Formal Methods in Computer-Aided Design 2020

Reductions for Strings and Regular Expressions Revisited

Andrew Reynolds*, Andres Nötzli†, Clark Barrett†, and Cesare Tinelli*

*The University of Iowa, †Stanford University

***Abstract*—The theory of strings supported by solvers in formal methods contains a large number of operators. Instead of implementing a semi-decision procedure that reasons about all the operators directly, string solvers often reduce operators to a core fragment and implement a semi-decision procedure over that fragment. These reductions considerably increase the number of constraints and thus have to be done carefully to achieve good performance. We propose novel reductions from regular expressions to string constraints and a framework for minimizing the introduction of new variables in current reductions of string constraints. The reductions of regular expression constraints enable string solvers to handle a significant fragment of such constraints without using dedicated reasoning over regular expressions. Minimizing the number of variables in the reduced constraints makes those constraints significantly cheaper to solve by the core solver. An experimental evaluation of our implementation of both techniques in CVC4, a state-of-the-art SMT solver with extensive support for the theory of strings, shows that they significantly improve the solver's performance.**

I. Introduction

Most software processes strings in some fashion, and as a result, modern programming languages include functionality to manipulate strings in various ways. The semantics of these string manipulations are often complex, which makes automated reasoning about programs that use them challenging. In recent years, researchers have proposed various approaches to tackle this challenge with dedicated solvers for string constraints [18], [20], [5], [11], [4], [3]. Dedicated solvers have been successfully used in a wide range of applications such as finding or proving the absence of SQL injections and XSS vulnerabilities in web applications [25], [23], [30], reasoning about access policies in cloud infrastructure [7], [6], and generating database tables from SQL queries for unit testing [28].

Modern string solvers natively support an extensive set of high-level string operations commonly found in programming languages, such as regular language membership, string replacement, and computing the index of one string in another. Reasoning about string constraints can be roughly divided into three areas: (i) reasoning about basic word equations with length constraints, (ii) reasoning about extended string constraints, and (iii) reasoning about regular membership constraints. One common approach to handling extended string constraints is to reduce the high-level operators to a set of basic operators and implement a semi-decision procedure for the latter. In such a design, the overall performance of a string solver depends on the efficiency of those reductions. In particular, these reductions tend to introduce fresh string variables, which affect the difficulty of the problem for the solver for basic constraints.

The expressive power of the signature for string constraints often enables the user to write the same constraints in multiple equivalent ways. As a simple example, consider the following three formulas, each stating in effect that string y is the result of removing the first character from another string x:

$$\exists z.\, x \approx z \cdot y \wedge |z| \approx 1 \tag{1}$$

$$\mathsf{substr}(x, 1, |x| - 1) \approx y \tag{2}$$

$$x \in \mathsf{rcon}(\Sigma, \mathsf{to\_re}(y)) \tag{3}$$

Equation (1) states that there exists some string z of length one such that x is the result of concatenating that string and y. Equation (2) uses the extended string function substr to state that y is the substring of x starting at position one and having length $|x| - 1$. Equation (3) states that x is in the regular language consisting of the set of strings obtained by concatenating (rcon) the regular language of single character strings (Σ) with the (singleton) regular language containing just y. In this work, we observe that many string constraints like those above share common properties and can be handled based on reductions that lead to a more effective collaboration between the various subsolvers in current string solvers.

The contributions of this paper are as follows:

- We introduce *witness sharing*, a novel technique that can significantly reduce the number of variables introduced by string solvers that reason about combinations of word equations, extended string constraints, and regular expressions.
- We verify the correctness of our technique by generating verification conditions that encode some of its soundness properties and solve them using multiple string solvers.
- We describe new techniques for encoding regular expressions using extended functions whose reductions take advantage of witness sharing.
- We implement these techniques in the state-of-the-art string subsolver of the SMT solver CVC4 [9], showing that they lead to significant performance improvements.

In the remainder of this section, we discuss related work. We discuss preliminaries in Section II, introduce the concept of witness sharing in Section III, and discuss the reduction of regular expression constraints to extended string functions in Section IV. Finally, we evaluate our approach in Section V.

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_30

$$\begin{array}{llll}
n : \mathsf{Int} \text{ for all } n \in \mathbb{N} & & l : & \mathsf{Str} \text{ for all } l \in \mathcal{A}^* \\
+ : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int} & - : \mathsf{Int} \to \mathsf{Int} & \geqslant : & \mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool} \\
\_ \cdot \ldots \cdot \_ : \mathsf{Str} \times \cdots \times \mathsf{Str} \to \mathsf{Str} & & |\_| : & \mathsf{Str} \to \mathsf{Int}
\end{array}$$

$$\begin{array}{llll}
\mathsf{substr} : & \mathsf{Str} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{Str} & \mathsf{ctn} : & \mathsf{Str} \times \mathsf{Str} \to \mathsf{Bool} \\
\mathsf{indexof} : & \mathsf{Str} \times \mathsf{Str} \times \mathsf{Int} \to \mathsf{Int} & \mathsf{replace} : & \mathsf{Str} \times \mathsf{Str} \times \mathsf{Str} \to \mathsf{Str}
\end{array}$$

$$\begin{array}{llll}
\_ \in \_ : & \mathsf{Str} \times \mathsf{Lan} \to \mathsf{Bool} & \Sigma : & \mathsf{Lan} \\
\mathsf{rcon} : & \mathsf{Lan} \times \cdots \times \mathsf{Lan} \to \mathsf{Lan} & \mathsf{to\_re} : & \mathsf{Str} \to \mathsf{Lan} \\
\mathsf{inter} : & \mathsf{Lan} \times \cdots \times \mathsf{Lan} \to \mathsf{Lan} & \mathsf{star} : & \mathsf{Lan} \to \mathsf{Lan} \\
\mathsf{union} : & \mathsf{Lan} \times \cdots \times \mathsf{Lan} \to \mathsf{Lan} & \mathsf{range}_{c_1,c_2} : & \mathsf{Lan}
\end{array}$$

Fig. 1. Functions in the signature of the theory of strings T_{S}.

Related Work String solvers typically reduce the input constraints to a basic representation. Common basic representations include finite automata [24], [16], [17], [27], [14]; a variation of word equations and length constraints [22], [12], [31], [25]; bit-vectors [18]; and arrays [19]. The reductions to word equations and length constraints are similar to those studied in this work, and our techniques would apply there in a similar manner.

To the best of our knowledge, improving the efficiency of reductions themselves was not a major factor in previous work, although there is work on avoiding unnecessary reductions. Reynolds et al. [21] propose the use of aggressive rewriting to eliminate or simplify extended string constraints before performing reductions. In earlier work, Reynolds et al. [22] describe an approach to perform reductions lazily after simplifying extended functions based on other constraints in the current solving context. The general approach proposed here tackles the cost of reductions from a different angle and can be combined with these approaches.

Backes et al. [7] reduce a fragment of regular expression constraints to extended string constraints. In contrast to our approach, their technique is not integrated within a solver and is restricted to a smaller fragment.

II. Preliminaries

We work in the context of many-sorted first-order logic with equality and assume the reader is familiar with the notions signature, term, literal, (quantified) formula, and free variable. We consider many-sorted signatures Σ that contain an (infix) logical symbol $\approx$ for equality—which has type $\sigma \times \sigma$ for all sorts σ in Σ and is always interpreted as the identity relation. 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 Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. We write $\models_T \varphi$ to denote that the Σ-formula φ is T*-valid*, i.e., is satisfied in every model of T. By convention and unless otherwise stated, we use letters x, y, z to denote variables and s, t to denote terms.

We consider an (extended) theory T_{S} of strings whose signature Σ_{S} is given in Figure 1. We fix a totally ordered finite alphabet $\mathcal{A}$ of characters. The signature includes the sorts Str, Lan, and Int denoting $\mathcal{A}^*$, regular languages over $\mathcal{A}$, and integers, respectively. The *core* signature is given on the first three lines in the figure. It includes the usual symbols of linear integer arithmetic, interpreted as expected. We will write $t_1 \bowtie t_2$, with $\bowtie \in \{>, <, \leqslant\}$, as syntactic sugar for the equivalent inequality between t_1 and t_2 expressed using only $\geqslant$. The core string symbols are given on the first and third line. They consist of a constant symbol, or *string constant*, for each word of $\mathcal{A}^*$ (including ϵ for the empty word), interpreted as that word; a variadic function symbol $\_ \cdot \ldots \cdot \_ : \mathsf{Str} \times \ldots \times \mathsf{Str} \to \mathsf{Str}$, interpreted as word concatenation; and a function symbol $|\_| : \mathsf{Str} \to \mathsf{Int}$, interpreted as the word length function.

The four function symbols in the next two lines of Figure 1 encode operations on strings that often occur in applications. We refer to these function symbols as *extended functions*. Informally, their semantics are as follows. A *position* in a string x is a non-negative integer smaller than the length of x that identifies a character in x—with 0 identifying the first character, 1 the second, and so on. For all x, y, z, n, m, the term $\mathsf{substr}(x, n, m)$ is interpreted as the maximal substring of x starting at position n with length at most m, or the empty string if n is an invalid position or m is negative; the predicate $\mathsf{ctn}(x, y)$ is interpreted as true if and only if x contains y, i.e., if y is a substring of x (every string contains the empty string); $\mathsf{indexof}(x, y, n)$ is interpreted as the position of the first occurrence of y in x starting at position n, or -1 if y is empty, n is an invalid position, or if no such occurrence exists; $\mathsf{replace}(x, y, z)$ is interpreted as the result of replacing the first occurrence in x of y by z, or just x if x does not contain y. We write $\mathsf{substr}(x, n)$ as a shorthand for $\mathsf{substr}(x, n, |x| - n)$.

The signature includes an infix binary predicate symbol $\_ \in \_ : \mathsf{Str} \times \mathsf{Lan} \to \mathsf{Bool}$, which denotes word membership in the given regular language. The remaining symbols are used to construct regular expressions. In particular, Σ denotes (the language of) all strings of length one; $\mathsf{to\_re}(s)$ denotes the singleton language containing just the word denoted by s; $\mathsf{rcon}(R_1, \ldots, R_n)$ denotes all strings that are a concatenation of the strings in the languages denoted by $R_1, \ldots, R_n$; the Kleene star operator $\mathsf{star}(R)$ denotes all strings that are obtained as the concatenation of zero or more repetitions of the strings denoted by R; $\mathsf{inter}(R_1, \ldots, R_n)$ and $\mathsf{union}(R_1, \ldots, R_n)$ denote respectively the intersection and the union of the languages denoted by their arguments; Finally, we include the class of indexed regular expression symbols of the form range_{c_1,c_2} where c_1 and c_2 are strings of length one. We call this a *regular expression range*, which is interpreted as the language containing all strings of length one that are between c_1 and c_2 (inclusive) in the ordering associated with $\mathcal{A}$. We refer to atomic or negated atomic formulas over the signature above as *string constraints*.

III. Witness Sharing for String Solving

In this section, we introduce a technique we call *witness sharing*, which can be used to improve the performance of string solvers that reason in logics that combine: (i) word equations with length constraints; (ii) extended string constraints (with operators like ctn, $\mathsf{replace}$, and so on); and (iii) regular membership constraints. The goal of this technique is to reduce the number of variables introduced internally by SMT solvers when solving various kinds of string constraints. Our key observation is that these variables have common properties, and consequently they can often be shared across multiple inferences, according to a policy that preserves the soundness of the solver. Before describing the technique, it is helpful to review how CDCL(T)-based string solvers operate.

CDCL(T) A CDCL(T)-based solver [10] with support for string constraints works via a cooperation between a propositional SAT solver and a *theory solver*. A theory solver checks the satisfiability of constraints in a background theory T such as arithmetic or strings (the theory solver may consist of multiple cooperating solvers when T is a combination of theories). For a given input formula F, the SAT solver is responsible for determining whether F is propositionally unsatisfiable, that is, unsatisfiable when treating its atomic subformulas as propositional variables. In that case, F is also T-unsatisfiable. Otherwise, the SAT solver generates a propositionally satisfying assignment for the atoms of F in the form of a set of theory literals M. The theory solver then tries to determine if M is consistent with the theory T. If so, F is T-satisfiable; otherwise, the theory solver adds a new (T-valid) formula φ to F, and the above loop repeats.

The formula φ, usually called a *theory lemma*, may correspond to a *conflict clause*, that is, a clause of the form $\ell_1 \vee \ldots \vee \ell_n$, where each literal ℓ_i is forced to be false by M. The addition of a conflict clause causes the SAT solver to choose a new satisfying assignment. Note that not all theory lemmas are conflict clauses. Some are simply T-valid formulas added to F to help the SAT solver refocus its search to assignments that satisfy those lemmas too. The theory solvers for strings we describe next produce this sort of lemmas.

Theory Solvers for String Constraints In this section, we focus on the behavior of the theory solver for strings in a CDCL(T) loop. Such solvers are often designed with subsolvers that handle word equations, extended string constraints, and regular expressions over the signature for T_S provided in Figure 1, or some variant of it. Their design and implementation have been thoroughly described in previous work [20], [5], [26]. For the purposes of this paper, it suffices to view a theory solver for strings as a method that takes as input a set M_S of string constraints, which we also refer to as the *context*, and either (a) returns (a set of) theory lemmas φ to be added to the set of constraints F maintained by the SAT solver, or (b) returns sat, indicating that M_S is T_S-satisfiable.

We can view a string solver abstractly as a set $\mathcal{S}$ of *inference schemas*. An inference schema is a mapping from T_S-literals ℓ (called its *premise*) to a list of the form $(C_1 \Rightarrow \varphi_1), \ldots, (C_n \Rightarrow \varphi_n)$ where $C_1, \ldots, C_n$ and $\varphi_1, \ldots, \varphi_n$ are formulas. We assume without loss of generality that all models of T_S satisfy exactly one of $C_1, \ldots, C_n$. Intuitively, an inference schema specifies that a list of conclusions $\varphi_1, \ldots, \varphi_n$ are implied by literal ℓ under the conditions $C_1, \ldots, C_n$ respectively. An abstract procedure for a theory solver for strings can be summarized by the following definition.

Definition 1 (Theory Solver for Strings)**.** *A theory solver for T_S based on an inference schema set $\mathcal{S}$ takes as input a set of T_S-literals M_S and adds formulas to an initially empty set F as follows. For each inference schema of the form $\ell \mapsto (C_1 \Rightarrow \varphi_1, \ldots, C_n \Rightarrow \varphi_n)$ and literal $\ell\sigma \in M_\mathsf{S}$, where σ is a substitution mapping the variables of ℓ to ground terms:*

1) *if $M_\mathsf{S} \models C_i\sigma$ for some i, then add $((\ell \wedge C_i) \Rightarrow \varphi_i)\sigma$ to F unless this lemma is already in F;*
2) *otherwise, add $(C_1 \vee \ldots \vee C_n)\sigma$ to F.*

If no formulas were added to F, return sat.

In other words, for each inference schema for which there exists a ground T_S-literal $\ell\sigma$ contained in (or, more generally, entailed by) the current context M_S that matches the premise ℓ, if any condition C_i is implied by the current assertions, we add a theory lemma stating that the conclusion φ_i must hold when the premise and its condition hold (under substitution σ). The theory lemma is added to the set of formulas F known by the SAT solver if it does not already occur in F. If none of the conditions $C_1, \ldots, C_n$ are implied, the solver adds the *splitting lemma* $(C_1 \vee \ldots \vee C_n)\sigma$, which will force the SAT solver to pick a condition to satisfy, which in turn will force the theory solver to derive one of the conclusions $\varphi_1\sigma, \ldots, \varphi_n\sigma$. A theory solver for strings is *refutation-sound* if it adds only T_S-valid formulas to F. It is *model-sound* if it returns sat only when M_S is T_S-satisfiable. We do not provide the details on the strategies used by a theory solver for strings in this paper and instead refer the reader to previous work [20], [5], [26].

It is important to note that, in contrast to traditional theory solvers, many state-of-the-art theory solvers for strings generate lemmas that do not necessarily correspond to conflict clauses. In fact, the generated lemmas may contain new literals or even literals with new (string) variables. A common example is the lemma for handling equality between two string concatenations.

Example 1. *Consider the T_S-literal ℓ of the form $x \cdot x' \approx y \cdot y'$, where x, y, x', y' are variables. A possible inference schema maps ℓ to:*

$$((|x| \approx |y| \Rightarrow x \approx y), (|x| > |y| \Rightarrow \exists k_1.\, x \approx y \cdot k_1),$$
$$(|x| < |y| \Rightarrow \exists k_2.\, x \cdot k_2 \approx y))$$

When $x \cdot x' \approx y \cdot y'$ holds, if x and y have the same length then they must be equal. If x is longer than y then y is a prefix of x, a fact expressed by the formula $\exists k_1.\, x \approx y \cdot k_1$, stating that x is the concatenation of y with some other string k_1. The case for when y is longer than x is analogous.

Notice that conclusions in the inference schema described above contain existentially quantified variables. In practice, existential quantifiers are eliminated eagerly by *Skolemization*, i.e., by instantiating them by fresh variables before the theory lemma is added to the set F. Thus, in the above example, a theory solver for strings may return $(x \cdot x' \approx y \cdot y' \wedge |x| > |y|) \Rightarrow x \approx y \cdot v_1$ where v_1 is a fresh variable. Later in this section, we argue that variables introduced in lemmas such as this one can be shared amongst multiple theory lemmas based on a careful analysis of the inference schemas.

	Premise	Conclusion	Condition	Witness Terms
(V-Split)	$x \cdot x' \approx y \cdot y'$	$x \approx y \wedge x' \approx y'$	$\lvert x\rvert \approx \lvert y\rvert$	
		$\exists k_1. x \approx y \cdot k_1 \wedge k_1 \cdot x' \approx y'$	$\lvert x\rvert > \lvert y\rvert$	$k_1 \mapsto \mathsf{suf}(x, \lvert y\rvert)$
		$\exists k_2. y \approx x \cdot k_2 \wedge x' \approx k_2 \cdot y'$	$\lvert x\rvert < \lvert y\rvert$	$k_2 \mapsto \mathsf{suf}(y, \lvert x\rvert)$
(C-Split)	$x \cdot x' \approx c \cdot y'$	$x \approx c \wedge x' \approx y'$	$\lvert x\rvert \approx 1$	
		$\exists k_1. x \approx c \cdot k_1 \wedge k_1 \cdot x' \approx y'$	$\lvert x\rvert > 1$	$k_1 \mapsto \mathsf{suf}(x, 1)$
		$x' \approx c \cdot y'$	$\lvert x\rvert \approx 0$	
(Deq-V-Split)	$x \cdot x' \not\approx y \cdot y'$	$x \not\approx y \vee x' \not\approx y'$	$\lvert x\rvert \approx \lvert y\rvert$	
		$\exists k_1 k_2. x \approx k_1 \cdot k_2 \wedge \lvert k_1\rvert \approx \lvert y\rvert$	$\lvert x\rvert > \lvert y\rvert$	$k_1 \mapsto \mathsf{pre}(x, \lvert y\rvert)$
				$k_2 \mapsto \mathsf{suf}(x, \lvert y\rvert)$
		$\exists k_3 k_4. y \approx k_3 \cdot k_4 \wedge \lvert k_3\rvert \approx \lvert x\rvert$	$\lvert x\rvert < \lvert y\rvert$	$k_3 \mapsto \mathsf{pre}(y, \lvert x\rvert)$
				$k_4 \mapsto \mathsf{suf}(y, \lvert x\rvert)$
(Deq-C-Split)	$x \cdot x' \not\approx c \cdot y'$	$x \not\approx c \vee x' \not\approx y'$	$\lvert x\rvert \approx 1$	
		$\exists k_1 k_2. x \approx k_1 \cdot k_2 \wedge \lvert k_1\rvert \approx 1$	$\lvert x\rvert > 1$	$k_1 \mapsto \mathsf{pre}(x, 1)$
		$x' \not\approx c \cdot y'$	$\lvert x\rvert \approx 0$	$k_2 \mapsto \mathsf{suf}(x, 1)$

Fig. 2. Inference schemas that introduce existential variables in string solvers for word equations.

Inference Schemas for String Solvers To give further context for how theory solvers for strings operate, we describe a representative list of inference schemas that introduce new variables in theory lemmas in a typical state-of-the-art string solver. Figures 2 to 4 list commonly applied inferences in the core equation solver (Figure 2), the solver for extended string functions (Figure 3), and the solver for regular expression memberships (Figure 4). In these figures, the first column gives the premise of the inference, and the second column gives (possibly multiple) conclusions that can be derived from that premise, given the conditions in the third column. We will address the fourth column in later parts of this section.

In Figure 2, the first inference schema V-Split is used when we have inferred an equality between two string terms of the form $x \cdot x'$ and $y \cdot y'$. Given this constraint, the string solver may be also able to infer whether x is equal to y, y is a prefix of x or vice versa, as discussed in Example 1. Based on these three cases, a (set of) equalities can be inferred possibly involving a new existentially quantified variable k_1 or k_2. The inference schema C-Split is similar to V-Split and handles the case where one side of an equality begins with a character constant c. There are two analogous schemas for string disequalities. The schema Deq-V-Split handles disequalities where both sides of the disequality begin with a variable (x and y). As in the equality case, the conditions split on the subcases where the length of x is equal, greater, or less than that of y. If they have equal length, the disequality is satisfied if and only if x and y differ or their remainders differ. If x is longer than y, then x can be decomposed into two parts k_1 and k_2 where k_1 has the same length as y. The case when y is longer than x is analogous. Schema Deq-C-Split is similar and handles the case where one side of the disequality begins with a constant. These four schemas do case-splitting based on the *first* argument of concatenation terms; although not shown here, four analogous inference schemas are used for splitting based on the *last* argument of concatenation terms. In practice, when splitting a string in the schemas for disequalities, there is no need to include the literal ℓ in the lemma since it is valid without ℓ.

The inference schemas in Figure 3 cover the support for reducing the extended string functions ctn, substr, $\mathsf{replace}$, and $\mathsf{indexof}$ respectively. To simplify the exposition, we assume with no loss of generality that for every extended string term t in the input set M_S of constraints, M_S contains an equality of the form $t \approx x$ for some variable x, which we call the *purification variable* for term t. The schema R-Ctn states that if x contains y then it must be equal to the concatenation term $k_1 \cdot y \cdot k_2$ for some (possibly empty) k_1 and k_2. The schema R-Substr relates the purification variable y for a substring term $\mathsf{substr}(x, n, m)$ with its arguments. Namely, the first conclusion holds when n is a valid position and m is positive, as expressed by its condition. It states that x must be of the form $k_1 \cdot y \cdot k_2$, where k_1 must have length n (to ensure y is a substring of x starting at position n). The remainder of the conclusion ensures that the length of y matches the semantics of substr. The length of the remainder string k_2 must equal either the length of the remaining portion of x after position $n + m$, or 0 (in the case that $n + m \geqslant |x|$). Moreover, unless y equals the empty string, it must have length at most m.[1] The schema R-Replace applies to premise $\mathsf{replace}(x, y, z) \approx w$ and introduces a conclusion with existential variables when x contains a non-empty string y. In that case, the first occurrence of y in x is immediately preceded by some prefix k_1 of x. This is expressed by the constraint $x \approx k_1 \cdot y \cdot k_2 \wedge \neg\mathsf{ctn}(k_1 \cdot \mathsf{pre}(y, |y| - 1), y)$, where $\mathsf{pre}(y, |y| - 1)$ is shorthand for $\mathsf{substr}(y, 0, |y| - 1)$, which denotes the result of removing the last character from y. If y is empty, the result of $\mathsf{replace}$ is to prepend z to x. If x does not contain y at all, the result of $\mathsf{replace}$ is the original string x. The schema R-Indexof introduces one conclusion with

[1]Note that the form of the conclusion here is slightly different from that found in analagous rules provided in previous work [22].

	Premise	Conclusion	Condition	Witness Terms
(R-Ctn)	$\mathsf{ctn}(x, y)$	$\exists k_1 k_2.\, x \approx k_1 \cdot y \cdot k_2$	$\top$	$k_1 \mapsto \mathsf{pre}_C(x, y)$ $k_2 \mapsto \mathsf{suf}_C(x, y)$
(R-Substr)	$\mathsf{substr}(x, n, m) \approx y$	$\exists k_1 k_2. x \approx k_1 \cdot y \cdot k_2 \wedge \lvert k_1 \rvert \approx n \wedge \lvert y \rvert \leqslant m \wedge (\lvert k_2 \rvert \approx \lvert x \rvert - (n + m) \vee \lvert k_2 \rvert \approx 0)$	$0 \leqslant n < \lvert x \rvert \wedge m > 0$	$k_1 \mapsto \mathsf{pre}(x, n)$ $k_2 \mapsto \mathsf{suf}(x, n + m)$
		$y \approx \epsilon$	otherwise	
(R-Replace)	$\mathsf{replace}(x, y, z) \approx w$	$\exists k_1 k_2.\, w \approx k_1 \cdot z \cdot k_2 \wedge x \approx k_1 \cdot y \cdot k_2 \wedge \neg\mathsf{ctn}(k_1 \cdot \mathsf{pre}(y, \lvert y \rvert - 1), y)$	$\mathsf{ctn}(x, y) \wedge y \not\approx \epsilon$	$k_1 \mapsto \mathsf{pre}_C(x, y)$ $k_2 \mapsto \mathsf{suf}_C(x, y)$
		$w \approx z \cdot x$	$y \approx \epsilon$	
		$w \approx x$	$\neg\mathsf{ctn}(x, y)$	
(R-Indexof)	$\mathsf{indexof}(x, y, n) \approx m$	$\exists k_1 k_2.\, \neg\mathsf{ctn}(k_1 \cdot \mathsf{pre}(y, \lvert y \rvert - 1), y) \wedge m \approx n + \lvert k_1 \rvert \wedge \mathsf{suf}(x, n) \approx k_1 \cdot y \cdot k_2$	$0 \leqslant n \leqslant \lvert x \rvert \wedge y \not\approx \epsilon \wedge \mathsf{ctn}(\mathsf{suf}(x, n), y)$	$k_1 \mapsto \mathsf{pre}_C(\mathsf{suf}(x, n), y)$ $k_2 \mapsto \mathsf{suf}_C(\mathsf{suf}(x, n), y)$
		$m \approx n$	$0 \leqslant n \leqslant \lvert x \rvert \wedge y \approx \epsilon$	
		$m \approx -1$	otherwise	

Fig. 3. Inference schemas that introduce existential extended functions.

existential variables for premise $\mathsf{indexof}(x, y, n) \approx m$ when n is a valid position in x and the substring of x after position n (written $\mathsf{suf}(x, n)$) contains a non-empty string y. In this case, the variable k_1 is introduced as the prefix of $\mathsf{suf}(x, n)$ before the first occurrence of y in $\mathsf{suf}(x, n)$. If y is empty and n is a valid position in x, the result is n. If n is an invalid position, the result is -1.

The inference schemas in Figure 4 introduce existential variables when reasoning about regular expressions. U-RCon is applied to reduce (positively asserted) membership constraints in a language expressed as the concatenation of two regular expressions R_1 and R_2. In this case, x must consist of two strings k_1 and k_2 that occur in R_1 and R_2, respectively. Finally, the rule for Kleene star U-RStar is similar to the rule U-RCon: if x occurs in R or is empty, then $x \in R^*$ holds trivially (so the conclusion is just $\top$). Otherwise x must be decomposable into three pieces k_1, k_2 and k_3, where k_1 and k_3 occur in R, and k_2 occurs in R^*.

Example 2. *Using double quotes to denote string constants, let M_S be $\{x \approx \text{“}a\text{”} \cdot y, x \in \mathsf{rcon}(\Sigma, R), y \notin R, |x| > 1\}$. We may apply U-RCon to literal $x \in \mathsf{rcon}(\Sigma, R)$, which matches the premise of that schema, to obtain its conclusion:*

$$\exists k_1 k_2.\, (x \approx k_1 \cdot k_2 \wedge k_1 \in \Sigma \wedge k_2 \in R) \tag{4}$$

Similarly we may C-Split[2] for literal $x \approx \text{“}a\text{”} \cdot y$ to obtain:

$$\exists k_3.\, x \approx \text{“}a\text{”} \cdot k_3 \wedge k_3 \approx y \tag{5}$$

After passing theory lemmas with these conclusions to the SAT solver, where existential variables k_1, k_2, k_3 are Skolemized respectively with fresh variables v_1, v_2, v_3, the string solver will be invoked again with a context extended with the set $\{x \approx v_1 \cdot v_2, v_1 \in \Sigma, v_2 \in R, x \approx \text{“}a\text{”} \cdot v_3, v_3 \approx y\}$.

[2]We assume matching is modulo empty strings in concatenation terms, so that string t matches $x \cdot x'$ under the substitution $\{x \mapsto t, x' \mapsto \epsilon\}$.

In the above example, observe that both v_2 and v_3 represent the result of removing the first character from x. Thus, it is sound to use the same Skolem variable to witness both k_2 and k_3. This can easily be inferred based on a policy that we describe in the following, which will make it easier for the string solver to conclude that sets of assertions like the one above are unsatisfiable.

A. Witness Sharing by Smart Quantifier Elimination

In total, there are 22 places where the string solver in CVC4 introduces existentially quantified variables in its inference schemas (9 for word equations, 8 for extended string functions, 5 for regular expressions). A naive approach for Skolemizing those variables would replace each of them by a fresh Skolem variable for each derived conclusion. However, in the following, we argue that the witnesses for existential quantified formulas in these rules can be *shared* across multiple formulas. A majority of the 22 kinds of variables fall into one of four categories: (i) the prefix of a string s up to some fixed position n; (ii) the suffix of a string s after some fixed position n; (iii) the prefix of a string s up to the position of a substring t; and (iv) the suffix of a string s after the position of a substring t.

One way to view it is that the quantified formulas introduced by the various inference schemas admit quantifier elimination in the extended string signature. For example, in the second conclusion of schema V-Split, the formula

$$\exists k_1 . x \approx y \cdot k_1 \wedge k_1 \cdot x' \approx y'$$

is equivalent to

$$x \approx y \cdot \mathsf{substr}(x, |y|) \wedge \mathsf{substr}(x, |y|) \cdot x' \approx y' \ ,$$

when the premise and corresponding condition for that schema hold. In principle, we could eliminate those quantifiers instead of Skolemizing them. This would not be efficient, however, because of the cost of processing terms with extended functions such as $\mathsf{substr}(x, |y|)$. Instead, we observe that each existential variable in a inference schema conclusion has a *witness term*,

Premise	Conclusion	Condition	Witness Terms
(U-RCon) $x \in \mathsf{rcon}(R_1, R_2)$	$\exists k_1 k_2.\, x \approx k_1 \cdot k_2 \wedge k_1 \in R_1 \wedge k_2 \in R_2$	$\top$	$k_1 \mapsto \mathsf{pre}(x, \|R_1\|)$ $k_2 \mapsto \mathsf{suf}(x, \|R_1\|)$
(U-RStar) $x \in R^*$	$\exists k_1 k_2 k_3.\, x \approx k_1 \cdot k_2 \cdot k_3 \wedge k_1 \in R \wedge k_2 \in R^* \wedge k_3 \in R$	$x \not\approx \epsilon \wedge x \notin R$	$k_1 \mapsto \mathsf{pre}(x, \|R\|)$ $k_2 \mapsto \mathsf{substr}(x, \|R\|, \lvert x\rvert - 2 * \|R\|)$ $k_3 \mapsto \mathsf{suf}(x, \lvert x\rvert - \|R\|)$
	$\top$	otherwise	

Fig. 4. Inference schemas that introduce existential variables in string solvers for regular expressions.

i.e., can be equivalently replaced by a term over the extended string signature, as is the case for k_1 above.

Based on this observation, instead of eliminating existential variables by instantiating them with their witness term t, we instantiate them with a *witness variable*, a Skolem variable that is associated with t. We do that by constructing and maintaining a mapping from witness terms to Skolem variables with the goal of mapping pairs of witness terms to the *same* Skolem variable whenever we recognize (inexpensively, as described in Section III-B) that the two witness terms are equivalent. This way, we can *recycle* Skolem variables introduced earlier, and keep their number low, without loss of generality.

Witness Terms For variables that represent the prefix (resp., suffix) of string x before (resp., after) a given position n, the corresponding witness term can be expressed using the substring operator, namely with terms of the form $\mathsf{substr}(s, 0, n)$ and $\mathsf{substr}(s, n)$. For convenience, we write $\mathsf{pre}(s, n)$ and $\mathsf{suf}(s, n)$ as shorthand for these terms. Furthermore, we write $\mathsf{pre}_C(s, t)$ to abbreviate $\mathsf{pre}(s, \mathsf{indexof}(s, t, 0))$ which denotes the term equivalent to the prefix of s before the first occurrence of t in s when one exists. We additionally write $\mathsf{suf}_C(s, t)$ to denote the suffix of s after the first occurrence of t in s if one exists, which abbreviates $\mathsf{suf}(s, |\mathsf{pre}_C(s, t)| + |t|)$.

The last column in Figures 2 to 4 lists the witness terms for each inference schema. The justifications for most witness terms are straightforward. R-Ctn, R-Replace, and R-Indexof use pre_C and suf_C because they involve reasoning about the occurrence of one string in another. Witness terms for the regular expression schema U-RCon can be constructed for regular expressions R for which there exists a term of integer type, which we denote by $\|R\|$ here, such that all strings that belong to R have length $\|R\|$. For example, $\|\mathsf{to\_re}(x)\| = |x|$. We call $\|R\|$ the *regular expression length* of R. We use a simple (incomplete) recursive method, summarized in Figure 5, to infer $\|R\|$ for a regular expression R when possible. For U-RCon, which applies to the premise $x \in \mathsf{rcon}(R_1, R_2)$, multiple choices for witness terms may exist. If a regular expression length can be computed for R_1, then we know that k_1 and k_2 can be given witness terms $\mathsf{pre}(x, \|R_1\|)$ and $\mathsf{suf}(x, \|R_1\|)$ respectively. Although not shown in the figure, witness terms $\mathsf{pre}(x, |x| - \|R_2\|)$ and $\mathsf{suf}(x, |x| - \|R_2\|)$ can be given when $\|R_2\|$ can be inferred. For U-RStar, we assume witness terms are used only when $\|R\|$ can be inferred. For this rule, k_1 is the prefix of x whose length is $\|R\|$, k_3 is the suffix of x whose length is $\|R\|$, and k_2 is remaining string after removing these two substrings.

$$
\begin{array}{rcl}
\|\Sigma\| & = & 1 \\
\|\mathsf{range}(c_1, c_2)\| & = & 1 \\
\|\mathsf{to\_re}(s)\| & = & |s| \\
\|\mathsf{union}(R_1, \cdots, R_k)\| & = & u, \text{if } \forall i.\, \|R_i\| = u \\
\|\mathsf{inter}(R_1, \cdots, R_k)\| & = & u, \text{if } \exists i.\, \|R_i\| = u \\
\|\mathsf{rcon}(R_1, \cdots, R_k)\| & = & \|R_1\| + \cdots + \|R_k\|
\end{array}
$$

Fig. 5. Definition of $\|R\|$ for cases in which a regular expression R only accepts strings of a fixed length.

Example 3. *We revisit the inference schemas applied for Example 2. In that example, we applied U-RCon to $x \in \mathsf{rcon}(\Sigma, R)$ to obtain the conclusion given by (4) over existentially quantified variables k_1 and k_2. According to Figure 4, since $\|\Sigma\| = 1$, the witness terms for k_1 and k_2 are $\mathsf{pre}(x, 1)$ and $\mathsf{suf}(x, 1)$ respectively. Similarly, we applied C-Split to the equality $x \approx$ "a" $\cdot\, y$ to obtain the conclusion given by (5) over the existentially quantified variable k_3. According to Figure 2, the witness term for k_3 is $\mathsf{suf}(x, 1)$. Since k_2 and k_3 have the same witness term, they can be witnessed by the same variable $v_{\mathsf{suf}(x,1)}$. Using this (shared) variable results in a context where the string solver is given as input the set of assertions $\{v_{\mathsf{suf}(x,1)} \in R, v_{\mathsf{suf}(x,1)} \approx y, y \notin R\}$, which can be easily shown to be unsatisfiable: the first two constraints imply that $y \in R$ which contradicts the third constraint.*

In the above example, the string solver was able to derive a contradiction in the state resulting from the application of two inference schemas. This was made possible by witnessing existential variables for two inference schemas with the same variable $v_{\mathsf{suf}(x,1)}$. A solver without witness sharing requires further case splitting before finding a similar contradiction. In practice, the use of witness sharing to minimize the number of witness variables leads to significant performance improvements, as we show in Section V.

B. Implementation Details

We list some of the important optimizations and implementation details for witness sharing in the following.

Witness Sharing based on Term Rewriting Two existential variables can be witnessed by the same variable when their witness terms s and t are equivalent. String solvers implement aggressive rewriting techniques on string terms (see, e.g., [21]), which we can leverage to perform fast but incomplete checks of the validity of the constraint $s \approx t$. We write $s{\downarrow}$ to denote the *rewritten form* of term s, which in practice is computed by a component of the SMT solver we call the *rewriter*. A rewriter is designed to be sound, that is, $s{\downarrow} = t{\downarrow}$ implies $s \approx t$. It is, however, typically incomplete for performance reasons, which means that two equivalent terms may have different rewritten forms. We apply the rewriter to witness terms before mapping them to witness variables to obtain improved sharing of witness variables.

Relaxing the Witness for the First Occurrence It is important to note that the witness variable v_t corresponding to witness term m is not necessarily constrained to be equal to t in the solver, which allows models where they indeed differ. This is not a problem because the value of a witness variable in any model is guaranteed to be a witness for the corresponding existentially quantified variable. We can use this fact to avoid introducing additional constraints on witness variables. Recall that term $\mathsf{pre}_C(x, y)$ is the prefix of x before the *first* occurrence of y in x if there is one. Constraints for witness variables are derived from the conclusions of rules. Indeed, R-Replace from Figure 3 introduces the constraint $\neg\mathsf{ctn}(v_{\mathsf{pre}_C(x,y)} \cdot \mathsf{substr}(y, 0, |y|-1), y)$ to insist that $v_{\mathsf{pre}_C(x,y)}$ be the prefix of x before the first occurrence. It is, however, not necessary to add the same constraint in the conclusion of R-Ctn. Instead, it is sufficient to insist that $v_{\mathsf{pre}_C(x,y)}$ be the prefix of x before *any* such occurrence. Applying the latter schema in isolation may permit models where $v_{\mathsf{pre}_C(x,y)}$ corresponds to a prefix of x prior to an occurrence of y in x other than the first one. Nevertheless, the inference schema R-Ctn may use $\mathsf{pre}_C(x, y)$ as a witness term because $v_{\mathsf{pre}_C(x,y)}$ can be assumed (when necessary, and without loss of generality) to be the prefix before the first occurrence. Avoiding additional constraints is important in practice because negative containment constraints like the one above are notoriously expensive to reason about. This can be seen as constraining the witness variables lazily.

Equivalence of Witness Variables and Substring Terms If we have a constraint of the form $y \approx t$ in the context where y is a variable and t is a witness term t, we can use y as the witness variable for t instead of introducing a fresh variable v_t. This insight is particularly useful for applications of substring. Recall that we assume that we purify extended string terms, so applications of substring only appear in assertions of the form $\mathsf{substr}(x, n, m) \approx y$ where y is the purification variable. As a result, we can use y as the witness variable if we have a witness term of the form $\mathsf{substr}(x, n, m)$. This means that witness variables are entailed to be equal to existing substring terms that occur in M_S whenever applicable.

Propagation Based on Adjacent Literals While not shown in Figure 2, a solver for word equations can be optimized by inferring when a string must contain a constant prefix. This can be inferred for equalities where one side has the form $x \cdot l_1 \cdot x'$, and the other side begins with a constant that cannot overlap with l_1. We demonstrate this in the following example.

Example 4. *Let ℓ be the literal $x \cdot$ "b" $\cdot\, x' \approx$ "aaaa" $\cdot\, y'$. Since x is followed by "b" on the left-hand side of ℓ, it must be the case that x begins with "aaaa" or otherwise "b" would overlap with "aaaa" and the two strings would be disequal. Thus, the conclusion $\exists k_1.\, x \approx$ "aaaa" $\cdot\, k_1$ is implied by ℓ.*

CVC4 implements an inference schema where $\exists k_1.\, x \approx l_1 \cdot k_1$ is derived as a conclusion from the premise $x \cdot l_2 \cdot x' \approx l_1 \cdot l_3 \cdot y'$ under the condition that no non-empty prefix of l_2 is a suffix of l_1, nor is l_2 contained in l_1. While the justification of this conclusion is complex, witness sharing can be applied in a straightforward way. Namely, k_1 in the above conclusion can be mapped to the witness term $\mathsf{suf}(x, |l_1|)$ and shared with variables from other inference schemas as explained earlier.

C. Checking Soundness for Witness Terms

As we have seen, witness sharing derives (implicit) equivalences between witnesses for existential variables. It is critical that the implementation of witness sharing preserve the soundness of the solver. To verify that this is indeed the case, we have constructed a set of 8 benchmarks expressing the correctness of inference schemas that leverage witness sharing. In particular, for each inference schema from Figures 2 and 3 with premise ℓ and conclusion $\exists k_1, \ldots, k_n.\, \varphi$ under condition C_i, we have generated a formula that expresses the entailment:

$$\ell \wedge C_i \models_{T_S} \varphi\{k_1 \mapsto t_1, \ldots, k_n \mapsto t_n\},$$

where $t_1, \ldots, t_n$ are the witness terms for $k_1, \ldots, k_n$. If this entailment does not hold, then there is a case where adding the conclusion with the witness terms to a set of assertions makes them unsatisfiable despite the original set of assertions being satisfiable, that is, the schema makes the solver refutation-unsound. On the other hand, if this entailment holds, then the soundness of the inference schema (using witness sharing) is confirmed. To see why this is the case, notice the entailment check with witness terms is strictly stronger than the same check with witness variables. This is because every model for the variant with witness terms $\varphi\{k_1 \mapsto t_1, \ldots, k_n \mapsto t_n\}$ can be extended to a model for the variant with witness variables $\varphi\{k_1 \mapsto v_{t_1} \ldots, k_n \mapsto v_{t_n}\}$ by interpreting witness variables $v_{t_1}, \ldots, v_{t_n}$ the same way as the corresponding witness terms. This is always possible because the variables themselves are unconstrained. In other words, $\varphi\{k_1 \mapsto t_1, \ldots, k_n \mapsto t_n\}$ entails $\varphi\{k_1 \mapsto v_{t_1}, \ldots, k_n \mapsto v_{t_n}\}$.

We generated one benchmark for each of the inference schemas in Figures 2 and 3. We generated only one benchmark for schemas that have multiple (symmetric) conclusions. We did not consider the verification of the regular expression rules, since neither of the solvers we used for the analysis, CVC4 and Z3 [15], currently support reasoning over regular expression variables. Overall, CVC4 (with witness sharing disabled) and Z3 are capable of showing that the entailment expressed by each of the 8 benchmarks holds, thus corroborating the correctness of our approach.

IV. Regular Expression Elimination

In this section, we discuss an alternate approach to solving regular membership constraints by reducing them to extended string operators. The key insight is that instead of using the

$$
\begin{aligned}
x \in \mathsf{rcon}(R_1, R_2) &\rightarrow \mathsf{pre}(x, \|R_1\|) \in R_1 \wedge \\
&\quad\ \mathsf{suf}(x, \|R_1\|) \in R_2 \\
x \in \mathsf{rcon}(R_1, R_2) &\rightarrow \mathsf{pre}(x, |x| - \|R_2\|) \in R_1 \wedge \\
&\quad\ \mathsf{suf}(x, |x| - \|R_2\|) \in R_2 \\
x \in \mathsf{rcon}(\Sigma^*, \mathsf{to\_re}(y), \Sigma^*, R) &\rightarrow \mathsf{indexof}(x, y, 0) \not\approx -1 \wedge \\
&\quad\ \mathsf{suf}_C(x, y) \in \mathsf{rcon}(\Sigma^*, R) \\
x \in \mathsf{rcon}(R_1, \mathsf{to\_re}(y), R_2) &\rightarrow \\
&\exists i.\ 0 \leqslant i < |x| - |y| \wedge \\
&\quad \mathsf{pre}(x, i) \in R_1 \wedge \\
&\quad \mathsf{substr}(x, i, |y|) \approx y \wedge \mathsf{suf}(x, i + |y|) \in R_2 \\
x \in R^* &\rightarrow \forall k.\ 0 \leqslant k < \mathsf{div}(|x|, \|R\|) \implies \\
&\quad \mathsf{substr}(x, k * \|R\|, \|R\|) \in R
\end{aligned}
$$

Fig. 6. Rules for regular expression elimination

inference schemas from the previous section to generate theory lemmas while solving, we can specialize them and apply them eagerly to eliminate certain types of regular membership constraints. The advantage of this eager elimination is that we do not need to rely on cooperation between the regular membership subsolver and the other subsolvers. The techniques from the previous section can then be applied more readily. The following example demonstrates this point.

Example 5. *Consider the constraint:*

$$x \in \mathsf{rcon}(\Sigma, \Sigma^*, \mathsf{to\_re}(\text{“}abc\text{”}), \Sigma^*)$$

If we applied the rule U-RCon*, we would introduce variables that are matched by the* Σ^* *components. If we look at this constraint through the lens of extended string operators, it is straightforward to show that it is equivalent to* $\mathsf{ctn}(\mathsf{substr}(x, 1), \text{“}abc\text{”})$. *Our techniques for regular expression elimination may eagerly replace the membership constraint above with this extended string constraint, which can be subsequently processed while leveraging our strategy for witness terms described in the previous section.*

To start, all membership constraints with a regular expression whose top symbol is not concatenation or Kleene star can be eliminated eagerly by rewriting. For example, $x \in \mathsf{inter}(\Sigma, \mathsf{union}(R, \mathsf{to\_re}(\text{“abc”})))$ is equivalent to $|x| \approx 1 \wedge (x \in R \vee x \approx \text{“abc”})$. We have extended CVC4 with a set of rules for reducing the other kinds of regular expression memberships (for rcon and Kleene star) to constraints involving extended functions. The most prominent of these rules are given in Figure 6. We give these rules in a form $x \in R \rightarrow \varphi$ where φ is a constraint involving extended string constraints that is equivalent to $x \in R$ and does not contain the top symbol of R.

The first two rules can be applied to constraints of the form $x \in \mathsf{rcon}(R_1, R_2)$ when all strings belonging to R_1 or R_2 are of a fixed length. These rules parallel the use of witness terms for U-RCon when $\|R_1\|$ or $\|R_2\|$ is defined. The next rule applies to the case where the regular expression requires a string y followed by arbitrary characters in some prefix of x. Its conclusion assumes the suffix x after the *first* occurrence of y in x occurs in R. This is with no loss of generality since the regular expression allows us to match an arbitrary number of characters after the position y occurs in x. The final rule for rcon is applicable to a larger set of regular expressions, where it cannot be assumed that the occurrence of $\mathsf{to\_re}(y)$ matches the position where it occurs in x. It says that if the membership constraint requires some string y to appear in x, we can split x in three parts: the prefix before the match on y (which occurs at some position i between 0 and $|x| - |y|$), the match itself, and the suffix after the match. In practice, the rules for regular expression concatenation are ordered with decreasing order of precedence: to reduce a constraint, we apply the first rule among those listed that matches a given membership constraint. For $x \in R^*$, if $\|R\|$ is defined, we can turn such constraints into a (bounded) quantifier that ensures that each substring of x at positions that are multiples of $\|R\|$ and have length $\|R\|$ are in R.

We observe in our evaluation in Section V that regular expression elimination leads to further performance improvements when combined with witness sharing. We attribute this to the fact that replacing regular expression membership constraints with extended string constraints may lead to a reduction in the number of unique constraints that must be processed by the SMT solver for inputs that combine regular expressions and extended functions. In other words, eliminating regular expressions may in some cases enable the solver to detect conflicts at the propositional level or by using high-level theory reasoning even before shared witness variables are introduced, in particular for input constraints that combine regular expression memberships and extended string functions.

V. Evaluation

In this section, we evaluate the impact of witness sharing and regular expression elimination. To this end, we have implemented our approach in CVC4, a state-of-the-art SMT solver with extensive support for the theory of strings.

We evaluate our implementation on three benchmark sets: PyEx, a benchmark set originating from the symbolic execution of Python code [22]; FStrInt, a benchmark set [1] originating from the concolic execution of Python code with Py-Conbyte [29]; and Transf, which consists of industrial benchmarks that were transformed using StringFuzz [13]. From Transf, we omit 438 benchmarks that use regular expression ranges with non-constant bounds and benchmarks that define functions over regular expression arguments. Both of those features are not supported by CVC4.

We compare four configurations of CVC4: **cvc4+wr** uses both regular expression elimination and witness sharing; **cvc4+r** uses just regular expression elimination; **cvc4+w** uses witness sharing only; and **cvc4** does not use the new techniques. As a point of reference, we compare our approach against Z3 4.8.8, another state-of-the-art string solver. We omit a comparison with Z3STR3 4.8.8 and Z3-Trau 1.1 [2] (the new version of Trau) because our experiments have shown that these versions

Set		**cvc4+wr**	**cvc4+r**	**cvc4+w**	**cvc4**	z3	R%
PyEx	sat	**21256**	20117	21254	20116	20214	
	unsat	**3866**	3847	**3866**	3847	3691	10%
	×	299	1457	301	1458	1516	
FStrInt	sat	4403	4410	4404	**4412**	4323	
	unsat	**17095**	17085	**17095**	17089	16834	8%
	×	75	78	74	72	416	
Transf	sat	3690	3688	3670	3663	**3771**	
	unsat	**4796**	4780	4769	4771	4780	7%
	×	259	277	306	311	194	
Total	sat	**29349**	28215	29328	28191	28308	
	unsat	**25757**	25712	25730	25707	25305	
	×	633	1812	681	1841	2126	

TABLE I
NUMBER OF SOLVED PROBLEMS PER BENCHMARK SET. BEST RESULTS ARE IN BOLD. ALL BENCHMARKS RAN WITH A TIMEOUT OF 300 SECONDS.

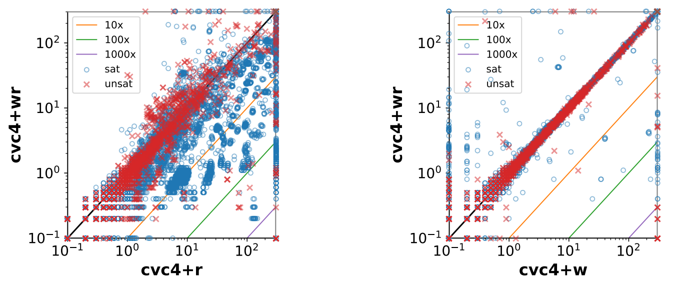

Fig. 7. Scatter plots of runtimes showing the impact of disabling witness sharing and regular expression elimination. All benchmarks ran with a timeout of 300 seconds.

are unsound.[3]

We ran our experiments on a cluster with Intel Xeon CPU E5-2620 v4 CPUs running Ubuntu 16.04 and allocated a physical CPU core, 8 GB of RAM, and 300 seconds for each job.

Table I summarizes our results. It lists the number of satisfiable and unsatisfiable answers as well as timeouts (×) for each configuration and benchmark set. For solved problems, we report the cumulative decrease in fresh variables introduced in the column "R%." To measure this, we instrument the code of **cvc4+wr** to record how many fresh variables were created by the inference schemas discussed in Section III using witness sharing, and compare it to the number of variables that would have been created with witness sharing disabled. Note that this measurement does not take into account compounding effects: Generating fewer variables at an earlier stage may prevent the introduction of fresh variables later in the solving process. Figure 7 shows the impact of disabling witness sharing and regular expression elimination by providing scatter plots that compare the performance of **cvc4+wr** with **cvc4+r** and **cvc4+w**. It differentiates between satisfiable and unsatisfiable instances. Overall, **cvc4** performs better than Z3 and the other configurations only improve on that, which shows that our approach has the potential of improving a solver that is already competitive with the state-of-the-art.

Witness sharing has a major impact on performance, especially for satisfiable instances as the scatter plot in Figure 7 visualizes. Without witness sharing, **cvc4+r** solves significantly fewer satisfiable problems from PyEx and increases the number of timeouts by over four times. The impact is less pronounced on the other benchmark sets, although it makes a noticeable impact on unsatisfiable benchmarks from the Transf set. As expected, the performance impact depends on the structure of the problem. The benchmarks in Transf primarily consist of regular expression membership constraints, so there are fewer opportunities for witness sharing. On the FStrInt benchmarks, **cvc4+wr** does not improve performance over **cvc4+r** despite eliminating a similar amount of variables. Nevertheless, witness sharing cumulatively over these three sets decreases the number of timeouts of CVC4 from 1812 to 633. We believe this indicates the importance of the use of witness sharing for advancing the state of the art in current string solvers.

Although less impactful, comparing **cvc4+wr** and **cvc4+w** indicates that our techniques for regular expression elimination lead to gains in both the overall number of satisfiable and unsatisfiable benchmarks. Regular expression elimination has no impact on the PyEx benchmarks because they lack regular expression membership constraints. Regular expression elimination has the biggest positive impact on the Transf benchmarks, where it decreases the number of unsolved instances from 306 to 259. Notice those benchmarks are generated with a fuzzing tool. Thus, they include regular expressions such as $\mathsf{rcon}([\mathsf{to\_re}(\text{“Q”})]^*, \mathsf{to\_re}(\text{“q”}))^*$ that are less amendable to regular expression elimination than real-world benchmarks. Overall, we believe these results demonstrate the value of exploring alternate encodings of regular expressions in combination with extended string function constraints.

VI. Conclusion

We have presented an approach for CDCL(T) theory solvers for strings that leverages the observation that many variables introduced by these solvers can be shared. Our implementation in the SMT solver CVC4 of witness sharing for these variables, as well as related techniques for recasting regular expressions as extended string constraints, leads to significant performance gains with respect to the state of the art, both in terms of number of benchmarks solved and run times.

As ongoing work, we are further investigating optimizations to the reductions used in this paper. We believe that the principle of witness sharing can be applied even more aggressively to infer when (pairs of) *input* variables are constrained to be equivalent to witness terms and hence can be equated as a preprocessing step. More generally, it can be used as a way of optimizing other CDCL(T) theory solvers that introduce fresh variables within theory lemmas they generate. For example, some procedures for reasoning about finite sets [8] use fresh variables to witness when two sets are disequal. We conjecture that witness sharing can be applied fruitfully there as well.

[3]Overall, CVC4 and Z3STR3 disagreed on 440 FStrInt and 22 Transf benchmarks whereas CVC4 and Z3-Trau disagreed on 416 Transf benchmarks. Out of those cases, Z3STR3 accepted all 325 models produced by CVC4 and rejected all 137 of its own models while Z3-Trau accepted all 343 models produced by CVC4 and rejected all 73 of its own models.

References

[1] str_int_benchmarks. https://github.com/plfm-iis/str_int_benchmarks, 2019.

[2] Z3-Trau. https://github.com/guluchen/z3/tree/new_trau, 2019.

[3] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, J. Dolby, P. Janku, H. Lin, L. Holík, and W. Wu. Efficient handling of string-number conversion. In A. F. Donaldson and E. Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 943–957. ACM, 2020.

[4] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In A. Cohen and M. T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 602–617. ACM, 2017.

[5] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 150–166, 2014.

[6] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. S. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan. Stratified abstraction of access control policies. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2020.

[7] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. S. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In N. Bjørner and A. Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[8] K. Bansal, A. Reynolds, C. W. Barrett, and C. Tinelli. A new decision procedure for finite sets and cardinality constraints in SMT. In *Proceedings of IJCAR'16*, volume 9706 of *LNCS*, pages 82–98. Springer, 2016.

[9] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[10] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. Clarke, T. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*. Springer, 2018.

[11] M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In D. Stewart and G. Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59. IEEE, 2017.

[12] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2009.

[13] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh. Stringfuzz: A fuzzer for string solvers. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 45–51. Springer, 2018.

[14] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *PACMPL*, 3(POPL):49:1–49:30, 2019.

[15] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[16] X. Fu and C. Li. A string constraint solver for detecting web application vulnerability. In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*, SEKE'2010. Knowledge Systems Institute Graduate School, 2010.

[17] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, pages 248–262. Springer-Verlag, 2011.

[18] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, 2012.

[19] G. Li and I. Ghosh. Pass: string solving with parameterized array and interval automaton. In *Haifa Verification Conference*, pages 15–31. Springer, 2013.

[20] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 646–662, 2014.

[21] A. Reynolds, A. Nötzli, C. W. Barrett, and C. Tinelli. High-level abstractions for simplifying extended string constraints in SMT. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2019.

[22] A. Reynolds, M. Woo, C. W. Barrett, D. Brumley, T. Liang, and C. Tinelli. Scaling up DPLL(T) string solvers using context-dependent simplification. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2017.

[23] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 513–528. IEEE Computer Society, 2010.

[24] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 13–22. IEEE, 2007.

[25] M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1232–1243. ACM, 2014.

[26] M. Trinh, D. Chu, and J. Jaffar. Progressive reasoning over recursively-defined strings. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 218–240, 2016.

[27] M. Veanes, N. Bjørner, and L. De Moura. Symbolic automata constraint solving. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 640–654. Springer-Verlag, 2010.

[28] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 425–446. Springer, 2010.

[29] Wei-Cheng Wu. Py-Conbyte. https://github.com/spencerwuwu/py-conbyte, 2019.

[30] 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, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 154–157, 2010.

[31] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In B. Meyer, L. Baresi, and M. Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering,*

ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pages 114–124. ACM, 2013.