Solving String Constraints Using SAT

Kevin Lotz[1], Amit Goel[2], Bruno Dutertre[2], Benjamin Kiesl-Reiter[2], Soonho Kong[2], Rupak Majumdar[2], and Dirk Nowotka[1]

[1] Department of Computer Science, Kiel University, Kiel, Germany
{kel,dn}@informatik.uni-kiel.com
[2] Amazon Web Services
{amgoel,dutebrun,benkiesl,soonho,rumajumd}@amazon.com

Abstract. String solvers are automated-reasoning tools that can solve combinatorial problems over formal languages. They typically operate on restricted first-order logic formulas that include operations such as string concatenation, substring relationship, and regular expression matching. String solving thus amounts to deciding the satisfiability of such formulas. While there exists a variety of different string solvers, many string problems cannot be solved efficiently by any of them. We present a new approach to string solving that encodes input problems into propositional logic and leverages incremental SAT solving. We evaluate our approach on a broad set of benchmarks. On the logical fragment that our tool supports, it is competitive with state-of-the-art solvers. Our experiments also demonstrate that an eager SAT-based approach complements existing approaches to string solving in this specific fragment.

1 Introduction

Many problems in software verification require reasoning about strings. To tackle these problems, numerous *string solvers*—automated decision procedures for quantifier-free first-order theories of strings and string operations—have been developed over the last years. These solvers form the workhorse of automated-reasoning tools in several domains, including web-application security [19,31,33], software model checking [15], and conformance checking for cloud-access-control policies [2,30].

The general theory of strings relies on deep results in combinatorics on words [23,29,16,5]; unfortunately, the related decision procedures remain intractable in practice. Practical string solvers achieve scalability through a judicious mix of heuristics and restrictions on the language of constraints.

We present a new approach to string solving that relies on an *eager* reduction to the Boolean satisfiability problem (SAT), using incremental solving and unsatisfiable-core analysis for completeness and scalability. Our approach supports a theory that contains Boolean combinations of regular membership constraints and equality constraints on string variables, and captures a large set of practical queries [6].

Our solving method iteratively searches for satisfying assignments up to a length bound on each string variable; it stops and reports unsatisfiability when

the search reaches computed upper bounds without finding a solution. Similar to the solver WOORPJE [12], we formulate regular membership constraints as reachability problems in nondeterministic finite automata. By bounding the number of transitions allowed by each automaton, we obtain a finite problem that we encode into propositional logic. To cut down the search space of the underlying SAT problem, we perform an *alphabet reduction* step (SMT-LIB string constraints are defined over an alphabet of $3 \cdot 2^{16}$ letters and a naive reduction to SAT does not scale). Inspired by bounded model checking [8], we iteratively increase bounds and utilize an incremental SAT solver to solve the resulting series of propositional formulas. We perform an unsatisfiable-core analysis after each unsatisfiable incremental call to increase only the bounds of a minimal subset of variables until a theoretical upper bound is reached.

We have evaluated our solver on a large set of benchmarks. The results show that our SAT-based approach is competitive with state-of-the-art SMT solvers in the logical fragment that we support. It is particularly effective on satisfiable instances.

Closest to our work is the WOORPJE solver [12], which also employs an eager reduction to SAT. WOORPJE reduces systems of word equations with linear constraints to a single Boolean formula and calls a SAT solver. An extension can also handle regular membership constraints [21]. However, WOORPJE does not handle the full language of constraints considered here and does not employ the reduction and incremental solving techniques that make our tool scale in practice. More importantly, in contrast to our solver, WOORPJE is not complete—it does not terminate on unsatisfiable instances.

Other solvers such as Hampi [19] and Kaluza [31] encode string problems into constraints on fixed-size bit-vector, which can be solved by reduction to SAT. These tools support expressive constraints but they require a user-provided bound on the length of string variables.

Further from our work are approaches based on the *lazy* SMT paradigm, which tightly integrates dedicated, heuristic, theory solvers for strings using the CDCL(T) architecture (also called DPLL(T) in early papers). Solvers that follow this paradigm include OSTRICH [11], Z3 [25], Z3STR4 [24], CVC5 [3], Z3STR3RE [7], TRAU [1], and CERTISTR [17]. Our evaluation shows that our eager approach is competitive with lazy solvers overall, but it also shows that combining both types of solvers in a portfolio is most effective. Our eager approach tends to perform best on satisfiable instances while lazy approaches work better on unsatisfiable problems.

2 Preliminaries

We assume a fixed alphabet Σ and a fixed set of variables Γ. Words of Σ^* are denoted by w, w', w'', etc. Variables are denoted by $\mathsf{x}, \mathsf{y}, \mathsf{z}$. Our decision procedure supports the theory described in Figure 1. Atoms in this theory include *regular membership constraints* (or *regular constraints* for short) of the form $\mathsf{x} \dot{\in} RE$,

$$F := F \vee F \mid F \wedge F \mid \neg F \mid \text{Atom}$$
$$\text{Atom} := \mathsf{x} \mathrel{\dot\in} RE \mid \mathsf{x} \doteq \mathsf{y}$$
$$RE := RE \cup RE \mid RE \cdot RE \mid RE^* \mid RE \cap RE \mid ? \mid w$$

Fig. 1: Syntax: x and y denote string variables and w denotes a word of Σ^*. The symbol ? is the wildcard character.

where RE is a regular expression, and *variable equations* of the form $\mathsf{x} \doteq \mathsf{y}$. Concatenation is not allowed in equations.

Regular expressions are defined inductively using union, concatenation, intersection, and the Kleene star. Atomic regular expressions are constant words $w \in \Sigma^*$ and the wildcard character ?, which is a placeholder for an arbitrary symbol $c \in \Sigma$. All regular expressions are grounded, meaning that they do not contain variables. We use the symbols $\dot\notin$ and $\dot\neq$ as a shorthand notation for negations of atoms using the respective predicate symbols. The following is an example formula in our language: $\neg(\mathsf{x} \mathrel{\dot\in} \mathrm{a} \cdot ?^* \wedge \mathsf{y} \mathrel{\dot\in} ?^* \cdot \mathrm{b}) \vee \mathsf{x} \mathrel{\dot\neq} \mathsf{y} \vee \mathsf{x} \mathrel{\dot\in} \mathrm{a} \cdot \mathrm{b}$.

Using our basic syntax, we can define additional relations, such as *constant equations* $\mathsf{x} \doteq w$, and *prefix and suffix constraints*, written $w \mathrel{\dot\sqsubseteq} \mathsf{x}$ and $w \mathrel{\dot\sqsupseteq} \mathsf{x}$, respectively. Even though these relations can be expressed as regular constraints (e.g., the prefix constraint $\mathrm{ab} \mathrel{\dot\sqsubseteq} \mathsf{x}$ can be expressed as $\mathsf{x} \mathrel{\dot\in} \mathrm{a} \cdot \mathrm{b} \cdot ?^*$), we can generate more efficient reductions to SAT by encoding them explicitly.

This string theory is not as expressive as others, since it does not include string concatenation, but it still has important practical applications. It is used in the Zelkova tool described by Backes, et al. [2] to support analysis of AWS security policies. Zelkova is a major industrial application of SMT solvers [30].

Given a formula ψ, we denote by $atoms(\psi)$ the set of atoms occurring in ψ, by $V(\psi)$ the set of variables occurring in ψ, and by $\Sigma(\psi)$ the set of constant symbols occurring in ψ. We call $\Sigma(\psi)$ the *alphabet of* ψ. Similarly, given a regular expression R, we denote by $\Sigma(R)$ the set of characters occurring in R. In particular, we have $\Sigma(?) = \emptyset$.

We call a formula *conjunctive* if it is a conjunction of literals and we call it a *clause* if it is a disjunction of literals. We say that a formula is in *normal form* if it is a conjunctive formula without unnegated variable equations. Every conjunctive formula can be turned into normal form by substitution, i.e., by repeatedly rewriting $\psi \wedge \mathsf{x} \doteq \mathsf{y}$ to $\psi[\mathsf{x} := \mathsf{y}]$. If ψ is in negation normal form (NNF), meaning that the negation symbol occurs only directly in front of atoms, we denote by $lits(\psi)$ the set of literals occurring in ψ. We say that an atom a occurs with *positive polarity* in ψ if $a \in lits(\psi)$ and that it occurs with *negative polarity* in ψ if $\neg a \in lits(\psi)$; we denote the respective sets of atoms of ψ by $atoms^+(\psi)$ and $atoms^-(\psi)$. The notion of polarity can be extended to arbitrary formulas (not necessarily in NNF), intuitively by considering polarity in a formula's corresponding NNF (see [26] for details).

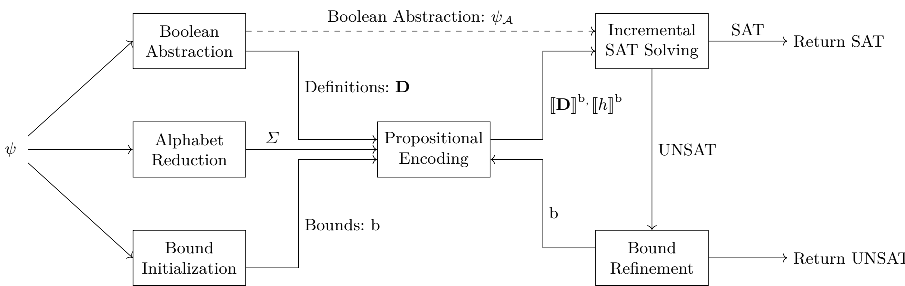


Fig. 2: Overview of the solving process.

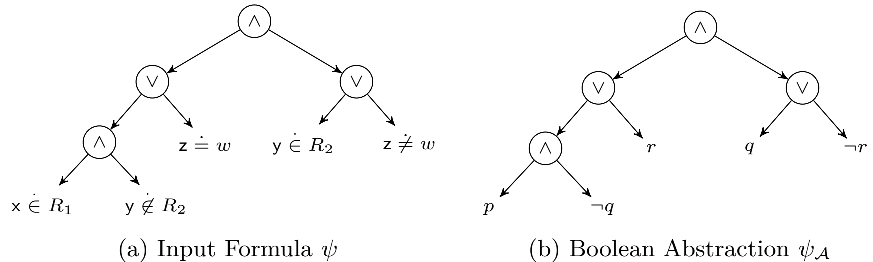
The semantics of our language is standard. A regular expression R defines a regular language $\mathcal{L}(R)$ over Σ in the usual way. An *interpretation* is a mapping (also called a *substitution*) $h\colon \Gamma \to \Sigma^*$ from string variables to words. Atoms are interpreted as usual, and a *model* (also called a *solution*) is an interpretation that makes a formula evaluate to true under the usual semantics of the Boolean connectives.

3 Overview

Our solving method is illustrated in Figure 2. It first performs three preprocessing steps that generate a Boolean abstraction of the input formula, reduce the size of the input alphabet, and initialize bounds on the lengths of all string variables. After preprocessing, we enter an encode-solve-and-refine loop that iteratively queries a SAT solver with a problem encoding based on the current bounds and refines the bounds after each unsatisfiable solver call. We repeat this loop until either the propositional encoding is satisfiable, in which case we conclude satisfiability of the input formula, or each bound has reached a theoretical upper bound, in which case we conclude unsatisfiability.

Generating the Boolean Abstraction. We abstract the input formula ψ by replacing each theory atom $a \in atoms(\psi)$ with a new Boolean variable $\mathbf{d}(a)$, and keep track of the mapping between a and $\mathbf{d}(a)$. This gives us a *Boolean abstraction* $\psi_{\mathcal{A}}$ of ψ and a set $\mathbf{D}$ of definitions, where each definition expresses the relationship between an atom a and its corresponding Boolean variable $\mathbf{d}(a)$. If a occurs with only one polarity in ψ, we encode the corresponding definition as an implication, i.e., as $\mathbf{d}(a) \to a$ or as $\neg\,\mathbf{d}(a) \to \neg a$, depending on the polarity of a. Otherwise, if a occurs with both polarities, we encode it as an equivalence consisting of both implications. This encoding, which is based on ideas behind the well-known *Plaisted-Greenbaum transformation* [28], ensures that the formulas ψ and $\psi_{\mathcal{A}} \wedge \bigwedge_{d\in\mathbf{D}} d$ are equisatisfiable. An example is shown in Figure 3.

Reducing the Alphabet. In the SMT-LIB theory of strings [4], the alphabet Σ comprises $3{\cdot}2^{16}$ letters, but we can typically use a much smaller alphabet without



(a) Input Formula ψ (b) Boolean Abstraction $\psi_{\mathcal{A}}$

Fig. 3: Example of Boolean abstraction. The formula ψ, whose expression tree is shown on the left, results in the Boolean abstraction illustrated on the right, where p, q, and r are fresh Boolean variables. We additionally get the definitions $p \rightarrow \mathsf{x} \mathrel{\dot\in} R_1$, $q \leftrightarrow \mathsf{y} \mathrel{\dot\in} R_2$, and $r \leftrightarrow \mathsf{z} \doteq w$. We use an implication (instead of an equivalence) for atom $\mathsf{x} \mathrel{\dot\in} R_1$ since it occurs only with positive polarity within ψ.

affecting satisfiability. In Section 4, we show that using $\Sigma(\psi)$ and one extra character per string variable is sufficient. Reducing the alphabet is critical for our SAT encoding to be practical.

Initializing Bounds. A model for the original first-order formula ψ is a substitution $h : \Gamma \rightarrow \Sigma^*$ that maps each string variable to a word of arbitrary length such that ψ evaluates to true. As we use a SAT solver to find such substitutions, we need to bound the lengths of strings, which we do by defining a bound function $\mathrm{b} : \Gamma \rightarrow \mathbb{N}$ that assigns an upper bound to each string variable. We initialize a small upper bound for each variable, relying on simple heuristics. If the bounds are too small, we increase them in a later refinement step.

Encoding, Solving, and Refining Bounds. Given a bound function b, we build a propositional formula $[\![\psi]\!]^{\mathrm{b}}$ that is satisfiable if and only if the original formula ψ has a solution h such that $|h(\mathsf{x})| \leq \mathrm{b}(\mathsf{x})$ for all $\mathsf{x} \in \Gamma$. We encode $[\![\psi]\!]^{\mathrm{b}}$ as the conjunction $\psi_{\mathcal{A}} \wedge [\![\mathbf{D}]\!]^{\mathrm{b}} \wedge [\![h]\!]^{\mathrm{b}}$, where $\psi_{\mathcal{A}}$ is the Boolean abstraction of ψ, $[\![\mathbf{D}]\!]^{\mathrm{b}}$ is an encoding of the definitions $\mathbf{D}$, and $[\![h]\!]^{\mathrm{b}}$ is an encoding of the set of possible substitutions. We discuss details of the encoding in Section 5. A key property is that it relies on *incremental SAT solving under assumptions* [13]. Increasing bounds amounts to adding new clauses to the formula $[\![\psi]\!]^{\mathrm{b}}$ and fixing a set of assumptions, i.e., temporarily fixing the truth values of a set of Boolean variables. If $[\![\psi]\!]^{\mathrm{b}}$ is satisfiable, we can construct a substitution h from a Boolean model ω of $[\![\psi]\!]^{\mathrm{b}}$. Otherwise, we examine an unsatisfiable core (i.e., an unsatisfiable subformula) of $[\![\psi]\!]^{\mathrm{b}}$ to determine whether increasing the bounds may give a solution and, if so, to identify the variables whose bounds must be increased. In Section 6, we explain in detail how we analyze unsatisfiable cores, increase bounds, and conclude unsatisfiability.

4 Reducing the Alphabet

In many applications, the alphabet Σ is large—typically Unicode or an approximation of Unicode as defined in the SMT-LIB standard—but formulas use much

fewer symbols (less than 100 symbols is common in our experiments). In order to check the satisfiability of a formula ψ, we can restrict the alphabet to the symbols that occur in ψ and add one extra character per variable. This allows us to produce compact propositional encodings that can be solved efficiently in practice.

To prove that such a reduced alphabet A is sufficient, we show that a model $h\colon \Gamma \to \Sigma^*$ of ψ can be transformed into a model $h'\colon \Gamma \to A^*$ of ψ by replacing characters of Σ that do not occur in ψ by new symbols—one new symbol per variable of ψ. For example, suppose $V(\psi) = \{\mathsf{x}_1, \mathsf{x}_2\}$, $\Sigma(\psi) = \{\mathrm{a}, \mathrm{c}, \mathrm{d}\}$, and h is a model of ψ such that $h(\mathsf{x}_1) = \mathrm{abcdef}$ and $h(\mathsf{x}_2) = \mathrm{abbd}$. We introduce two new symbols $\alpha_1, \alpha_2 \in \Sigma \setminus \Sigma(\psi)$, define $h'(\mathsf{x}_1) = \mathrm{a}\alpha_1\mathrm{cd}\alpha_1\alpha_1$ and $h'(\mathsf{x}_2) = \mathrm{a}\alpha_2\alpha_2\mathrm{d}$, and argue that h' is a model as well.

More generally, assume B is a subset of Σ and n is a positive integer such that $|B| \leq |\Sigma| - n$. We can then pick n distinct symbols $\alpha_1, \ldots, \alpha_n$ from $\Sigma \setminus B$. Let A be the set $B \cup \{\alpha_1, \ldots, \alpha_n\}$. We construct n functions $f_1, \ldots, f_n$ from Σ to A by setting $f_i(a) = a$ if $a \in B$, and $f_i(a) = \alpha_i$ otherwise. We extend f_i to words of Σ^* in the natural way: $f_i(\varepsilon) = \varepsilon$ and $f_i(a \cdot w) = f_i(a) \cdot f_i(w)$. This construction satisfies the following property:

Lemma 4.1. *Let $f_1, \ldots, f_n$ be mappings as defined above, and let $i, j \in 1, \ldots, n$ such that $i \neq j$. Then, the following holds:*

1. *If a and b are distinct symbols of Σ, then $f_i(a) \neq f_j(b)$.*
2. *If w and w' are distinct words of Σ^*, then $f_i(w) \neq f_j(w')$.*

Proof. The first part is an easy case analysis. For the second part, we have that $|f_i(w)| = |w|$ and $|f_j(w')| = |w'|$, so the statement holds if w and w' have different lengths. Assume now that w and w' have the same length and let v be the longest common prefix of w and w'. Since w and w' are distinct, we have that $w = v \cdot a \cdot u$ and $w' = v \cdot b \cdot u'$, where $a \neq b$ are symbols of Σ and u and u' are words of Σ^*. By the first part, we have $f_i(a) \neq f_j(b)$, so $f_i(w)$ and $f_j(w')$ must be distinct. □

The following lemma can be proved by induction on R.

Lemma 4.2. *Let $f_1, \ldots, f_n$ be mappings as defined above and let R be a regular expression with $\Sigma(R) \subseteq B$. Then, for all words $w \in \Sigma^*$ and all $i \in 1, \ldots, n$, $w \in \mathcal{L}(R)$ if and only if $f_i(w) \in \mathcal{L}(R)$.*

Given a subset A of Σ, we say that ψ is satisfiable in A if there is a model $h\colon V(\psi) \to A^*$ of ψ. We can now prove the main theorem of this section, which shows how to reduce the alphabet while maintaining satisfiability.

Theorem 4.3. *Let ψ be a formula with at most n string variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$ such that $|\Sigma(\psi)| + n \leq |\Sigma|$. Then, ψ is satisfiable if and only if it is satisfiable in an alphabet $A \subseteq \Sigma$ of cardinality $|A| = |\Sigma(\psi)| + n$.*

Proof. We set $B = \Sigma(\psi)$ and use the previous construction. So the alphabet $A = B \cup \{\alpha_1, \ldots, \alpha_n\}$ has cardinality $|\Sigma(\psi)| + n$, where $\alpha_1, \ldots \alpha_n$ are distinct

symbols of $\Sigma \setminus B$. We can assume that ψ is in disjunctive normal form, meaning that it is a disjunction of the form $\psi = \psi_1 \vee \cdots \vee \psi_m$, where each ψ_t is a conjunctive formula. If ψ is satisfiable, then one of the disjuncts ψ_k is satisfiable and we have $\Sigma(\psi_k) \subseteq B$. We can turn ψ_k into normal form by eliminating all variable equalities of the form $\mathsf{x}_i \doteq \mathsf{x}_j$ from ψ_k, resulting in a conjunction φ_k of literals of the form $\mathsf{x}_i \mathrel{\dot\in} R$, $\mathsf{x}_i \mathrel{\dot\notin} R$, or $\mathsf{x}_i \mathrel{\dot\neq} \mathsf{x}_j$. Clearly, for any $A \subseteq \Sigma$, φ_k is satisfiable in A if and only if ψ_k is satisfiable in A.

Let $h\colon V(\varphi_k) \to \Sigma^*$ be a model of φ_k and define the mapping $h'\colon V(\varphi_k) \to A^*$ as $h'(\mathsf{x}_i) = f_i(h(\mathsf{x}_i))$. We show that h' is a model of φ_k. Consider a literal l of φ_k. We have three cases:

- l is of the form $\mathsf{x}_i \mathrel{\dot\in} R$ where $\Sigma(R) \subseteq \Sigma(\psi) = B$. Since h satisfies φ_k, we must have $h(\mathsf{x}_i) \in \mathcal{L}(R)$ so $h'(\mathsf{x}_i) = f_i(h(\mathsf{x}_i))$ is also in $\mathcal{L}(\mathcal{R})$ by Lemma 4.2.
- l is of the form $\mathsf{x}_i \mathrel{\dot\notin} R$ with $\Sigma(R) \subseteq B$. Then, $h'(\mathsf{x}_i) \notin \mathcal{L}(R)$ and we can conclude $h'(\mathsf{x}_i) \notin \mathcal{L}(R)$ again by Lemma 4.2.
- l is of the form $\mathsf{x}_i \mathrel{\dot\neq} \mathsf{x}_j$. Since h satisfies φ_k, we must have $i \neq j$ and $h(\mathsf{x}_i) \neq h(\mathsf{x}_j)$, which implies $h'(\mathsf{x}_i) = f_i(h(\mathsf{x}_i)) \neq f_j(h(\mathsf{x}_j)) = h'(\mathsf{x}_j)$ by Lemma 4.1.

All literals of φ_k are then satisfied by h', hence φ_k is satisfiable in A and thus so is ψ_k. It follows that ψ is satisfiable in A. ☐

The reduction presented here can be improved and generalized. For example, it can be worthwhile to use different alphabets for different variables or to reduce large character intervals to smaller sets.

5 Propositional Encodings

Our algorithm performs a series of calls to a SAT solver. Each call determines the satisfiability of the propositional encoding $[\![\psi]\!]^{\mathrm{b}}$ of ψ for some upper bounds b. Recall that $[\![\psi]\!]^{\mathrm{b}} = \psi_{\mathcal{A}} \wedge [\![h]\!]^{\mathrm{b}} \wedge [\![\mathbf{D}]\!]^{\mathrm{b}}$, where $\psi_{\mathcal{A}}$ is the Boolean abstraction of ψ, $[\![h]\!]^{\mathrm{b}}$ is an encoding of the set of possible substitutions, and $[\![\mathbf{D}]\!]^{\mathrm{b}}$ is an encoding of the theory-literal definitions, both bounded by b. Intuitively, $[\![h]\!]^{\mathrm{b}}$ tells the SAT solver to "guess" a substitution, $[\![\mathbf{D}]\!]^{\mathrm{b}}$ makes sure that all theory literals are assigned proper truth values according to the substitution, and $\psi_{\mathcal{A}}$ forces the evaluation of the whole formula under these truth values.

Suppose the algorithm performs n calls and let $\mathrm{b}_k : \Gamma \to \mathbb{N}$ for $k \in 1, \ldots, n$ denote the upper bounds used in the k-th call to the SAT solver. For convenience, we additionally define $\mathrm{b}_0(\mathsf{x}) = 0$ for all $\mathsf{x} \in \Gamma$. In the k-th call, the SAT solver decides whether $[\![\psi]\!]^{\mathrm{b}_k}$ is satisfiable. The Boolean abstraction $\psi_{\mathcal{A}}$, which we already discussed in Section 3, stays the same for each call. In the following, we thus discuss the encodings of the substitutions $[\![h]\!]^{\mathrm{b}_k}$ and of the various theory literals $[\![a]\!]^{\mathrm{b}_k}$ and $[\![\neg a]\!]^{\mathrm{b}_k}$ that are part of $[\![\mathbf{D}]\!]^{\mathrm{b}_k}$. Even though SAT solvers expect their input in CNF, we do not present the encodings in CNF to simplify the presentation, but they can be converted to CNF using simple equivalence transformations.

Most of our encodings are *incremental* in the sense that the formula for call k is constructed by only adding clauses to the formula for call $k-1$. In other words, for substitution encodings we have $[\![h]\!]^{\mathrm{b}_k} = [\![h]\!]^{\mathrm{b}_{k-1}} \wedge [\![h]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$ and for literals we have $[\![l]\!]^{\mathrm{b}_k} = [\![l]\!]^{\mathrm{b}_{k-1}} \wedge [\![l]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$, with the base case $[\![h]\!]^{\mathrm{b}_0} = [\![l]\!]^{\mathrm{b}_0} = \top$. In these cases, it is thus enough to encode the incremental additions $[\![l]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$ and $[\![h]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$ for each call to the SAT solver. Some of our encodings, however, introduce clauses that are valid only for a specific bound b_k and thus become invalid for larger bounds. We handle the deactivation of these encodings with *selector variables* as is common in incremental SAT solving.

Our encodings are correct in the following sense.[3]

Theorem 5.1. *Let l be a literal and let $\mathrm{b} : \Gamma \to \mathbb{N}$ be a bound function. Then, l has a model that is bounded by* b *if and only if $[\![h]\!]^{\mathrm{b}} \wedge [\![l]\!]^{\mathrm{b}}$ is satisfiable.*

5.1 Substitutions

We encode substitutions by defining for each variable $\mathsf{x} \in \Gamma$ the characters to which each of x's positions is mapped. Specifically, given x and its corresponding upper bound $\mathrm{b}(\mathsf{x})$, we represent the substitution $h(\mathsf{x})$ by introducing new variables $\mathsf{x}[1], \dots, \mathsf{x}[\mathrm{b}(\mathsf{x})]$, one for each symbol $h(\mathsf{x})[i]$ of the word $h(\mathsf{x})$. We call these variables *filler variables* and we denote the set of all filler variables by $\check{\Gamma}$. By introducing a new symbol $\lambda \notin \Sigma$, which stands for an unused filler variable, we can define h based on a substitution $\check{h} \colon \check{\Gamma} \to \Sigma_\lambda$ over the filler variables, where $\Sigma_\lambda = \Sigma \cup \{\lambda\}$:

$$h(\mathsf{x})[i] = \begin{cases} \varepsilon & \text{if } \check{h}(\mathsf{x}[i]) = \lambda \\ \check{h}(\mathsf{x}[i]) & \text{otherwise} \end{cases}$$

We use this representation of substitutions (known as “filling the positions” [18]) because it has a straightforward propositional encoding: For each variable $\mathsf{x} \in \Gamma$ and each position $i \in 1, \dots, \mathrm{b}(\mathsf{x})$, we create a set $\{h^a_{\mathsf{x}[i]} \mid a \in \Sigma_\lambda\}$ of Boolean variables, where $h^a_{\mathsf{x}[i]}$ is true if $\check{h}(\mathsf{x}[i]) = a$. We then use a propositional encoding of an *exactly-one* (EO) constraint (e.g., [20]) to assert that exactly one variable in this set must be true:

$$[\![h]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \bigwedge_{\mathsf{x} \in \Gamma} \bigwedge_{i=\mathrm{b}_{k-1}(\mathsf{x})+1}^{\mathrm{b}_k(\mathsf{x})} \mathrm{EO}(\{h^a_{\mathsf{x}[i]} \mid a \in \Sigma_\lambda\}) \tag{1}$$

$$\wedge \bigwedge_{\mathsf{x} \in \Gamma} \bigwedge_{i=\mathrm{b}_{k-1}(\mathsf{x})}^{\mathrm{b}_k(\mathsf{x})-1} h^\lambda_{\mathsf{x}[i]} \to h^\lambda_{\mathsf{x}[i+1]} \tag{2}$$

Constraint (2) prevents the SAT solver from considering filled substitutions that are equivalent modulo λ-substitutions—it enforces that if a position i is mapped

[3] Proof is omitted due to space constraints but made available for review purposes.

to λ, all following positions are mapped to λ too. For instance, $ab\lambda\lambda$, $a\lambda b\lambda$, and $\lambda\lambda ab$ all correspond to the same word ab, but our encoding allows only $ab\lambda\lambda$. Thus, every Boolean assignment ω that satisfies $[\![h]\!]^{\mathrm{b}}$ encodes exactly one substitution h_ω, and for every substitution h (bounded by b) there exists a corresponding assignment ω_h that satisfies $[\![h]\!]^{\mathrm{b}}$.

5.2 Theory Literals

The only theory literals of our core language are regular constraints ($\mathsf{x} \mathrel{\dot\in} R$) and variable equations ($\mathsf{x} \doteq \mathsf{y}$) with their negations. Constant equations ($\mathsf{x} \doteq w$) as well as prefix and suffix constraints ($w \mathrel{\dot\sqsubseteq} \mathsf{x}$ and $w \mathrel{\dot\sqsupseteq} \mathsf{x}$) could be expressed as regular constraints, but we encode them explicitly to improve performance.

Regular Constraints We encode a regular constraint $\mathsf{x} \mathrel{\dot\in} R$ by constructing a propositional formula that is true if and only if the word $h(\mathsf{x})$ is accepted by a specific nondeterministic finite automaton that accepts the language $\mathcal{L}(R)$. Let $\mathsf{x} \mathrel{\dot\in} R$ be a regular constraint and let $M = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton (with states Q, alphabet Σ, transition relation δ, initial state q_0, and accepting states F) that accepts $\mathcal{L}(R)$ and that additionally allows λ-self-transitions on every state. Given that λ is a placeholder for the empty symbol, λ-transitions do not change the language accepted by M. We allow them so that M performs exactly $\mathrm{b}(\mathsf{x})$ transitions, even for substitutions of length less than $\mathrm{b}(\mathsf{x})$. This reduces checking whether the automaton accepts a word to only evaluating the states reached after exactly $\mathrm{b}(\mathsf{x})$ transitions.

Given a model $\omega \models [\![h]\!]^{\mathrm{b}}$, we express the semantics of M in propositional logic by encoding which states are reachable after reading $h_\omega(\mathsf{x})$. To this end, we assign $\mathrm{b}(\mathsf{x}) + 1$ Boolean variables $\{S_q^0, S_q^1, \ldots, S_q^{\mathrm{b}(\mathsf{x})}\}$ to each state $q \in Q$ and assert that $\omega_h(S_q^i) = 1$ if and only if q can be reached by reading prefix $h_\omega(\mathsf{x})[1..i]$. We encode this as a conjunction $[\![(M;\mathsf{x})]\!] = [\![\mathrm{I}_{(M;\mathsf{x})}]\!] \wedge [\![\mathrm{T}_{(M;\mathsf{x})}]\!] \wedge [\![\mathrm{P}_{(M;\mathsf{x})}]\!]$ of three formulas, modelling the semantics of the initial state, the transition relation, and the predecessor relation of M. We assert that the initial state q_0 is the only state reachable after reading the prefix of length 0, i.e., $[\![\mathrm{I}_{(M;\mathsf{x})}]\!]^{b_1} = S_{q_0}^0 \wedge \bigwedge_{q \in Q \setminus \{q_0\}} \neg S_q^0$. The condition is independent of the bound on x, thus we set $[\![\mathrm{I}_{(M;\mathsf{x})}]\!]^{b_k}_{b_{k-1}} = \top$ for all $k > 1$.

We encode the transition relation of M by stating that if M is in some state q after reading $h_\omega(\mathsf{x})[1..i]$, and if there exists a transition from q to q' labelled with an a, then M can reach state q' after $i+1$ transitions if $h_\omega(\mathsf{x})[i+1] = a$. This is expressed in the following formula:

$$[\![\mathrm{T}_{(M;\mathsf{x})}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \bigwedge_{i=\mathrm{b}_{k-1}(\mathsf{x})}^{\mathrm{b}_k(\mathsf{x})-1} \ \bigwedge_{(q,a)\in\mathrm{dom}(\delta)} \ \bigwedge_{q' \in \delta(q,a)} (S_q^i \wedge h^a_{\mathsf{x}[i+1]}) \rightarrow S_{q'}^{i+1}$$

The formula captures all possible forward moves from each state. We must also ensure that a state is reachable only if it has a reachable predecessor, which we

encode with the following formula, where $\mathrm{pred}(q') = \{(q,a) \mid q' \in \delta(q,a)\}$:

$$[\![\mathrm{P}_{(M;\mathsf{x})}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \bigwedge_{i=\mathrm{b}_{k-1}(\mathsf{x})+1}^{\mathrm{b}_k(\mathsf{x})} \bigwedge_{q' \in Q} (S^i_{q'} \rightarrow \bigvee_{(q,a)\in\mathrm{pred}(q')} (S^{i-1}_q \wedge h^a_{\mathsf{x}[i]}))$$

The formula states that if state q' is reachable after $i \geq 1$ transitions, then there must be a reachable predecessor state $q \in \hat{\delta}(\{q_0\}, h_\omega(\mathsf{x})[1..i-1])$ such that $q' \in \delta(q, h_\omega(\mathsf{x})[i])$.

To decide whether the automaton accepts $h_\omega(\mathsf{x})$, we encode that it must reach an accepting state after $\mathrm{b}_k(\mathsf{x})$ transitions. Our corresponding encoding is only valid for the particular bound $\mathrm{b}_k(\mathsf{x})$. To account for this, we introduce a fresh selector variable s_k and define $[\![\mathrm{accept}_{\mathsf{x}\dot{\in}M}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = s_k \rightarrow \bigvee_{q_f\in F} S^{\mathrm{b}_k(\mathsf{x})}_{q_f}$. Analogously, we define $[\![\mathrm{reject}_{\mathsf{x}\dot{\in}M}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = s_k \rightarrow \bigwedge_{q_f\in F} \neg S^{\mathrm{b}_k(\mathsf{x})}_{q_f}$. In the k-th call to the SAT solver and all following calls with the same bound on x, we solve under the assumption that s_k is true. In the first call k' with $\mathrm{b}_k(\mathsf{x}) < \mathrm{b}_{k'}(\mathsf{x})$, we re-encode the condition using a new selector variable $s_{k'}$ and solve under the assumption that s_k is false and s'_k is true. The full encoding of the regular constraint $x \dot{\in} R$ is thus given by

$$[\![\mathsf{x} \dot{\in} R]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = [\![(M;\mathsf{x})]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} \wedge [\![\mathrm{accept}_{\mathsf{x}\dot{\in}M}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$$

and its negation $x \dot{\notin} R$ is encoded as

$$[\![\mathsf{x} \dot{\notin} R]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = [\![(M;\mathsf{x})]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} \wedge [\![\mathrm{reject}_{\mathsf{x}\dot{\in}M}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}.$$

Variable Equations Let $\mathsf{x}, \mathsf{y} \in \Gamma$ be two string variables, let $l = \min(\mathrm{b}_{k-1}(\mathsf{x}), \mathrm{b}_{k-1}(\mathsf{y}))$, and let $u = \min(\mathrm{b}_k(\mathsf{x}), \mathrm{b}_k(\mathsf{y}))$. We encode equality between x and y with respect to b_k position-wise up to u:

$$[\![\mathsf{x} \doteq \mathsf{y}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}} = \bigwedge_{i=l+1}^{u} \bigwedge_{a\in\Sigma_\lambda} (h^a_{\mathsf{x}[i]} \rightarrow h^a_{\mathsf{y}[i]}).$$

The formula asserts that for each position $i \in l+1, \ldots, u$, if $\mathsf{x}[i]$ is mapped to a symbol, then $\mathsf{y}[i]$ is mapped to the same symbol (including λ). Since our encoding of substitutions ensures that every position in a string variable is mapped to exactly one character, $[\![\mathsf{x} \doteq \mathsf{y}]\!]^{\mathrm{b}_k}_{\mathrm{b}_{k-1}}$ ensures $\mathsf{x}[i] = \mathsf{y}[i]$ for $i \in l+1, \ldots, u$. In conjunction with $[\![\mathsf{x} \doteq \mathsf{y}]\!]^{\mathrm{b}_{k-1}}$, which encodes equality up to the l-th position, we have symbol-wise equality of x and y up to bound u. Thus, if $\mathrm{b}_k(\mathsf{x}) = \mathrm{b}_k(\mathsf{y})$, then the formula ensures the equality of both variables. If $\mathrm{b}_k(\mathsf{x}) > \mathrm{b}_k(\mathsf{y})$, we add $h^\lambda_{\mathsf{x}[u+1]}$ as an assumption to the solver to ensure $\mathsf{x}[i] = \lambda$ for $i \in u+1, \ldots, \mathrm{b}_k(\mathsf{x})$ and, symmetrically, we add the assumption $h^\lambda_{\mathsf{y}[u+1]}$ if $\mathrm{b}_k(\mathsf{y}) > \mathrm{b}_k(\mathsf{x})$.

For the negation $\mathsf{x} \dot{\neq} \mathsf{y}$, we encode that $h(\mathsf{x})$ and $h(\mathsf{y})$ must disagree on at least one position, which can happen either because they map to different symbols or because the variable with the higher bound is mapped to a longer word. As

for the regular constraints, we again use selector variable s_k to deactivate the encoding for all later bounds, for which it will be re-encoded:

$$[\![\mathsf{x} \not\doteq \mathsf{y}]\!]_{\mathrm{b}_{k-1}}^{\mathrm{b}_k} = \begin{cases} s_k \to (\bigvee_{i=1}^{u} \bigvee_{a \in \Sigma_\lambda} (\neg h_{\mathsf{x}[i]}^{a} \wedge h_{\mathsf{y}[i]}^{a})) & \text{if } \mathrm{b}_k(\mathsf{x}) = \mathrm{b}_k(\mathsf{y}) \\ s_k \to (\bigvee_{i=1}^{u} \bigvee_{a \in \Sigma_\lambda} (\neg h_{\mathsf{x}[i]}^{a} \wedge h_{\mathsf{y}[i]}^{a})) \vee \neg h_{\mathsf{y}[u+1]}^{\lambda} & \text{if } \mathrm{b}_k(\mathsf{x}) < \mathrm{b}_k(\mathsf{y}) \\ s_k \to (\bigvee_{i=1}^{u} \bigvee_{a \in \Sigma_\lambda} (\neg h_{\mathsf{x}[i]}^{a} \wedge h_{\mathsf{y}[i]}^{a})) \vee \neg h_{\mathsf{x}[u+1]}^{\lambda} & \text{if } \mathrm{b}_k(\mathsf{x}) > \mathrm{b}_k(\mathsf{y}) \end{cases}$$

Constant Equations Given a constant equation $\mathsf{x} \doteq w$, if the upper bound of x is less than $|w|$, the atom is trivially unsatisfiable. Thus, for all i such that $b_i(\mathsf{x}) < |w|$, we encode $\mathsf{x} \doteq w$ with a simple literal $\neg s_{\mathsf{x},w}$ and add $s_{\mathsf{x},w}$ to the assumptions. For $\mathrm{b}_k(\mathsf{x}) \geq |w|$, the encoding is based on the value of $\mathrm{b}_{k-1}(\mathsf{x})$:

$$[\![\mathsf{x} \doteq w]\!]_{\mathrm{b}_{k-1}}^{\mathrm{b}_k} = \begin{cases} \bigwedge_{i=1}^{|w|} h_{\mathsf{x}[i]}^{w[i]} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) < |w| = \mathrm{b}_k(\mathsf{x}) \\ \bigwedge_{i=1}^{|w|} h_{\mathsf{x}[i]}^{w[i]} \wedge h_{\mathsf{x}[|w|+1]}^{\lambda} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) < |w| < \mathrm{b}_k(\mathsf{x}) \\ h_{\mathsf{x}[|w|+1]}^{\lambda} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) = |w| < \mathrm{b}_k(\mathsf{x}) \\ \top & \text{if } |w| < \mathrm{b}_{k-1}(\mathsf{x}) \end{cases}$$

If $\mathrm{b}_{k-1}(\mathsf{x}) < |w|$, then equality is encoded for all positions $1, \ldots, |w|$. Additionally, if $\mathrm{b}_k(\mathsf{x}) > |w|$, we ensure that the suffix of x is empty starting from position $|w| + 1$. If $\mathrm{b}_{k-1}(\mathsf{x}) = |w| < \mathrm{b}_k(\mathsf{x})$, then only the empty suffix has to be ensured. Lastly, if $|w| < \mathrm{b}_{k-1}(\mathsf{x})$, then $[\![\mathsf{x} \doteq w]\!]^{\mathrm{b}_{k-1}} \Leftrightarrow [\![\mathsf{x} \doteq w]\!]^{\mathrm{b}_k}$.

Conversely, for an inequality $\mathsf{x} \not\doteq w$, if $\mathrm{b}_k(\mathsf{x}) < |w|$, then any substitution trivially is a solution, which we simply encode with $\top$. Otherwise, we introduce a selector variable $s'_{\mathsf{x},w}$ and define

$$[\![\mathsf{x} \not\doteq w]\!]_{\mathrm{b}_{k-1}}^{\mathrm{b}_k} = \begin{cases} s'_{\mathsf{x},w} \to \bigvee_{i=1}^{|w|} \neg h_{\mathsf{x}[i]}^{w[i]} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) < |w| = \mathrm{b}_k(\mathsf{x}) \\ \bigvee_{i=1}^{|w|} \neg h_{\mathsf{x}[i]}^{w[i]} \vee \neg h_{\mathsf{x}[|w|+1]}^{\lambda} & \text{if } \mathrm{b}_{k-1}(\mathsf{x}) < |w| < \mathrm{b}_k(\mathsf{x}) \\ \top & \text{if } |w| < \mathrm{b}_{k-1}(\mathsf{x}) \leq \mathrm{b}_k(\mathsf{x}) \end{cases}$$

If $\mathrm{b}_k(\mathsf{x}) = |w|$, then a substitution h satisfies the constraint if and only if $h(\mathsf{x})[i] \neq w[i]$ for some $i \in 1, \ldots, |w|$. If $\mathrm{b}_k(\mathsf{x}) > |w|$, in addition, h satisfies the constraint if $|h(\mathsf{x})| > |w|$. Thus, if $\mathrm{b}_k(\mathsf{x}) = |w|$, we perform solver call k under the assumption $s'_{\mathsf{x},w}$, and if $\mathrm{b}_k(\mathsf{x}) > |w|$, we perform it under the assumption $\neg s'_{\mathsf{x},w}$. Again, if $|w| < \mathrm{b}_{k-1}(\mathsf{x})$, then $[\![\mathsf{x} \not\doteq w]\!]^{\mathrm{b}_{k-1}} \Leftrightarrow [\![\mathsf{x} \not\doteq w]\!]^{\mathrm{b}_k}$.

Prefix and Suffix Constraints A prefix constraint $w \mathrel{\dot\sqsubseteq} \mathsf{x}$ expresses that the first $|w|$ positions of x must be mapped exactly onto w. As with equations between a variable x and a constant word w, we could express this as a regular constraint of the form $\mathsf{x} \mathrel{\dot\in} w \cdot ?^*$. However, we achieve a more efficient encoding simply by dropping from the encoding of $[\![\mathsf{x} \doteq w]\!]$ the assertion that the suffix of x starting at $|w + 1|$ be empty. Accordingly, a negated prefix constraint $w \mathrel{\dot{\not\sqsubseteq}} \mathsf{x}$ expresses that there is an index $i \in 1, \ldots, |w|$ such that the i-th position of x

is mapped onto a symbol different from $w[i]$, which we encode by repurposing $[\![\mathsf{x} \not\doteq w]\!]$ in a similar manner. Suffix constraints $w \dot{\sqsupseteq} \mathsf{x}$ and $w \dot{\not\sqsupseteq} \mathsf{x}$ can be encoded by analogous modifications to the encodings of $\mathsf{x} \doteq w$ and $\mathsf{x} \not\doteq w$.

6 Refining Upper Bounds

Our procedure solves a series of SAT problems where the length bounds on string variables increase after each unsatisfiable solver call. The procedure terminates once the bounds are large enough so that further increasing them would be futile. To determine when this is the case, we rely on the upper bounds of a *shortest solution* to a formula ψ. We call a model h of ψ a shortest solution of ψ if ψ has no model h' such that $\sum_{\mathsf{x}\in\Gamma}|h'(\mathsf{x})| < \sum_{\mathsf{x}\in\Gamma}|h(\mathsf{x})|$. We first establish this bound for conjunctive formulas in normal form, where all literals are of the form $\mathsf{x} \not\doteq \mathsf{y}$, $\mathsf{x} \dot{\in} R$, or $\mathsf{x} \dot{\notin} R$. Once established, we show how the bound can be generalized to arbitrary formulas.

Let φ be a formula in normal form and let $\mathsf{x}_1, \ldots, \mathsf{x}_n$ be the variables of φ. For each variable x_i, we can collect all the regular constraints on x_i, that is, all the literals of the form $\mathsf{x}_i \dot{\in} R$ or $\mathsf{x}_i \dot{\notin} R$ that occur in φ. We can characterize the solutions to all these constraints by a single nondeterministic finite automaton M_i. If the constraints on x_i are $\mathsf{x}_i \dot{\in} R_1, \ldots, \mathsf{x}_i \dot{\in} R_k, \mathsf{x}_i \dot{\notin} R'_1 \ldots, \mathsf{x}_i \dot{\notin} R'_l$, then M_i is an NFA that accepts the regular language $\bigcap_{t=1}^{k} \mathcal{L}(R_t) \cap \bigcap_{t=1}^{l} \overline{\mathcal{L}(R'_t)}$, where $\overline{\mathcal{L}(R)}$ denotes the complement of $\mathcal{L}(R)$. We say that M_i accepts the regular constraints on x_i in φ. If there are no such constraints on x_i, then M_i is the one-state NFA that accepts the full language Σ^*. Let Q_i denote the set of states of M_i. If we do not take inequalities into account and if the regular constraints on x_i are satisfiable, then a shortest solution h has length $|h(\mathsf{x}_i)| \leq |Q_i|$.

Theorem 6.1 gives a bound for the general case with variable inequalities. Intuitively, we prove the theorem by constructing a single automaton $\mathcal{P}$ that takes as input a vector of words $W = (w_1, ..., w_n)^T$ and accepts W iff the substitution h_W with $h_W(\mathsf{x}_i) = w_i$ satisfies φ. To construct $\mathcal{P}$, we introduce one two-state NFA for each inequality and we then form the product of these NFAs with (slightly modified versions of) the NFAs $M_1, \ldots, M_n$. We can then derive the bound of a shortest solution from the number of states of $\mathcal{P}$.

Theorem 6.1. *Let φ be a conjunctive formula in normal form over variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$. Let $M_i = (Q_i, \Sigma, \delta_i, q_{0,i}, F_i)$ be an NFA that accepts the regular constraints on x_i in φ and let k be the number of inequalities occurring in φ. If φ is satisfiable, then it has a model h such that*

$$|h(\mathsf{x}_i)| \leq 2^k \times |Q_1| \times \ldots \times |Q_n|.$$

Proof. Let λ be a symbol that does not belong to Σ and define $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. As previously, we use λ to extend words of Σ^* by padding. Given a word $w \in \Sigma_\lambda^*$, we denote by $\hat{w}$ the word of Σ^* obtained by removing all occurrences of λ from w. We say that w is well-formed if it can be written as $w = v \cdot \lambda^t$ with $v \in \Sigma^*$ and

$t \geq 0$. In this case, we have $\hat{w} = v$. Thus a well-formed word w consists of a prefix in Σ^* followed by a sequence of λs.

Let Δ be the alphabet Σ_λ^n, i.e., the letters of Δ are the n-letter words over Σ_λ. We can then represent a letter u of Δ as an n-element vector $(u_1, \ldots, u_n)$, and a word W of Δ^t can be written as an $n \times t$ matrix

$$W = \begin{pmatrix} u_{11} & \ldots & u_{t1} \\ \vdots & & \vdots \\ u_{1n} & \ldots & u_{tn} \end{pmatrix}$$

where $u_{ij} \in \Sigma_\lambda$. Each column of this matrix is a letter in Δ and each row is a word in Σ_λ^t. We denote by $p_i(W)$ the i-th row of this matrix and by $\hat{p_i}(W) = \widehat{p_i(W)}$ the word $p_i(W)$ with all occurrences of λ removed. We say that W is well-formed if the words $p_1(W), \ldots, p_n(W)$ are all well-formed. Given a well-formed word W, we can construct a mapping $h_W : \{\mathsf{x}_1, \ldots, \mathsf{x}_n\} \to \Sigma^*$ by setting $h_W(\mathsf{x}_i) = \hat{p_i}(W)$ and we have $|h_W(\mathsf{x}_i)| \leq |W| = t$.

To prove the theorem, we build an NFA $\mathcal{P}$ with alphabet Δ such that a well-formed word W is accepted by $\mathcal{P}$ iff h_W satisfies φ. The shortest well-formed W accepted by $\mathcal{P}$ has length no more than the number of states of $\mathcal{P}$ and the bound will follow.

We first extend the NFA $M_i = (Q_i, \Sigma, \delta_i, q_{0,i}, F_i)$ to an automaton M_i' with alphabet Δ. M_i' has the same set of states, initial state, and final states as M_i. Its transition relation δ_i' is defined by

$$\delta_i'(q, u) = \begin{cases} \delta_i(q, u_i) & \text{if } u_i \in \Sigma \\ \{q\} & \text{if } u_i = \lambda \end{cases}$$

One can easily check that M_i' accepts a word W iff M_i accepts $\hat{p_i}(W)$.

For an inequality $\mathsf{x}_i \not\equiv \mathsf{x}_j$, we construct an NFA $D_{i,j} = (\{e, d\}, \Delta, \delta, e, \{d\})$ with transition function defined as follows:

$$\begin{aligned} \delta(e, u) &= \{e\} \;\; \text{if } u_i = u_j \\ \delta(e, u) &= \{d\} \;\; \text{if } u_i \neq u_j \\ \delta(d, u) &= \{d\}. \end{aligned}$$

This NFA has two states. It starts in state e (for "equal") and stays in e as long as the characters u_i and u_j are equal. It transitions to state d (for "different") on the first u where $u_i \neq u_j$ and stays in state d from that point. Since d is the final state, a word W is accepted by $D_{i,j}$ iff $p_i(W) \neq p_j(W)$. If W is well-formed, we also have that W is accepted by $D_{i,j}$ iff $\hat{p_i}(W) \neq \hat{p_j}(W)$.

Let $\mathsf{x}_{i_1} \not\equiv \mathsf{x}_{j_1}, \ldots, \mathsf{x}_{i_k} \not\equiv \mathsf{x}_{j_k}$ denote the k inequalities of φ. We define $\mathcal{P}$ to be the product of the NFAs $M_1', \ldots, M_n'$ and $D_{i_1,j_1}, \ldots, D_{i_k,j_k}$. A well-formed word W is accepted by $\mathcal{P}$ if it is accepted by all M_i' and all D_{i_t,j_t}, which means that $\mathcal{P}$ accepts a well-formed word W iff h_W satisfies φ.

Let P be the set of states of $\mathcal{P}$. We then have $|P| \leq 2^k \times |Q_1| \times \ldots \times |Q_n|$. Assume φ is satisfiable, so $\mathcal{P}$ accepts a well-formed word W. The shortest well-formed word accepted by $\mathcal{P}$ has an accepting run that does not visit the same

state twice. So the length of this well-formed word W is no more than $|P|$. The mapping h_W satisfies φ and for every x_i, it satisfies $|h_W(\mathsf{x}_i)| = |\hat{p}_i(W)| \leq |W| \leq |P| \leq 2^k \times |Q_1| \times \ldots \times |Q_n|$. □

The bound given by Theorem 6.1 holds if φ is in normal form but it also holds for a general conjunctive formula ψ. This follows from the observation that converting conjunctive formulas to normal form preserves the length of solutions. In particular, we convert $\psi \wedge \mathsf{x} \doteq \mathsf{y}$ to formula $\psi' = \psi[x := y]$ so x does not occur in ψ', but clearly, a bound for y in ψ' gives us the same bound for x in ψ.

In practice, before we apply the theorem we decompose the conjunctive formula φ into subformulas that have disjoint sets of variables. We write φ as $\varphi_1 \wedge \ldots \wedge \varphi_m$ where the conjuncts have no common variables. Then, φ is satisfiable if each conjunct φ_t is satisfiable and we derive upper bounds on the shortest solution for the variables of φ_t, which gives more precise bounds than deriving bounds from φ directly. In particular, if a variable x_i of ψ does not occur in any inequality, then the bound on $|h(\mathsf{x}_i)|$ is $|Q_i|$.

Theorem 6.1 only holds for conjunctive formulas. For an arbitrary (non-conjunctive) formula ψ, a generalization is to convert ψ into disjunctive normal form. Alternatively, it is sufficient to enumerate the subsets of $\mathit{lits}(\psi)$. Given a subset A of $\mathit{lits}(\psi)$, let us denote by d_A a mapping that bounds the length of solutions to A, i.e., any solution h to A satisfies $|h(x)| \leq d_A(x)$. This mapping d_A can be computed from Theorem 6.1. The following property gives a bound for ψ.

Proposition 6.2. *If ψ is satisfiable, then it has a model h such that for all $\mathsf{x} \in \Gamma$, it holds that $|h(\mathsf{x})| \leq \max\{d_A(x) \mid A \subseteq \mathit{lits}(\psi)\}$.*

Proof. We can assume that ψ is in negation normal form. We can then convert ψ to disjunctive normal form $\psi \Leftrightarrow \psi_1 \vee \cdots \vee \psi_n$ and we have $\mathit{lits}(\psi_i) \subseteq \mathit{lits}(\psi)$. Also, ψ is satisfiable if and only if at least one ψ_i is satisfiable and the proposition follows. □

Since there are $2^{|\mathit{lits}(\psi)|}$ subsets of $\mathit{lits}(\psi)$, a direct application of Proposition 6.2 is rarely feasible in practice. Fortunately, we can use unsatisfiable cores to reduce the number of subsets to consider.

6.1 Unsatisfiable-Core Analysis

Instead of calculating the bounds upfront, we use the unsatisfiable core produced by the SAT solver after each incremental call to evaluate whether the upper bounds on the variables exceed the upper bounds of the shortest solution. If $[\![\psi]\!]^{\mathrm{b}}$ is unsatisfiable for bounds b, then it has an unsatisfiable core

$$C = C_{\mathcal{A}} \wedge C_h \wedge \bigwedge_{a \in atoms^+(\psi)} C_a \wedge \bigwedge_{a \in atoms^-(\psi)} C_{\bar{a}}$$

with (possibly empty) subsets of clauses $C_{\mathcal{A}} \subseteq \psi_{\mathcal{A}}$, $C_h \subseteq [\![h]\!]^{\mathrm{b}}$, $C_a \subseteq (\mathbf{d}(a) \rightarrow [\![a]\!]^{\mathrm{b}})$, and $C_{\bar{a}} \subseteq (\neg\, \mathbf{d}(a) \rightarrow [\![\neg a]\!]^{\mathrm{b}})$. Here we implicitly assume $\psi_{\mathcal{A}}$, $\mathbf{d}(a) \rightarrow [\![a]\!]^{\mathrm{b}}$,

and $\neg\, \mathbf{d}(a) \to [\![\neg a]\!]^{\mathrm{b}}$ to be in CNF. Let $\mathcal{C}^+ = \{a \mid C_a \neq \emptyset\}$ and $\mathcal{C}^- = \{\neg a \mid C_{\bar{a}} \neq \emptyset\}$ be the sets of literals whose encodings contain at least one clause of the core C. Using these sets, we construct the formula

$$\psi^{\mathcal{C}} = \psi_{\mathcal{A}} \wedge \bigwedge_{a \in \mathcal{C}^+} \mathbf{d}(a) \to a \wedge \bigwedge_{\neg a \in \mathcal{C}^-} \neg\, \mathbf{d}(a) \to \neg a,$$

which consists of the conjunction of the abstraction and the definitions of the literals that are contained in $\mathcal{C}^+$, respectively $\mathcal{C}^-$. Recall that ψ is equisatisfiable to the conjunction $\psi_{\mathcal{A}} \wedge \bigwedge_{d \in \mathbf{D}} d$ of the abstraction and all definitions in $\mathbf{D}$. Let ψ' denote this formula, i.e.,

$$\psi' = \psi_{\mathcal{A}} \wedge \bigwedge_{a \in \mathit{atoms}^+(\psi)} \mathbf{d}(a) \to a \wedge \bigwedge_{\neg a \in \mathit{atoms}^-(\psi)} \neg\, \mathbf{d}(a) \to \neg a.$$

The following proposition shows that it suffices to refine the bounds according to $\psi^{\mathcal{C}}$.

Proposition 6.3. *Let ψ be unsatisfiable with respect to* b *and let C be an unsatisfiable core of $[\![\psi]\!]^{\mathrm{b}}$. Then, $\psi^{\mathcal{C}}$ is unsatisfiable with respect to* b *and $\psi' \models \psi^{\mathcal{C}}$.*

Proof. By definition, we have $[\![\psi^{\mathcal{C}}]\!]^{\mathrm{b}} = \psi_{\mathcal{A}} \wedge [\![h]\!]^{\mathrm{b}} \wedge \bigwedge_{a \in \mathcal{C}^+} \mathbf{d}(a) \to [\![a]\!]^{\mathrm{b}} \wedge \bigwedge_{\neg a \in \mathcal{C}^-} \neg\, \mathbf{d}(a) \to \neg [\![\neg a]\!]^{\mathrm{b}}$. This implies $C \subseteq [\![\psi^{\mathcal{C}}]\!]^{\mathrm{b}}$ and, since C is an unsatisfiable core, $[\![\psi^{\mathcal{C}}]\!]^{\mathrm{b}}$ is unsatisfiable. That is, $\psi^{\mathcal{C}}$ is unsatisfiable with respect to b. We also have $\psi' \models \psi^{\mathcal{C}}$ since $\mathcal{C}^+ \subseteq \mathit{atoms}^+(\psi)$ and $\mathcal{C}^- \subseteq \mathit{atoms}^-(\psi)$. $\square$

Applying Proposition 6.2 to $\psi^{\mathcal{C}}$ results in the upper bounds of the shortest solution h_C for $\psi^{\mathcal{C}}$. If $|h_C(\mathsf{x})| \leq \mathrm{b}(\mathsf{x})$ holds for all $\mathsf{x} \in \Gamma$, then ψ^C has no solution and unsatisfiability of ψ' follows from Proposition 6.3. Because ψ and ψ' are equisatisfiable, we can conclude that ψ is unsatisfiable.

Otherwise, we increase the bounds on the variables that occur in ψ^C while keeping bounds on the other variables unchanged: We construct b_{k+1} with $\mathrm{b}_k(\mathsf{x}) \leq \mathrm{b}_{k+1}(\mathsf{x}) \leq |h_C(\mathsf{x})|$ for all $\mathsf{x} \in \Gamma$, such that $\mathrm{b}_k(\mathsf{y}) < \mathrm{b}_{k+1}(\mathsf{y})$ holds for at least one $\mathsf{y} \in V(\psi^{\mathcal{C}})$. By strictly increasing at least one variable's bound, we eventually either reach the upper bounds of $\psi^{\mathcal{C}}$ and return unsatisfiability, or we eliminate it as an unsatisfiable implication of ψ. As there are only finitely many possibilities for $\mathcal{C}$ and thus for $\psi^{\mathcal{C}}$, our procedure is guaranteed to terminate.

We do not explicitly construct formula $\psi^{\mathcal{C}}$ to compute bounds on h_C as we know the set $\mathit{lits}(\psi^{\mathcal{C}}) = \mathcal{C}^+ \cup \mathcal{C}^-$. Finding upper bounds still requires enumerating all subsets of $\mathit{lits}(\psi^{\mathcal{C}})$, but we have $|\mathit{lits}(\psi^{\mathcal{C}})| \leq |\mathit{lits}(\psi)|$ and usually $\mathit{lits}(\psi^{\mathcal{C}})$ is much smaller than $\mathit{lits}(\psi)$. For example, consider the formula

$$\psi = \mathsf{z} \not\doteq abd \wedge (\mathsf{x} \doteq a \vee \mathsf{x} \mathrel{\dot\in} ab^*) \wedge \mathsf{x} \doteq \mathsf{y} \wedge (\mathsf{y} \doteq bbc \vee \mathsf{z} \in a(b|c)^*d) \wedge \mathsf{y} \mathrel{\dot\in} ab\cdot?^*$$

which is unsatisfiable for the bounds $\mathrm{b}(\mathsf{x}) = \mathrm{b}(\mathsf{y}) = 1$ and $\mathrm{b}(\mathsf{z}) = 4$. The unsatisfiable core C returned after solving $[\![\psi]\!]^{\mathrm{b}}$ results in the formula

$\psi^{\mathcal{C}} = (\mathsf{x} \doteq a \vee \mathsf{x} \mathrel{\dot\in} ab^*) \wedge \mathsf{x} \doteq \mathsf{y} \wedge \mathsf{y} \mathrel{\dot\in} ab{\cdot}?^*$ containing four literals. Finding upper bounds for $\psi^{\mathcal{C}}$ thus amounts to enumerating just 2^4 subsets, which is substantially less than considering all 2^7 subsets of $lits(\psi)$ upfront. The conjunction of a subset of $lits(\psi^{\mathcal{C}})$ yielding the largest upper bounds is $\mathsf{x} \mathrel{\dot\in} ab^* \wedge \mathsf{x} \doteq \mathsf{y} \wedge \mathsf{y} \mathrel{\dot\in} ab{\cdot}?^*$, which simplifies to $\mathsf{x} \mathrel{\dot\in} ab^* \cap ab{\cdot}?^*$ and has a solution of length at most 2 for x and y. With bounds $\mathrm{b}(\mathsf{x}) = \mathrm{b}(\mathsf{y}) = 2$ and $\mathrm{b}(\mathsf{z}) = 4$, the formula is satisfiable.

7 Implementation

We have implemented our approach in a solver called NFA2SAT. NFA2SAT is written in RUST and uses CADICAL [9] as the backend SAT solver. We use the incremental API provided by CADICAL to solve problems under assumptions. Soundness of NFA2SAT follows from Theorem 5.1. For completeness, we rely on CADICAL's *failed* function to efficiently determine *failed assumptions*, i.e., assumption literals that were used to conclude unsatisfiability.

The procedure works as follows. Given a formula ψ, we first introduce one fresh Boolean selector variable s_l for each theory literal $l \in lits(\psi)$. Then, instead of adding the encoded definitions of the theory literals directly to the SAT solver, we precede them with their corresponding selector variables: for a positive literal a, we add $s_a \rightarrow (\mathbf{d}(a) \rightarrow [\![a]\!])$, and for a negative literal $\neg a$, we add $s_{\neg a} \rightarrow (\neg\, \mathbf{d}(a) \rightarrow [\![\neg a]\!])$ (considering assumptions introduced by $[\![a]\!]$ as unit clauses). In the resulting CNF formula, the new selector variables are present in all clauses that encode their corresponding definition, and we use them as assumptions for every incremental call to the SAT solver, which does not affect satisfiability. If such an assumption failed, then we know that at least one of the corresponding clauses in the propositional formula was part of an unsatisfiable core, which enables us to efficiently construct the sets $\mathcal{C}^+$ and $\mathcal{C}^-$ of positive and negative atoms present in the unsatisfiable core. As noted previously, we have $lits(\psi^C) = \mathcal{C}^+ \cup \mathcal{C}^-$ and hence the sets are sufficient to find bounds on a shortest model for ψ^C.

This approach is efficient for obtaining $lits(\psi^C)$ but since CADICAL does not guarantee that the set of failed assumptions is minimal, $lits(\psi^C)$ is not minimal in general. Moreover, even a minimal $lits(\psi^C)$ can contain too many elements for processing all subsets. To address this issue, we enumerate the subsets only if $lits(\psi^C)$ is small (by default, we use a limit of ten literals). In this case, we construct the automata M_i used in Theorem 6.1 for each subset, facilitating the techniques described in [7] for quickly ruling out unsatisfiable ones. Otherwise, instead of enumerating the subsets, we resort to sound approximations of upper bounds, which amounts to over-approximating the number of states without explicitly constructing the automata (c.f. [14]).

Once we have obtained upper bounds on the length of the solution of ψ^C, we increment bounds on all variables involved, except those that have reached their maximum. Our default heuristics computes a new bound that is either double the current bound of a variable or its maximum, whichever is smaller.

	cvc5	Z3	nfa2sat	cvc5 Z3	nfa2sat cvc5	nfa2sat Z3
SAT	22895	22927	22922	22934	22934	22934
UNSAT	6259	6486	6405	6526	6598	6645
Timeout	445	185	206	139	67	20
Out-of-memory	0	1	66	n/a	n/a	n/a
Total Solved	29154	29413	29327	29460	29532	29579
Total Runtime (s)	655877	283942	275420	169553	126655	28914

Table 1: Evaluation on ZaligVinder benchmarks. The three left columns show results of individual solvers. The other three columns show results of portfolios combining two solvers.

8 Experimental Evaluation

We have evaluated our solver on a large set of benchmarks from the ZaligVinder [22] repository[4]. The repository contains 120,287 benchmarks stemming from both academic and industrial applications. In particular, all the string problems from the SMT-LIB repository,[5] are included in the ZaligVinder repository. We converted the ZaligVinder problems to the SMT-LIB 2.6 syntax and removed duplicates. This resulted in 82,632 unique problems out of which 29,599 are in the logical fragment we support.

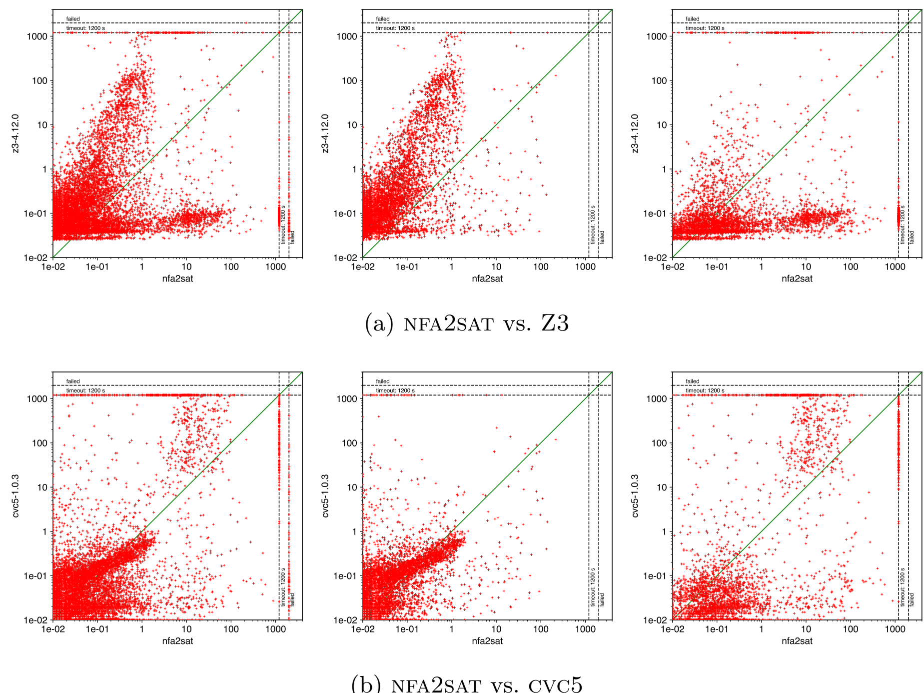
We compare nfa2sat with the state-of-the-art solvers cvc5 (version 1.0.3) and Z3 (version 4.12.0). The comparison is limited to these two solvers because they are widely adopted and because they had the best performance in our evaluation. Other string solvers either don't support our logical fragment (CertiStr, Woorpje) or gave incorrect answers on the benchmark problems considered here. Older, no-longer maintained, solvers have known soundness problems, as reported in [7] and [27].

We ran our experiment on a Linux server, with a timeout of 1200 seconds CPU time and a memory limit of 16 GB. Table 1 shows the results. As a single tool, nfa2sat solves more problems than cvc5 but not as many as Z3. All three tools solve more than 98% of the problems.

The table also shows results of portfolios that combine two solvers. In a portfolio configuration, the best setting is to use both Z3 and nfa2sat. This combination solves all but 20 problems within the timeout. It also reduces the total run-time from 283,942 seconds for Z3 (about 79 hours) to 28,914 seconds for the portfolio (about 8 hours), that is, a 90% reduction in total solve time. The other two portfolios—namely, Z3 with cvc5 and nfa2sat with cvc5—also have better performance than a single solver, but the improvement in runtime and number of timeouts is not as large.

[4] `https://github.com/zaligvinder/zaligvinder`

[5] `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S`



(a) NFA2SAT vs. Z3

(b) NFA2SAT vs. CVC5

Fig. 4: Comparison of runtime (in seconds) with Z3 and CVC5. The left plots include all problems, the middle plots include only satisfiable problems, and the right plots include only unsatisfiable problems. The lines marked "failed" correspond to problems that are not solved because a solver ran out of memory. The lines marked "timeout" correspond to problems not solved because of a timeout (1200 seconds).

Figure 4a illustrates why NFA2SAT and Z3 complement each other well. The figure shows three scatter plots that compare the runtime of NFA2SAT and Z3 on our problems. The plot on the left compares the two solvers on *all* problems, the one in the middle compares them on *satisfiable* problems, and the one on the right compares them on *unsatisfiable* problems. Points in the left plot are concentrated close to the axes, with a smaller number of points near the diagonal, meaning that Z3 and NFA2SAT have different runtime on most problems. The other two plots show this even more clearly: NFA2SAT is faster on satisfiable problems while Z3 is faster on unsatisfiable problems. Figure 4b shows analogous scatter plots comparing NFA2SAT and CVC5. The two solvers show similar performance on a large set of easy benchmarks although CVC5 is faster on problems that both solvers can solve in less than 1 second. However, CVC5 times out on 38 problems that NFA2SAT solves in less than 2 seconds. On unsatisfiable problems, CVC5 tends to be faster than NFA2SAT, but there is a class of problems for which NFA2SAT takes between 10 and 100 seconds whereas CVC5 is slower.

Overall, the comparison shows that NFA2SAT is competitive with CVC5 and Z3 on these benchmarks. We also observe that NFA2SAT tends to work better on satisfiable problems. For best overall performance, our experiments show that a portfolio of Z3 and NFA2SAT would solve all but 20 problems within the timeout, and reduce the total solve time by 90%.

9 Conclusion

We have presented the first eager SAT-based approach to string solving that is both sound and complete for a reasonably expressive fragment of string theory. Our experimental evaluation shows that our approach is competitive with the state-of-the-art lazy SMT solvers Z3 and CVC5, outperforming them on satisfiable problems but falling behind on unsatisfiable ones. A portfolio that combines our approach with these solvers—particularly with Z3—would thus yield strong performance across both types of problems.

In future work, we plan to extend our approach to a more expressive logical fragment, including more general word equations. Other avenues of research include the adaption of model checking techniques such as IC3 [10] to string problems, which we hope would lead to better performance on unsatisfiable instances. A particular benefit of the eager approach is that it enables the use of mature techniques from the SAT world, especially for proof generation and parallel solving. Producing proofs of unsatisfiability is complex for traditional CDCL(T) solvers because of the complex rewriting and deduction rules they employ. In contrast, efficiently generating and checking proofs produced by SAT solvers (using the DRAT format [32]) is well-established and practicable. A challenge in this respect would be to combine unsatisfiability proofs from a SAT solver with proof that our reduction to SAT is sound. For parallel solving, we plan to explore the use of a parallel incremental solver (such as iLingeling [9]) as well as other possible ways to solve multiple bounds in parallel.

References

1. Abdulla, P.A., Faouzi Atig, M., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–5 (2018). https://doi.org/10.23919/FMCAD.2018.8602997
2. 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
3. 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, https://doi.org/10.1007/978-3-030-99524-9_24
4. 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
5. 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)
6. Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: Towards more efficient methods for solving regular-expression heavy string constraints. Theoretical Computer Science **943**, 50–72 (2023). https://doi.org/https://doi.org/10.1016/j.tcs.2022.12.009, https://www.sciencedirect.com/science/article/pii/S030439752200723X
7. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An SMT solver for regular expressions and linear arithmetic over string length. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 289–312. Springer International Publishing, Cham (2021)
8. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-457, https://doi.org/10.3233/978-1-58603-929-5-457
9. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
10. Bradley, A.R.: SAT-based model model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI 2011). Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011)

11. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290362, https://doi.org/10.1145/3290362
12. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In: Filiot, E., Jungers, R., Potapov, I. (eds.) Reachability Problems. pp. 93–106. Springer International Publishing, Cham (2019)
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4), 543–560 (2003). https://doi.org/https://doi.org/10.1016/S1571-0661(05)82542-3, https://www.sciencedirect.com/science/article/pii/S1571066105825423, bMC'2003, First International Workshop on Bounded Model Checking
14. Gao, Y., Moreira, N., Reis, R., Yu, S.: A survey on operational state complexity. CoRR **abs/1509.03254** (2015), http://arxiv.org/abs/1509.03254
15. 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)
16. 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, http://drops.dagstuhl.de/opus/volltexte/2017/7408
17. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: A certified string solver. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 210–224. CPP 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3497775.3503691, https://doi.org/10.1145/3497775.3503691
18. 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, https://doi.org/10.1145/337244.337255
19. 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, https://doi.org/10.1145/1572272.1572286
20. Klieber, W., Kwon, G.: Efficient CNF encoding for selecting 1 from N objects. In: Fourth Workshop on Constraints in Formal Verification (CFV) (2007)
21. 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)
22. Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: Zaligvinder: A generic test framework for string solvers. Journal of Software: Evolution and Process **n/a**(n/a), e2400. https://doi.org/https://doi.org/10.1002/smr.2400, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2400
23. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Math. USSR, Sb. **32**, 129–198 (1977). https://doi.org/10.1070/SM1977v032n02ABEH002376

24. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A multi-armed string solver. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) Formal Methods. pp. 389–406. Springer International Publishing, Cham (2021)
25. 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)
26. Murray, N.V.: Completely non-clausal theorem proving. Artificial Intelligence **18**(1), 67–85 (1982). https://doi.org/https://doi.org/10.1016/0004-3702(82)90011-X, https://www.sciencedirect.com/science/article/pii/000437028290011X
27. Nötzli, A., Reynolds, A., Barbosa, H., Barrett, C.W., Tinelli, C.: Even faster conflicts and lazier reductions for string solvers. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 205–226. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_11, https://doi.org/10.1007/978-3-031-13188-2_11
28. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation **2**(3), 293–304 (1986). https://doi.org/https://doi.org/10.1016/S0747-7171(86)80028-1, https://www.sciencedirect.com/science/article/pii/S0747717186800281
29. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. In: 40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039). pp. 495–500 (1999). https://doi.org/10.1109/SFFCS.1999.814622
30. 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
31. 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
32. Wetzler, N., Heule, M., Jr., W.A.H.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8561, pp. 422–429. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31, https://doi.org/10.1007/978-3-319-09284-3_31
33. 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, https://doi.org/10.1007/s10703-013-0189-1