Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification

Andrew Reynolds[1], Maverick Woo[2], Clark Barrett[3], David Brumley[2], Tianyi Liang[4], and Cesare Tinelli[1(✉)]

[1] Department of Computer Science, The University of Iowa, Iowa City, USA
cesare-tinelli@uiowa.edu
[2] CyLab, Carnegie Mellon University, Pittsburgh, USA
[3] Department of Computer Science, Stanford University, Stanford, USA
[4] Two Sigma, New York, USA

Abstract. Efficient reasoning about strings is essential to a growing number of security and verification applications. We describe satisfiability checking techniques in an extended theory of strings that includes operators commonly occurring in these applications, such as contains, index_of and replace. We introduce a novel context-dependent simplification technique that improves the scalability of string solvers on challenging constraints coming from real-world problems. Our evaluation shows that an implementation of these techniques in the SMT solver CVC4 significantly outperforms state-of-the-art string solvers on benchmarks generated using PyEx, a symbolic execution engine for Python programs. Using a test suite sampled from four popular Python packages, we show that PyEx uses only 41% of the runtime when coupled with CVC4 than when coupled with CVC4's closest competitor while achieving comparable program coverage.

1 Introduction

A growing number of applications of static analysis techniques have benefited from automated reasoning tools for string constraints. The effect of such tools on symbolic execution in particular has been transformative. At a high level, symbolic execution *runs* a program under analysis by representing its input values symbolically and tracking the program variables as expressions over these symbolic values, together with other concrete values in the program. The collected expressions are then analyzed by an automated reasoning tool to determine path feasibility at branches or other properties such as security vulnerabilities at points of interest. With the ever-increasing hardware and automated reasoning capabilities, symbolic execution has enjoyed much success in practice. A recent example in the cybersecurity realm was the DARPA Cyber Grand Challenge,[1] which featured a number of Cyber Reasoning Systems that heavily relied on symbolic execution techniques (see, e.g., [7,24]).

[1] See http://www.darpa.mil/program/cyber-grand-challenge.

R. Majumdar and V. Kunčak (Eds.): CAV 2017, Part II, LNCS 10427, pp. 453–474, 2017.
DOI: 10.1007/978-3-319-63390-9_24

Prior to the availability of string-capable reasoning tools, developers of symbolic execution engines had to adopt various ad-hoc heuristics to cope with strings and other variable-length inputs. One popular heuristic is to impose an artificial upper-bound on the length of these inputs. Unfortunately, this not only compromises analysis accuracy, since the chosen upper bounds may be too low in face of adversarial inputs, but it also leads to inefficiencies in solvers when practitioners, in an attempt to mitigate this problem, may end up setting the upper bounds too high. To address this issue, a number of SMT solvers have been extended recently with native support for *unbounded* strings and length constraints [2,19,26,30]. These solvers have dramatically improved both in performance and robustness over the past few years, enabling a new generation of symbolic execution tools that support faithful string reasoning.

This paper revisits approaches for solving *extended string constraints*, which allow a rich language of string terms over operators such as $\mathsf{contains}$, $\mathsf{index\_of}$ and $\mathsf{replace}$. Earlier techniques for extended string constraints [5,18,27,30] often rely on eager reductions to a core language of constraints, with the effect of requiring the solver to deal with fairly large or complex combination of basic constraints. For instance, encoding the constraint $\neg\mathsf{contains}(x, y)$ commonly involves bounded universal quantification over the integers, to state that string y does not occur at any position in x. In this work, we start with the observation that DPLL(T)-based SMT solvers [12] often reason in contexts where string variables are equated to (partially) concrete values. Based on this, we have developed a way to leverage efficient context-dependent simplification techniques and reduce extended constraints to a core language lazily instead.

Contribution and Significance. We extend a calculus by Liang et al. [19] to handle *extended string constraints* (i.e. constraints over substr, $\mathsf{contains}$, $\mathsf{index\_of}$ and $\mathsf{replace}$) using a combination of two techniques:

- a reduction of extended string constraints to basic ones involving bounded quantification (Sect. 3.1), and
- an inference technique based on context-dependent simplification (Sect. 3.2) which supplements this reduction and in practice significantly improves the scalability of our approach on constraints coming from symbolic execution.

Additionally, we provide a new set of 25,421 publicly-available benchmarks over extended string constraints. These benchmarks were generated by running PyEx, a new symbolic executor for Python programs based on PyExZ3 [3], over a test suite of 19 target functions sampled from four popular Python packages. We discuss an experimental evaluation showing that our implementation in the SMT solver CVC4 significantly outperforms other state-of-the-art string solvers in finding models for these benchmarks.

Finally, we discuss how the superior performance of CVC4 in comparison to other solvers translates into real-life benefits for Python developers using PyEx.

Structure of the Paper. After some formal preliminaries, we briefly review a calculus for basic string constraints in Sect. 2 that is an abbreviated version of [19].

We present new techniques for extended string constraints in Sect. 3, and evaluate these techniques on real-world queries generated by PyEx in Sect. 5.

1.1 Related Work

The satisfiability of word equations was proven decidable by Makanin [21] and then given a PSPACE algorithm by Plandowski [22]. The decidability of the fairly restricted language of word equations with length constraints is an open problem [11]. In practice, a number of approaches for solving string and regular expression constraints rely on reductions to automata [10,15,28] or bit-vector constraints for fixed-length strings [16]. More recently, new approaches have been developed for the satisfiability problem for (unbounded) word equations and memberships with length [2,19,20,27,30] within SMT solvers. Among these, Z3-STR [30] and S3 [27] are third-party extensions of the SMT solver Z3 [9] adding support for string constraints via reductions to linear arithmetic and uninterpreted functions. This support includes extended string constraints over a signature similar to the one we consider in this paper. With respect to these solvers, our string solver is fully integrated into the architecture of CVC4, meaning it can be combined with other theories of CVC4, such as algebraic datatypes and arrays.

This paper is similar in scope to work by Bjørner et al. [5], which gives decidability results and an approach for string library functions, including contains and replace. As in that work, we reduce the satisfiability problem for extended string constraints to a core language with bounded quantification. We also incorporate simplification techniques that improve performance by completely or partially avoiding this reduction.

Our target application, symbolic execution, has a rich history, starting from the seminal work of King [17]. Common modern symbolic execution tools include SAGE [13], KLEE [6], S2E [8], and Mayhem [7], which are all designed to analyze low-level binary or source code. In contrast, this paper considers constraints generated from the symbolic executor PyEx, which is designed to analyze Python code and includes support for string variables.

1.2 Formal 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. We also assume signatures Σ contain the Boolean sort Bool and Boolean constant symbols $\top$ and $\bot$ for true and false. Without loss of generality, we assume $\approx$ is the only predicate symbol in Σ, as all other predicates may be modeled as functions with return sort Bool. If P is a function with return sort Bool, we will commonly write $P(\boldsymbol{t})$ as shorthand for $P(\boldsymbol{t}) \approx \top$. If e is a term or a formula, we denote by $\mathcal{V}(e)$ and $\mathcal{T}(e)$ the set of free variables and subterms of e respectively, extending these notations to tuples and sets of terms or formulas as expected.

Σ_{A} $n : \mathsf{Int}$ for all $n \in \mathbb{N}$ $+ : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ $- : \mathsf{Int} \to \mathsf{Int}$ $\geqslant : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$

Σ_{S} $l : \mathsf{Str}$ for all $l \in \mathcal{A}^*$ $\mathsf{con} : \mathsf{Str} \times \cdots \times \mathsf{Str} \to \mathsf{Str}$ $\mathsf{len} : \mathsf{Str} \to \mathsf{Int}$

Σ_{X} $\mathsf{substr} : \mathsf{Str} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{Str}$ $\mathsf{contains} : \mathsf{Str} \times \mathsf{Str} \to \mathsf{Bool}$
$\mathsf{index\_of} : \mathsf{Str} \times \mathsf{Str} \times \mathsf{Int} \to \mathsf{Int}$ $\mathsf{replace} : \mathsf{Str} \times \mathsf{Str} \times \mathsf{Str} \to \mathsf{Str}$

Fig. 1. Functions in signature Σ_{ASX}. Str and Int denote strings and integers respectively.

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 $\mathbf{I}$. A set Γ of Σ-formulas *entails in* T a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. We write $\Gamma \models \varphi$ to denote entailment in the (empty) theory of equality. We say Γ *propositionally entails* φ, written $\Gamma \models_p \varphi$, if Γ entails φ when considering all atoms in γ and φ as propositional variables. Two Σ-terms or Σ-formulas are *equivalent in* T if they are satisfied by the same models of T. Two formulas φ_1 and φ_2 are *equisatisfiable in* T if $\exists \boldsymbol{x}_1.\, \varphi_1$ and $\exists \boldsymbol{x}_2.\, \varphi_2$ are equivalent in T where $\boldsymbol{x}_i$ collects the free variables of φ_i that do not occur free in φ_j with $i \neq j$.

We consider an extended theory T_{ASX} of strings and length equations, whose signature Σ_{ASX} is given in Fig. 1. We assume a fixed finite alphabet $\mathcal{A}$ of characters. The signature includes the sorts Str and Int denoting strings and integers respectively. Figure 1 divides the signature Σ_{ASX} into three parts, which we denote by $\Sigma_{\mathsf{A}}, \Sigma_{\mathsf{S}}$ and Σ_{X}. We will write Σ_{AS} to denote $\Sigma_{\mathsf{A}} \cup \Sigma_{\mathsf{S}}$ and so on. The subsignature Σ_{A} is provided on the top line of Fig. 1 and includes the usual symbols of *linear* integer arithmetic, interpreted as expected. We will commonly 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 subsignature Σ_{S} is provided on the second line and includes: 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 $\mathsf{con} : \mathsf{Str} \times \ldots \times \mathsf{Str} \to \mathsf{Str}$, interpreted as word concatenation; and a function symbol $\mathsf{len} : \mathsf{Str} \to \mathsf{Int}$, interpreted as the word length function. The subsignature Σ_{X} is provided in the remainder of the figure.

We refer to the function symbols in Σ_{X} as *extended functions*, and terms whose top symbol is in Σ_{X} as *extended function terms*. 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; $\mathsf{contains}(x, y)$ is interpreted as true if and only if string x contains string y; $\mathsf{index\_of}(x, y, n)$ 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 y is empty or x does not contain y.

An *atomic term* is either a constant or a variable. A *flat term* is a term of the form $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are variables. A *string term* is one

$$\begin{array}{ll} \mathsf{con}(\boldsymbol{s}, \epsilon, \boldsymbol{u}) \to \mathsf{con}(\boldsymbol{s}, \boldsymbol{u}) & \mathsf{con}() \to \epsilon \quad \mathsf{con}(s) \to s \\ \mathsf{con}(\boldsymbol{s}, \mathsf{con}(\boldsymbol{t}), \boldsymbol{u}) \to \mathsf{con}(\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u}) & \mathsf{con}(\boldsymbol{s}, c_1 \cdots c_i, c_{i+1} \cdots c_n, \boldsymbol{u}) \to \mathsf{con}(\boldsymbol{s}, c_1 \cdots c_n, \boldsymbol{u}) \\ \mathsf{len}(c_1 \cdots c_n) \to n & \mathsf{len}(\mathsf{con}(s_1, \ldots, s_n)) \to \mathsf{len}\, s_1 + \cdots + \mathsf{len}\, s_n \end{array}$$

Fig. 2. Simplification rules for Σ_{AS}-terms.

that contains function symbols from Σ_{SX} only. A string term is *basic* if contains function symbols from Σ_{S} only, and *extended* otherwise. A *(basic, extended) string constraint* is a (dis)equality between (basic, extended) string terms. An *arithmetic constraint* is an inequality or (dis)equality between linear combinations of atomic and/or string terms with integer sort. Notice that (dis)equalities between integer variables and constraints such as $\mathsf{len}\, x \approx \mathsf{len}\, y$ are both string and arithmetic constraints. A T_{ASX}-constraint is either a string or an arithmetic constraint. Without loss of generality, we consider the satisfiability problem for Σ_{ASX}-formulas composed of T_{ASX}-constraints only.

If E is a finite set of basic string constraints, the *congruence closure* of E is the set

$$\begin{aligned} \widehat{E} = \{&s \approx t \mid s, t \in \mathcal{T}(E),\ E \models s \approx t\} \cup \\ &\{s \not\approx t \mid s, t \in \mathcal{T}(E),\ s' \not\approx t' \in E \cup L,\ E \models s \approx s' \wedge t \approx t' \text{ for some } s', t'\} \end{aligned}$$

where $L = \{l_1 \not\approx l_2 \mid \text{for all distinct } l_1, l_2 \in \mathcal{A}^*\} \cup \{\top \not\approx \bot\}$. The congruence closure of E induces an equivalence relation over $\mathcal{T}(E)$ where two terms s, t are equivalent iff $s \approx t \in \widehat{E}$. For all $t \in \mathcal{T}(E)$, we denote its equivalence class in $\widehat{E}$ by $[t]_E$ or just $[t]$ when E is clear.

Given a term t, we write $t{\downarrow}$ to denote its *simplified form*, where $t{\downarrow}$ is a term that is equivalent to t in T_{ASX} and is not simplifiable further (i.e., $(t{\downarrow}){\downarrow} = t{\downarrow}$). We do not insist that simplified forms be canonical, that is, equivalent terms need not have the same simplified form. Rules for computing the simplified form of Σ_{AS}-terms are given in Fig. 2. Rules for computing the simplified form of other Σ_{ASX}-terms are fairly sophisticated and will be described in detail in Sect. 3.2. Given a tuple of basic string terms $\boldsymbol{t}$, we write $\boldsymbol{t}{\downarrow}$ to denote a tuple of atomic string terms corresponding to the arguments of $\mathsf{con}(\boldsymbol{t}){\downarrow}$. For example, if $c_1, c_2 \in \mathcal{A}$, $(c_1, \mathsf{con}(c_2, x), y){\downarrow} = (c_1 c_2, x, y)$ and $(x, \epsilon){\downarrow} = (x)$. Given an arbitrary Σ_{ASX}-formula φ, we write $\lfloor \varphi \rfloor$ to denote an equisatisfiable (purified) formula whose atoms are T_{ASX}-constraints, and where t is in simplified form (i.e., $t = t{\downarrow}$) for all its subterms t. For example, $\lfloor \mathsf{substr}(x, n_0, n_1{+}1) \approx y \rfloor$ is $\mathsf{substr}(x, n_0, n_2) \approx y \wedge n_2 \approx n_1 + 1$ for a fresh integer variable n_2.

2 A Calculus for Basic String Constraints

Liang et al. [19] developed a calculus for the satisfiability of finite conjunctions of constraints in a theory of strings with length and regular expressions. This section presents a modified version of that calculus. We focus on the portion of that calculus that handles string equalities and length constraints, and omit

discussion of its other aspects. Furthermore, we extend that calculus with support for constraints involving extended functions. To simplify the description of this extension and make it self-contained, we also extend the calculus to model propositional reasoning as well, making it applicable to Σ_{ASX}-formulas instead of just conjunctions of T_{ASX}-constraints.

Definition 1 (Configurations). *A configuration is either the distinguished symbol* unsat *or a tuple of the form* $\langle \mathsf{G}, \mathsf{S}, \mathsf{A} \rangle$ *where* G *is a set of* Σ_{ASX}*-formulas,* A *is a set of arithmetic constraints, and* S *is a tuple of the form* $(\mathsf{E}, \mathsf{X}, \mathsf{F}, \mathsf{N})$*, where:*

- E *is a set of basic string equalities;*
- X *is a set of equalities of the form* $x \approx t$*, where* x *is a variable and* t *is a flat extended function term;*
- F *is a set of pairs* $s \mapsto \boldsymbol{a}$ *where* $s \in \mathcal{T}(\mathsf{E})$ *and* $\boldsymbol{a}$ *is a tuple of atomic string terms;*
- N *is a set of pairs* $e \mapsto \boldsymbol{a}$ *where* e *is an equivalence class of* $\widehat{\mathsf{E}}$ *and* $\boldsymbol{a}$ *is a tuple of atomic string terms.* □

A configuration $\langle \mathsf{G}, \mathsf{S}, \mathsf{A} \rangle$ models the internal state of various modules of a DPLL(T)-based solver. The component G collects the formulas being processed by the solver's propositional satisfiability (SAT) engine; S models the state of a theory solver for strings; and A collects the constraints given to a solver for linear integer arithmetic. Initial configurations have the form $\langle \{\varphi\}, \overline{\varnothing}, \varnothing \rangle$ where φ is a *quantifier-free* T_{ASX}-formula to be checked for satisfiability and $\overline{\varnothing}$ abbreviates the tuple $(\varnothing, \varnothing, \varnothing, \varnothing)$.

We describe a calculus for the satisfiability of string constraints by a set of derivation rules that modify configurations. The rules are given in *guarded assignment form*, where the top of the rule describes the conditions under which the rule can be applied, and the bottom of the rule either is unsat, or otherwise describes the resulting modifications to the components of our configuration. A rule may have multiple, alternative conclusions separated by $\|$. An application of a rule is *redundant* if it has a conclusion where each component in the derived configuration is a subset of the corresponding component in the premise configuration. A configuration other than unsat is *saturated* if every possible application of a derivation rule to it is redundant. A *derivation tree* is a tree where each node is a configuration whose children, if any, are obtained by a non-redundant application of a rule of the calculus. A *closed* derivation tree (where all terminal nodes are unsat) with root node $\langle \{\varphi\}, \overline{\varnothing}, \varnothing \rangle$ is a proof that φ is unsatisfiable in T_{ASX}. A derivation tree with root node $\langle \{\varphi\}, \overline{\varnothing}, \varnothing \rangle$ and a saturated leaf is, under certain assumptions (see Theorem 1) a witness that φ is satisfiable in T_{ASX}.

To discuss the rules, we first introduce the notation for updating the internal state $S = (E, X, F, N)$ of our string solver. Let M be a set of T_{ASX}-constraints. By introducing enough fresh variables, we can construct an equisatisfiable set $M_E \cup M_X$ where M_E is a set of Σ_{AS}-literals and M_X is a set of equalities of the form $y \approx t$, with y a variable and t a flat extended function term. For simplicity,

$$\textsf{Prop-Assign}\ \frac{\mathsf{G} = \mathsf{G}', \varphi \qquad \varphi \text{ is quantifier-free}}{\|_{M \models_p \varphi}\ \mathsf{G} := \mathsf{G}' \quad \mathsf{S} := \mathsf{S} \oplus M|_S \quad \mathsf{A} := \mathsf{A} \cup M|_A} \qquad \textsf{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\mathsf{unsat}}$$

$$\textsf{A-Prop}\ \frac{\mathsf{S} \models s \approx t \quad s,t : \mathsf{Int}}{\mathsf{A} := \mathsf{A}, s \approx t} \qquad \textsf{S-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} s \approx t \quad s,t : \mathsf{Int}}{\mathsf{S} := \mathsf{S} \oplus \{s \approx t\}} \qquad \textsf{S-Conf}\ \frac{s \not\approx s \in \widehat{\mathsf{E}}}{\mathsf{unsat}}$$

$$\textsf{L-Eq}\ \frac{x \approx t \in \widehat{\mathsf{E}} \quad x : \mathsf{Str}}{\mathsf{A} := \mathsf{A}, \mathsf{len}\, x \approx (\mathsf{len}\, t)\downarrow} \qquad \textsf{L-Geq}\ \frac{x \in \mathcal{V}(\mathsf{S} \cup \mathsf{A}) \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S} \oplus \{x \approx \epsilon\} \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}\, x > 0}$$

Fig. 3. Rules modeling the interaction between the propositional, string and arithmetic subsolvers.

we assume here that M_E does not contain string disequalities.[2] We define the *external update of* S *with* M, written $S \oplus M$, to be the tuple $(E \cup M_E, X \cup M_X, \varnothing, \varnothing)$. For Σ_{AS}-terms t_1, t_2, we define the *internal update of* S *with* $t_1 \approx t_2$, written $S \odot t_1 \approx t_2$, to be the tuple $(E \cup \{t_1 \approx t_2\}, X, F\sigma, \varnothing)$, where σ is the substitution $\{t_1 \mapsto t_2\}$ if t_1 is a string variable and is the empty substitution otherwise, and $F\sigma$ is the result of replacing the right hand side of all pairs $y \mapsto \boldsymbol{a}$ in F with $(\boldsymbol{a}\sigma)\downarrow$.

The rules in Fig. 3 model the basic interaction between the various subsolvers in our approach. The rule Prop-Assign considers each *propositional satisfying assignment* M for a quantifier-free formula $\varphi \in \mathsf{G}$, i.e., each set M of literals such as every atom of φ occurs either positively or negatively (but not both) in M, all the atoms in M occur in φ, and $M \models_p \varphi$. For each such M, Prop-Assign has a conclusion where the string constraints in M (denoted as $M|_S$) are given to the string subsolver and the arithmetic constraints (denoted as $M|_A$) are given to the arithmetic subsolver. The arithmetic and string solvers use rules A-Prop and S-Prop to share equalities between (shared) arithmetic terms, and use A-Conf and S-Conf to report that their respective set of constraints is unsatisfiable. In those rules and in the rest of the paper, $\models_{\mathsf{LIA}}$ denotes entailment in linear integer arithmetic. The rules L-Eq and L-Geq respectively infer and guess arithmetic constraints involving string length.

Example 1. Consider the formula φ of the form $\mathsf{len}\, x > \mathsf{len}\, y \wedge y \approx \mathsf{con}(x, \mathsf{a})$. Starting from configuration $\langle \{\varphi\}, \overline{\varnothing}, \varnothing \rangle$, we may apply Prop-Assign to remove φ from G and update S and A based the propositional satisfying assignment for φ. We obtain a configuration where A is $\{\mathsf{len}\, x > \mathsf{len}\, y\}$ and the E component of S is $\{y \approx \mathsf{con}(x, \mathsf{a})\}$. Since $(\mathsf{len}(\mathsf{con}(x, \mathsf{a})))\downarrow = \mathsf{len}\, x + 1$, we may add $\mathsf{len}\, y \approx \mathsf{len}\, x + 1$ to A by the rule L-Eq. Since $\mathsf{len}\, x > \mathsf{len}\, y, \mathsf{len}\, y \approx \mathsf{len}\, x + 1 \models_{\mathsf{LIA}} \bot$, we may apply A-Conf to derive the unsat configuration, establishing that φ is unsatisfiable in T_{ASX}. □

[2] String disequalities can be reduced to a finite disjunction of equalities, e.g. see [1]. More sophisticated and efficient methods for handling string disequalities are given in [19].

$$\textsf{F-Form1}\ \frac{\begin{array}{c}\mathsf{con}(t_1,\ldots,t_n)\in\mathcal{T}(\mathsf{E})\\ \mathsf{N}\,[t_i]=\boldsymbol{s}_i \text{ for } i=1,\ldots,n\end{array}}{\mathsf{F}:=\mathsf{F},\mathsf{con}(t_1,\ldots,t_n)\mapsto(\boldsymbol{s}_1,\ldots,\boldsymbol{s}_n)\downarrow}\qquad \textsf{F-Form2}\ \frac{l\in\mathcal{T}(\mathsf{E})}{\mathsf{F}:=\mathsf{F},l\mapsto(l)\downarrow}$$

$$\textsf{N-Form1}\ \frac{\begin{array}{c}e\not\subseteq\mathcal{V}(\mathsf{E})\\ \mathsf{F}\,t=\boldsymbol{a}\ \text{for all}\ t\in e\backslash\mathcal{V}(\mathsf{E})\end{array}}{\mathsf{N}:=\mathsf{N},e\mapsto\boldsymbol{a}}\qquad \textsf{N-Form2}\ \frac{\begin{array}{c}[x]\subseteq\mathcal{V}(\mathsf{E})\\ x\preceq y\ \text{for all}\ y\in[x]\end{array}}{\mathsf{N}:=\mathsf{N},[x]\mapsto(x)}$$

$$\textsf{F-Unify}\ \frac{\mathsf{F}\,s=(\boldsymbol{w},u,\boldsymbol{u}_1)\quad \mathsf{F}\,t=(\boldsymbol{w},v,\boldsymbol{v}_1)\quad s\approx t\in\widehat{\mathsf{E}}\quad u\prec v\quad \mathsf{E}\models\mathsf{len}\,u\approx\mathsf{len}\,v}{\mathsf{S}:=\mathsf{S}\odot v\approx u}$$

$$\textsf{F-Split}\ \frac{\begin{array}{c}\mathsf{F}\,s=(\boldsymbol{w},u,\boldsymbol{u}_1)\quad \mathsf{F}\,t=(\boldsymbol{w},v,\boldsymbol{v}_1)\quad s\approx t\in\widehat{\mathsf{E}}\quad \mathsf{E}\models\mathsf{len}\,u\not\approx\mathsf{len}\,v\\ u\notin\mathcal{V}(\boldsymbol{v}_1)\quad v\notin\mathcal{V}(\boldsymbol{u}_1)\end{array}}{\mathsf{S}:=\mathsf{S}\odot u\approx\mathsf{con}(v,z)\quad\|\quad\mathsf{S}:=\mathsf{S}\odot v\approx\mathsf{con}(u,z)}$$

$$\textsf{L-Split}\ \frac{x,y\in\mathcal{V}(\mathsf{E})\quad x,y:\mathsf{Str}}{\mathsf{E}:=\mathsf{E},\mathsf{len}\,x\approx\mathsf{len}\,y\quad\|\quad\mathsf{E}:=\mathsf{E},\mathsf{len}\,x\not\approx\mathsf{len}\,y}$$

Fig. 4. String derivation rules. The rules construct flat forms F and normal forms N for string terms. The letter l denotes a string constant, and z denotes a fresh string variable.

The rules in Fig. 4 are used by the string solver for building the mappings F and N. We defer discussion of the X component of configurations until Sect. 3. In the rules, we assume a total ordering $\prec$ on string terms, whose only restriction is that $t_1\prec t_2$ if t_1 is variable and t_2 is not. For a term t, we call $\mathsf{F}\,t$ the *flat form* of t, and for an equivalence class e, we call $\mathsf{N}\,e$ the *normal form* of e. We construct the mappings F and N using the rules F-Form1, F-Form2, N-Form1 and N-Form2 in a mutually recursive fashion. The remaining three rules apply to cases where the above rules do not result in complete mappings for F and N. In the case where we compute flat forms for two terms s and t in the same equivalence class that share a common prefix $\boldsymbol{w}$ followed by two distinct variables u and v, we apply F-Unify to infer $u\approx v$ in the case that the lengths of u and v are equal, otherwise we apply F-Split to infer that u is a prefix of v or vice versa in the case that the lengths of u and v are disequal. The rule L-Split splits the derivation based on equality between the lengths of string variables x and y, which we use to derive configurations where one of these two rules applies.

Example 2. Consider a configuration where E is $\{y\approx\mathsf{con}(\mathsf{a},x),x\approx\mathsf{con}(u,z),y\approx\mathsf{con}(\mathsf{a},v,z),w\approx\mathsf{a},\mathsf{len}\,u\approx\mathsf{len}\,v\}$. The equivalence classes of $\widehat{\mathsf{E}}$ are

$$\{y,\mathsf{con}(\mathsf{a},x),\mathsf{con}(\mathsf{a},v,\mathsf{a})\}\quad\{x,\mathsf{con}(u,\mathsf{a})\}\quad\{u\}\quad\{v\}\quad\{w,\mathsf{a}\}$$

Using N-Form2, we obtain $\mathsf{N}\,[u]=(u)$ and $\mathsf{N}\,[v]=(v)$. Using F-Form1 and N-Form1, we obtain $\mathsf{F}\,\mathsf{con}(u,\mathsf{a})=\mathsf{N}\,[x]=(u,z)$ and $\mathsf{F}\,\mathsf{a}=\mathsf{N}\,[w]=(\mathsf{a})$. For $[y]$, we

use F-Form1 to obtain $\mathsf{F}\,\mathsf{con}(\mathsf{a}, x) = (\mathsf{a}, u, \mathsf{a})$ and $\mathsf{F}\,\mathsf{con}(\mathsf{a}, v, \mathsf{a}) = (\mathsf{a}, v, \mathsf{a})$. Since the flat forms of these terms are not the same and $\mathsf{len}\, u \approx \mathsf{len}\, v \in \mathsf{E}$, we may apply F-Unify to conclude $v \approx u$. We update S to $\mathsf{S} \odot u \approx v$, after which the equivalence classes are:

$$\{y, \mathsf{con}(\mathsf{a}, x), \mathsf{con}(\mathsf{a}, v, \mathsf{a})\} \quad \{x, \mathsf{con}(u, \mathsf{a})\} \quad \{u, v\} \quad \{w, \mathsf{a}\}$$

and $\mathsf{F}\,\mathsf{con}(\mathsf{a}, v, \mathsf{a})$ is now $(\mathsf{a}, u, \mathsf{a})$. Then we can use N-Form1 and N-Form2 to reconstruct N. Since $\mathsf{F}\,\mathsf{con}(\mathsf{a}, x) = \mathsf{F}\,\mathsf{con}(\mathsf{a}, v, \mathsf{a}) = (\mathsf{a}, u, \mathsf{a})$, we can obtain $\mathsf{N}[y] = (\mathsf{a}, u, \mathsf{a})$. This results in a configuration where N is a complete mapping over the equivalence classes and no more rules apply, indicating that E is satisfiable in T_{ASX}. □

We say a configuration is *cyclic* if $\widehat{\mathsf{E}}$ either contains a chain of equalities of the form $s \approx \mathsf{con}(\boldsymbol{t_1}), s_1 \approx \mathsf{con}(\boldsymbol{t_2}), \ldots, s_{n-1} \approx \mathsf{con}(\boldsymbol{t_n}), s_n \approx s$ where s_i is a term from $\boldsymbol{t_i}$ for each i, or an equality of the form $s \approx t$ where $\mathsf{F}\,s = (\boldsymbol{w}, \boldsymbol{u}), \mathsf{F}\,t = (\boldsymbol{w}, \boldsymbol{v}), \boldsymbol{w}$ is the maximal common prefix of $\mathsf{F}\,s$, and $\mathsf{F}\,t$ and $\mathcal{V}(\boldsymbol{u}) \cap \mathcal{V}(\boldsymbol{v})$ is non-empty. Recent techniques have been proposed for cyclic string constraints [19,29]. We instead focus primarily on acyclic string constraints in the following result.

Theorem 1. *For all quantifier-free Σ_{AS}-formulas φ, the following hold.*

1. *There is a closed derivation tree with root $\langle\{\varphi\}, \overline{\varnothing}, \varnothing\rangle$ only if φ is unsat in T_{ASX}.*
2. *There is a derivation tree with root $\langle\{\varphi\}, \overline{\varnothing}, \varnothing\rangle$ containing an acyclic saturated configuration only if φ is sat in T_{ASX}.*

3 Techniques for Extended String Constraints

This section gives a novel extension of the calculus in the previous section for determining the satisfiability of T_{ASX}-constraints. While the decidability of this problem is not known [5], we focus on techniques that are both refutation-sound and model-sound but guaranteed to terminate in general. We introduce two techniques for establishing the (un)satisfiability of T_{ASX}-constraints S, described by the additional derivation rules in Figs. 5 and 6 which supplement those from Sect. 2.

3.1 Expanding Extended Function Terms to Bounded Integer Quantification

The satisfiability problem for equalities over Σ_{ASX}-terms can be reduced to the satisfiability problem for possibly quantified Σ_{AS}-formulas. This reduction is provided by rule Ext-Expand in Fig. 5 which, given $x \approx t \in \mathsf{X}$, adds an equisatisfiable formula $[\![x \approx t]\!]$ to G, the *expanded form* of $x \approx t$. The rules also removes $x \approx t$ from X, effectively marking it as processed. Since $[\![x \approx t]\!]$ keeps the (free) variables of $x \approx t$, any interpretation later found to satisfy $[\![x \approx t]\!]$ will also satisfy $x \approx t$.

$$\textsf{Ext-Expand}\ \frac{x \approx t \in \mathsf{X}}{\mathsf{G} := \mathsf{G}, \lfloor [\![x \approx t]\!] \rfloor \qquad \mathsf{X} := \mathsf{X} \backslash \{x \approx t\}} \quad \text{where}$$

$$[\![x \approx \mathsf{substr}(y,n,m)]\!] = \mathsf{ite}(\ 0 \leqslant n < \mathsf{len}\, y \wedge 0 < m,\ y \approx \mathsf{con}(z_1, x, z_2) \wedge \mathsf{len}\, z_1 \approx n \wedge \mathsf{len}\, z_2 \approx \mathsf{len}\, y \dot{-} m,\ x \approx \epsilon\ )$$

$$[\![x \approx \mathsf{contains}(y,z)]\!] = (x \not\approx \top) \Leftrightarrow \forall k.\, 0 \leqslant k \leqslant \mathsf{len}\, y - \mathsf{len}\, z \Rightarrow \mathsf{substr}(y,k,\mathsf{len}\, z) \not\approx z$$

$$[\![x \approx \mathsf{index\_of}(y,z,n)]\!] = \mathsf{ite}(\ 0 \leqslant n \wedge z \not\approx \epsilon \wedge \mathsf{contains}(y',z),\ \mathsf{substr}(y',x',\mathsf{len}\, z) \approx z \wedge \neg\mathsf{contains}(\mathsf{substr}(y',0,x' + \mathsf{len}\, z - 1), z),\ x \approx -1\ )$$
$$\text{with } y' = \mathsf{substr}(y,n,\mathsf{len}\, y - n) \text{ and } x' = x - n$$

$$[\![x \approx \mathsf{replace}(y,z,w)]\!] = \mathsf{ite}(\ \mathsf{contains}(y,z) \wedge z \not\approx \epsilon,\ x \approx \mathsf{con}(z_1, w, z_2) \wedge y \approx \mathsf{con}(z_1, z, z_2) \wedge \mathsf{index\_of}(y,z,0) \approx \mathsf{len}\, z_1,\ x \approx y\ )$$

$$\textsf{B-Val}\ \frac{t : \mathsf{Int} \quad \mathsf{n} \text{ is a numeral}}{\mathsf{A} := \mathsf{A}, t \leqslant \mathsf{n} \quad \| \quad \mathsf{A} := \mathsf{A}, t > \mathsf{n}} \qquad \textsf{B-Inst}\ \frac{\mathsf{G} = \mathsf{G}', \varphi[\forall k.\, 0 \leqslant k \leqslant t \Rightarrow \psi] \quad \mathsf{A} \models_{\mathsf{LIA}} t \leqslant \mathsf{n} \text{ for some numeral } \mathsf{n}}{\mathsf{G} := \mathsf{G}', \lfloor \varphi[\wedge_{i=0}^{\mathsf{n}} \psi\{k \mapsto i\}] \rfloor}$$

Fig. 5. Rules for reducing Σ_{ASX}-constraints to Σ_{AS}-constraints, where z_1, z_2 are fresh variables, $n_1 \dot{-} n_2$ denotes the maximum of $n_1 - n_2$ and 0, and ite is the if-then-else connective.

The definition of expanded form for the possible cases of t are given below rule Ext-Expand.[3] We remark that this rule can be applied only finitely many times. For an intuition of why, consider any ordering $\prec$ on function symbols such that $f \prec g$ if g is an extended function and f is not, and substr $\prec$ contains $\prec$ index_of $\prec$ replace. In all cases, all function symbols of $[\![x \approx f(\boldsymbol{t})]\!]$ are smaller than f in this ordering.

Note that the reduction introduces formulas with *bounded integer quantification*, that is, formulas of the form $\forall k.\, 0 \leqslant k \leqslant t \Rightarrow \varphi$, where t does not contain k and φ is quantifier-free. Special consideration is needed to handle formulas of this form. We employ a pragmatic approach, modeled by the other two rules in Fig. 5, which guesses upper bounds on certain arithmetic terms and eliminates those quantified formulas based on these bounds. Specifically, rule B-Val splits the search into two cases $t \leqslant \mathsf{n}$ and $t > \mathsf{n}$, where t is an integer term and n is a numeral. In the rule B-Inst, if G contains a formula φ having a subformula $\forall k.\, 0 \leqslant k \leqslant t \Rightarrow \psi$ and A entails a (concrete) upper bound on t, then that subformula is replaced by a finite conjunction. Since all quantifiers introduced in a configuration are bounded, these two rules used in combination suffice to eliminate them.

Example 3. Consider the formula $\varphi = \mathsf{contains}(y,z) \wedge 0 < \mathsf{len}\, y \leqslant 3 \wedge 0 < \mathsf{len}\, z$. Applying Prop-Assign to φ results in a configuration where E, X and A respectively are $\{x \approx \top\}, \{x \approx \mathsf{contains}(y,z)\}$, and $\{0 < \mathsf{len}\, y \leqslant 3, 0 < \mathsf{len}\, z\}$. Using Ext-Expand, we remove $x \approx \top$ from X and add to G, $[\![x \approx \mathsf{contains}(y,z)]\!]$ which is:

$$(x \not\approx \top) \Leftrightarrow \forall k.\, 0 \leqslant k \leqslant \mathsf{len}\, y - \mathsf{len}\, z \Rightarrow \mathsf{substr}(y,k,\mathsf{len}\, z) \not\approx z.$$

[3] We use a number of optimizations of this encoding in our implementation.

Since $\mathsf{A} \models_{\mathsf{LIA}} \mathsf{len}\, y - \mathsf{len}\, z \leqslant 2$, by B-Inst we can replace this formula with:

$$((x \not\approx \top) \Leftrightarrow \wedge_{j=0}^{2} \mathsf{substr}(y, n_j, \mathsf{len}\, z) \not\approx z) \wedge \bigwedge_{j=0}^{2} n_j \approx j$$

where n_j is a fresh integer variable for $j = 0, 1, 2$. Applying Prop-Assign to this formula gives a branch where E and X are updated to $\{x \approx \top, m_0 \approx \mathsf{len}\, z, x_0 \approx z\}$ and $\{x_0 \approx \mathsf{substr}(y, n_0, m_0)\}$ for fresh variables x_0 and m_0. Using the rule Ext-Expand, we remove the equality from X and add $[\![x_0 \approx \mathsf{substr}(y, n_0, m_0)]\!]$ to G, which is:

$$\begin{aligned}&\mathsf{ite}(\ 0 \leqslant n_0 < \mathsf{len}\, y \wedge 0 \leqslant m_0,\\ &\quad y \approx \mathsf{con}(z_1, x_0, z_2) \wedge \mathsf{len}\, z_1 \approx 0 \wedge \mathsf{len}\, z_2 \approx \mathsf{len}\, y \dot{-} m_0,\ x_0 \approx \epsilon)\end{aligned}$$

with z_1, z_2 fresh string variables. Applying Prop-Assign again produces a branch with $\mathsf{E} = \{x \approx \top, x_0 \approx z, m_0 \approx \mathsf{len}\, z, y \approx \mathsf{con}(z_1, x_0, z_2)\}$ and X empty. The set E is satisfiable in T_{ASX}. Deriving a saturated configuration from this point indicates that φ is satisfiable in T_{ASX} as well. Indeed, all interpretations that satisfy both $y \approx \mathsf{con}(z_1, x_0, z_2)$ and $x_0 \approx z$ also satisfy $\mathsf{contains}(y, z)$. □

Although these rules give the general idea of our approach, our implementation actually handles the bounded quantifiers in a more sophisticated way, using model-based quantifier instantiation [23]. In a nutshell, we avoid generating all the instances of a quantified formula either by backtracking when a subset of them are unsatisfiable, or by determining that (sets of) instances are already satisfied by a candidate model.

3.2 Context-Dependent Simplification of Extended Function Terms

The reductions described above may be impractical due to the size and complexity of the formulas they introduce. For this reason, we have developed techniques for recognizing when the interpretation of an extended function term can be deduced based on the constraints in the current context. As a simple example, if $\mathsf{contains}(\mathsf{abc}, x)$ is a term belonging to the string theory (with abc a string constant), and the string solver has inferred that x is equal to a concrete value (e.g., d) or even a partially concrete value (e.g., $\mathsf{con}(\mathsf{d}, y)$), then we can already infer that $\mathsf{contains}(\mathsf{abc}, x)$ is equivalent to $\bot$, thereby avoiding the construction of its expanded form. We present next a generic technique for inferring such facts that has a substantial performance impact in practice.

The rule Ext-Simplify from Fig. 6 applies to configurations in which an extended function term t can be simplified to an equivalent form, modulo the current set of constraints, that does not involve extended functions. In this rule, we derive a set of equalities $\boldsymbol{y} \approx \boldsymbol{s}$ that are consequences of our current set of string constraints E, where typically $\boldsymbol{y}$ are the (free) variables of t. We will refer to $\{\boldsymbol{y} \mapsto \boldsymbol{s}\}$ as a *derivable substitution (in* E*)*. If the simplified form of t under this substitution is a Σ_{AS}-term, then we add the equality $x \approx (t\{\boldsymbol{y} \mapsto \boldsymbol{s}\})\downarrow$ to G, and remove $x \approx t$ from X. Similarly when t is of sort Bool, the rule Ext-Simplify-Pred adds an equivalence to G based on the result of simplifying the formula

$$\textsf{Ext-Simplify}\ \frac{x \approx t \in \mathsf{X} \qquad \mathsf{E} \models \boldsymbol{y} \approx \boldsymbol{s} \qquad (t\{\boldsymbol{y} \mapsto \boldsymbol{s}\})\downarrow \text{ is a } \Sigma_{\mathsf{AS}}\text{-term}}{\mathsf{G} := \mathsf{G}, \lfloor x \approx (t\{\boldsymbol{y} \mapsto \boldsymbol{s}\})\downarrow \rfloor \qquad \mathsf{X} := \mathsf{X} \backslash \{x \approx t\}}$$

$$\textsf{Ext-Simplify-Pred}\ \frac{x \approx t \in \mathsf{X} \qquad \mathsf{E} \models \boldsymbol{y} \approx \boldsymbol{s} \qquad t : \mathsf{Bool}}{\mathsf{G} := \mathsf{G}, \lfloor (x \approx \top) \Leftrightarrow ((t \approx \top)\{\boldsymbol{y} \mapsto \boldsymbol{s}\})\downarrow \rfloor \qquad \mathsf{X} := \mathsf{X} \backslash \{x \approx t\}}$$

$$\textsf{Ext-Eq}\ \frac{x_1 \approx t_1, x_2 \approx t_2 \in \mathsf{X} \qquad \mathsf{E} \models \boldsymbol{y} \approx \boldsymbol{s} \qquad (t_1\{\boldsymbol{y} \mapsto \boldsymbol{s}\})\downarrow = (t_2\{\boldsymbol{y} \mapsto \boldsymbol{s}\})\downarrow}{\mathsf{S} := \mathsf{S} \oplus \{x_1 \approx x_2\}}$$

Fig. 6. Rules for context-dependent simplification of extended functions terms.

$$\begin{array}{rll}
\mathsf{contains}(l_1, l_2) \to & \top & \text{if } l_1 \text{ contains } l_2 \\
\mathsf{contains}(l_1, l_2) \to & \bot & \text{if } l_1 \text{ does not contain } l_2 \\
\mathsf{contains}(l_1, \mathsf{con}(l_2, \boldsymbol{t})) \to & \bot & \text{if } l_1 \text{ does not contain } l_2 \\
\mathsf{contains}(l_1, \mathsf{con}(l_2, \boldsymbol{t})) \to & \bot & \text{if } \mathsf{contains}(l_1 \backslash l_2, \mathsf{con}(\boldsymbol{t})) \to^* \bot \\
\mathsf{contains}(l_1, \mathsf{con}(x, \boldsymbol{t})) \to & \bot & \text{if } \mathsf{contains}(l_1, \mathsf{con}(\boldsymbol{t})) \to^* \bot \\
\mathsf{contains}(\mathsf{con}(l_1, \boldsymbol{t}), l_2) \to & \top & \text{if } l_1 \text{ contains } l_2 \\
\mathsf{contains}(\mathsf{con}(x, \boldsymbol{t}), s) \to & \top & \text{if } \mathsf{contains}(\mathsf{con}(\boldsymbol{t}), s) \to^* \top \\
\mathsf{contains}(\mathsf{con}(t, \boldsymbol{s}), \mathsf{con}(t, \boldsymbol{u})) \to & \top & \text{if } \mathsf{contains}(\mathsf{con}(\boldsymbol{s}), \mathsf{con}(\boldsymbol{u})) \to^* \top \\
\mathsf{contains}(\mathsf{con}(l_1, \boldsymbol{t}), l_2) \to & \mathsf{contains}(\mathsf{con}(\boldsymbol{t}), l_2) & \text{if } l_1 \sqcup_l l_2 = \epsilon \\
\mathsf{contains}(\mathsf{con}(\boldsymbol{t}, l_1), l_2) \to & \mathsf{contains}(\mathsf{con}(\boldsymbol{t}), l_2) & \text{if } l_1 \sqcup_r l_2 = \epsilon
\end{array}$$

$$\begin{array}{rll}
\mathsf{contains}(\epsilon, t) \approx \top \to & \epsilon \approx t & \\
\mathsf{contains}(\mathsf{con}(\boldsymbol{t}_1, l_1, \boldsymbol{t}_2), l_2) \approx \top \to & \vee_{i=1}^{2} \mathsf{contains}(\mathsf{con}(\boldsymbol{t}_i), l_2) \approx \top & \text{if } l_1 \sqcup_r l_2 = l_1 \sqcup_l l_2 = \epsilon
\end{array}$$

Fig. 7. Examples of simplification rules for contains.

$t \approx \top$ under a derivable substitution, and removes $x \approx t$ from X. The rule Ext-Eq is used to deduce equalities between extended terms that are syntactically identical after simplification under a derivable substitution.

These rules require methods for computing the simplified form $t\downarrow$ of Σ_{ASX}-terms t, as well as for choosing substitutions $\{\boldsymbol{y} \mapsto \boldsymbol{s}\}$. We describe these methods in the following, and give several examples.

Simplification Rules for Extended String Functions. Recall that by construction, a term t and its simplified form $t\downarrow$ are equivalent in T_{ASX}. It is generally advantageous to use techniques that often simplify Σ_{ASX}-terms t to Σ_{AS}-terms $t\downarrow$, since this eliminates the need to apply Ext-Expand to compute the expanded form of t. For this reason, we use aggressive and non-trivial simplification techniques when considering Σ_{ASX}-terms.

Examples of some of the simplification rules for contains are given in Fig. 7. There, for string constants l_1, l_2, we write $l_1 \setminus l_2$ to denote the empty string if l_1 does not contain l_2, and the remainder obtained from removing the largest prefix of l_1 containing l_2 otherwise. We use $l_1 \sqcup_l l_2$ (resp. $l_1 \sqcup_r l_2$) to denote l_2 if l_1 contains l_2, and the largest suffix (resp. prefix) of l_1 that is a prefix (resp. suffix) of l_2 otherwise. For example, $(\mathsf{abcde} \setminus \mathsf{cd}) = \mathsf{e}$, $(\mathsf{abcde} \setminus \mathsf{ba}) = \epsilon$, $(\mathsf{abcde} \sqcup_l \mathsf{def}) = \mathsf{de}$, $(\mathsf{abcde} \sqcup_r \mathsf{def}) = \epsilon$, and $(\mathsf{abcdc} \sqcup_l \mathsf{cd}) = \mathsf{cd}$. Also, $s \to^* t$ indicates that t can

be obtained from s by zero or more applications of the rules in the figure. One can prove that his rewrite system is terminating by noting that all conditions and right hand side of each non-trivial rule involve only concatenation terms with strictly fewer arguments than its left hand side.

In practice, the rules are implemented by a handful of recursive passes over the arguments of contains terms. Computing the simplified form of other operators is also fairly sophisticated and not shown here. (Our simplifier is around 2000 lines of C++ code.) Despite its complexity, simplification often results in significant performance improvements, by eliminating the need to generate the expanded form of Σ_{ASX}-terms. We illustrate this in the following examples.

Example 4. Given input $y \approx \mathsf{bc} \wedge \mathsf{contains}(\mathsf{con}(\mathsf{a}, y), \mathsf{con}(\mathsf{b}, z, \mathsf{a}))$, our calculus considers a configuration where E and X respectively are

$$\{y \approx \mathsf{bc}, x_1 \approx \top, z_1 \approx \mathsf{con}(\mathsf{a}, y), z_2 \approx \mathsf{con}(\mathsf{b}, z, \mathsf{a})\} \text{ and } \{x_1 \approx \mathsf{contains}(z_1, z_2)\}$$

where x_1, z_1, z_2 are fresh string variables. We have that $\mathsf{E} \models z_1 \approx \mathsf{abc} \wedge z_2 \approx \mathsf{con}(\mathsf{b}, z, \mathsf{a})$. Hence the substitution $\sigma = \{z_1 \mapsto \mathsf{abc}, z_2 \mapsto \mathsf{con}(\mathsf{b}, z, \mathsf{a})\}$ is derivable in this configuration. Since $(\mathsf{contains}(z_1, z_2)\sigma)\downarrow = \mathsf{contains}(\mathsf{abc}, \mathsf{con}(\mathsf{b}, z, \mathsf{a}))\downarrow = \bot$, we may apply Ext-Simplify to remove $x_1 \approx \mathsf{contains}(z_1, z_2)$ from X and add $x_1 \approx \bot$ to E, after which unsat may be derived, since $x_1 \approx \top \in \mathsf{E}$. In this example, we have avoided expanding the input formula by reasoning that $\mathsf{con}(\mathsf{a}, y)$ does not contain $\mathsf{con}(\mathsf{b}, z, \mathsf{a})$ in the context where y is bc. □

Example 5. Given input $y \approx \mathsf{con}(\mathsf{a}, z) \wedge \mathsf{contains}(\mathsf{con}(x, y), \mathsf{bc})$, our calculus considers the configuration where E and X respectively are

$$\{y \approx \mathsf{con}(\mathsf{a}, z), x_1 \approx \top, z_1 \approx \mathsf{con}(x, y), z_2 \approx \mathsf{bc}\} \text{ and } \{x_1 \approx \mathsf{contains}(z_1, z_2)\}$$

The substitution $\sigma = \{z_1 \approx \mathsf{con}(x, \mathsf{a}, z), z_2 \mapsto \mathsf{bc}\}$ is derivable in this configuration. Computing $(\mathsf{contains}(z_1, z_2)\sigma)\downarrow$ results in $\mathsf{contains}(x, \mathsf{bc}) \vee \mathsf{contains}(z, \mathsf{bc})$. We may apply Ext-Simplify-Pred to remove $x_1 \approx \mathsf{contains}(z_1, z_2)$ from X and add this formula to G, after which we consider the two disjuncts independently. □

Example 6. Given input $y \approx \mathsf{ab} \wedge \mathsf{contains}(\mathsf{con}(\mathsf{b}, z), y) \wedge \neg\mathsf{contains}(z, y)$, our calculus considers the configuration where E and X respectively are

$$\{y \approx \mathsf{ab}, x_1 \approx \top, x_2 \approx \bot, z_1 \approx \mathsf{con}(\mathsf{b}, z)\} \text{ and } \{x_1 \approx \mathsf{contains}(z_1, y), x_2 \approx \mathsf{contains}(z, y)\}$$

The substitution $\sigma = \{y \approx \mathsf{ab}, z_1 \approx \mathsf{con}(\mathsf{b}, z)\}$ is derivable in this configuration, and $(\mathsf{contains}(z_1, y)\sigma)\downarrow = \mathsf{contains}(z, \mathsf{ab}) = (\mathsf{contains}(z, y)\sigma)\downarrow$. Hence we can apply Ext-Eq to add $x_1 \approx x_2$ to E, after which unsat can be derived. □

Choosing Substitutions. A simple and general heuristic for choosing substitutions $\{\boldsymbol{y} \mapsto \boldsymbol{s}\}$ for terms t in the rules from Fig. 6 is to map each variable y in t to some representative of its equivalence class $[y]$. We assume string constants are chosen as representatives whenever possible. We call this the *representative substitution* for t (in E). Representative substitutions are both easy to compute

and often enough for reducing Σ_{ASX}-terms. A more powerful method for choosing substitutions is to consider substitutions that map each free variable y in t to $\mathsf{con}(a_1,\ldots,a_n)\downarrow$ where $\mathsf{N}[y] = (a_1,\ldots,a_n)$. We call this the *normal form substitution* for t. Intuitively, the normal form of t is a schema representing all known information about t. In this sense, a substitution mapping variables to their normal forms gives the highest likelihood of enabling our simplification techniques. In practice, our implementation takes advantage of both of these heuristics for choosing substitutions.

Example 7. Say we are in a configuration where the equivalence classes of $\widehat{\mathsf{E}}$ are:

$$\{y, \mathsf{con}(\mathsf{a}, x)\} \quad \{x, \mathsf{con}(z, \mathsf{c}), u\} \quad \{z, w, \mathsf{b}\}$$

and the normal forms are $\mathsf{N}[y] = \mathsf{abc}, \mathsf{N}[x] = \mathsf{bc}$ and $\mathsf{N}[z] = \mathsf{b}$. If y, x, and b are chosen as the representatives of these classes, the representative substitution σ_r for this configuration is $\{y \mapsto y, x \mapsto x, u \mapsto x, z \mapsto \mathsf{b}, w \mapsto \mathsf{b}\}$, whereas the normal form substitution σ_n is $\{y \mapsto \mathsf{abc}, x \mapsto \mathsf{bc}, u \mapsto \mathsf{bc}, z \mapsto \mathsf{b}, w \mapsto \mathsf{b}\}$. Only the latter substitution suffices to show that $\mathsf{contains}(y, \mathsf{con}(z, z))$ is false in the current context, noting $(\mathsf{contains}(y, \mathsf{con}(z, z))\sigma_r)\downarrow = \mathsf{contains}(y, \mathsf{bb})$ and $(\mathsf{contains}(y, \mathsf{con}(z, z))\sigma_n)\downarrow = \bot$. □

4 Implementation

We have implemented all of these techniques in the DPLL(T)-based SMT solver CVC4 [4]. At a high level, our implementation can be summarized as a particular strategy for applying the rules of the calculus, which we outline in the following.

Strategy 1. *Start with a derivation tree consisting of (root) node* $\langle\{\varphi\}, \varnothing, \varnothing\rangle$. *Let* t_{len} *be* $\mathsf{len}\, x_1 + \ldots + \mathsf{len}\, x_m$ *where* $x_1, \ldots, x_m$ *are the string variables of* φ. *While the tree is not closed, consider as current configuration the left-most leaf in the tree that is not* unsat *and apply to it a derivation rule to that configuration, based on the steps below.*

1. *Let* n *be the smallest numeral such that* $t_{\mathsf{len}} > \mathsf{n} \notin \mathsf{A}$. *If* $t_{\mathsf{len}} \leqslant \mathsf{n} \notin \mathsf{A}$, *apply B-Val for* t_{len} *and* n.
2. *If* G *contains a formula* φ *with subformula* $\forall k.\, 0 \leqslant k \leqslant t \Rightarrow \psi$, *then let* n *be the smallest numeral such that* $t > \mathsf{n} \notin \mathsf{A}$. *If* $t \leqslant \mathsf{n} \in \mathsf{A}$, *apply B-Inst for* φ *and* n. *Otherwise, apply B-Val for* t *and* n.
3. *If possible, apply a rule from Fig. 3, giving priority to A-Conf and S-Conf.*
4. *If possible, apply a rule from Fig. 6 based on representative substitutions.*
5. *If possible, apply a rule from Fig. 4.*
6. *If possible, apply a rule from Fig. 6 based on normal form substitutions.*
7. *If* X *is non-empty, apply the rule Ext-Expand for some equality* $x \approx t$ *in* X.

If no rule applies and the current configuration is acyclic, return sat. *If the tree is closed, return* unsat. □

The strategy above is sound both for refutations and models, although it is not terminating in general.

Theorem 2. *For all initial configurations* $\langle\{\varphi\}, \varnothing, \varnothing\rangle$ *where* φ *is a quantifier-free* Σ_{ASX}*-formula:*

1. *Strategy 1 returns* unsat *only if* φ *is unsatisfiable in* T_{ASX}.
2. *Strategy 1 returns* sat *only if* φ *is satisfiable in* T_{ASX}.

Implementation. While a comprehensive description of our implementation is beyond the scope of this work, we mention a few salient implementation details. The rule Prop-Assign is implemented by converting G to clausal normal form and giving the resulting clauses to a SAT solver with support for conflict-driven clause learning. The rule A-Conf is implemented by a standard theory solver for linear integer arithmetic. The rules of the calculus that modify the S component of our configuration are implemented in a dedicated DPLL(T) *theory solver* for strings which generates *conflict clauses* when branches of a derivation tree are closed, and *theory lemmas* for rules that add formulas to G or A and those that have multiple conclusions. Conflict clauses are generated by tracking *explanations* so that each literal internally added to S can be justified in terms of input literals. Finally, we do not explicitly introduce fresh variables when constructing the set X, and instead record the set of extended terms that occur in E, which are implicitly treated as variables. We now revisit a few of the examples, giving concrete details on the operation of the solver.

Example 8. In Example 1, our input was $\mathsf{len}\, x > \mathsf{len}\, y \wedge y \approx \mathsf{con}(x, \mathsf{a})$. For this input, the SAT solver finds a propositionally satisfying assignment that assigns both conjuncts to true, which causes the literal $\mathsf{len}\, x > \mathsf{len}\, y$ to be given to the theory solver for linear integer arithmetic, and $y \approx \mathsf{con}(x, \mathsf{a})$ to be given to the theory solver for strings. This corresponds to an application of the rule Prop-Assign. The string solver sends $(\neg y \approx \mathsf{con}(x, \mathsf{a}) \vee \mathsf{len}\, y \approx \mathsf{len}\, x + 1)$ as a theory lemma to the SAT solver, corresponding to an application of the rule L-Eq. After that, the SAT solver assigns $\mathsf{len}\, y \approx \mathsf{len}\, x + 1$ to true, causing that literal to be asserted to the arithmetic solver, which subsequently generates a conflict clause of the form $(\neg \mathsf{len}\, x > \mathsf{len}\, y \vee \neg \mathsf{len}\, y \approx \mathsf{len}\, x + 1)$ corresponding to an application of A-Conf. After receiving this clause, the SAT solver is unable to find another satisfying assignment and causes the system to terminate with "unsat." □

Example 9. In Example 4, the string solver is given as input the literals $y \approx \mathsf{bc}$ and $\mathsf{contains}(\mathsf{con}(\mathsf{a}, y), \mathsf{con}(\mathsf{b}, z, \mathsf{a})) \approx \top$. The intermediate variables z_1 and z_2 are not explicitly introduced. Instead, using the substitution $\sigma = \{y \mapsto \mathsf{bc}\}$, the solver directly infers that $\mathsf{contains}(\mathsf{con}(\mathsf{a}, y), \mathsf{con}(\mathsf{b}, z, \mathsf{a}))\sigma\downarrow = \bot$. Based on this simplification, it infers $\mathsf{contains}(\mathsf{con}(\mathsf{a}, y), \mathsf{con}(\mathsf{b}, z, \mathsf{a})) \approx \bot$ with the explanation $y \approx \mathsf{bc}$. Since the inferred literal conflicts with the second input literal, the string solver reports the conflict clause $\neg y \approx \mathsf{bc} \vee \mathsf{contains}(\mathsf{con}(\mathsf{a}, y), \mathsf{con}(\mathsf{b}, z, \mathsf{a})) \approx \top$. □

Example 10. In Example 6, the equality $y \approx \mathsf{ab}$ is the explanation for the substitution $\sigma = \{y \mapsto \mathsf{ab}\}$ under which $\mathsf{contains}(\mathsf{con}(b, z), y)\sigma\downarrow = \mathsf{contains}(z, \mathsf{ab}) = \mathsf{contains}(\mathsf{con}(z, y))\sigma\downarrow$. Hence, the solver reports $(\neg y \approx \mathsf{ab} \vee \neg\mathsf{contains}(\mathsf{con}(\mathsf{b}, z), y) \vee \mathsf{contains}(z, y))$ as a conflict clause in this example. □

Example 11. Explanations are tracked for normal form substitutions as well. In Example 7, a possible explanation for the substitution σ_n is $y \approx \mathsf{con}(\mathsf{a}, x) \wedge x \approx \mathsf{con}(z, \mathsf{c}) \wedge u \approx x \wedge z \approx \mathsf{b} \wedge w \approx z$. Explanations for simplifications that occur under the substitution σ_n must include these equalities. □

In practice, we minimize explanations by only including the variables in substitutions that are relevant for certain inferences. In particular, the domain of derivable substitutions is restricted to the free variables of the terms they apply to. We further reduce this set based on dependency analysis. For example, $\mathsf{contains}(\mathsf{abc}, \mathsf{con}(x, y)) \approx \bot$ can be explained by $x \approx \mathsf{d} \wedge y \approx \mathsf{a}$. However, $x \approx \mathsf{d}$ alone is enough.

5 Evaluation

This section reports on our experimental evaluation of our approach for extended string constraints as implemented in the SMT solver CVC4.[4] We used benchmark queries generated by running PyEx, a symbolic executor for Python programs, over a test suite that mimics the usage by a real-world Python developer. The technical details of our benchmark generation process are provided in Sect. 5.2.

We considered several configurations of CVC4 that differ in the subset of steps from Strategy 1 they apply. The default configuration, denoted **cvc4**, performs Steps 2, 3, 5 and 7 only. Configurations with suffix **f** (for "finite model finding") perform Step 1, and configurations with suffix **s** (for "simplification") perform Steps 4 and 6. For example, configuration **cvc4+fs** performs all seven steps. We consider other solvers for string constraints, including Z3-STR [30] (git revision e398f81) and Z3 [9] (version 4.5, git revision 24eae3f) which was recently extended with native support for strings.

5.1 Comparison with Other String Solvers

We first evaluated the raw performance of CVC4, Z3, and Z3-STR on the string benchmarks we collected with PyEx. We considered three sets of benchmarks produced by PyEx using **cvc4+fs**, Z3, and Z3-STR as the path constraint solver during program exploration. We denote these sets, which collectively consisted of 25,421 benchmark problems, as **PyEx-cfs**, **PyEx-z3** and **PyEx-z32**, respectively. We omit a small number (35) of these benchmarks for the following reasons: for 13 of them, at least one of the solvers produced a parse error; for the other 22, one solver returned a model (i.e., a satisfying assignment for the variables in the input problem) and agreed with its own model, but another solver answered "unsat" and disagreed with the model of the first solver.[5] We attribute the parse errors to how the solvers process certain escape sequences in string constants, and the model discrepancies to minor differences in the semantics of substr when input indices are out of bounds.

[4] For details, see http://cvc4.cs.stanford.edu/papers/CAV2017-strings/.

[5] We say solver A *(dis)agrees* with solver B's model for input formula φ if A finds that $\varphi \wedge \mathcal{M}_B$ is (un)sat, where $\mathcal{M}_B$ is a conjunction of equalities encoding B's.

	PyEx-cfs (5557)			**PyEx-z3** (8399)			**PyEx-z32** (11430)			**Total** (25386)			
Solver	sat	unsat	×	sat	unsat	×	sat	unsat	×	sat	time	unsat	time
cvc4+fs	**4229**	1256	72	**5694**	1325	1380	**10104**	1194	132	**20027**	5h1m	3775	7m
cvc4+s	4133	1270	154	5461	1325	1613	9884	1193	353	19478	8h40m	3788	6m
cvc4+f	4160	1217	180	5571	1308	1520	9210	1145	1075	18941	6h38m	3670	6m
cvc4	4160	1213	184	5570	1308	1521	9211	1145	1074	18941	6h32m	3666	6m
z3	3421	1274	862	4925	**1333**	2141	7219	**1196**	3015	15565	11h30m	**3803**	3m
z3str2	2013	**1278**	2266	2803	1333	4263	4726	1182	5522	9542	15h47m	3793	13m

Fig. 8. Results of running each solver over benchmarks generated by PyEx over our test suite. All benchmarks run with a 30 s timeout.

All results were produced on StarExec [25], a public execution service for running comparative evaluations of logical solvers. The results for the three solvers on the three benchmark sets are shown in Fig. 8 based on a 30 s timeout. The columns show the number of benchmarks that were determined to be satisfiable and unsatisfiable by each solver. The column with heading × indicates the number of times the solver either timed out or terminated with an inconclusive response such as "unknown." The best configuration of CVC4 (**cvc4+fs**) had a factor of 3.8 fewer timeouts than Z3, and a factor of 7.6 fewer timeouts or failures than Z3-STR in total over all benchmark sets. In particular, we note that **cvc4+fs** solved 1,451 unique benchmarks with respect to Z3 among those generated during a symbolic execution run using Z3 as the solver (**PyEx-z3**). Since PyEx supports concurrent solver invocation, this suggests a mixed-solver strategy that employs both **cvc+fs** and Z3 would likely have reduced the number of failed queries for that run. For unsatisfiable benchmarks, the solvers were relatively closer in performance, where Z3 solved 24 more unsatisfiable benchmarks than **cvc4+fs**, which is not tuned for the unsatisfiable case due to its use of finite model finding. This further suggests a mixed-solver strategy would likely be beneficial for symbolic execution since it is often used for both program exploration (where sat leads to progress) and vulnerability checking (where unsat implies safety).

In addition to solving more benchmarks, **cvc4+fs** was significantly faster over them. Figure 9 plots the cumulative run time of the three solvers on benchmarks that each solves. With respect to Z3, which took 11 h and 33 min on the 19,368 benchmarks its solves, **cvc4+fs** solved its first 19,368 benchmarks in 1 h and 23 min, and overall took only 5 h and 8 min on the 23,802 benchmarks it solves.

Using the context-dependent simplification techniques from Sect. 3.2, **cvc4+s** was able to solve 663 more benchmarks than **cvc4**, which does not apply simplification. By incorporating finite model finding, **cvc4+fs** was able to solve 536 more benchmarks in significantly less time, with a cumulative difference of more than 3 h on solved benchmarks compared to **cvc4+s**. Taking the virtual best configuration of CVC4, our techniques find 20,594 benchmarks to be satisfiable, 567 more than **cvc4+fs**, indicating that a portfolio approach for the various configurations would be advantageous.

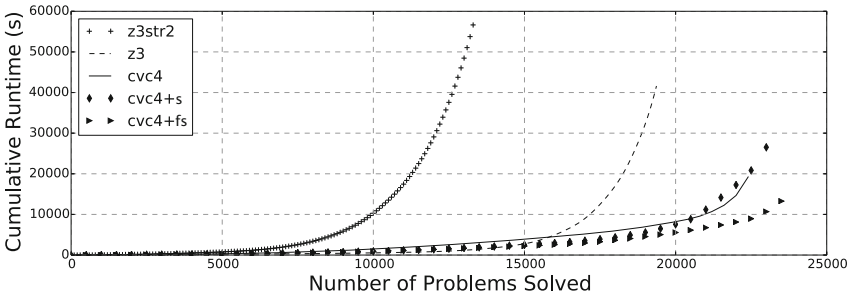

Fig. 9. Cactus plot of configurations of CVC4, Z3 and Z3-STR on solved benchmarks across all three benchmark sets.

We also measured how often CVC4 resorted to expanding extended function terms. We considered a modified configuration **cvc4+fs'** that is identical to **cvc4+fs** except that it does not use the rule Ext-Simplify. The 23,738 benchmarks solved by both **cvc4+fs** and **cvc4+fs'** had 619.2 extended function terms on average. On average over these benchmarks, the configuration **cvc4+fs'** found that 63.5 unique extended functions terms were relevant to satisfiability (e.g., were added to a configuration), and of these 24.3 were expanded (38%). Likewise, **cvc4+fs** found that 66.4 unique extended functions terms were relevant to satisfiability, and of these 12.6 were expanded (19%). With Ext-Simplify, it inferred 405.2 equalities per benchmark on average based on context-dependent simplification, showing that simplification is possible in a majority of contexts (97%). Limited to expansions that introduce universal quantification, which excludes expansions of substr and positively asserted contains, **cvc4+fs'** considered 11.4 expansions on average compared to 2.7 considered by **cvc4+fs**. This means that approximately 4 times fewer quantified formulas were introduced thanks to context-dependent simplification in **cvc4+fs**.

5.2 Symbolic Execution for Python

Our benchmarks were generated by PyEx, which is a symbolic executor designed to assist Python developers achieve high-coverage testing in their daily development routine, e.g., as part of the nightly tests run on the most recent version of a code base. To demonstrate the relative performance of CVC4, Z3, and Z3-STR in our nightly tests scenario, we ran PyEx on a test suite of 19 functions sampled from 4 popular Python packages: httplib2, pip, pymongo, and requests. The set of queries generated during this experiment was used for our evaluation of the raw solver performance in Sect. 5.1. In this section, we show that the superior raw performance of CVC4 over other current solvers also translates into *real-life benefits* for PyEx users.

Our experiment was conducted on a developer machine featuring an Intel E3-1275 v3 quad-core processor (3.5 GHz) and 32 GB of memory. PyEx was run on each of the 19 functions for a maximum CPU time of 2 h. By design, PyEx

issues concurrent queries using Python multiprocessing when multiple queries are pending. To reflect the configuration of our test machine, we capped the number of concurrent processes to 8. Note that, due to the nature of our test infrastructure, each of the 19 functions was tested in sequence and thus concurrency happened only within the testing of each individual function. In addition, PyEx has a notion of a per-path timeout, which is a heuristic to steer code exploration away from code paths that are stuck with hard queries. For this experiment, that timeout was set to 10 min.

We argue that the most important metrics for a developer are (*i*) the wall-clock time to run PyEx over the test suite and (*ii*) the coverage achieved over this time. In our experiments, PyEx with Z3 and with Z3-STR finished in 717 min and 829 min, respectively. By comparison, PyEx with the recommended configuration of CVC4 (**cvc4+fs**) finished in 295 min, which represents a speedup of 59% and 64% respectively over the other solvers. To compare coverage, we used the Python coverage library to measure both *line coverage*, the percentage of executed source lines, and *branch coverage*, the number of witnessed branch outcomes, of the test suite during symbolic execution. The line coverage of PyEx with **cvc4+fs**, Z3, and Z3-STR was respectively 8.48%, 8.41%, and 8.34%,[6] whereas the branch coverage was 3612, 3895, and 3500. Taking both metrics into account, we conclude that **cvc4+fs** is highly effective for PyEx since it achieves similar coverage as the other tools while running significantly faster.

6 Concluding Remarks and Future Work

We have presented a calculus for extended string constraints that relies on both bounded quantifier elimination and context-dependent simplification techniques. The latter led to significant performance benefits for constraints coming from symbolic execution of Python programs. An implementation of these techniques in CVC4 has 3.8 times fewer timeouts and enables the PyEx symbolic executor to achieve comparable program coverage on our test suite while using only 41% of the runtime compared to other solvers.

Our analysis on program coverage indicates that an interesting research avenue for future work would be to determine correlations between certain features of models generated by string solvers and their utility in a symbolic executor, since different models may lead to different symbolic executions and hence different overall analyses.

We plan to develop context-dependent simplification techniques for other string functions, including conversion functions str_to_int and int_to_str, and to adapt these techniques to other SMT theories. Notably, we would like to use context-dependent simplification to optimize lazy approaches for fixed-width bit-vectors [14] where it is beneficial to avoid bit-blasting bit-vector operators, such as multiplication, that require elaborate encodings.

[6] Overall coverage appears to be low because we tested only some functions from each library.

Acknowledgments. This work was supported in part by the National Science Foundation under grants CNS-1228765, CNS-1228768, and CNS-1228827. We express our immense gratitude to Peter Chapman, who served as the first lead developer of PyEx.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_10
2. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Cham (2015). doi:10.1007/978-3-319-21690-4_29
3. Ball, T., Daniel, J.: Deconstructing dynamic symbolic execution. In: Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering. IOS Press (2014)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_14
5. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00768-2_27
6. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation, pp. 209–224. USENIX (2008)
7. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing Mayhem on binary code. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy, pp. 380–394. IEEE (2012)
8. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: a platform for in-vivo multi-path analysis of software systems. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 265–278. ACM (2011)
9. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
10. Fu, X., Li, C.: 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)
11. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 209–226. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39611-3_21
12. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27813-9_14

13. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium. Internet Society (2008)
14. Hadarean, L., Bansal, K., Jovanović, D., Barrett, C., Tinelli, C.: A tale of two solvers: eager and lazy approaches to bit-vectors. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 680–695. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_45
15. Hooimeijer, P., Veanes, M.: An evaluation of automata algorithms for string analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011). doi:10.1007/978-3-642-18275-4_18
16. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 105–116. ACM (2009)
17. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
18. Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 15–31. Springer, Cham (2013). doi:10.1007/978-3-319-03077-7_2
19. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_43
20. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: Lutz, C., Ranise, S. (eds.) FroCoS 2015. LNCS, vol. 9322, pp. 135–150. Springer, Cham (2015). doi:10.1007/978-3-319-24246-0_9
21. Makanin, G.S.: The problem of solvability of equations in a free semigroup. English transl. in Math USSR Sbornik **32**, 147–236 (1977)
22. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3), 483–496 (2004)
23. Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 377–391. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38574-2_26
24. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium (2016)
25. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 367–373. Springer, Cham (2014). doi:10.1007/978-3-319-08587-6_28
26. Trinh, M.-T., Chu, D.-H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 218–240. Springer, Cham (2016). doi:10.1007/978-3-319-41528-4_12
27. Trinh, M.-T., Chu, D.-H., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: Yung, M., Li, N. (eds.) Proceedings of the 21st ACM Conference on Computer and Communications Security (2014)
28. Veanes, M., Bjørner, N., Moura, L.: Symbolic automata constraint solving. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6397, pp. 640–654. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16242-8_45

29. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 235–254. Springer, Cham (2015). doi:10.1007/978-3-319-21690-4_14
30. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 114–124. ACM (2013)