, https://elib.uni-stuttgart.de/bitstream/11682/2436/1/420_1.pdf

[9] Diestel, R.: Graph Theory, Graduate Texts in Mathematics, vol. 173. Springer (1997)

[10] Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4), 543–560 (2003). https://doi.org/10.1016/S1571-0661(05)82542-3, bMC'2003, First International Workshop on Bounded Model Checking

[11] Hojjat, H., Rümmer, P., Shamakhi, A.: On strings in software model checking. In: Lin, A.W. (ed.) Programming Languages and Systems. pp. 19–30. Springer International Publishing, Cham (2019)

[12] Jez, A.: Word Equations in Nondeterministic Linear Space. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 80, pp. 95:1–95:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.95

[13] Jovanović, D., de Moura, L.: Cutting to the Chase Solving Linear Integer Arithmetic, pp. 338–353. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_26

[14] Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: CertiStr: a certified string solver. In: Popescu, A., Zdancewic, S. (eds.) CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022. pp. 210–224. ACM (2022). https://doi.org/10.1145/3497775.3503691

[15] Karhumäki, J., Mignosi, F., Plandowski, W.: The expressibility of languages and relations by word equations. J. ACM **47**(3), 483–505 (may 2000). https://doi.org/10.1145/337244.337255

[16] Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: A solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. p. 105–116. ISSTA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1572272.1572286

[17] Klieber, W., Kwon, G.: Efficient CNF encoding for selecting 1 from N objects. In: Fourth Workshop on Constraints in Formal Verification (CFV) (2007)

[18] Kulczynski, M., Lotz, K., Nowotka, D., Poulsen, D.B.: Solving string theories involving regular membership predicates using SAT. In: Legunsen, O., Rosu, G. (eds.) Model Checking Software. pp. 134–151. Springer International Publishing, Cham (2022)

[19] Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: ZaligVinder: A generic test framework for string solvers. Journal of Software: Evolution and Process **35**(4), e2400 (2023). https://doi.org/https://doi.org/10.1002/smr.2400

[20] Lotz, K., Goel, A., Dutertre, B., Kiesl-Reiter, B., Kong, S., Majumdar, R., Nowotka, D.: Solving string constraints using SAT. In: Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II. pp. 187–208. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-37703-7_9

[21] Lu, Z., Siemer, S., Jha, P., Manea, F., Day, J., Ganesh, V.: Z3-alpha: a reinforcement learning guided smt solver. System Description: SMT-COMP 2023 (July 2023), https://smt-comp.github.io/2023/system-descriptions/z3-alpha.pdf

[22] Makanin, G.S.: The problem of solvability of equations in a free semigroup. Mathematics of The Ussr-sbornik **32**, 129–198 (1977), https://api.semanticscholar.org/CorpusID:7073856

[23] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)

[24] Plandowski, W.: Satisfiability of word equations with constants is in nexptime. In: Proceedings of the thirty-first annual ACM symposium on Theory of Computing. STOC99, ACM (May 1999). https://doi.org/10.1145/301250.301443

[25] Plandowski, W., Rytter, W.: Application of Lempel-Ziv encodings to the solution of word equations, p. 731–742. Springer Berlin Heidelberg (1998). https://doi.org/10.1007/bfb0055097

[26] Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 3–18. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_1

[27] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: 2010 IEEE Symposium on Security and Privacy. pp. 513–528 (2010). https://doi.org/10.1109/SP.2010.38

[28] Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization, pp. 85–150. Springer Berlin Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_4

[29] Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods in System Design **44**(1), 44–70 (2014). https://doi.org/10.1007/s10703-013-0189-1