



Mathematical Programming Modulo Strings

Ankit Kumar  and Panagiotis Manolios 
Northeastern University

Email: {ankitk, pete}@ccs.neu.edu

Abstract—We introduce **TranSeq**, a non-deterministic, branching transition system for deciding the satisfiability of conjunctions of string equations. **TranSeq** is an extension of the **Mathematical Programming Modulo Theories (MPMT)** constraint solving framework and is designed to enable useful and computationally efficient inferences that reduce the search space, that encode certain string constraints and theory lemmas as integer linear constraints and that otherwise split problems into simpler cases, via branching. We have implemented a prototype, **SeqSolve**, in **ACL2s**, which uses **Z3** as a back-end solver. String solvers have numerous applications, including in security, software engineering, programming languages and verification. We evaluated **SeqSolve** by comparing it with existing tools on a set of benchmark problems and our experimental results show that **SeqSolve** is both practical and efficient.

I. INTRODUCTION

The problem of solving string equations has interested mathematicians and computer scientists for decades. Security, software engineering and verification applications, in particular, have generated a renewed interest in string solvers. Security applications include finding cross-site scripting vulnerabilities in Web applications, SQL injection attacks and fuzzing [1], [2], [3], [4], [5]. Software engineering applications include testcase generation, symbolic evaluation and flow analysis [6], [7], [8]. Programming language applications include type inference for array processing languages [9][10].

The basic problem is easy to define. Let Γ be a non-empty set of constants. The elements of Γ^* form a free monoid, *i.e.*, a structure with a single associative operation, corresponding to concatenation, and an identity element ϵ . Elements of Γ^* are called strings or words. Let \mathcal{X} be a set of variables over Γ^* and let \mathcal{Y} be a set of variables over Γ such that Γ , \mathcal{X} and \mathcal{Y} are disjoint. Elements in \mathcal{Y} are also called *unit variables*. Let $\mathcal{Z} = \mathcal{X} \cup \mathcal{Y}$. Elements of the free monoid $(\Gamma \cup \mathcal{Z})^*$ are called sequences, again with ϵ as the identity. A *normal substitution* is a partial function $\rho : \mathcal{Z} \rightarrow (\Gamma \cup \mathcal{Z})^*$. Every substitution can be extended to the domain $(\Gamma \cup \mathcal{Z})$, by defining $\rho(a) = a$ for all a not in the domain of ρ . We can also extend the domain to $(\Gamma \cup \mathcal{Z})^*$ in the standard way. $w\rho$ stands for the application of substitution ρ to the sequence w and it extends naturally to sequence equations. A solution of a set of equations $\{u_1 = v_1, u_2 = v_2, \dots, u_n = v_n\}$ is a substitution ρ that when applied to each equation yields identical sequences, *i.e.*, $\{u_1\rho = v_1\rho, u_2\rho = v_2\rho, \dots, u_n\rho = v_n\rho\}$ is a set of syntactic equivalences over $(\Gamma \cup \mathcal{Z})^*$. The problem statement is: given a set of sequence equations $\{u_1 = v_1, u_2 = v_2, \dots, u_n = v_n\}$ find a solution if there exists one, otherwise return *unsat*.

Related Work. Makanin, in 1977, proved that the satisfiability of string equations is decidable [11]. A series of results on complexity followed, after which Plandowski showed that the problem is in polynomial space [12]. String solvers supporting a variety of theories are available, *e.g.*, **Z3Str3** [13], **CVC4** [14], [15], **S3P** [16], **Norn** [17], **TRAU** [18], **StrSolve** [19], **Sloth** [2], **Kepler₂₂** [20] and **HAMPI** [1]. **Z3Str3** and **CVC4** are multi-theory SMT solvers which consider unbounded string equations with concatenation, substring, replace and length functionality. Together with **S3P** and **Norn**, these tools handle a variety of string constraints including string equations, length constraints and regular language membership. However, these tools are incomplete. **HAMPI** works only for problems with one string variable of fixed size. **Kepler₂₂** is a decision procedure for the straight line and quadratic fragments of string equations. **Norn** and **TRAU** can decide only the acyclic fragment whereas **Sloth** decides straight line and acyclic fragments. To the best of our knowledge, there is no solver that for decidable fragments is both theoretically and practically complete, *e.g.*, none of the above solvers are able to solve the string equation $xcyczvycya = yacwazvbux$. Therefore it is important to explore new techniques for solving string equations. One of the most promising existing techniques uses context-dependent techniques to improve the reasoning of string constraints in the context of **DPLL(T)**-based SMT solvers [15]. Similarly, our work introduces new techniques for reasoning in the context of **BC(T)**-based (Branch and Cut Modulo T) **MPMT** solvers [21], [22].

Contributions. Our contributions include (1) **TranSeq**, a new non-deterministic, branching transition system that can be used as part of the **MPMT** framework for combining decision procedures, (2) the **SeqSolve** solver, an implementation of **TranSeq** which resolves non-deterministic choices in a way designed to infer as much as possible with as few computational resources as possible, (3) proof sketches of soundness, completeness and termination for **TranSeq** and (4) an evaluation of **SeqSolve** using a set of benchmarks from related work, as well as **Remora** examples [9], [10]. We use publicly available benchmarks, being careful to evaluate only the string solving capabilities of our tool, not irrelevant aspects of the underlying SMT/**MPMT** tools. The integration of our solver into SMT/**MPMT** tools is briefly discussed. There are over 1,100 problems in our benchmark and no existing string solver can solve all of them. Experimental results show that **SeqSolve** is more efficient and complete than existing solvers.

Paper Outline. Section II illustrates some techniques we use to reason about string equations through motivating examples. Section III defines basic terms used to define our transition system and algorithm. Section IV describes TranSeq and SeqSolve. Section V gives proofs sketches of correctness and termination; due to space limitations full definitions and proofs will appear in a full version of the paper. Section VI describes implementation considerations of our prototype and Section VII contains our evaluation. Conclusions and future work appear in Section VIII.

II. ILLUSTRATIVE EXAMPLES

In this section, we highlight some of the techniques used in our string equation solver, via a collection of examples, where a, b, c, \dots are constants (elements of Γ) and u, v, w, x, y and z are string variables (elements of \mathcal{X}).

Example 1 [ConstUnsat] Consider the string equation $b = a$. The constant b differs from the constant a so this equation is unsatisfiable. Our algorithm determines by performing partial evaluation that includes evaluating constant prefixes and suffixes of equations.

Example 2 [Trim] Consider $xab = xbb$. Our algorithm trims common prefixes and suffixes from both sides of the input equation to get $a = b$ which is unsatisfiable by ConstUnsat.

Example 3 [Decompose] Consider $xyazy = yxubyz$. Prefixes xy and yx have provably equal lengths. So do the suffixes zy and yz . Therefore our algorithm decomposes the input equation into three equations: $xy = yx$, $a = ub$ and $zy = yz$. Equation $a = ub$ can be further decomposed into $a = b$ and $u = \epsilon$, which is unsatisfiable by ConstUnsat.

Example 4 [EqLength] Consider $uvxayvu = vuyxuv$. Decomposition generates the two *distinct* equations $uv = vu$ and $xay = yx$. Notice that if an equation is satisfiable, then both sides have to have the same length and our algorithm generates the constraint $l_x + 1 + l_y = l_y + l_x$ where l_x and l_y denote the lengths of x and y , respectively, which is unsatisfiable.

Example 5 [EqConsts] Consider $ax = xb$. If the equation is satisfiable, then both sides of the equation must have the same number of occurrences of each constant. To enforce this, our algorithm generates the constraint $1 + c_a^x = c_a^x$, where c_a^x is the number of a 's in x , which is unsatisfiable.

Example 6 [VarElim] Consider the set of (implicitly conjoined) string equations $\{uv = vu, xa = ax, cy = x\}$. The last equation has the form of a definition and this allows our algorithm to eliminate x by applying the appropriate substitution to the set of equations, giving us $\{uv = vu, cya = acy\}$. Since $cya = acy$ is unsatisfiable, so is the set.

Example 7 [VarSplit] Consider $xxa = cyx$. One side starts with the constant c so the other side must also start with c , which means x cannot be empty and must start with a c . Our algorithm detects this and adds the equation $x = c\hat{x}$, where \hat{x}

is a new string variable. After eliminating x and trimming, we wind up with the equation $\hat{x}c\hat{x}a = yc\hat{x}$, which decomposes into $\hat{x}c = y$ and $\hat{x}a = c\hat{x}$. The EqConsts analysis (Example 5) infers that the second equation is unsatisfiable. Our algorithm also does this for suffixes.

Example 8 [VarSubst] Consider $wuzwuz = cywuz$. The equation is equi-satisfiable with $xxa = cyx$: we substitute a new string variable, x , for the sequence of string variables, wuz , thereby eliminating all occurrences of w, u and z from all string equations. The resulting equation is unsatisfiable by VarSplit (see Example 7).

Example 9 [Rewrite] Consider the set of (implicitly conjoined) string equations $\{zv = ba, xxazv = cyxba\}$. The first equality can be used to rewrite the second equality to $xxazv = cyxzv$ which can be trimmed to $xxa = cyx$, which is unsatisfiable, as per Example 7.

Example 10 [LenSplit] Consider $xbyu = caxzb$. The length of the prefix xb is strictly less than the length of the prefix cax , which allows us to infer that $yu = \hat{y}zb$ for some new string variable $\hat{y} \neq \epsilon$. We can rewrite yu to $\hat{y}zb$ (see Example 9) and after trimming, we wind up with the equation $x\hat{y}z = cax$, which is unsatisfiable (see Example 5).

Example 11 [EqWords] Consider $xbca = ycba$. Let W_{ca}^x and W_{ca}^y be the number of occurrences of a word ca in x and y respectively. If the equation is satisfiable, then both sides must have the same number of ca occurrences. To enforce this, our algorithm generates the constraint $W_{ca}^x + 1 + W_{ca}^y = W_{ca}^y + W_{ca}^x$, which is unsatisfiable. Consider $bubxaccv = vbabxcw$, which shows that counting words requires more care than what the above example suggests, *e.g.*, to count the occurrences of bc , we have to take into account whether c is a prefix of w , whether b is a suffix of x , whether x is empty, and so on. We use 0-1 indicator variables P_c^w, S_b^x and ϵ_x , denoting the above conditions, respectively. Now, with just the ab occurrence analysis, we can use variable splitting on w (w ends in an a) and then on v (v ends in an a) to derive a contradiction.

Example 12 [SAT] None of the string solvers we tried are able to solve the string equation $xcyczvyeya = yacwazvbux$. This equation is outside the scope of Kepler₂₂, StrSolve, Hampi and Sloth. Sloth, TRAU and S3P return *unsat*, which is wrong. Norn, Z3Str3 and CVC4 timed out after 1,000 seconds, which shows that existing tools are incomplete, in a practical sense. Our solver finds the assignment $x = aba, y = ab, u = cab$ and $v, w, z = \epsilon$ in a fraction of a second.

III. BLOCKS, SUBSTITUTIONS AND THEORIES

Suppose that a sequence u has an l length subsequence of consecutive occurrences of the constant a . This subsequence can be compactly represented by the pair (a, l) , which we refer to as a *block*: pairs in $\Gamma \times PExp$ where

$$PExp := \mathbb{P} \mid x \mid PExp + PExp \mid PExp - PExp$$

and x is a variable over positive natural numbers, \mathbb{P} . We require that a $PExp$ is positive. A sequence that allows blocks

is called an *extended sequence* (es); an *extended sequence equation* (ese) is similarly defined. The set of extended sequences es is $(\Gamma \cup (\Gamma \times PExp) \cup \mathcal{Z})^*$. We define a function $compress : es \rightarrow ((\Gamma, PExp) \cup \mathcal{Z})^*$ which given an (extended) sequence, replaces contiguous occurrences of each constant by its block such that no two blocks of the same constant are adjacent to each other, thus returning a unique *maximally compressed sequence*. We define the following useful functions, which given an extended sequence U : (1) $Elms : es \rightarrow 2^{\Gamma \cup \mathcal{Z} \cup (\Gamma, PExp)}$ returns the set of elements of U ; (2) $Atoms : es \rightarrow 2^{\Gamma \cup \mathcal{Z}}$ returns the set of variables and constants occurring in U ; (3) $Consts : es \rightarrow 2^\Gamma$ returns the set of constants in U . (4) $Vars : es \rightarrow 2^{\mathcal{Z}}$ returns the set of variables in U . These functions extend naturally to $eses$ and to sets of ess and $eses$. An extended sequence U represents a sequence u if u is obtained from U by replacing every block (α, n) by α repeated n times. Note that n needs to be a positive integer. Extended sequences U and V are syntactically equivalent if they represent the same sequence. We use \equiv to denote syntactic equivalence. For example, $(\alpha, 2)\alpha X \equiv \alpha(\alpha, 2)X$, as both of them represent the sequence $\alpha\alpha\alpha X$. Notice that syntactic equivalence is an equivalence relation.

We define a substitution σ to be a partial function of the form $\sigma : es \rightarrow es$. Given substitution σ , let σ_v be σ restricted to \mathcal{Z} and let σ_s be $\sigma \setminus \sigma_v$. Let $dom(f)$ and $cod(f)$ be the domain and codomain of function f , respectively. Note that $dom(\sigma_v) \subseteq \mathcal{Z}$, so σ_v is a normal substitution. Substitutions σ_v and σ_s partition σ and have disjoint domains. We say that σ_s is an *extended substitution*, as its domain may contain sequences. We require substitutions to be *well-typed*, i.e., σ_v must map unit variables to sequences of unit length. $U\sigma$ stands for the application of substitution σ to $U \in es$. This notation extends naturally to equations and sets of equations. In order for application to be well-defined, we require that σ is *consistent*, as defined below. We say that σ is *uniquely defined* if for all $x, y \in dom(\sigma)$, if $x \neq y$ then $Atoms(x) \cap Atoms(y) = \emptyset$. To see why we require this, consider the case where $\sigma_v = \{x:ab, y:a\}$ and $\sigma_s = \{yax:aba\}$; note that $(yax)\sigma$ is ambiguous.

Given two uniquely defined substitutions, σ and τ , we say that they are *equivalent*, written $\sigma \equiv \tau$, if for all $U \in es$, we have $U\sigma \equiv U\tau$. We say that σ is *consistent* if it is uniquely defined and $\langle \exists \tau :: dom(\tau) \subseteq \mathcal{Z} \wedge \sigma \equiv \tau \rangle$, i.e., σ is equivalent to a normal substitution. Consider $\sigma = \{xay:bbb\}$. Even though σ is uniquely defined, it can not be expressed as a normal substitution. From now on, unless we say otherwise, all substitutions are implicitly assumed to be consistent. A substitution σ is said to solve an *ese* $U = V$ if $U\sigma \equiv V\sigma$; σ solves Q , a set of *eses*, if σ solves every *ese* in Q . A *word* ab is an *es* in which no prefix is a suffix.

Theorem 1. *If σ is a consistent substitution and $x_1, \dots, x_n \in \mathcal{Z}$ are distinct variables such that $n \geq 0$ and $\{x_1, \dots, x_n\} \cap Vars(dom(\sigma)) = \emptyset$, then $\sigma \cup \{x_1:V_1, \dots, x_n:V_n\}$ (where V_1, \dots, V_n are extended sequences of the right type) is a*

consistent substitution.

A theory is a pair $T = (\Sigma, \mathbf{I})$, where Σ is a signature and \mathbf{I} is a class of Σ -interpretations, the models of T . A set of formulas, Ψ , entails in T a Σ -formula ϕ , written $\Psi \models_T \phi$, if every interpretation in \mathbf{I} that satisfies all formulas in Ψ satisfies ϕ as well. The set Ψ is unsatisfiable in T if $\Psi \models_T \perp$.

Let LIA be a theory with signature $(0, 1, +, -, \leq)$ interpreted over the standard model of integers \mathbb{Z} . A linear constraint is a formula of the form $\sum_{i \in [1..n]} a_i x_i \leq b$, where x_i are variables and a_i and b are integer constants. For a collection of linear constraints C , $C \models_{LIA} \perp$ means that C is unsatisfiable in LIA , whereas $C \not\models_{LIA} \perp$ means that a model exists for C . Our algorithm accepts and generates linear constraints on the conjunction of input string equations. It assumes a sound, complete and terminating backend ILP solver for such constraints. Let ES be a theory of (extended) sequences over a signature Σ_{ES} with two sorts: extended sequences (es) and integers (\mathbb{Z}) along with an infinite set of variables over each sort. Σ_{ES} also includes constants in Γ , $PExp$ expressions, blocks, (extended) sequences and functions len interpreted as the string length function, $countConst$ interpreted as a function counting the number of a specified constant in a sequence and $countWords$ interpreted as a function counting the number of specified words in a sequence.

IV. MPMT-BASED STRING SOLVER

Our algorithm, $SeqSolve$, accepts a conjunction of string equations Q as well as initial constraints C_{init} and returns either *unsat*, *unknown* or *sat* along with a solution. C_{init} is a set of *initemp*'s defined as

$$\begin{aligned} LExp &:= \mathbb{Z} \mid x \mid len(u) \mid LExp + LExp \mid LExp - LExp \\ initemp &:= LExp (< \mid \leq \mid > \mid \geq \mid = \mid \neq) LExp \end{aligned}$$

where x is an integer variable (\mathbb{Z}), u is an (extended) sequence and $len : es \rightarrow \mathbb{N}$ is a function that returns length of u . We refer to variables occurring in $PExp$ and $LExp$ expressions as *numeric variables*. Central to the algorithm is a non-deterministic transition system $TranSeq$ with rules that operate on configurations consisting of (extended) sequence equations and sets of LIA constraints.

Our decision procedure can be integrated into MPMT solvers in a fine-grained way since MPMT is based on branching, using the branch-and-cut framework. However, in order to make the paper more self contained, we present $TranSeq$ and $SeqSolve$ with as few dependencies on the MPMT framework as possible.

Our decision procedure can be integrated into SMT solvers using the idea of *recursive solvers*: these are solvers whose decision procedures may depend on the solvers themselves. For example, we can integrate our decision procedure into Z3, even though our decision procedure uses Z3 as a backend solver, by using a separate Z3 process to handle the LIA constraints and one can use this integration as a backend solver

for yet another decision procedure, and so on. As far as we know, we are the first to propose the idea of recursive solvers. For SMT solvers like Z3 that provide contexts and a stack with a push-pop interface to manage constraints, integration can be achieved using these features by creating a new context or stack frame, thereby allowing decision procedures to query the SMT solver without polluting its state.

A. Configurations

The algorithm works on configurations that include tuples of the form $\langle \text{unsat} \rangle$, $\langle \text{unknown} \rangle$, $\langle \text{sat}, \sigma, \mathcal{C} \rangle$ and $\langle Q, \sigma, \text{vars}, \mathcal{C} \rangle$ where (1) Q is a set of *eses*, (2) $\sigma : es \rightarrow es$ is a (consistent) substitution, (3) vars is a superset of the variables in \mathcal{Z} which occur in Q , (4) \mathcal{C} is a union of constraints $C_{len}, C_{consts}, C_{words}$ and a set of linear constraints corresponding to C_{init} , where (i) C_{len} is a set of linear constraints regarding the lengths of variables in vars . For $x \in \text{vars}$, l_x is an integer variable denoting the length of x and ϵ_x is a 0-1 indicator variable indicating whether x is empty. Linear constraints in C_{len} and C_{init} are over these integer variables and over *PExp* variables; (ii) C_{consts} is a set of linear constraints regarding the number of occurrences of constants in variables from vars . For $x \in \text{vars}$, n_a^x is an integer variable denoting the number of occurrences of the constant a in x . Linear constraints in C_{consts} are over these variables as well as over variables of C_{len} ; (iii) C_{words} is a set of linear constraints regarding the number of words occurring in variables from vars . Let $x \in \text{vars}$ and $s \in \text{consts}^*$. Then W_s^x denotes the number of s occurrences in x ; P_s^x and S_s^x are 0-1 indicator variables indicating whether x begins with s and ends with s , respectively. Linear constraints in C_{words} are over these variables as well as over variables of C_{len} .

The reason why we distinguish between C_{len}, C_{consts} and C_{words} is that it makes it easier to consider simplified transition systems that include only a subset of these kinds of constraints. We define sets consts and C_{fuel} where (1) consts is a superset of the constants from Γ occurring in Q and (2) C_{fuel} is a set of linear constraints over the l_x variables, used to guarantee termination. Both consts and C_{fuel} are generated once and never modified by our transition system. The rules in TranSeq depend on auxiliary functions that are used to generate LIA constraints or to simplify equations. All of these functions are described in the full version of this paper.

B. Transition System TranSeq

We describe a non-deterministic transition system TranSeq. TranSeq consists of a set of rules called derivation rules. A derivation rule applies to a configuration K if all of the rule's premises are satisfied by K . Such a rule is *enabled* for K . A derivation tree is a tree where each node is a configuration and the children of any non-leaf node are exactly the configurations obtained by applying one of the derivation rules to the node. A configuration is *terminal* if no rules can be applied to it. We prove that terminal configurations are either of the form $\langle \text{unsat} \rangle$, in which case we call them *unsat* terminal nodes, $\langle \text{unknown} \rangle$, in which case we call them *unknown* terminal

nodes, or of the form $\langle \text{sat}, \sigma, \mathcal{C} \rangle$, in which case we call them *sat* terminal nodes and σ, \mathcal{C} can be used to generate a satisfying assignment to the equations appearing in the root of the tree.

A configuration $K = \langle Q, \sigma, \text{vars}, \mathcal{C} \rangle$ is *sat* (*unsat*) iff $Q \cup \mathcal{C} \cup C_{fuel}$ is *sat* (*unsat*). K is \mathcal{C} -*sat* iff $Q \cup \mathcal{C}$ is *sat*. Notice that an *unknown* terminal node may be *sat* (or *unsat*). This discrepancy is due to the C_{fuel} constraints, which are provable upper bounds on the lengths of minimal solutions, but only if we have no length constraints in the input, so it is possible that K is \mathcal{C} -*sat*, but the configuration is *unsat* and we generate an *unknown* terminal node. The derivation rules of TranSeq are given in guarded assignment form and can be categorized into three groups: (1) **Terminal rules:** Rules that yield terminal nodes. (2) **Inference rules:** Rules that generate new inferences. (3) **Branching rules:** Rules that generate multiple subproblems.

A derivation tree is *closed* if all its leaf nodes are terminal nodes. A derivation tree is *unsat-closed* if it is closed and all of its leaf nodes are *unsat*-terminal nodes. A derivation tree is *unknown-closed* if it is closed, has at least one *unknown* terminal node and has no *sat*-terminal nodes. We prove that if a derivation tree is *unsat-closed*, then the conjunction of the equations and constraints appearing in the root of the tree are unsatisfiable. A derivation tree for a set of sequence equations $Q = \{u_1=v_1, u_2=v_2, \dots, u_n=v_n\}$ and some initial length constraints C_{init} (if provided) is a tree whose root, $\text{genRoot}(Q, C_{init})$, is defined in Algorithm 1, where $\text{Choose}(X)$ is a function that given a non-empty set X , returns an element of X . C_{len}, C_{consts} and C_{words} are initialized with linear constraints by functions $\text{initLen}, \text{initConsts}$ and initWordCount respectively. These functions generate constraints which are satisfiable for any string variable. C_{fuel} comprises of constraints on the size of the minimum solution of each equation in Q which are calculated in function initFuel and are based on results from [23]. The sets consts and vars are supersets of the constants and variables occurring in Q , respectively.

We define the function toLIA , which given an *initexp* returns a linear constraint. Given $\text{len}(x)$, where x is a sequence variable, toLIA returns l_x ; we extend this to *initexp* expressions in the obvious way and use toLIA to also generate fuel constraints. We denote the set of words we are interested in counting as \mathcal{W} , which is global.

C. Rules in TranSeq

We now describe each rule in TranSeq. The conclusion of a rule describes how each component of a configuration is changed, if it does. Rules with two or more conclusions separated by \parallel , are branching rules, where each of the configurations are starting configurations for new branches in their derivation tree. In derivation rules, if Q is relevant, it appears on the top-left corner in the premise and as the last line of a concluding branch. A, t is an abbreviation for $A \cup \{t\}$ and $A \sim t$ abbreviates $A \setminus \{t\}$. We use $\equiv (\neq)$ for syntactic equivalence

Algorithm 1 $\text{genRoot}(Q, C_{init})$: Given input set of string equations Q , genRoot generates the root node of a derivation tree.

```

1:  $\sigma \leftarrow \{\}$ 
2:  $\text{vars} \leftarrow \{x \mid x \in \mathcal{Z} \wedge x \in uv \wedge u=v \in Q\}$ 
3:  $\text{consts} \leftarrow \{a \mid a \in uv \wedge a \in \Gamma \wedge u=v \in Q\}$ 
4: if  $\text{consts} = \emptyset \wedge \text{vars} \cap \mathcal{Y} \neq \emptyset$  then
5:    $\text{consts} \leftarrow \{\text{Choose}(\Gamma)\}$ 
6:  $C_{len} \leftarrow \bigcup_{v \in \text{vars}} \text{initLen}(v)$ 
7:  $C_{consts} \leftarrow \bigcup_{v \in \text{vars}} \text{initConsts}(v, \text{consts})$ 
8:  $C_{words} \leftarrow \bigcup_{v \in \text{vars}} \text{initWordCount}(v, w)$ 
9:  $\mathcal{C} \leftarrow \text{toLIA}(C_{init}) \cup C_{len} \cup C_{consts} \cup C_{words}$ 
10:  $C_{fuel} \leftarrow \text{initFuel}(Q)$ 
11: return  $\langle Q, \sigma, \text{vars}, \mathcal{C} \rangle$ 

```

(in-equivalence) and $= (\neq)$ for semantic equality (inequality).

Terminal rules When Q is empty, if \mathcal{C} is unsatisfiable, LIAUnsat infers unsat otherwise Sat returns a sat configuration.

$$\frac{\mathcal{C} \models_{\text{LIA}} \perp}{\langle \text{unsat} \rangle} \text{LIAUnsat} \quad \frac{\{\} \quad \mathcal{C} \not\models_{\text{LIA}} \perp}{\langle \text{sat}, \sigma, \mathcal{C} \rangle} \text{Sat}$$

If the fuel constraints are needed to show unsatisfiability, then the rule FuelUnsat returns unsat if no initial linear constraints were provided, otherwise the rule Unknown returns unknown . Terminal rules are subject to fairness constraints, as described later.

$$\frac{C_{init} = \emptyset \quad \mathcal{C} \cup C_{fuel} \models_{\text{LIA}} \perp}{\langle \text{unsat} \rangle} \text{FuelUnsat}$$

$$\frac{C_{init} \neq \emptyset \quad \mathcal{C} \not\models_{\text{LIA}} \perp \quad \mathcal{C} \cup C_{fuel} \models_{\text{LIA}} \perp}{\langle \text{unknown} \rangle} \text{Unknown}$$

If there exists an equation with syntactically different extended sequences on both sides, ConstUnsat infers unsat .

$$\frac{\{U=V, \dots\} \quad U \neq V \quad \text{Vars}(UV) = \emptyset}{\langle \text{unsat} \rangle} \text{ConstUnsat}$$

Note that we do not apply substitution σ to U and V when checking for syntactic equivalence, as shown below.

$$\frac{\{U=V, \dots\} \quad U\sigma \neq V\sigma \quad \text{Vars}(UV) = \emptyset}{\langle \text{unsat} \rangle} \text{ConstUnsat}$$

This is because, for any equation $U=V \in Q$, we get the original rule due to $U\sigma = U$ as a result of the invariant $Q\sigma = Q$, which we prove later.

When one side of an extended equation contains a constant or a block, while the other side is empty, ConstEmpty deduces unsat . If both sides begin with blocks of unequal constants, DiffConsts deduces unsat .

$$\frac{\{U=\epsilon, \dots\} \quad \alpha \in \text{Atoms}(U) \quad \alpha \in \text{consts}}{\langle \text{unsat} \rangle} \text{ConstEmpty}$$

$$\frac{\{(\alpha, l)U=(\beta, m)V, \dots\} \quad \alpha \neq \beta}{\langle \text{unsat} \rangle} \text{DiffConsts}$$

If one side of an equation contains a unit variable while the other side is empty, then YVarEmpty infers $\langle \text{unsat} \rangle$.

$$\frac{\{U=\epsilon, \dots\} \quad e \in U \quad e \in \mathcal{Y}}{\langle \text{unsat} \rangle} \text{YVarEmpty}$$

The rules ConstEmpty and DiffConsts deduce unsat based on how terms in an equation start, but there is a symmetry here that allows us to define rules that make the same deduction based on how terms end. For example, the symmetric version of DiffConsts would start with $\{U(\alpha, l) = V(\beta, m), \dots\}$, but would otherwise be identical to DiffConsts . When rules have this kind of symmetry, we denote it by underlining the name of the rule in its definition. These symmetric rules help with efficiency, but are not needed for completeness, so to simplify the rest of the presentation, we proceed as if they do not exist.

Inference rules Trim removes syntactically equal prefixes and suffixes from both sides of an equation; note that one of a, b can be ϵ . EqElim removes eses whose both sides are syntactically equivalent. Observe that Trim can be used to reduce an equation $U=V$ which is syntactically equivalent on both sides, to get $\epsilon=\epsilon$, in which case we get syntactic equivalence of both sides trivially.

$$\frac{\{aUb=cVd, \dots\} \quad a \equiv c \quad \{U=U, \dots\}}{\{U=V, \dots\}} \text{Trim} \quad \frac{\{U=U, \dots\}}{\{\dots\}} \text{EqElim}$$

Decompose splits an ese $U=V$ into multiple equations using length constraints. A simple example is given in Example 3.

$$\frac{\{U=V, \dots\} \quad |\text{splitEq}(U, V, \mathcal{C})| > 1}{\text{splitEq}(U, V, \mathcal{C}) \cup \{\dots\}} \text{Decompose}$$

Compress converts an equation $u=v \in Q$ into a maximally compressed sequence. Observe that the premise requires that there is at least one constant element in $u=v$. Note that blocks such as $(a, 1)$ are not constants, as they are not elements of Γ .

$$\frac{\{u=v, \dots\} \quad \text{Elems}(uv) \cap \Gamma \neq \emptyset}{\{\text{compress}(u)=\text{compress}(v), \dots\}} \text{Compress}$$

VarSubst formalizes the idea from Example 8. Given W , a non-empty subsequence in Q satisfying the conditions below, the rule replaces W with a new variable z . We show later that for every node in a derivation tree generated by our algorithm, $Q\sigma = Q$ holds; hence, the first condition for consistency of substitutions is satisfied. The second consistency condition is satisfied due to the premise that requires atoms of W and $Q\{W:z\}$ to be disjoint. Hence, the substitution in the new configuration is consistent. The LIANewVar procedure generates numeric constraints for new variables. After this rule, it is called implicitly whenever a new variable is introduced.

$$\frac{\{U=V, \dots\} \quad \langle \exists S, T :: SWT=U \wedge |W| > 1 \rangle \quad \text{Atoms}(W) \subseteq \text{vars} \quad z \in \mathcal{X} \quad z \notin \text{vars} \quad \text{Atoms}(W) \cap \text{Atoms}(\{U=V, \dots\}\{W:z\}) = \emptyset}{\text{LIANewVar}(z) \quad \sigma \leftarrow \sigma, W:z \quad \{U=V, \dots\}\{W:z\}} \text{VarSubst}$$

Rewrite replaces a subsequence S of U by T , given that $S=T$ is an equation in Q . Rewrite can choose which occurrences to replace. Infinite derivation trees are ruled out with a fairness requirement that only allows us to use the Rewrite rule a finite number of times.

$$\frac{\{U=V, S=T, \dots\} \quad S \in U}{\{U\{S:T\}=V, S=T, \dots\}} \text{ Rewrite}$$

EqLength, EqConsts and EqWords generate length, constant count and word count constraints implied by an equation. Function `equateWordCount` returns a linear constraint equating the number of occurrences of a word w in U and V .

$$\frac{\{U=V, \dots\} \quad \text{equateLen}(U, V) \not\subseteq \mathcal{C}}{C_{len} \leftarrow C_{len} \cup \text{equateLen}(U, V)} \text{ EqLength}$$

$$\frac{\{U=V, \dots\} \quad \text{equateConsts}(U, V) \not\subseteq \mathcal{C}}{C_{consts} \leftarrow C_{consts} \cup \text{equateConsts}(U, V, consts)} \text{ EqConsts}$$

$$\frac{\{U=V, \dots\} \quad w \in consts \geq 2 \quad \text{equateWordCount}(U, V, w) \not\subseteq \mathcal{C}}{C_{words} \leftarrow C_{words} \cup \text{equateWordCount}(U, V, w)} \text{ EqWords}$$

VarElim allows us to eliminate variables.

$$\frac{\{x=V, \dots\} \quad x \notin V \quad x \in \mathcal{X}}{\sigma \leftarrow \sigma, x:V \quad \{\dots\}\{x:V\}} \text{ VarElim}$$

Given an equation where one side starts with c occurrences of variable x and the other starts with m occurrences of constant β , the rule `VarSplit` infers shape information about x involving fresh variable y . x can not be empty, and the prefix of x^c must be syntactically equivalent to (β, m) . Hence, `VarSplit` infers that x is $(\beta, k)y$, where $c * k \geq m$. Note that c is a constant, hence expressions such as $c * k$ do not take us out of the LIA fragment. Also note that if $k < m$, y will have to start with β as well, which we do not want. Hence we add an implication that if $k < m$ then y is empty. We extend the set of equations with $x=(\beta, k)y$. Anytime we extend a the set of equations with an equation of the form $x = \dots$, we call `VarElim` to eliminate the variable x .

$$\frac{\{x^c(\alpha, l)U=(\beta, m)V, \dots\} \quad \alpha \neq \beta, c > 0 \quad x, y \in \mathcal{X} \quad y \notin vars}{C_{len} \leftarrow C_{len}, k > 0, (c-1) * k < m \leq c * k, k < m \Rightarrow \epsilon_y = 1} \text{ VarSplit}$$

$$C_{words} \leftarrow C_{words}, k < m \Rightarrow S_\beta^x = 1$$

$$\{x=(\beta, k)y, x^c(\alpha, l)U=(\beta, m)V, \dots\}$$

Length constraints alone may not always be enough to split an equation. `LenSplit` introduces a new variable on one side of an equation such that the resulting equation is clearly split into smaller and possibly more tractable equations. Example 10 illustrates a simple example.

$$\frac{\{UW=SzV, \dots\} \quad C \models_{LIA} len(U) < len(Sz) \quad y, z \in \mathcal{X} \quad y \notin vars}{C_{len} \leftarrow C_{len}, \epsilon_y = 0} \text{ LenSplit}$$

$$\{Uy=Sz, W=yV, \dots\}$$

Inferences made by the backend LIA solver can be used to infer sequence variables. `LIAEmpty` concludes that a variable x is empty if $\epsilon_x = 1$ is derived by the solver. Similarly, x starts (ends) with α iff the solver derives $P_\alpha^x = 1$ ($S_\alpha^x = 1$).

$$\frac{C \models_{LIA} \epsilon_x = 1 \quad x \in vars}{\{x=\epsilon, \dots\}} \text{ LIAEmpty} \quad \frac{C \models_{LIA} P_\alpha^x = 1 \quad y \in \mathcal{X} \quad x \in vars \quad y \notin vars}{\{x=\alpha y, \dots\}} \text{ LIABegin}$$

$$\frac{C \models_{LIA} S_\alpha^x = 1 \quad y \in \mathcal{X} \quad x \in vars \quad y \notin vars}{\{x=y\alpha, \dots\}} \text{ LIAEnd}$$

Given an equation where one side is empty, `XVarEmpty` infers that a variable $x \in \mathcal{X}$ in the other side must also be empty. If the two sides of an `ese` start with unit variables x and y , then `DiffYVars` infers that both the variables must be equal.

$$\frac{\{U=\epsilon, \dots\} \quad x \in U \quad x \in \mathcal{X}}{\{x=\epsilon, U=\epsilon, \dots\}} \text{ XVarEmpty} \quad \frac{\{xU=yV, \dots\} \quad x \neq y \quad x, y \in \mathcal{Y}}{\{x=y, U=V, \dots\}} \text{ DiffYVars}$$

Branching rules Given an equation where one side starts with a block of α , while the other side starts with a unit variable e , `UnitConst` infers that either the length of the α block is greater than one, or equal to one. Observe that some constraints in this rule are emphasized with a wavy underline. If such constraints are implied by \mathcal{C} , we can directly jump to their corresponding branch. Practically, it helps to not branch, if one of the underlined constraints can be derived in the premise.

$$\frac{\{eU=(\alpha, l)V, \dots\} \quad e \in \mathcal{Y}}{C_{len} \leftarrow C_{len}, \underline{l=1} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{l \geq 1}} \text{ UnitConst}$$

$$\{e=\alpha, U=V, \dots\} \quad \{e=\alpha, U=(\alpha, l-1)V, \dots\}$$

Given an equation where one side starts with a unit variable e while the other side starts with sequence variable y , `UnitVar` infers that either y is empty, or e is a prefix of y .

$$\frac{\{eU=yV, \dots\} \quad e \in \mathcal{Y} \quad y, z \in \mathcal{X} \quad z \notin vars}{C_{len} \leftarrow C_{len}, \underline{\epsilon_y = 1} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{\epsilon_y = 0}} \text{ UnitVar}$$

$$\{y=\epsilon, eU=V, \dots\} \quad \{y=ez, U=zV, \dots\}$$

If both sides of an equation start with blocks of the same constant α , `SimConsts` infers that either both blocks have the same length or one of them has length more than the other. So this rule should have three branches, one equating l and m , while the other two deducing a strict inequality between them. However, there are two branches, one equating l and m , while the other deducing $\hat{m} > \hat{l}$. This is because, for the sake of conciseness we introduce ‘‘hatted’’ variables $\hat{U}, \hat{V}, \hat{l}, \hat{m}$ and $\hat{\beta}$. A branch with hatted variables signifies the presence of another branch where the hatted variables are replaced by their substitutions defined as:

$$\{\hat{x}:y, \hat{y}:x, \hat{X}:Y, \hat{Y}:X, \hat{U}:V, \hat{V}:U, \hat{l}:m, \hat{m}:l, \hat{\alpha}:\beta, \hat{\beta}:\alpha\}$$

Notice that we also have underlined constraints in the conclusion. So, the rule `SimConsts` represents six rules, three after expanding hatted variables where none of the underlines constraints is implied by \mathcal{C} , and the rest considering presence of each of the underlined constraints in the premise of its corresponding rule.

$$\frac{\{(\alpha, l)U=(\alpha, m)V, \dots\}}{C_{len} \leftarrow C_{len}, \underline{m=l} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{\hat{m} \geq \hat{l}} \quad \text{SimConsts}} \\ \{U=V, \dots\} \quad \{\hat{U}=(\alpha, \hat{m} - \hat{l})\hat{V}, \dots\}$$

Similar to `SimConsts`, `DiffXVars` also uses both hatted variables and underlined constraints which gives rise to a total of ten rules. If both sides of an equation start with syntactically different variables $x, y \in \mathcal{X}$, and none of the underlined constraints is implied by \mathcal{C} , then `DiffXVars` infers that either one of them is empty or they are semantically equal or one of them is a prefix of the other.

$$\frac{\{xU=yV, \dots\} \quad x \neq y \quad z \notin \text{vars} \quad x, y \in \mathcal{X} \quad z \in \mathcal{X}}{C_{len} \leftarrow C_{len}, \underline{l_{\hat{x}} > l_{\hat{y}}}, \parallel C_{len} \leftarrow C_{len}, \underline{l_x = l_y}, \quad \text{DiffXVars}} \\ \{\underline{\epsilon_{\hat{x}} = \epsilon_{\hat{y}} = \epsilon_z = 0, \hat{x} = l_{\hat{y}} + l_z} \quad \epsilon_x = \epsilon_y = 0\} \\ \{\hat{x} = \hat{y}z, z\hat{U} = \hat{V}, \dots\} \quad \{x = y, U = V, \dots\} \\ \parallel C_{len} \leftarrow C_{len}, \underline{\epsilon_{\hat{x}} = 1} \\ \{\hat{x} = \epsilon, \hat{U} = \hat{y}\hat{V}, \dots\}$$

Finally, `VarConst` fires when one side of an equation starts with a constant block (α, l) while the other side starts with a variable x . Again, `VarConst` represents eight rules due to the presence of underlined constraints in its branching conclusions. Assuming none of these constraints is implied by \mathcal{C} , the first branch sets x empty; second branch sets length of x less than l ; third branch equated x to (α, l) , while the last branch sets x as a block of α whose length is greater than l , possibly followed by another variable y that does not start with α .

$$\frac{\{xU=(\alpha, l)V, \dots\} \quad x, y \in \mathcal{X} \quad y \notin \text{vars}}{C_{len} \leftarrow C_{len}, \underline{\epsilon_x = 1} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{0 < l_x < l} \quad \text{VarConst}} \\ \{x = \epsilon, U = (\alpha, l)V, \dots\} \quad \{x = (\alpha, l_x), U = (\alpha, l - l_x)V, \dots\} \\ C_{len} \leftarrow C_{len}, \underline{0 < l_x = l} \quad \parallel \quad C_{len} \leftarrow C_{len}, \underline{0 < l < l_x} \\ \{x = (\alpha, l), U = V, \dots\} \quad \{x = (\alpha, l_x)y, xU = (\alpha, l)V, \dots\}$$

D. SeqSolve definition

We define `SeqSolve` in Algorithm 2. It takes a set of sequence equations W and an optional set of length constraints C_{init} as input and either returns a *sat* with a solution, *unknown* or *unsat*.

V. CORRECTNESS OF SEQ SOLVE

Full proofs of correctness of `SeqSolve` appear in the full version of this paper. In the interest of brevity, we outline the structure of proofs in this section. First, we define correctness.

Algorithm 2 `SeqSolve` takes a set of (extended) sequence equations W and optionally a set of linear constraints C_{init} as input and either returns a *sat* with a solution, *unknown* or *unsat*.

```

1:  $T \leftarrow \text{genRoot}(W, C_{init})$ 
2: while  $\exists$  a non-terminal leaf node  $n \in T$  do
3:   apply an enabled TranSeq rule to  $n$ 
4:   if sat terminal node  $\langle \text{sat}, \sigma, \mathcal{C} \rangle$  generated then
5:     generate a satisfying assignment  $\psi$  from  $\sigma, \mathcal{C}$ 
6:     return sat,  $\psi$ 
7:   if  $\exists$  leaf node  $\langle \text{unknown} \rangle \in T$  then
8:     return unknown
9:   else
10:    return unsat

```

Definition 1. A string equation solver is an algorithm that takes as input a set of string equations and a set of linear constraints. Its output is either “*Unsat*,” “*Unknown*,” or “*Sat*” and an assignment.

Definition 2. A string equation solver is sound if it never lies, by which we mean: (1) when it returns “*Sat*,” the conjunction of the string equations and the linear constraints is satisfiable and the assignment returned is a satisfying assignment and (2) when it returns “*Unsat*,” the conjunction of the string equations and the linear constraints is unsatisfiable.

Definition 3. A string equation solver is partially correct if it is sound and terminating.

Definition 4. A string equation solver is fully correct if it is sound, terminating and never returns “*Unknown*.”

Note that a sound solver can be turned into a partially correct solver by adding a timeout, which results in the solver returning “*Unknown*.” We prove that our solver is fully correct for the theory of string equations by showing that when the input consists of only a conjunction of string equations Q , our transition system generates a derivation tree that is unsat-closed iff the input is unsatisfiable; otherwise it generates a derivation tree containing a *sat* terminal node, from which we can extract a satisfying assignment for the input. When the input also includes linear constraints, our solver is partially correct as it may also generate an *unknown*-closed derivation tree. We show that `SeqSolve` is sound using the following theorems.

Theorem 5. Given inputs Q, C_{init} such that `SeqSolve` generates a tree T with a *sat* terminal node $\langle \text{sat}, \sigma, \mathcal{C} \rangle$, then σ, \mathcal{C} can be used to generate a solution for Q, C_{init} .

A configuration is *var-compliant* iff it is of the form $\langle Q, \sigma, \text{vars}, \dots \rangle$ where $\text{Vars}(\sigma) \subseteq \text{vars}$ (by $\text{Vars}(\sigma)$ we mean $\text{Vars}(\text{dom}(\sigma)) \cup \text{Vars}(\text{cod}(\sigma))$). A configuration is *numvar-compliant* iff (1) it is of the form $\langle Q, \sigma, \text{vars}, \mathcal{C} \rangle$ and all numeric variables appearing in it are also in \mathcal{C} and (2) for a variable $x \in \text{vars}$, $\text{initLen}(x) \cup \text{initConsts}(x, \text{consts}) \cup \text{initWordCount}(x, \text{consts}) \subseteq \mathcal{C}$. A configuration is *good* iff it is either terminal or it is disjoint, *var-compliant* and *numvar-compliant*. A derivation tree is *good* if all of its nodes are

good configurations. It turns out that all SeqSolve-generated derivation trees are good.

Lemma 7. *Given input Q, C_{init} where Q is a set of (extended) sequence equations and C_{init} is a set of linear constraints, $genRoot$ returns a good, non-terminal configuration.*

Lemma 12. *TranSeq rules preserve goodness, i.e., when applied to a good configuration, they produce good configurations.*

SeqSolve is subject to the following fairness conditions: (1) LIAUnsat, FuelUnsat and Unknown are *weakly-fair* rules. First note that once any of these rules is enabled, it stays enabled. We require that no branch of a derivation tree contains a suffix in which a weakly-fair rule is infinitely enabled, yet never applied. (2) Rewrite can only be applied a finite number of times along any branch.

A *fair* derivation tree is one which respects the above fairness conditions. SeqSolve generates fair and good derivation trees. We use good derivation trees to show that TranSeq is sound.

Theorem 6. *Every TranSeq rule is sound when applied to a good configuration.*

The termination of SeqSolve (and TranSeq) depends on a bound on the minimum lengths of solutions of string equations as described in [23] and on fair derivation trees.

Theorem 9. *SeqSolve is terminating.*

Theorem 10. *SeqSolve is a partially correct string equation solver.*

Theorem 11. *SeqSolve is a fully correct string equation solver when the input does not include any linear constraints.*

VI. IMPLEMENTATION OF SEQ SOLVE

Our implementation of SeqSolve along with all the benchmarks used is publicly available [24]. SeqSolve is implemented in ACL2s [25] which allows us to (1) define datatypes like blocks, sequences and valid Z3 expressions (used to query Z3) (2) define TranSeq rules, which requires proving termination and input/output contracts (input/output types) (3) prove basic theorems relating datatypes (subtypes, etc) and properties needed for above proofs and (4) make essential use of the Z3 interface ACL2s provides to solve ILP constraints. SeqSolve provides various settings that can be used to control how aggressively it generates linear constraints; however, all of the results reported in this paper are with the default settings. We implemented SeqSolve as a standalone decision procedure as opposed to making it a part of an MPMT solver. This makes it easier to compare our tool with other string solvers in an apples-to-apples way, avoiding the complications that would arise from the use of different underlying solvers and frameworks.

We apply a few TranSeq rules until we reach a fixpoint before generating the derivation tree in order to simplify the input problem. These preprocessing steps include Decompose,

VarElim, VarSubst and Compress. After reaching a fixpoint, we use LIAUnsat to check if the set of initial constraints and the linear constraints we generated above are *unsat*.

In our implementation of the rule EqWords, we only use words with the property that no non-empty prefix of w is a suffix of w . Since our solver makes many low-level calls to Z3, it does this in an incremental way. In addition, care is taken to avoid unnecessary calls to Z3, e.g., LIAUnsat is not checked after running Trim, EqElim, Decompose, Compress, VarSubst, Rewrite and VarElim, because in all of these rules, we do not update \mathcal{C} . We do not apply any branching rules, unless we have no other options. Our implementation supports string operations like charAt, contains, indexOf, substr, prefixOf and suffixOf. Each of these operations can be converted to a problem in the theory of extended sequences e.g., given charAt constraint $e = (str.at\ s\ n)$, we convert it into the conjunction of the string equation $s = xey$ and $len(x) = n$, where $e \in \mathcal{V}$ and $x, y \in \mathcal{X}$. Given the constraint $(str.contains\ s\ t)$, we convert it into the string equation $s = xty$ where $x, y \in \mathcal{X}$.

VII. EVALUATION

We compared our solver against Z3Str2 and Z3Str3 (Z3 version 4.8.8), Norn 1.0, Z3-Trau, Sloth 1.0 and CVC4 1.7. These are the only string solvers we know of that solve string equations with length constraints and ran without crashing. In [26], the tools CVC4, Z3Str2 and S3 are evaluated in which S3 is found to be 5 times slower than Z3Str2 and crashed on about 4.5% of problems in the Kaluza [27] benchmarks. We ran all of the selected tools on Kaluza and Stringfuzz-generated [28] benchmarks, as well as on benchmarks consisting of problem instances pertinent to type inference in Remora [9], [10], a dependently typed array processing language. The type of an array term in Remora encodes the shape of the array as a list of dimensions (natural numbers). Our work was motivated by the problem of inferencing these shapes which reduces to solving string equations. For example, suppose that X has dimensions $[a\ 3]b$ and Y has dimensions $b[3]z$, where a is a single dimension, while b and z are lists of dimensions, and juxtaposition indicates concatenation. If X and Y are used in a context where they must have the same dimensions, then for the program to be well-typed, we require that the string equation $a3b = b3z$ is satisfiable. One solution is $b = []$, $z = [3]$ and $a = 3$, in which case X and Y are 2-dimensional matrices with shape $[3\ 3]$.

We used all of the problems in the above mentioned benchmarks that were in the extended sequence theory, thus, excluding problems in Kaluza that used other constructs. This allows us to evaluate only our contribution, the string solver, not the underlying solvers. In total, we have 1,178 problems, of which 903 are *sat* problems and 275 are *unsat* problems. We cross-verified the tools and for all benchmark problems, all tools that gave definitive answers agreed on the classification of the problem. All experiments were performed on the same machine, which was running macOS Catalina 10.14.6 with a 2.7GHz Intel Core i5 CPU and 8 GB of memory. The timeout

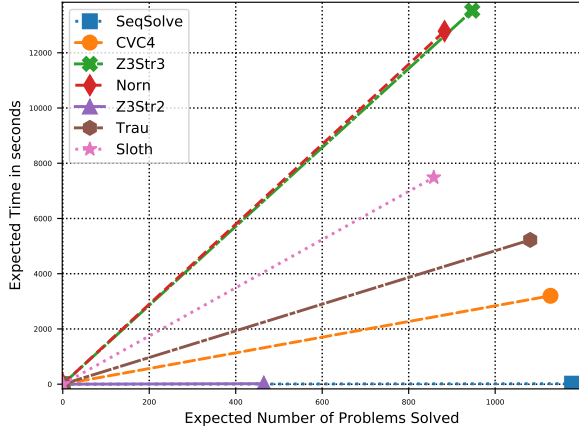


Fig. 1. Performance of SeqSolve, CVC4, Z3-Str3, Norn, Sloth, Trau and Z3-Str2 on solved benchmarks across all three benchmark sets.

for each problem was set to 60 seconds. Figure 1 shows the results of the performance evaluation, using what we call a *ray plot*. Ray plots are designed to visually depict the results of the evaluation in as simple a way as possible. On the x -axis we have the expected number of problems solved and on the y -axis we have the expected time in seconds. Suppose you want to determine how long it will take to solve n benchmark problems, say 800; just look at the line $x = 800$ and you will see that SeqSolve will take about 100 seconds, CVC4 will take over 2,000 seconds, Z3Str3 will take just under 12,000 seconds, Norn will take about 5,500 seconds and Z3Str2 can only solve about 500 problems, so it will never solve 800 problems. Symmetrically, if you want to determine how many problems you can expect to solve in t seconds, just look at the line $y = t$. This is a simpler plot than a cactus plot, which shows similar information, but with problems ordered, on a per-tool basis, from easiest to hardest. These orderings can vary significantly from tool to tool and there is no way for a user of the tool to determine how easy or difficult a problem will be, so it is not clear what benefit there is to this extra complexity. It is easy to generate ray plots; just run all the benchmark problems and draw a ray from the origin to the (p, t) coordinate, where p is the number of problems solved and t is the time taken. This is equivalent to shuffling the problems many times and taking the average of the running times for the shufflings.

In Table I, we show a table version of the experimental evaluation. Tuples under “Solved” give the number of problems solved for the Stringfuzz-generated, Kaluza and handcrafted benchmarks, respectively. In addition to the time in seconds, we also show the number of problems for which solvers returned *unknown*, timed out or returned incorrect result (X). We ran the tools without giving them a timeout and our scripts killed jobs that were taking too long, but some

TABLE I
PERFORMANCE OF SOLVERS ON ALL BENCHMARKS

Solver	Solved	Time (s)	Unknown	Timeout	X
SeqSolve	1,178: 780/344/54	176	0	0	0
CVC4	1,128: 736/344/48	3,200	0	50	0
Z3Str3	947: 552/344/51	13,527	6	225	0
Norn	883: 492/344/47	12,783	120	175	0
Z3Str2	465: 121/332/12	18	713	0	0
Trau	1,081: 692/344/45	5,223	18	78	1
Sloth	858: 462/344/52	7,486	0	319	64

tools returned *unknown* before timeouts occurred. Notice that SeqSolve beats all the other string solvers in terms of the standard ordering, which is based on first the number incorrect results, then on the number of problems solved and finally on the time taken.

Acknowledgements: We thank Andrew Walter for integrating Z3 with ACL2s, which was indispensable.

VIII. CONCLUSION AND FUTURE WORK

We introduced a new non-deterministic, branching transition system, TranSeq, for deciding the satisfiability of conjunctions of string equations and length constraints. TranSeq extends the MPMT framework for combining decision procedures and we prove that it is both sound and complete. We implemented a prototype, SeqSolve, which is based on TranSeq and resolves non-deterministic choices in a way designed to infer as much as possible with as few computational resources as possible. We evaluated SeqSolve by comparing it with existing tools on a suite of benchmark problems and found that SeqSolve solved more problems and was faster than existing solvers. In our ongoing work, we plan to extend the scope of TranSeq so that it supports richer classes of constraints. We also plan to reason about the implementation, as it is mostly written in ACL2s, which is built on top of the ACL2 theorem prover.

REFERENCES

- [1] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, “HAMPI: A solver for string constraints,” in *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.
- [2] L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar, “String constraints with concatenation and transducers solved efficiently,” in *Proceedings of the ACM on Programming Languages (PACMPL)*, 2018.
- [3] F. Yu, M. Alkhalaf, and T. Bultan, “Stranger: An automata-based string analysis tool for php,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2010.
- [4] P. Godefroid, N. Klarlund, and K. Sen, “Dart: Directed automated random testing,” in *Programming Language Design and Implementation*, 2005.
- [5] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing,” in *Programming Language Design and Implementation*, 2008.
- [6] A. S. Christensen, A. Møller, and M. I. Schwartzbach, “Precise analysis of string expressions,” in *Proceedings of the 10th International Conference on Static Analysis*, 2003.

- [7] R. Majumdar and R. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, 2007.
- [8] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.
- [9] J. Slepak, O. Shivers, and P. Manolios, "An array-oriented language with static rank polymorphism," in *European Symposium on Programming (ESOP)*, 2014.
- [10] J. Slepak, P. Manolios, and O. Shivers, "Rank polymorphism viewed as a constraint problem," in *International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI*, 2018.
- [11] G. S. Makanin, "The problem of solvability of equations in a free semigroup," *Mathematics of the USSR-Sbornik*, 1977.
- [12] W. Plandowski, "Satisfiability of word equations with constants is in PSPACE," in *Foundations of Computer Science (FOCS)*, 1999, pp. 495–500.
- [13] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," in *Formal Methods in Computer Aided Design, FMCAD*, 2017.
- [14] T. Liang, A. Reynolds, C. Tinelli, C. W. Barrett, and M. Deters, "A DPLL(T) theory solver for a theory of strings and regular expressions," in *Computer Aided Verification (CAV)*, 2014.
- [15] A. Reynolds, M. Woo, C. Barrett, D. Brumley, T. Liang, and C. Tinelli, "Scaling up DPLL(T) string solvers using context-dependent simplification," in *International Conference on Computer Aided Verification (CAV)*, 2017.
- [16] M. Trinh, D. Chu, and J. Jaffar, "Progressive reasoning over recursively-defined strings," in *International Conference on Computer Aided Verification (CAV)*, 2016.
- [17] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "String constraints for verification," in *International Conference on Computer Aided Verification (CAV)*, 2014.
- [18] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "TRAU: SMT solver for string constraints," in *Formal Methods in Computer Aided Design (FMCAD)*, 2018.
- [19] P. Hooimeijer and W. Weimer, "StrSolve: Solving string constraints lazily," in *Automated Software Engineering (ASE)*, 2012.
- [20] S. Eguchi, N. K. B., and T. Tsukada, "Automated synthesis of functional programs with auxiliary functions," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2018.
- [21] P. Manolios and V. Papavasileiou, "ILP modulo theories," in *International Conference on Computer Aided Verification (CAV)*, 2013, pp. 662–677.
- [22] P. Manolios, J. Pais, and V. Papavasileiou, "The Inez mathematical programming modulo theories framework," in *International Conference on Computer Aided Verification (CAV)*, 2015.
- [23] W. Plandowski, "Satisfiability of word equations with constants is in NEXPTIME," in *Symposium on Theory of Computing (STOC)*, 1999.
- [24] A. Kumar, "SeqSolve string solver with benchmarks," <https://github.com/ankitku/SeqSolve>.
- [25] H. Chamarithi, P. C. Dillinger, P. Manolios, and D. Vroon, "The ACL2 sedan theorem proving system," in *TACAS*, 2011.
- [26] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang, "Z3str2: an efficient solver for strings, regular expressions, and length constraints," in *Formal Methods in System Design*, 2017.
- [27] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "Kaluza benchmark," <http://webblaze.cs.berkeley.edu/2010/kaluza/>.
- [28] D. Blotsky, "StringFuzz-generated benchmark," <http://stringfuzz.dmitryblotsky.com/suites/generated/>.
- [29] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Symposium on Security and Privacy, S&P*, 2010.
- [30] A. Jez, "Recompression: A simple and powerful technique for word equations," *Journal of the ACM (JACM)*, 2016.
- [31] W. Plandowski and W. Rytter, "Application of Lempel-Ziv Encodings to the Solution of Words Equations," in *International Colloquium on Automata, Languages and Programming (ICALP)*, 1998.
- [32] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, "Stringfuzz: A fuzzer for string solvers," in *International Conference on Computer Aided Verification (CAV)*, 2018.
- [33] L. M. de Moura, B. Dutertre, and N. Shankar, "A tutorial on satisfiability modulo theories," in *International Conference on Computer Aided Verification (CAV)*, 2007.
- [34] H. Abdulrab, "Solving word equations," in *Informatique Théorique et Applications (ITA)*, 1990.
- [35] S. Subramanian, M. Berzish, O. Tripp, and V. Ganesh, "A solver for a theory of strings and bit-vectors," in *International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017.
- [36] N. Bjørner, "All strings attached: String and sequence constraints in Z3," in *Rewriting Logic and Its Applications*, 2016.
- [37] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore, "ACL2s: The ACL2 sedan," in *International Conference on Software Engineering (ICSE)*, 2007.
- [38] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *International Conference on Computer Aided Verification (CAV)*, 2011.
- [39] M. T. Trinh, D. H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *Computer and Communications Security (CCS)*, 2014.
- [40] A. Lin and P. Barceló, "String solving with word equations and transducers: Towards a logic for analysing mutation XSS," in *Principles of Programming Languages (POPL)*, 2016.
- [41] C. Gutiérrez, "Solving equations in strings: On makanin s algorithm," in *Theoretical Informatics, Third Latin American Symposium (LATIN)*, 1998.
- [42] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang, "Effective search-space pruning for solvers of string equations, regular expressions and length constraints," in *International Conference on Computer Aided Verification (CAV)*, 2015.
- [43] J. D. Day, F. Manea, and D. Nowotka, "The hardness of solving simple word equations," in *Leibniz International Proceedings in Informatics (LIPIcs)*, 2017.
- [44] P. Aziz Abdulla, M. Faouzi Atig, Y.-F. Chen, B. Phi Diep, L. Holik, A. Rezine, and P. Rümmer, "Flatten and conquer: A framework for efficient analysis of string constraints," in *Programming Language Design and Implementation (PLDI)*, 2017.
- [45] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.
- [46] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th International Conference on World Wide Web*, 2005.
- [47] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *IEEE Symposium on Security and Privacy*, 2010.