On Solving Word Equations Using SAT

Joel D. Day[1], Thorsten Ehlers[2], Mitja Kulczynski[3(✉)], Florin Manea[3], Dirk Nowotka[3], and Danny Bøgsted Poulsen[3]

[1] Department of Computer Science, Loughborough University, Loughborough, UK
j.day@lboro.ac.uk
[2] German Aerospace Center (DLR), Helmholtz Association, Hamburg, Germany
thorsten.ehlers@dlr.de
[3] Department of Computer Science, Kiel University, Kiel, Germany
{mku,flm,dn,dbp}@informatik.uni-kiel.de

Abstract. We present Woorpje, a string solver for bounded word equations (i.e., equations where the length of each variable is upper bounded by a given integer). Our algorithm works by reformulating the satisfiability of bounded word equations as a reachability problem for nondeterministic finite automata, and then carefully encoding this as a propositional satisfiability problem, which we then solve using the well-known Glucose SAT-solver. This approach has the advantage of allowing for the natural inclusion of additional linear length constraints. Our solver obtains reliable and competitive results and, remarkably, discovered several cases where state-of-the-art solvers exhibit a faulty behaviour.

1 Introduction

Over the past twenty years, applications of software verification have scaled from small academic programs to finding errors in the GNU Coreutils [7]. In principle, the employed verification strategies involve exploring the control-flow-graph of the program, gathering constraints over program variables and passing these constraints to a constraint solver. The primary worker of software verification is thus the constraint solver, and the scalability of software verification achieved by improving the efficiency of constraint solvers. The theories supported by constraint solvers are likewise highly influenced by the needs of software verification tools (e.g. array theory and bitvector arithmetic). A recent need of software verification tools is the ability to cope with equations involving string constraints, i.e. equations over string variables composing equality between concatenation of strings and string variables. This need arose from the desire to do software verification of languages with string manipulation as a core part of the language (e.g. JavaScript and Java) [9,19]. To accomplish this goal, we have seen the advent of

Florin Manea's work was supported by the DFG grant MA 5725/2-1. Danny Bøgsted Poulsen's work was supported by the BMBF through the ARAMiS2 (01IS160253) project.

© Springer Nature Switzerland AG 2019
E. Filiot et al. (Eds.): RP 2019, LNCS 11674, pp. 93–106, 2019.
https://doi.org/10.1007/978-3-030-30806-3_8

dedicated string solvers as well as constraint solvers implementing string solving techniques. As an incomplete list we mention HAMPI [15], CVC4 [4], Ostrich [8], Sloth [11], Norn [1], S3P [20] and Z3str3 [5].

Although the need for string solving only recently surfaced in the software verification community, the problem is in fact older and known as *Word Equations* (a term that we will use from now on). The word equation satisfiability problem is to determine whether we can unify the two strings, i.e., transform them into two equal strings containing constant letters only, by substituting the variables consistently by strings of constants. For example, consider the equation defined by the two strings $XabY$ and $aXYb$, denoted $XabY \doteq aXYb$, with variables X, Y and constants a and b. It is satisfiable because X can be substituted by a and Y by b, which produces the equality $aabb = aabb$. In fact, substituting X by an arbitrary amount of a's and Y by an arbitrary amount of b's unifies the two sides of the equation.

The word equation problem is decidable [16] and NP-hard. In a series of works, Jeż [12,13] showed that word equations can be solved in non-deterministic linear space. It has been shown by Plandowski [18] that there exists an upper bound of $2^{2^{O(n^4)}}$ for the smallest solution to a word equation of length n. Having this in mind, a standard method for solving word equations is known as *filling the positions* [14,17]. In this method a length for each of the string variables is non-deterministically selected. Having a fixed length of the variables reduces the problem to lining up the positions of the two sides of the equation, and filling the unknown positions of the variables with characters, making the two sides equal.

In this paper we present a new solver for word equation with linear length constraints, Woorpje. In particular, it guesses the maximal length of variables and encodes a variation of *filling the positions* method into an automata-construction, thereby reducing the search for a solution to a reachability question of this automata. Preliminary experiments with a pure automata-reachability-based approach revealed however, that this would not scale for even small word equations. Woorpje therefore encodes the automata into SAT and uses the tool Glucose [3] as a backend. Unlike other approaches based on the filling the positions method (e.g. [6,19]), Woorpje does not need an exact bound for each variable, but only an upper bound. Experiments indicate that Woorpje is not only reliable but also competitive with the more mature CVC4 and Z3. Results indicate that Woorpje is quicker on pure word equations (no linear length constraints), and that CVC4 and Z3 mainly have an edge on word equations with linear constraints. This may be due to our naive solution for solving linear length constraints.

2 Preliminaries

Let $\mathbb{N}$ be the set of natural numbers, let $[n]$ be the set $\{ 0, 1, 2, \ldots, n-1 \}$ and $[n]_0$ the set $[n] \setminus \{ 0 \}$. For a finite set Δ of symbols, we let Δ^* be the set of all words over Δ and ε be the empty word. For an alphabet Δ and $a \notin \Delta$, we let

Δ_a denote the set $\Delta \cup \{ a \}$. For a word $w = x_0 x_1 \dots x_{n-1}$ we let $|w| = n$ refer to its length. For $i \in [|w|]$ we denote by $w[i]$ the symbol on the i^{th} position of w i.e. $w[i] = x_i$. For $a \in \Delta$ and $w \in \Delta^*$ we let $|w|_a$ denote the number of as in w. If $w = v_1 v_2$ for some words $v_1, v_2 \in \Delta^*$, then v_1 is called a *prefix* of w and v_2 is a *suffix* of w. In the remainder of the paper, we let $\Xi = \Sigma \cup \Gamma$ where Σ (Γ) is a set of symbols called letters (variables) and $\Sigma \cap \Gamma = \emptyset$. We call a word $w \in \Xi^*$ a *pattern* over Ξ. For a pattern $w \in \Xi^*$ we let $\mathrm{var}(w) \subseteq \Gamma$ denote the set of variables from Γ occurring in w. A *substitution* for Ξ is a morphism $S : \Xi^* \to \Sigma^*$ with $S(a) = a$ for every $a \in \Sigma$ and $S(\varepsilon) = \varepsilon$. Note, that to define a substitution S, it suffices to define $S(X)$ for all $X \in \Gamma$.

A *word equation* over Ξ is a tuple $(u, v) \in \Xi^* \times \Xi^*$ written $u \doteq v$. A substitution S over Ξ is a *solution* to a word equation $u \doteq v$ (denoted $S \models u \doteq v$) if $S(u) = S(v)$. A word equation $u \doteq v$ is *satisfiable* if there exists a substitution S such that $S \models u \doteq v$. A *system of word equations* is a set of word equations $P \subseteq \Xi^* \times \Xi^*$. A system of word equations P is satisfiable if there exists a substitution S that is a solution to all word equations (denoted $S \models E$). Karhumäki et al. [14] showed that for every system of word equations, a single equation can be constructed which is satisfiable if and only if the initial formula was satisfiable. The solution to the constructed word equation can be directly transferred to a solution of the original word equation system.

Bounded Word Equations. A natural sub-problem of solving word equations is that of *Bounded Word Equations*. In this problem we are not only given a word equation $u \doteq v$ but also a set of length constraints $\{|X| \leq b_X \mid X \in \Gamma \wedge b_X \in \mathbb{N}\}$. The bounded word equation is satisfiable if there exists a substitution S such $S \models u \doteq v$ and $|S(X)| \leq b_X$ for each $X \in \Gamma$. For convenience, we shall sometimes refer to the set of bounds b_X as a function $B : \Gamma \to \mathbb{N}$ such that $b_X = B(X)$.

Word Equations with Linear Constraints. A word equation with linear constraints is a word equation $u \doteq v$ accompanied by a system θ of linear Diophantine equations, where the unknowns correspond to the lengths of possible substitutions of the variables in Γ. A word equation with linear constraints is satisfiable if there exists a substitution S such that $S \models u \doteq v$ and S satisfies θ. Note that the bounded word equation problem is in fact a special case of word equations with linear constraints.

SAT Solving. A Boolean formula φ with finitely many Boolean variables $\mathrm{var}(\varphi) = \{ x_1, \dots, x_n \}$ is usually given in conjunctive normal form. This is a conjunction over a set of disjunctions (called clauses), i.e. $\varphi = \bigwedge_i \bigvee_j l_{i,j}$, where $l_{i,j} \in \bigcup_{i \in [n]} \{ x_i, \neg x_i \}$ is a literal. A mapping $\beta : \mathrm{var}(\varphi) \to \{ 0, 1 \}$ is called an *assignment*; for such an assignment, the literal l evaluates to true if and only if $l = x_i$ and $\beta(x_i) = 1$, or $l = \neg x_i$ and $\beta(x_i) = 0$. A clause inside a formula in conjunctive normal form is evaluated to true if at least one of its literals evaluates true. We call a formula φ *satisfied* (under an assignment) if all clauses are evaluated to true. If there does not exists a satisfying assignment, φ is unsatisfiable.

3 Word Equation Solving

In this section we focus on solving *Bounded Word Equations* and *Word Equations with Linear Constraints*. We proceed by first solving bounded word equations, and secondly, we discuss a minor change, that allows solving word equations with linear constraints.

3.1 Solving Bounded Word Equation

Recall that a bounded word equation consists of a word equation $u \doteq v$ along with a set of equations $\{|X| \leq b_X\}$ providing upper bounds for the solution of each variable X. In our approach we use these bounds to create a finite automaton which has an accepting run if and only if the bounded word equation is satisfiable.

Before the actual automata construction, we need some convenient transformations of the word equation itself. For a variable X with length bound b_X, we replace X with a sequence of new *'filled variables'* $X^{(0)} \cdots X^{(b_X-1)}$ which we restrict to only be substituted by either a single letter or the empty word. A pattern containing only filled variables, as well as letters, is called a *filled pattern*. For a pattern $w \in \Xi^*$ we denote its corresponding filled pattern by w_ξ. In the following, we refer to the alphabet of filled variables by Γ_ξ and by $\Xi_\xi = \Sigma \cup \Gamma_\xi$ the alphabet of the filled patterns. Let $S : \Xi^* \to \Sigma^*$ be a substitution for $w \in \Xi^*$. We can canonically define the induced substitution for filled patterns as $S_\xi : (\Sigma \cup \Gamma_\xi) \to \Sigma_\lambda$ with $S_\xi(a) = S(a)$ for all $a \in \Sigma$, $S_\xi(X^{(i)}) = S(X)[i]$ for all $X^{(i)} \in \Gamma_\xi$ and $i < |S(X)|$, and $S_\xi(X^{(j)}) = \lambda$ for all $X^{(j)} \in \Gamma_\xi$ and $|S(X)| \leq j < b_X$. Here, λ is a new symbol ($\lambda \notin \Xi_\xi$) to indicate an unused position at the end of a filled variable. Note that the substitution of a single filled variable always maps to exactly one character from Σ_λ, and, as soon as we discover $S_\xi\left(X^{(j)}\right) = \lambda$ for $j \in [b_X]$ it also holds that $S_\xi\left(X^{(i)}\right) = \lambda$ for all $j \leq i < b_X$. In a sense, the new element λ behaves in the same way as the neutral element of the word monoid Σ^*, being actually a place holder for this element ε. In the other direction, if we have found a satisfying filled substitution to our word equation, the two filled patterns obtained from the left hand side and the right hand side of an equation, respectively, we can transform it to a substitution for our original word equation by defining $S(X)$ as the concatenation $S_\xi\left(X^{(0)}\right) \ldots S_\xi\left(X^{(i)}\right)$ in which each occurrence of λ is replaced by the empty word ε, for all $X \in \Gamma$ and $i \in [b_X]$.

Our goal is now to build an automaton which calculates a suitable substitution for a given equation. During the calculation there are situations where a substitution does not form a total function. To extend a partial substitution $S : \Xi \nrightarrow \Sigma^*$ we define for $X \in \Xi$ and $b \in \Sigma^*$ the notation $S\left[\frac{X}{b}\right] = S \cup \{\, X \mapsto b \,\}$ whenever $S(X)$ is undefined and otherwise $S\left[\frac{X}{b}\right] = S$. This definition can be naturally applied to filled substitutions. We define a congruence relation which sets variables and letters in relation whenever their substitution with respect to a partial substitution S_ξ is equal or undefined. For all $a, b \in \Xi_\xi \cup \{\lambda\}$ we define

$$a \overset{S_\xi}{\sim} b \text{ iff } S_\xi(a) = S_\xi(b) \text{ or } S_\xi(b) \notin \Sigma_\lambda^* \text{ or } S_\xi(a) \notin \Sigma_\lambda^*.$$

Definition 1. *For a word equation $u \doteq v$ for $u, v \in \Xi^*$ and a mapping $B : \Gamma \to \mathbb{N}$ defining the bounds $B(X) = b_X$, we define the* equation automaton *$A(u \doteq v, B) = (Q, \delta, I, F)$ where $Q = ([|u_\xi| + 1] \times [|v_\xi| + 1]) \times (\Xi_{\xi_\lambda} \nrightarrow \Sigma_\lambda)$ is a set of states consisting of two integers which indicate the position inside the two words u_ξ and v_ξ and a partial substitution, the transition function $\delta : Q \times \Sigma_\lambda \to Q$ defined by*

$$\delta(((i,j), S), a) = \begin{cases} \left((i+1, j+1), S\left[\frac{u_\xi[i]}{a}\right]\left[\frac{v_\xi[j]}{a}\right]\right) & \text{if } u_\xi[i] \overset{S_\xi}{\sim} v_\xi[j] \overset{S_\xi}{\sim} a, \\ \left((i+1, j), S\left[\frac{u_\xi[i]}{\lambda}\right]\right) & \text{if } u_\xi[i] \overset{S_\xi}{\sim} \lambda = a, \\ \left((i, j+1), S\left[\frac{v_\xi[j]}{\lambda}\right]\right) & \text{if } v_\xi[j] \overset{S_\xi}{\sim} \lambda = a. \end{cases}$$

an initial state $I = ((0,0), \{ a \mapsto a \mid a \in \Sigma_\lambda \})$ and the set of final states $F = \{ ((i,j), S_\xi) \mid i = |u_\xi|, j = |v_\xi| \}$.

The state space of our automaton is finite since the filled substitution S_ξ maps each input to exactly one character in Σ. The automaton is nondeterministic, as the three choices we have for a transition are not necessarily mutually exclusive.

As an addition to the above definition, we introduce the notion of *location* as a pair of integers (i, j) corresponding to two positions inside the two words u_ξ and v_ξ. A location (i, j) can also be seen as the set of states of the form $((i, j), S)$ for all possible partial substitutions S.

A run of the above nondeterministic automaton constructs a partial substitution for the given equation which is extended with each change of state. The equation has a solution if one of the accepting states $(|u_\xi|, |v_\xi|, S)$, where S is a total substitution, is reachable, because the automaton simulates a walk through our input equation left to right, with all its positions filled in a coherent way.

Example 1. Consider the equation $u \doteq v$ for $u = aZXb, v = aXaY \in \Xi^*$. We choose the bounds $b_X = b_Y = b_Z = 1$. This will give us the words $u_\xi = aZ^{(0)}X^{(0)}b$ and $v_\xi = aX^{(0)}aY^{(0)}$. Figure 1 visualizes the corresponding automaton. A run starting with the initial substitution $S_i = \{ a \mapsto a \mid a \in \Sigma_\lambda \}$ reaching one of the final states gives us a solution to the equation. In this example we get the substitutions $Z \mapsto a, X \mapsto a, Y \mapsto b$ and $Z \mapsto a, X \mapsto \varepsilon, Y \mapsto b$.

Theorem 1. *Given a bounded word equation $u \doteq v$ for $u, v \in \Xi^*$,with bounds B, then the automaton $A(u \doteq v, B)$ reaches an accepting state if and only if there exists S such that $S \models u \doteq v$ and $|S(X)| \leq B(X)$ for all $X \in \Gamma$.*

SAT Encoding. We now encode the solving process into propositional logic. For that we impose an ordering on the finite alphabets $\Sigma = \{ a_0, \ldots, a_{n-1} \}$ and $\Gamma = \{ X_0, \ldots, X_{m-1} \}$ for $n, m \in \mathbb{N}$. Using the upper bounds given for all variables $X \in \Gamma$, we create the filled variables alphabet Γ_ξ. Further, we create

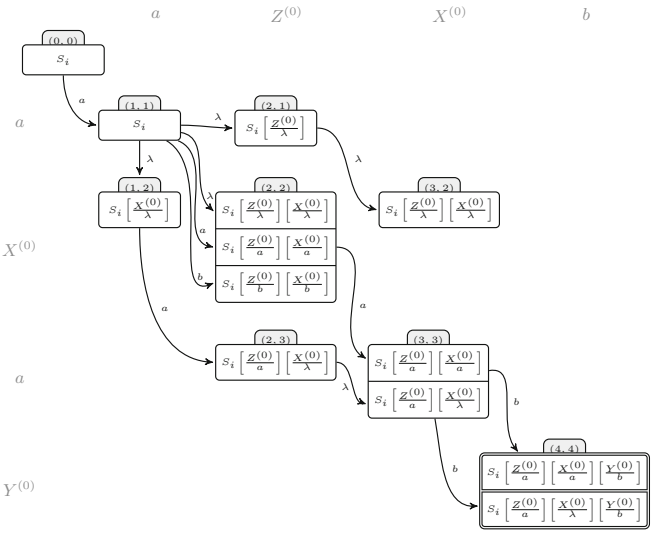

Fig. 1. Automaton for the word equation $aZXb \doteq aXaY$, with the states grouped according to their locations. Only reachable states are shown.

the Boolean variables $\mathsf{K}^a_{X^{(i)}}$, for all $X^{(i)} \in \Gamma_\xi$, $a \in \Sigma_\lambda$ and $i \in [b_X]$. Intuitively, we want to construct our formula such that an assignment β sets $\mathsf{K}^a_{X^{(i)}}$ to 1, if the solution of the word equation, which corresponds to the assignment β, is such that at position i of the variable X an a is found, meaning $S_\xi\left(X^{(i)}\right) = a$. To make sure $\mathsf{K}^a_{X^{(i)}}$ is set to 1 for exactly one $a \in \Sigma_\lambda$ we define the clause $\bigvee_{a\in\Sigma_\lambda} \mathsf{K}^a_{X^{(i)}}$ which needs to be assigned true, as well the constraints $\mathsf{K}^a_{X^{(i)}} \rightarrow \neg\mathsf{K}^b_{X^{(i)}}$, for all $a, b \in \Sigma_\lambda, X \in \Gamma, i \in [b_X]$ where $a \neq b$, which also need to be all true.

To match letters we add the variables $\mathsf{C}_{a,a} \leftrightarrow \top$ and $\mathsf{C}_{a,b} \leftrightarrow \bot$ for all $a, b \in \Sigma_\lambda$ with $a \neq b$. As such, the actual encoding of our equation can be defined as follows: for $w \in \{ u_\xi, v_\xi \}$ and each position i of w and letter $a \in \Sigma_\lambda$ we introduce a variable which is true if and only if $w[i]$ will correspond to an a in the solution of the word equation. More precisely, we make a distinction between constant letters and variable positions and define: $\mathsf{word}^a_{w[i]} \leftrightarrow \mathsf{C}_{w[i],a}$ if $w[i] \in \Sigma_\lambda$ and $\mathsf{word}^a_{w[i]} \leftrightarrow \mathsf{K}^a_{w[i]}$ if $w[i] \in \Gamma_\xi$. The equality of two characters, corresponding to position i in u and, respectively, j in v, is encoded by introducing a Boolean variable $\mathsf{wm}_{i,j} \leftrightarrow \bigvee_{a\in\Sigma_\lambda} \mathsf{word}^a_{u[i]} \wedge \mathsf{word}^a_{v[j]}$ for appropriate $i \in [|u_\xi|]$, $j \in [|v_\xi|]$.

Based on this setup, each location of the automaton is assigned a Boolean variable. As seen in Definition 1 we process both sides of the equation simultaneously, from left to right. As such, for a given equation $u \doteq v$ we create $n \cdot m = (|u_\xi| + 1) \cdot (|v_\xi| + 1)$ many Boolean variables $\mathsf{S}_{i,j}$ for $i \in [n]$ and $j \in [m]$. Each variable corresponds to a location in our automaton. The location $(0, 0)$ is our initial location and $(|u_\xi|, |v_\xi|)$ our accepting location. The goal is to find a path between those two locations, or, alternatively, a satisfying assignment β, which sets the variables corresponding to these locations to 1. Every path

between the location $(0,0)$ and another location corresponds to matching prefixes of u and v, under proper substitutions. We will call locations where an assignment β sets a variable $\mathsf{S}_{i,j}$ to 1, active locations. Our transitions are now defined by a set of constraints. We fix $i \in [n]$ and $j \in [m]$ in the following. The constraints are given as follows: The first constraint (1) ensures that every active location has at least one active successor. The next three constraints (2)–(4) ensure the validity of the paths we follow: from a location we can only proceed to exactly one other location, in order to find a satisfying assignment; therefore we disallow simultaneous steps in multiple directions. In (5), (6) we forbid using an λ-transition whenever there is another possibility of moving forward. In the same manner we proceed in the case of two matching λ in (7); this part is especially important for finding substitutions which are smaller than the given bounds. The idea applies in the same way for matching letters, whose encoding is given in (8). The actual transitions which are possible from one state to another are encoded in (9) by using our Boolean variables $\mathsf{wm}_{i,j}$ which are true for matching positions in the two sides of the equation. This constraint allows us to move forward in both words if there was a match of two letters in the previous location. When the transitions are pictured as movements in the plane, this corresponds to a diagonal move. A horizontal or vertical move corresponds to a match with the empty word. The last constraint (10) ensures a valid predecessor. This is supposed to help the solver in deciding the satisfiability of the obtained formula, i.e., to guide the search in an efficient way. It can be seen as a local optimization step.

$$\mathsf{S}_{i,j} \rightarrow \mathsf{S}_{i+1,j} \vee \mathsf{S}_{i,j+1} \vee \mathsf{S}_{i+1,j+1} \tag{1}$$

$$(\mathsf{S}_{i,j} \wedge \mathsf{S}_{i,j+1}) \rightarrow (\neg\mathsf{S}_{i+1,j+1} \wedge \neg\mathsf{S}_{i+1,j}) \tag{2}$$

$$(\mathsf{S}_{i,j} \wedge \mathsf{S}_{i+1,j}) \rightarrow (\neg\mathsf{S}_{i+1,j+1} \wedge \neg\mathsf{S}_{i,j+1}) \tag{3}$$

$$(\mathsf{S}_{i,j} \wedge \mathsf{S}_{i+1,j+1}) \rightarrow (\neg\mathsf{S}_{i,j+1} \wedge \neg\mathsf{S}_{i+1,j}) \tag{4}$$

$$\mathsf{S}_{i,j} \wedge \neg\mathsf{word}^{\lambda}_{u[i]} \rightarrow \neg\mathsf{S}_{i+1,j} \text{ and } \mathsf{S}_{i,j} \wedge \mathsf{word}^{\lambda}_{u[i]} \wedge \neg\mathsf{word}^{\lambda}_{v[j]} \rightarrow \mathsf{S}_{i+1,j} \tag{5}$$

$$\mathsf{S}_{i,j} \wedge \neg\mathsf{word}^{\lambda}_{v[j]} \rightarrow \neg\mathsf{S}_{i,j+1} \text{ and } \mathsf{S}_{i,j} \wedge \neg\mathsf{word}^{\lambda}_{u[i]} \wedge \mathsf{word}^{\lambda}_{v[j]} \rightarrow \mathsf{S}_{i,j+1} \tag{6}$$

$$\mathsf{S}_{i,j} \wedge \mathsf{word}^{\lambda}_{u[i]} \wedge \mathsf{word}^{\lambda}_{v[j]} \rightarrow \mathsf{S}_{i+1,j+1} \tag{7}$$

$$\mathsf{S}_{i,j} \wedge \mathsf{S}_{i+1,j+1} \rightarrow \mathsf{wm}_{i,j} \tag{8}$$

$$\mathsf{S}_{i,j} \leftrightarrow (\mathsf{S}_{i-1,j-1} \wedge \mathsf{wm}_{i-1,j-1}) \vee (\mathsf{S}_{i,j-1} \wedge \neg\mathsf{wm}_{i,j-1}) \vee (\mathsf{S}_{i-1,j} \wedge \neg\mathsf{wm}_{i-1,j}) \tag{9}$$

$$\mathsf{S}_{i+1,j+1} \rightarrow \mathsf{S}_{i,j} \vee \mathsf{S}_{i+1,j} \vee \mathsf{S}_{i,j+1} \tag{10}$$

The final formula is the conjunction of all constraints defined above. This formula is true iff location (n,m) is reachable from location $(0,0)$, and this is true iff the given word equation is satisfiable w.r.t. the given length bounds.

Lemma 1. *Let $u \doteq v$ be a word equation, B be the function giving the bounds for the word equation variable, and φ the corresponding formula consisting of the conjunction (1)–(10) and the earlier defined constraints in this section, then $\varphi \wedge \mathsf{S}_{0,0} \wedge \mathsf{S}_{|u_\xi|,|v_\xi|}$ has a satisfying assignment if and only if $A(u \doteq v, B)$ reaches an accepting state.*

Example 2. Consider the word equation $u \doteq v$ where $u = XaXbYbZ$ and $v = aXYYbZZbaa \in \Xi^*$ where $\Sigma = \{ a \}$ and $\Gamma = \{ X, Y, Z \}$. Using the approach discussed above, we find the solution $S(X) = aaaaaaaa$, $S(Y) = aaaa$ and $S(Z) = aa$ using the bounds $b_X = 8$ and $b_Y = b_Z = 6$. We set up an automaton with $32 \cdot 38 = 1216$ states to solve the equation. In Fig. 2 we show the computation of the SAT-Solver. Light grey markers indicate states considered in a run of the automaton. In this case only 261 states are needed. The dark grey markers visualize the actual path in the automaton leading to the substitution. Non-diagonal stretches are λ transitions.

3.2 Refining Bounds and Guiding the Search

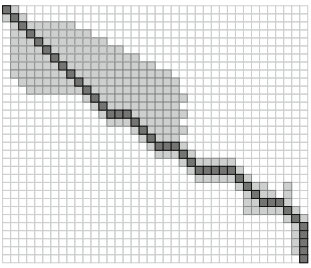

Fig. 2. Solver computation on $XaXbYbZ \doteq aXYYbZZbaa$

Initial experiments revealed a major inefficiency of our approach: most of the locations were not used during the search and only increased the encoding time. The many white markers in Fig. 2 indicating unused locations visualizes the problem. Since we create all required variables $x \in \Gamma$ and constraints for every position $i < b_X$, we can reduce the automaton size by lowering these upper bounds. Abstracting a word equation by the length of the variables gives us a way to refine the bounds b_X for some of the variables $X \in \Gamma$. By only considering length we obtain a Diophantine equation in the following manner. We assume an ordering on the variable alphabet $\Gamma = \{ X_0, \ldots, X_{n-1} \}$. We associate to each word equation variable X_j an integer variable I_j.

Definition 2. *For a word equation $u \doteq v$ with $\Gamma = \{ X_0, \ldots, X_{n-1} \}$ we define its* length abstraction *by $\sum_{j \in [n]} \left(|u|_{X_j} - |v|_{X_j} \right) \cdot I_j = \sum_{a \in \Sigma} |v|_a - |u|_a$ for $j \in [n]$.*

If a word equation has a solution S, then so does its length abstraction with variable $I_j = |S(X_j)|$. Our interest is computing upper bounds for each variable $X_k \in \Gamma$ relative to the upper bounds of the bounded word equation problem. To this end consider the following natural deductions

$$\begin{aligned} & \sum_{j \in [n]} \left(|u|_{X_j} - |v|_{X_j} \right) \cdot I_j = \sum_{a \in \Sigma} \left(|v|_a - |u|_a \right) \\ \iff \; & I_k = \frac{\sum_{a \in \Sigma} \left(|v|_a - |u|_a \right)}{|u|_{X_k} - |v|_{X_k}} - \frac{\sum_{j \in [n] \setminus k} \left(|u|_{X_j} - |v|_{X_j} \right) \cdot I_j}{|u|_{X_k} - |v|_{X_k}} \\ \implies \; & I_k \leq \frac{\sum_{a \in \Sigma} \left(|v|_a - |u|_a \right)}{\left(|u|_{X_k} - |v|_{X_k} \right)} - \frac{\sum_{j \in \kappa} \left(|u|_{X_j} - |v|_{X_j} \right) \cdot b_{X_j}}{\left(|u|_{X_k} - |v|_{X_k} \right)} = b^{\mathsf{S}}_{X_k}, \end{aligned}$$

where $\kappa = \{ m \in [n] \setminus k \mid (\mid u|_{X_k} - |v|_{X_k}) \cdot (|u|_{X_m} - |v|_{X_m}) < 0 \}$. Whenever $0 < b^{\mathsf{S}}_{X_k} < b_{X_k}$ holds, we use $b^{\mathsf{S}}_{X_k}$ instead of b_{x_k} to prune the search space.

The length abstraction is also useful because it might give information about the unsatisfiability of an equation: if there is no solution to the Diophantine equation, there is no solution to the word equation. We use this acquired knowledge and directly report this fact. Unfortunately whenever $|u|_X - |v|_X = 0$ holds for a variable X we cannot refine the bounds, as they are not influenced by the above Diophantine equation.

Guiding the Search. The length abstraction used to refine upper bounds can also be used to guide the search in the automaton. In particular it can restrict allowed length of one variable based on the length of others. We refer to the coefficient of variable I_j in Definition 2 by $\mathsf{Co}_{u,v}(I_j) = \left(|u|_{X_j} - |v|_{X_j}\right)$.

To benefit from the abstraction of the word equation inside our propositional logic encoding we use Reduced Ordered Multi-Decision Diagrams (MDD) [2]. An MDD is a directed acyclic graph, with two nodes having no outgoing edges (called `true` and `false` terminal nodes). A Node in the MDD is associated to exactly one variable I_j, and has an outgoing edge for each element of I_j's domain. In the MDD, a node labelled I_j is only connected to nodes labelled I_{j+1}. A row $(\mathsf{r}(I_j))$ in an MDD is a subset of nodes corresponding to a certain variable I_j.

We create the MDD following Definition 2. The following definition creates the rows of the MDD recursively. An MDD node is a tuple consisting of a variable I_j and an integer corresponding the partial sum which can be obtained using the coefficients and position information of all previous variables I_k for $k < j$. We introduce a new variable I_{-1} labelling the initial node of the MDD. The computation is done as follows:

$$\mathsf{r}(I_i) = \{\ (I_i, s + k \cdot \mathsf{Co}_{u,v}(X_i)) \mid s \in \{\ s' \mid (I_{i-1}, s') \in \mathsf{r}(I_{i-1})\ \}, k \in [b_{X_i}]\ \} \quad (11)$$

and $\mathsf{r}(I_{-1}) = \{\ (I_{-1}, 0)\ \}$. Since I_j is associated to the word equations variable X_j, we let $\mathsf{r}(X_j) = \mathsf{r}(I_j)$. We denote the whole set of nodes in the MDD by $M^C = \bigcup_{X \in \Gamma \cup \{\ I_{-1}\ \}} \mathsf{r}(X)$. The `true` node of the MDD is $(I_{n-1}, \mathsf{s}_\#)$, where $\mathsf{s}_\# = \sum_{a \in \Sigma} |v|_a - |u|_a$. If the initial creation of nodes did not add this node, the given equation (Definition 2) is not satisfiable hence the word equation has no solution given the set bounds. Furthermore there is no need to encode the full MDD, when only a subset of its nodes can reach $(I_{n-1}, \mathsf{s}_\#)$. For reducing the MDD nodes to this subset, we calculate all predecessors of a given node $(I_i, s) \in M^C$ as follows

$$\mathsf{pred}((I_i, s)) = \left\{\ (I_{i-1}, s') \;\middle|\; s' = s - k \cdot \mathsf{Co}_{u,v}(X_{i-1}), k \in [b_{X_{i-1}}]\ \right\}.$$

The minimized set $M = F(T)$ of reachable nodes starting at the only accepting node $T = \{\ (I_n, \mathsf{s}_\#)\ \}$ is afterwards defined through a fixed point by

$$T \subseteq F(T) \wedge \left(\forall\, p \in F(T) : q \in \mathsf{pred}(p) \wedge q \in M^C \Rightarrow q \in F(T)\right) \quad (12)$$

We continue by encoding this into a Boolean formula. For that we need information on the actual length of a possible substitution. We reuse the Boolean variables of our filled variables $X \in \Gamma_\xi$. The idea is to introduce $b_X + 1$ many Boolean

variables $(\mathsf{OH}_i(0) \dots \mathsf{OH}_i(b_X + 1))$ for each $X_i \in \Gamma$, where $\mathsf{OH}_i(j)$ is true if and only if X_i has length j in the actual substitution. This kind of encoding is known as a *one-hot encoding*. To achieve this we add a constraint forcing substitutions to have all λ in the end. We force our solver to adapt to this by adding clauses $\mathsf{K}^{\lambda}_{X^{(j)}} \rightarrow \mathsf{K}^{\lambda}_{X^{(j+1)}}$ for all $j \in [b_{X_i} - 1]$ and $X_i^{(j)} \in \Gamma_\xi$. The actual encoding is done by adding the constraints $\mathsf{OH}_i(0) \leftrightarrow \mathsf{K}^{\lambda} X_i^{(0)}$ and $\mathsf{OH}_i(b_{X_i}) \leftrightarrow \neg \mathsf{K}^{\lambda}_{X^{(b_{X_i}-1)}}$, which fixes the edge cases for the substitution by the empty word and no λ inside it. For all $j \in [b_{X_i}]_0$, we add the constraints $\mathsf{OH}_i(j) \leftrightarrow \mathsf{K}^{\lambda}_{X_i^{(j)}} \wedge \neg \mathsf{K}^{\lambda}_{X_i^{(j-1)}}$, which marks the first occurrence of λ. The encoding of the MDD is done nodewise by associating a Boolean variable $\mathsf{M}_{i,j}$ for each $i \in [|\Gamma|]$, where $(I_i, j) \in M$. Our goal is now to find a path inside the MDD from node $(I_{-1}, 0)$ to $(I_{n-1}, \mathsf{s}_\#)$. Therefore we enforce a true assignment for the corresponding variables $\mathsf{M}_{-1,0}$ and $\mathsf{M}_{n-1,\mathsf{s}_\#}$. A valid path is encoded by the constraint $\mathsf{M}_{i-1,j} \wedge \mathsf{OH}_i(k) \rightarrow \mathsf{M}_{i,s}$ for each variable $X_i \in \Gamma$, $k \in [b_{X_i}]_0$, where $s = j + k \cdot \mathsf{Co}_{u,v}(X_i)$ and $(I_i, s) \in M$. This encodes the fact that whenever we are at a node $(I_{i-1}, s) \in M$ and the substitution for a variable X_i has length k ($|S(X_i)| = k$), we move on to the next node, which corresponds to X_i and an integer obtained by taking the coefficient of the variable X_i, multiplying it by the substitution length, and adding it to the previous partial sum s. Whenever there is only one successor to a node (I_i, j) within our MDD, we directly force its corresponding one hot encoding to be true by adding $\mathsf{M}_{i-1,j} \rightarrow \mathsf{OH}_i(j)$. This reduces the amount of guesses on variables.

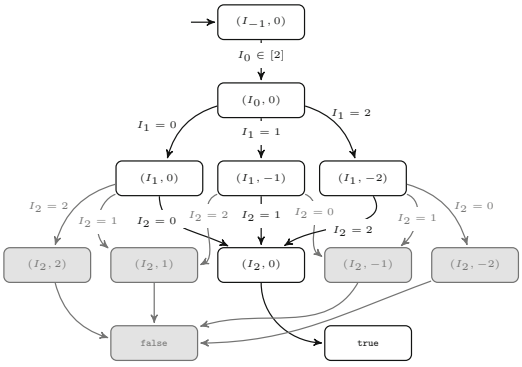

Fig. 3. The MDD for $aX_2X_0b \doteq aX_0aX_1$

Example 3. Consider the equation $u \doteq v$ for $u = aX_2X_0b, v = aX_0aX_1 \in \Xi^*$, where $\Sigma = \{a, b\}$ and $\Gamma = \{X_0, X_1, X_2\}$. The corresponding linear equation therefore has the form $0 \cdot I_0 + (-1) \cdot I_1 + 1 \cdot I_2 = 0$ which gives us the coefficients $\mathsf{Co}_{u,v}(X_0) = 0$, $\mathsf{Co}_{u,v}(X_1) = -1$ and $\mathsf{Co}_{u,v}(X_2) = 1$. For given bounds $b_{X_0} = b_{X_1} = b_{X_2} = 2$ the induced MDD has the form shown in Fig. 3. In this example $s_\# = 0$, and therefore $(I_2, 0)$ is the only node connected to the `true` node. The minimization of the MDD by using the fixed point decribed in (12) removes all grey nodes, since they are not reachable starting at the `true` node. The solver returns the substitution $S(X_0) = \varepsilon$, $S(X_0) = b$ and $S(X_0) = a$. It took the centred path consisting of the nodes $(I_{-1}, 0), (I_0, 0), (I_1, -1), (I_2, 0)$, `true` inside the MDD.

Adding Linear Length Constraints. Until now we have only concerned ourselves with bounded word equations. As mentioned in the introduction however, bounded equations with linear constraints are of interest as well. In particular, without

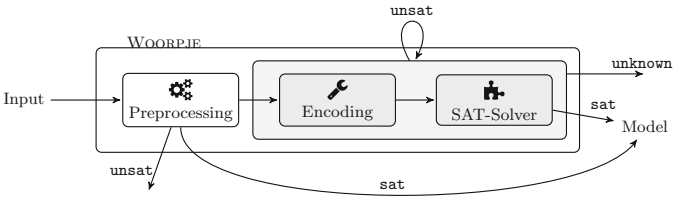

Fig. 4. Architecture of Woorpje

loss of generality we restrict to linear constraints of the form [2] $c_0 I_0 + \cdots + c_{n-1} I_{n-1} \leq c$ where $c, c_i \in \mathbb{Z}$ are integer coefficients and I_i are integer variables with a domain $D_i = \{ m \in \mathbb{N} \mid 0 \leq m \leq d_i \}$ and a corresponding $d_i \in \mathbb{N}$. Each I_i corresponds to the length of a substitution to a variable of the given word equation.

Notice that the structure of the linear length constraint is similar to that of Definition 2. For handling linear constraints we can adapt the generation of MDD nodes to keep track the partial sum of the linear constraint, and define the accepting node of the MDD to one where all rows have been visited and the inequality is true. We simply extend the set T which was used in the fix point iteration in (12) to the set $T = \{ (I_n, s) \mid (I_n, s) \in M^C \wedge s \leq \mathsf{s}_{\#} \}$.

4 Experiments

The approach described in the previous sections has been implemented in the tool Woorpje. The inner workings of Woorpje is visualised in Fig. 4. Woorpje first has a preprocessing step where obviously satisfiable/unsatisfiable word equations are immediately reported.

After the preprocessing step, Woorpje iteratively encodes the word equation into a propositional logic formula and solves it with Glucose [3] for increasing maximal variable lengths (i^2, where i is the current iteration). If a solution is found, it is reported. The maximal value of i is user definable, and by default set to 2^n where n is the length of the given equation. If Woorpje reaches the given bound without a verdict, it returns unknown.

We have run Woorpje and state of the art word equation solvers (CVC4 1.6, Norn 1.0.1, Sloth 1.0, Z3 4.8.4) on several word equation benchmarks with linear length constraints. The benchmarks range from theoretically-interesting cases to variations of the real-world application set Kaluza [19]. All tests were performed on Ubuntu Linux 18.04 with an Intel Xeon E5-2698 v4 @ 2.20 GHz CPU and 512 GB of memory with a timeout of 30 s.

We used five different kind of benchmarks. The first track (I) was produced by generating random strings, and replacing factors with variables at random, in a coherent fashion. This guarantees the existence of a solution. The generated word equations have at most 15 variables, 10 letters, and length 300. The second track (II) is based on the idea in Proposition 1 of [10], where the equation $X_n a X_n b X_{n-1} b X_{n-2} \cdots b X_1 \doteq a X_n X_{n-1} X_{n-1} b X_{n-2} X_{n-2} b \cdots b X_1 X_1 baa$ is

shown to have a minimal solution of exponential length w.r.t. the length of the equation. The third track (III) is based on the second track, but each letter b is replaced by the left hand side or the right hand side of some randomly generated word equation (e.g., with the methods from track (I)). In the fourth track (IV) each benchmark consists of a system of 100 small random word equations with at most 6 letters, 10 variables and length 60. The hard aspect of this track is solving multiple equations at the same time. Within the fifth track (V) each benchmark enriches a system of 30 word equations by suitable linear constraints, as presented in this paper. This track is inspired by the Kaluza benchmark set in terms of having many small equations enriched by linear length constraints. All tracks, except track II which holds 9 instances, consist of 200 benchmarks. The full benchmark set is available at https://www.informatik.uni-kiel.de/~mku/woorpje. Table 1 is read as follows: ✓ is the count of instances classified as correctly, where ■ marks the incorrect classified cases. For instances marked with ■ the solver returned no answer but terminated before the timeout of 30 s was reached, where in ⏸ marked cases the solver was killed after 30 s. The row marked by 🕓 states the overall solving time. The produced substitutions were checked regarding their correctness afterwards. The classification of ✗ was done by ad-hoc case inspection whenever not all solvers agreed on a result. In the cases one solver produced a valid solution, and others did not, we validated the substitutions manually. For the cases where one solver determined an equation is unsatisfied and all others timed out, we treated the unsat result as correct. This means that we only report errors if a solver reports unsat and we know the equation was satisfiable. During our evaluation of track I CVC4 crashed with a null-pointer exception regarding the word equation $dbebgddbecfcbbAadeeaecAgebegeecafegebdbagddaadbddcaeeebfabfefabfacdgAgaabgegagf \doteq dbebgddbeAfcbbAaIegeeAaDegagf$, where lowercase symbols are letters and uppercase symbols are variables. Worth mentioning is the reporting of 14 satisfiable benchmarks by the tool Sloth without being able to produce a valid model, while at least two other tools classified them as unsatisfiable. We treated this as an erroneous behaviour.

Table 1. Benchmark results (✓: correct classified, ■: reported unknown, ⏸: timed out after 30 s, ✗: incorrectly classified, 🕓: total Time in seconds)

	Track I					Track II					Track III					Track IV					Track V				
	✓	■	⏸	✗	🕓	✓	■	⏸	✗	🕓	✓	■	⏸	✗	🕓	✓	■	⏸	✗	🕓	✓	■	⏸	✗	🕓
woorpje	200	0	0	0	8.10	5	0	4	0	123.85	189	0	11	0	341.74	196	2	2	0	136.20	178	9	13	0	399.22
cvc4	182	0	17	0	543.32	1	0	8	0	240.03	165	0	35	0	1055.24	172	0	28	0	925.13	179	0	21	0	635.97
z3str3	197	1	2	0	105.07	0	9	0	0	0.33	93	43	58	6	2089.92	175	10	12	3	490.23	198	1	1	0	41.52
z3seq	183	0	17	0	545.24	9	0	0	0	1.81	126	0	73	1	2199.99	193	0	6	1	200.09	193	0	7	0	217.35
norn	176	0	20	4	1037.63	0	0	9	0	270.00	71	0	128	1	4038.83	60	0	72	68	3216.95	112	0	9	79	742.34
sloth	101	0	99	0	3658.34	7	0	2	0	124.56	121	0	67	12	2808.48	16	0	184	0	5615.81	9	2	187	2	5750.79

The result shows that Woorpje produces reliable results (0 errors) in competitive time. It outperforms the competitors in track I, III and IV and sticks

relatively tight to the leaders Z3str3, Z3Seq and CVC4 on track V. On track II Woorpje trails CVC4 and Z3Str3. The major inefficiency of Woorpje is related to multiple equations with large alphabets and linear length constraints.

It is worth emphasising, that the benchmarks developed here seem of intrinsic interest, as they challenge even established solvers.

5 Conclusion

In this paper we present a method for solving word equations by using a SAT-Solver. The method is implemented in our new tool Woorpje and experiments show it is competitive with state-of-the-art string solvers. Woorpje solves word equations instances that other solvers fail to solve. This indicates that our technique can complement existing techniques in a portfolio approach.

In the future, we aim to extend our approach to include regular constraints. As our approach relies on automata theory, it is expected that this could be achievable. Another step is the enrichment of our linear constraint solving. We currently do a basic analysis by using the MDDS. There are a few refinement steps described in [2] which seem applicable. A next major step is to develop a more efficient encoding of the alphabet of constants. Currently the state space explodes due to the massive branching caused by the usage of large alphabets.

References

1. Abdulla, P.A., et al.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_29
2. Abío, I., Stuckey, P.J.: Encoding linear constraints into SAT. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 75–91. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_9
3. Audemard, G., Simon, L.: On the glucose SAT solver. Int. J. Artif. Intell. Tools **27**(01), 1840001 (2018)
4. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD), pp. 55–59, October 2017
6. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_27
7. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, 8–10 December 2008, San Diego, California, USA, Proceedings, pp. 209–224. USENIX Association (2008). http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

8. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49 (2019)
9. Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., Trtik, M.: JBMC: a bounded model checking tool for verifying Java bytecode. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 183–190. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_10
10. Day, J.D., Manea, F., Nowotka, D.: The hardness of solving simple word equations. In: Proceedings of MFCS 2017. LIPIcs, vol. 83, pp. 18:1–18:14 (2017)
11. Holík, L., Jank P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proc. ACM Program. Lang. **2**(POPL), 4 (2017)
12. Jeż, A.: Recompression: a simple and powerful technique for word equations. In: 30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, 27 February- 2 March 2013, Kiel, Germany, pp. 233–244 (2013). https://doi.org/10.4230/LIPIcs.STACS.2013.233
13. Jeż, A.: Word equations in nondeterministic linear space. In: Proceedings of ICALP 2017. LIPIcs, vol. 80, pp. 95:1–95:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
14. Karhumäki, J., Mignosi, F., Plandowski, W.: The expressibility of languages and relations by word equations. J. ACM (JACM) **47**(3), 483–505 (2000)
15. 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, pp. 105–116. ACM (2009)
16. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Sbornik: Math. **32**(2), 129–198 (1977)
17. Plandowski, W., Rytter, W.: Application of Lempel-Ziv encodings to the solution of word equations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 731–742. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055097
18. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. In: 40th Annual Symposium on Foundations of Computer Science, pp. 495–500. IEEE (1999)
19. 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. IEEE (2010)
20. Trinh, M.-T., Chu, D.-H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 218–240. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_12