Even Faster Conflicts and Lazier Reductions for String Solvers

Andres Nötzli[1(✉)], Andrew Reynolds[2], Haniel Barbosa[3], Clark Barrett[1], and Cesare Tinelli[2]

[1] Department of Computer Science, Stanford University, Stanford, USA
noetzli@cs.standford.edu

[2] Department of Computer Science, The University of Iowa, Iowa City, USA

[3] Universidade Federal de Minas Gerais, Belo Horizonte, Brasil

Abstract. In the past decade, satisfiability modulo theories (SMT) solvers have been extended to support the theory of strings and regular expressions. This theory has proven to be useful in a wide range of applications in academia and industry. To accommodate the expressive nature of string constraints used in those applications, string solvers use a multi-layered architecture where extended operators are reduced to a set of core operators. These reductions, however, are often costly to reason about. In this work, we propose new techniques for eagerly discovering conflicts based on equality reasoning and lazily avoiding reductions for certain extended functions based on lightweight reasoning. We present a strategy for integrating and scheduling these techniques in a CDCL(T)-based theory solver for strings and regular expressions. We implement the techniques and the strategy in CVC5, a state-of-the-art SMT solver, and show that they lead to a significant performance improvement.

1 Introduction

Most software processes strings and, as a result, modern programming languages integrate rich functionality to represent and manipulate strings. The semantics of string-manipulating functions are often complex, which makes reasoning about them challenging. In recent years, researchers have proposed various approaches to tackle this challenge with dedicated solvers for string constraints [3,5,11,19,21], often as extensions of satisfiability modulo theories (SMT) solvers [10]. Dedicated solvers have been successfully used in a wide range of applications, including: finding or proving the absence of SQL injections and XSS vulnerabilities in web applications [30,32,35]; reasoning about access policies in cloud infrastructure [6,7,13]; and generating database tables from SQL queries for unit testing [34].

SMT solvers are frequently used as back ends for formal tools that reason about software or hardware. These tools typically produce a mix of easy and hard proof obligations that must be discharged by the solver. For many applications,

This work was supported by a gift from Amazon Web Services.

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13372, pp. 205–226, 2022.
https://doi.org/10.1007/978-3-031-13188-2_11

it is crucial that the SMT solver responds quickly, and modern solvers are finely tuned to deliver the required performance. String solvers often stratify reasoning about constraints by combining different reasoning techniques rather than relying on a single, monolithic procedure. Specifically, it is common for a string solver to have a core procedure that processes only a basic language of string constraints with a minimal set of string operators. *Extended constraints*, containing additional operators, are supported by applying transformations that reduce them to combinations of basic constraints. Optimizations to this design have been explored in previous work, e.g., by simplifying extended string constraints based on the current *context* (i.e., the current set of asserted constraints) [29]. However, existing techniques still sometimes fall short for industrial applications, which continue to require richer languages of constraints while expecting the underlying solvers to remain efficient. To meet these needs, string solvers must have an even greater understanding of extended constraints and be equipped with fast procedures that leverage this knowledge.

In this work, we focus on CDCL(T)-based SMT solvers [26], where solving is done through the cooperation of a SAT solver and one or more theory solvers. The SAT solver is responsible for finding truth assignments M that satisfy the Boolean abstraction of the input formula, and the theory solvers are responsible for returning *conflict clauses* (disjunctions of literals that are valid in the theory T but are falsified by M) and, optionally, *lemmas* (selected clauses that are valid in T). The conflict clauses and lemmas from theory solvers are then added to the original input formula, and the process of finding a satisfying assignment M is repeated until no conflicts are detected, indicating that the input formula is satisfiable in T, or an unrecoverable conflict is derived, indicating that the input is unsatisfiable in T. Theory reasoning done while the SAT solver is constructing the assignment M is characterized as *eager*. Theory reasoning done after a full assignment has been computed is called *lazy*.

Inspired by real-world benchmarks, we propose new techniques for string solvers that make them more eager, and hence *faster*, in their discovery of conflicts and *lazier* in reducing constraints that are hard to handle such as, for instance, negated regular expression membership constraints. For the former, we extend the congruence closure [24] module at the heart of the string solver to perform selected theory-specific forms of reasoning including eager evaluation, reasoning based on inferred prefixes and suffixes, and (integer) arithmetic approximations (Sect. 3). For the latter, we introduce several new techniques for avoiding reductions involving extended string operators (Sects. 4 and 5). This set of techniques is particularly useful for satisfiable benchmarks, where it is possible to determine that a (candidate) model indeed satisfies the input formula without having to fully process extended constraints. We have designed these techniques to be compatible with most existing solving techniques for strings. In Sect. 6, we propose an extended strategy that describes the integration of the new techniques within an existing string solver.

In summary, our contributions are as follows:

– We describe new techniques for eagerly detecting conflicts based on an enriched congruence closure procedure for the theory of strings.

- We describe a strategy for *model-based reductions*, which can be used to minimize the reductions considered during string solving.
- We describe a procedure for efficiently reasoning about inclusion relationships for a common fragment of regular membership constraints. This procedure is used both for detecting conflicts and for avoiding unfoldings of regular expressions.
- We evaluate an implementation of the new techniques in CVC5 [8], an open source state-of-the-art SMT solver, on a wide range of string benchmarks and show a significant improvement in overall performance.

1.1 Related Work

As mentioned above, string solvers typically reduce the input constraints to a basic form. Common basic representations include finite automata [14,17,18,31,33], bit-vectors [19], arrays [20], variations of word equations and length constraints [12,29,32,36], and hybrid approaches that combine word equations and bit-vector representations [23]. Our techniques for lazier reductions are primarily targeted at reductions to word equations, but our other techniques are more broadly applicable and could be used with any of the other basic representations.

In general, the theory of strings is undecidable [12], but modern solvers integrate a wide range of techniques to solve problems that appear in practice. One line of work has been exploring techniques that avoid reductions or make them more efficient. Reynolds et al. [29] describe an approach for lazily performing reductions after simplifying extended functions based on other constraints in the current context. In later work, Reynolds et al. [27] propose the use of aggressive rewriting to eliminate or simplify extended string constraints before performing reductions. In this work, we propose techniques that can be combined with that earlier work to perform reductions even more lazily. Reynolds et al. [28] also proposed a technique for improving the efficiency of reductions by introducing fewer fresh variables. Our approach is orthogonal to this work, because it further avoids reductions, but cannot avoid them entirely.

Both Reynolds et al. [28] and Backes et al. [7] reduce a fragment of regular expression constraints to extended string constraints. In contrast, our approach avoids reductions of certain regular membership constraints.

2 Preliminaries

We work in many-sorted first-order logic with equality and assume the reader is familiar with the notions of signature, term, literal, (quantified) formula, and free variable (see, e.g., [16]). We consider many-sorted signatures Σ, each containing a family of logical symbols $\approx$ for equality and interpreted as the identity relation, with input sort $\sigma \times \sigma$ for all sorts σ in Σ. A Σ-interpretation is a Σ-structure that additionally assigns a value to each variable. A *theory* is a pair $T = (\Sigma, \mathbf{I})$, in which 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

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

$\mathsf{substr} : \mathsf{Str} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{Str}$ $\quad \mathsf{ctn} : \mathsf{Str} \times \mathsf{Str} \to \mathsf{Bool}$
$\mathsf{indexof} : \mathsf{Str} \times \mathsf{Str} \times \mathsf{Int} \to \mathsf{Int}$ $\quad \mathsf{replace} : \mathsf{Str} \times \mathsf{Str} \times \mathsf{Str} \to \mathsf{Str}$

$\_ \in \_ : \mathsf{Str} \times \mathsf{Lan} \to \mathsf{Bool}$ $\quad \Sigma : \mathsf{Lan}$
$\mathsf{rcon} : \mathsf{Lan} \times \cdots \times \mathsf{Lan} \to \mathsf{Lan}$ $\quad \mathsf{re} : \mathsf{Str} \to \mathsf{Lan}$
$\mathsf{inter} : \mathsf{Lan} \times \cdots \times \mathsf{Lan} \to \mathsf{Lan}$ $\quad \_^* : \mathsf{Lan} \to \mathsf{Lan}$
$\mathsf{union} : \mathsf{Lan} \times \cdots \times \mathsf{Lan} \to \mathsf{Lan}$ $\quad \mathsf{range}_{c_1,c_2} : \mathsf{Lan}$

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

some (resp., no) interpretation in $\mathbf{I}$. 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 Fig. 1. We fix a totally ordered finite alphabet $\mathcal{A}$ of characters. The signature includes the sorts Str, Lan, Int, and Bool, denoting $\mathcal{A}^*$, regular languages over $\mathcal{A}$, integers, and Booleans respectively. The *core* signature is given on the first two lines. 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 second line, and include a constant symbol, or *string constant*, for each word of $\mathcal{A}^*$ 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. In our examples, we will take a $\mathcal{A}$ to be the set of ASCII characters and denote string constants by double-quote-delimited string literals (as in `"abc"`).

The four function symbols in the next two lines of Fig. 1 encode operations on strings that often occur in applications: a substring operator, a string containment predicate, an operation to find the position of one string in another, and one to replace a substring with another. We refer to these function symbols as *extended functions*. For details on the semantics of these operators, see for example [29].

The remainder of the signature covers regular expressions. It includes an infix binary predicate symbol $\_ \in \_ : \mathsf{Str} \times \mathsf{Lan} \to \mathsf{Bool}$, which denotes word membership in a given regular language. The remaining symbols are used to construct regular expressions. In particular, Σ denotes (the language of) all strings of length one; $\mathsf{re}(s)$ denotes the singleton language containing just the word denoted by s; $\mathsf{rcon}(R_1, \cdots, R_n)$ denotes all strings that are a concatenation of strings denoted by the arguments; the Kleene star operator R^* denotes all strings that are obtained as the concatenation of zero or more repetitions of the strings denoted by R; $\mathsf{inter}(R_1, \cdots, R_n)$ denotes the intersection of the languages denoted its arguments; and $\mathsf{union}(R_1, \cdots, R_n)$ denotes the union of the languages denoted by its arguments. Finally, we include the class of all indexed regular expression symbols of the form range_{c_1,c_2} where c_1 and c_2 are string constants of length one. We call this a *regular expression range* and interpret it

as the language containing all strings of length one that are between c_1 and c_2 (inclusive) in the ordering associated with $\mathcal{A}$.

3 Eager Equality-Based Conflicts for Strings

We consider theory solvers for strings like those described by Liang et al. [21], which have at their core a congruence closure algorithm that determines whether a set of string constraints S is satisfiable in the empty theory (i.e., all function symbols, including string operations, are treated as uninterpreted). In this section, we describe two enhancements to such congruence closure algorithms, which can help detect theory-inconsistencies in S. We stress that our extended congruence closure is computed eagerly and *incrementally* as the SAT solver assigns truth values to string equalities. This enables the enhanced congruence closure algorithm to detect theory inconsistencies early, when the truth assignment is still only partially specified. We elaborate on how this enables eager backtracking in Sect. 6.

3.1 Enhancing Congruence Closure with Evaluation

The string solver implements a procedure to compute the congruence closure $\mathcal{C}(\mathsf{S})$ over the set S of currently asserted string equalities. Let $\mathcal{T}(\mathsf{S})$ be the set of all terms and subterms in S. Formally, $\mathcal{C}(\mathsf{S})$ is the set of all equalities between terms in $\mathcal{T}(\mathsf{S})$ that are entailed by the empty theory:

$$\mathcal{C}(\mathsf{S}) = \{s \approx t \mid s, t \in \mathcal{T}(\mathsf{S}), \mathsf{S} \models s \approx t\}$$

The output of the procedure that computes $\mathcal{C}(\mathsf{S})$ can be represented as a set of *equivalence classes*, that is, a partition of $\mathcal{T}(\mathsf{S})$ where each block of the partition is a maximal set of equivalent terms. For each equivalence class, we designate a unique term in it as the *representative* for that class; if the class contains at least one constant term, then the representative must be one of them. We will denote by $[t]$ the equivalence class of a term t induced by $\mathcal{C}(\mathsf{S})$. By a slight abuse of notation we will use $[t]$ also to denote the representative of that class.

Computing the congruence closure $\mathcal{C}(\mathsf{S})$ allows the string solver to detect theory conflicts in the current context which occur when the context contains a disequality $s \not\approx t$, where $[s] = [t]$. It also allows the string solver to propagate to the SAT solver entailed equalities that occur in the input formula but have not been explicitly asserted yet.

By default, congruence closure procedures effectively treat theory symbols as uninterpreted functions. Here, we propose a lightweight approach for injecting some theory-specific reasoning by *evaluating* string terms whenever possible. Specifically, for every term that is a function application $f(t_1, \ldots, t_n)$, where f is a string theory symbol, if the representatives $[t_1], \ldots, [t_n]$ are all constants, the enhanced congruence closure procedure adds the equality $f(t_1, \ldots, t_n) \approx f([t_1], \ldots, [t_n])\downarrow$ to $\mathcal{C}(\mathsf{S})$, where $f([t_1], \ldots, [t_n])\downarrow$ is the constant resulting from

the evaluation of $f([t_1], \ldots, [t_n])$. Adding these equalities improves the ability of the congruence closure layer to detect more theory conflicts and propagations, as illustrated in the following example.

Example 1. Consider the constraints $\{y \approx \texttt{"b"}, z \approx \mathsf{replace}(x, y, \texttt{"d"}), x \approx z, x \approx \texttt{"abc"}\}$, where the term $\mathsf{replace}(x, y, \texttt{"d"})$ denotes the result of replacing the first occurrence of y in x by `"d"` if one exists. The congruence closure for this set of constraints determines the following equivalence classes, each with a constant representative:

$$\{\texttt{"b"}, y\}, \quad \{\texttt{"d"}\}, \quad \{\texttt{"abc"}, x, z, \mathsf{replace}(x, y, \texttt{"d"})\} \ .$$

This means that the term $\mathsf{replace}(x, y, \texttt{"d"})$ is equivalent to the concrete term $\mathsf{replace}(\texttt{"abc"}, \texttt{"b"}, \texttt{"d"})$. Evaluating the latter results in the constant `"adc"`. Hence, the congruence closure procedure will add the equality $\mathsf{replace}(x, y, \texttt{"d"}) \approx \texttt{"adc"}$ to its input set of equalities and recompute the congruence closure. This will cause the third equivalence class in the list above to contain the (distinct) string constants `"abc"` and `"adc"`, thus resulting in a conflict.

In our implementation, we must track explanations for inferred equalities for the purposes of reporting conflict clauses. In the above example, the equality $\mathsf{replace}(x, y, \texttt{"d"}) \approx \texttt{"adc"}$ is added to the congruence with the explanation $x \approx \texttt{"abc"} \wedge y \approx \texttt{"b"}$, which is then used in the standard technique for constructing explanations for congruence-closure-based reasoning [25].

We remark that enhancing congruence closure with evaluation is not specific to the theory of strings, and can be leveraged by other theory solvers based on congruence closure. Further exploration of this technique and its impact on other theories is left as future work.

3.2 Tracking Properties of Equivalence Classes

In addition to the use of evaluation, we enhance our congruence closure procedure with further information that can be used to discover conflicts eagerly based on string-specific reasoning. We describe two examples of this mechanism below.

First, we maintain a mapping $\mathcal{Z}$ from integer equivalence classes e to intervals of the form $[\ell, u]$, indicating concrete lower and upper bounds on the value that the terms in e can have. Open intervals are achieved by letting ℓ and u be $-\infty$ and ∞ respectively. The interval can be inferred using string-specific reasoning over the terms in e.

Second, we maintain a mapping $\mathcal{S}$ from string equivalence classes e to a pair of string constants (l_1, l_2) denoting the maximal known prefix l_1 and suffix l_2 of the value that the terms in e can have. For example, if e contains the term $\texttt{"abc"} \cdot x$ then l_1 for e is, at least, `"abc"`. When no prefix is known, l_1 is the empty string. The suffix l_2 is handled similarly.

Figure 2 shows how the maps $\mathcal{Z}$ and $\mathcal{S}$ are updated when new equivalence classes are created (newEqc) and when equivalence classes are merged

$\mathsf{newEqc}(t)$:

$$t : \mathsf{Int} \quad \mathcal{Z}\,[t] := \begin{cases} [n,n] & \text{if } t = n \\ [\ell_{|s|}, u_{|s|}] & \text{if } t = |s| \\ [-\infty, \infty] & \text{otherwise} \end{cases}$$

$$t : \mathsf{Str} \quad \mathcal{S}\,[t] := \begin{cases} (t,t) & \text{if } t \text{ is a constant} \\ (l_1, l_2) & \text{if } t \text{ reduces to } l_1 \cdot t' \cdot l_2 \text{ with } l_1, l_2 \text{ constants} \end{cases}$$

$\mathsf{mergeEqc}([t_1], [t_2])$:

$t_1, t_2 : \mathsf{Bool}$ — if $(t_1, t_2) = (\top, x \in R)$ where $R = \mathsf{rcon}(\mathsf{re}(l_1), R', \mathsf{re}(l_2))$ then
$\mathsf{mergeEntry}(\mathcal{Z}\,[x], [\ell_{|R|}, u_{|R|}])$
$\mathsf{mergeEntry}(\mathcal{S}\,[x], (l_1, l_2))$

$t_1, t_2 : \mathsf{Int}$ — $\mathsf{mergeEntry}(\mathcal{Z}\,[t_1], \mathcal{Z}\,[t_2])$

$t_1, t_2 : \mathsf{Str}$ — $\mathsf{mergeEntry}(\mathcal{S}\,[t_1], \mathcal{S}\,[t_2])$

$\mathsf{mergeEntry}(E_1, E_2)$:

$E_1, E_2 = [\ell_1, u_1], [\ell_2, u_2]$ — if $\ell_1 > u_2$ or $\ell_2 > u_1$ then CONFLICT
else $E_1 := [\mathsf{max}(\ell_1, \ell_2), \mathsf{min}(u_1, u_2)]$

$E_1, E_2 = (p_1, s_1), (p_2, s_2)$ — if $p_1 \not\prec_{pre} p_2$ or $s_1 \not\prec_{suf} s_2$ then CONFLICT
else $E_1 := (\mathsf{max}_{|\_|}(p_1, p_2), \mathsf{max}_{|\_|}(s_1, s_2))$

Fig. 2. Methods for tracking intervals, prefixes, and suffixes for equivalence classes.

(mergeEqc), the two basic methods that are used when computing congruence closures. For the second method, a helper method (mergeEntry) is used to combine the contents of the entries in two maps. We assume without loss of generality that when mergeEqc is called on equivalence classes $([t_1], [t_2])$, $[t_1]$ becomes the new representative for the merged class.

We now look at these methods in more detail. When a new equivalence class for term t is created, we look at the type of t. If t has integer type, there are three cases. If t is a numeral n, it is mapped to the interval $[n, n]$. If t is a length term of the form $|s|$, then we compute an interval $[\ell_{|s|}, u_{|s|}]$ where $\ell_{|s|}$ (resp., $u_{|s|}$) is a sound under-approximation (resp., over-approximation) of the length of s. We use the procedure described by Reynolds et al. [27] to compute these approximations. We use it because it is available, well-tested, and designed to be fast, but any sound approximation could be used. Otherwise, t is mapped to the open interval $[-\infty, \infty]$. If t has string type, we consider two cases. If t is a string constant, its prefix and suffix are both set to t. If t can be normalized using a simple set of rewrite rules to a concatenation term of the form $l_1 \cdot t' \cdot l_2$, where l_1 and l_2 are string constants of maximal length and t' is a non-constant term,

then t is mapped to the pair (l_1, l_2). Note that the notation $l_1 \cdot t' \cdot l_2$ is meant to include the case where either l_1 or l_2 (or both) is the empty string.[1]

When two equivalence classes $[t_1]$ and $[t_2]$ are merged, first, if $[t_1]$ is $\top$ and $[t_2]$ is a regular expression membership predicate $x \in R$, then we may infer information about x, because $x \in R$ is now known to be true in the current context. We compute upper and lower bounds $[\ell_{|R|}, u_{|R|}]$ on the length of all strings that occur in R. We use fast approximate techniques for computing these bounds (e.g., sum the length of constant components of concatenations to infer lower bounds). Note that these techniques are context-independent and are solely based on the structure of R. We update the entry $\mathcal{Z}[x]$ based on this information. Similarly, we update the entry $\mathcal{S}[x]$ with information about the constant prefix and suffix of the regular expression R. On the other hand, when $[t_1]$ and $[t_2]$ are integer or string equivalence classes, we merge the entries for the appropriate mapping. We stress that the entry for $[t_1]$ is updated with the information from the entry for $[t_2]$ and not vice versa. This is because $[t_1]$ is the new representative of the merged equivalence class, and further merges may refer to it, while $[t_2]$ is subsequently unused.

When merging entries, we may determine that the constraints represented by the two entries are inconsistent, in which case we have found a conflict. For example, when merging integer equivalence classes, if the lower bound for one equivalence class is greater than the upper bound for the other, we raise a conflict. For string equivalence classes, a conflict is raised if the prefixes for the two equivalence classes are incompatible (i.e., neither is a prefix of the other) and similarly for suffixes. We write $p_1 \not\sim_{pre} p_2$ (resp., $s_1 \not\sim_{suf} s_2$) to denote that p_1 is not a prefix of p_2 or vice versa (resp., s_1 is not a suffix of s_2 or vice versa), and $\mathsf{max}_{|\text{-}|}$ to denote the function returning the string constant having maximum length. If no conflict is raised, then the new entry E_1 is updated to contain the merged information: for integers, we take the maximal lower bound and minimal upper bound; and for strings, we take the prefix or suffix of maximal length.

In the context of CDCL(T), when the procedure raises a conflict, it is required to return a *conflict clause*, which in turn will cause the solver to backtrack. To make it possible to compute conflict clauses in the methods described above, each component of the entries for an equivalence class e in the two maps $\mathcal{Z}$ and $\mathcal{S}$ is additionally annotated with an explanation pair (t, φ), where t is a term in e and φ entails that t has the property represented by the component. This is maintained independently for each lower bound, upper bound, prefix and suffix. In most cases, this pair is of the form $(t, \top)$, where t is the source of the annotation. When inferring annotations from an asserted membership constraint $x \in R$ during mergeEqc above, their explanations are the pair $(x, x \in R)$. Explanations are updated when entries E_1 and E_2 are merged, where, e.g., the explanation for the lower bound is taken from E_2 when $\ell_2 > \ell_1$. When

[1] It is possible to produce tighter prefixes and suffixes recursively—for instance for terms $t_1 \cdot t' \cdot t_2$ where the equivalence class of t_1 (resp., t_2) is assigned a constant prefix (resp., suffix). However, in our experiments, this did not turn out to be worth the extra effort.

two entries are in conflict, the explanations are used to generate the conflict. For example, assuming two entries have explanations (t_1, φ_1) and (t_2, φ_2), we send the conflict clause $\neg(t_1 \approx t_2 \wedge \varphi_1 \wedge \varphi_2)$. The equality $t_1 \approx t_2$ may be further expanded using standard methods for explanations during congruence closure [25].

Example 2. Consider the constraints $\{x \in \mathsf{rcon}(\mathsf{re}(\texttt{"a"}), \Sigma^*, \mathsf{re}(\texttt{"b"})), z \approx \texttt{"bcd"} \cdot w, x \approx z\}$. The state of the map $\mathcal{S}$ after processing each assertion is as follows:

#	Assertion	$\mathcal{S}$	Conflict?
1	$x \in \mathsf{rcon}(\mathsf{re}(\texttt{"a"}), \Sigma^*, \mathsf{re}(\texttt{"b"}))$	$[x] \mapsto (\texttt{"a"}, \texttt{"b"})$	
2	$z \approx \texttt{"bcd"} \cdot w$	$\mathcal{S}_1 \cup [z] \mapsto (\texttt{"bcd"}, \epsilon)$	
3	$x \approx z$	$\mathcal{S}_2$	$\mathcal{S}_2([x]), \mathcal{S}_2([z]))$

When the first constraint $x \in \mathsf{rcon}(\mathsf{re}(\texttt{"a"}), \Sigma^*, \mathsf{re}(\texttt{"b"}))$ is asserted, we construct the (Boolean) equivalence class for this constraint and merge it with $[\top]$. Based on the mergeEqc method, we infer that the prefix and suffix for the string equivalence class $[x]$ are `"a"` and `"b"` respectively, which are added to $\mathcal{S}$ to obtain $\mathcal{S}_1$ When the second constraint is asserted, we infer the prefix `"bcd"` for $[z]$ and add it to $\mathcal{S}_1$ to get $\mathcal{S}_2$; no suffix is inferred since we do not know the value of w. When the third constraint is asserted, the equivalence classes $[x]$ and $[z]$ merge. Since we have inferred that `"a"` is a prefix of $[x]$ and `"bcd"` is a prefix of $[z]$, we have a conflict, as these two strings do not have a common prefix. Our procedure will thus report a conflict containing the three constraints.

Example 3. Consider the constraints $\{|s| \not\approx 0, |\texttt{"abc"} \cdot w| \not\approx 0, x \approx s, x \approx \texttt{"abc"} \cdot w\}$, where s is the term $\mathsf{substr}(y, 0, 2)$, which takes the substring of y at position 0 of length (at most) 2. The state of the map $\mathcal{Z}$ after processing each assertion is as follows:

#	Assertion	$\mathcal{Z}$	Conflict?
1	$\lvert s\rvert \not\approx 0$	$[0] \mapsto [0,0], [\lvert s\rvert] \mapsto [0,2]$	
2	$\lvert\texttt{"abc"} \cdot w\rvert \not\approx 0$	$\mathcal{Z}_1 \cup [\lvert\texttt{"abc"} \cdot w\rvert] \mapsto [3,\infty]$	
3	$x \approx s$	$\mathcal{Z}_2$	
4	$x \approx \texttt{"abc"} \cdot w$	$\mathcal{Z}_3$	$\mathcal{Z}([\lvert s\rvert]), \mathcal{Z}([\lvert\texttt{"abc"} \cdot w\rvert])$

When the first constraint $|s| \not\approx 0$ is asserted, we construct the equivalence classes $[0]$ and $[|s|]$. The former trivially has bounds $[0, 0]$. For the latter, we use the methods from [27] to infer lower and upper bounds for $|s|$. Note that every string has a lower length bound of 0. The upper bound for the length of $\mathsf{substr}(y, 0, 2)$ can easily be inferred to be 2. Similarly, when $|\texttt{"abc"} \cdot w| \not\approx 0$ is asserted, the equivalence class $[|\texttt{"abc"} \cdot w|]$ is created, whose length has a lower bound of 3 and no upper bound. After the latter two constraints are asserted, note that s becomes equal to $\texttt{"abc"} \cdot w$ by transitivity, and hence $|s|$ is equal to $|\texttt{"abc"} \cdot w|$ by congruence. When these two equivalence classes merge, we obtain

a conflict from their respective entries in $\mathcal{Z}$, since the former has an upper bound of 2 and the latter has a lower bound of 3. Thus, our procedure returns the latter two constraints as a conflict.

4 Model-Based Reductions for Strings

The bottleneck for string solving often lies in reasoning about the reductions of extended string functions. Context-dependent simplification can greatly improve the scalability of string solvers for extended string constraints [29]. At a high level, this approach attempts to simplify extended terms based on information that holds in the current context, which can preempt the need for potentially expensive reasoning. In this work, we extend this strategy by additionally reasoning about candidate models.

First, we briefly review how extended string terms are reduced to more basic constructs. A *reduction formula* for term t is a formula $\varphi \wedge t \approx k$, where k is a fresh variable and φ is a formula over terms $k, t_1, \ldots, t_n$ that *characterizes* the meaning of t in the sense that a theory interpretation satisfies φ if and only if it satisfies $t \approx k$. As a result, the formula $\exists\, k.\, (\varphi \wedge t \approx k)$ is valid in the theory, and hence its Skolemized version can be given to the SAT solver as a lemma. This effectively reduces the satisfiability of constraints of the form $c[t]$ to the satisfiability of $c[k] \wedge \varphi$, where t has been replaced by k.

Example 4. Let t be the regular expression membership constraint $x \in \mathsf{re}(\texttt{"a"})^*$. The formula $(k \approx (x \approx \epsilon \vee x \in \mathsf{re}(\texttt{"a"}) \vee \psi)) \wedge t \approx k$ where ψ is

$$\exists k_1 k_2 k_3.\, x \approx k_1 \cdot k_2 \cdot k_3 \wedge k_1 \in \mathsf{re}(\texttt{"a"}) \wedge k_2 \in \mathsf{re}(\texttt{"a"})^* \wedge k_3 \in \mathsf{re}(\texttt{"a"})$$

is a reduction for t.

Reductions like the one above can be expensive to reason about, since they may introduce fresh (possibly universally) quantified variables. Context-dependent simplifications can avoid these reductions in some cases.

Given a string term t of the form $f(t_1, \ldots, t_n)$, where f is an extended function, a *context-dependent simplification* is a formula of the form $(t_1 \approx s_1 \wedge \ldots \wedge t_n \approx s_n) \Rightarrow t \approx l$ where l is the constant value obtained by evaluating or rewriting $f(s_1, \ldots, s_n)$. Whenever possible, we use context-dependent simplifications for extended string terms, where $t_1 \approx s_1, \ldots, t_n \approx s_n$ are equalities that hold in the current context. The same approach can be applied to regular expression memberships as well, where a membership constraint of the form $x \in R$ can be simplified to $\top$ or $\bot$ whenever x is inferred to be equal to a concrete string literal.

Example 5. Let t be as in the previous example. The formula $x \approx \texttt{"b"} \Rightarrow t \approx \bot$ is a context-dependent simplification for t.[2]

While context-dependent simplification eliminates some reductions, in this paper we propose making certain reductions even lazier by taking into account *candidate* models. If a candidate model can be built that already satisfies a constraint with extended terms, it is not necessary to reduce it.

To elaborate, existing procedures for strings [21] are able to construct candidate models $\mathcal{M}$ (or, more precisely, interpretations) for satisfiable sets of string constraints before reductions are considered by treating all (sub)terms headed by an extended function as fresh variables, and by ignoring regular expression membership constraints. A strategy for *model-based* reduction only considers reductions for t if the candidate model $\mathcal{M}$ is inconsistent with the semantics of t—something that can be easily checked by evaluating t in the model and verifying that the computed value coincides with the value that $\mathcal{M}$ assigns to t as a variable. This allows us to avoid reductions for cases where a candidate model is correctly guessed in the presence of extended functions and regular expression membership constraints. A concrete instantiation of this strategy is described in Sect. 6.

Example 6. Consider the constraints $\{x \approx y \cdot \texttt{"c"}, \neg x \in \mathsf{rcon}(\Sigma^*, \mathsf{re}(\texttt{"j"}), \Sigma^*)\}$. A model-based reduction strategy would first construct a candidate model that satisfies the first constraint, e.g., $\mathcal{M} = \{x \mapsto \texttt{"abc"}, y \mapsto \texttt{"ab"}\}$. It would then check whether the membership constraint $x \in \mathsf{rcon}(\Sigma^*, \mathsf{re}(\texttt{"j"}), \Sigma^*)$ evaluates to false in $\mathcal{M}$. This is indeed the case, since $x^{\mathcal{M}} = \texttt{"abc"}$, making $\mathcal{M}$ a model for the full set of constraints. Hence, the reduction for the regular membership constraint in this example can be avoided altogether.

5 Fast Techniques for Regular Expression Inclusion

As mentioned in Sect. 4, regular expression memberships are handled by a lazy reduction, which can be seen as a single-step unfolding. While model-based reductions can avoid some reductions, the remaining ones may still be expensive. In this section, we show another technique to avoid reductions, based on the observation that most regular expressions in real programs are relatively simple. We focus on those of the form $\mathsf{rcon}(R_1, \ldots, R_n)$, where each R_i corresponds to a fixed or arbitrary number of range or constant regular expressions. Such regular expressions are frequently used to match a string that is made up of multiple segments, each with a different alphabet. For this fragment of regular expressions, our procedure allows us to detect conflicts before unfolding and may additionally tell us which regular expression memberships are entailed by others, and hence can be discarded.

We use the notation $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$ to denote that R_1 matches a subset of the strings matched by R_2. The derivation rules in Fig. 3 can be used to implement a

[2] We omit from the implication the trivial antecedent $\mathsf{re}(\texttt{"a"})^* \approx \mathsf{re}(\texttt{"a"})^*$.

$$\textsf{Emp}\frac{}{\mathcal{L}(\texttt{""}) \subseteq \mathcal{L}(R^*)} \qquad \textsf{Star}\frac{}{\mathcal{L}(R) \subseteq \mathcal{L}(R^*)}$$

$$\textsf{All}\frac{}{\mathcal{L}(R) \subseteq \mathcal{L}(\Sigma^*)} \qquad \textsf{Refl}\frac{}{\mathcal{L}(R) \subseteq \mathcal{L}(R)}$$

$$\textsf{Trans}\frac{\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2) \quad \mathcal{L}(R_2) \subseteq \mathcal{L}(R_3)}{\mathcal{L}(R_1) \subseteq \mathcal{L}(R_3)} \qquad \textsf{CongStar}\frac{\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)}{\mathcal{L}(R_1^*) \subseteq \mathcal{L}(R_2^*)}$$

$$\textsf{Char}\frac{\text{For each } x \in \mathcal{L}(R),\ |x| = 1}{\mathcal{L}(R) \subseteq \mathcal{L}(\Sigma)} \qquad \textsf{Range}\frac{c_1 \geqslant c_3 \quad c_2 \leqslant c_4}{\mathcal{L}(\textsf{range}_{c_1,c_2}) \subseteq \mathcal{L}(\textsf{range}_{c_3,c_4})}$$

$$\textsf{Concat}\frac{\mathcal{L}(R_1) \subseteq \mathcal{L}(R_3) \quad \mathcal{L}(R_2) \subseteq \mathcal{L}(R_4)}{\mathcal{L}(\textsf{rcon}(R_1, R_2)) \subseteq \mathcal{L}(\textsf{rcon}(R_3, R_4))}$$

Fig. 3. Rules for deriving $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$.

fast, incomplete procedure to prove $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$. The procedure applies the rules bottom-up to build a derivation tree with $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$ as the root. The statement is proven if a derivation tree is found where all leaves have no preconditions. For any given pair of regular expressions, the number of possible rule applications is finite, and whether a rule applies can be checked in polynomial time w.r.t. the number of elements in the regular expression concatenations.

The first four rules in Fig. 3 have no preconditions. A regular expression R matches zero or more occurrences of R and the rules Emp and Star use that fact to conclude that (the language generated by) R^* includes the empty string, corresponding to zero occurrences of R, and (the language generated by) R, corresponding to a single occurrence of R. The third rule, All, concludes that every R is included in Σ^*, which matches all strings. Finally, Refl captures the reflexivity of the regular expression inclusion relation. Regular expression inclusion is transitive, which is captured by Trans. Additionally, CongStar captures that applying the Kleene star to regular expressions preserves the inclusion relation. The next two rules are related to regular expressions that match single characters: Char concludes that if a regular expression matches only single characters then it is included in Σ, which matches all characters; Range compares the bounds of two ranges to determine if one is included in the other. Finally, the rule Concat splits regular expression concatenations into two parts and ensures that the parts on the right-hand side include the parts on the left-hand side. Note that the splits themselves can be concatenations, so there is a choice regarding how those concatenations are split into two parts. In the context of this rule, we treat regular expressions that match a single word as a concatenation of the individual letters of that word. For example, for $\mathcal{L}(\texttt{"abc"}) \subseteq \mathcal{L}(\textsf{rcon}(\texttt{"ab"}, \Sigma))$, we could choose the subgoal $\mathcal{L}(\texttt{"c"}) \subseteq \mathcal{L}(\Sigma)$ after applying Concat.

Given a regular expression inclusion $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$, the above procedure may potentially derive conflicts or propagate regular membership constraints, avoiding reducing them. A conflict can be derived from membership constraints $x \in R_1$ and $\neg y \in R_2$ if $x \approx y$ is entailed by the current context. Similarly, from $x \approx y$ being entailed and $y \in R_1$ being asserted, we can propagate the regular membership constraint $x \in R_2$; and from $x \approx y$ and $\neg y \in R_2$ we can propagate $\neg x \in R_1$.

Example 7. Consider the following theory literals:

$$x \in \mathsf{rcon}((\mathsf{range}_{0,9})^*, \Sigma^*, \texttt{"b"}, \Sigma^*) \tag{1}$$

$$\neg x \in \mathsf{rcon}((\mathsf{range}_{0,9})^*, \Sigma^*) \tag{2}$$

We can apply Concat, Refl, and All to the two regular expressions:

$$\textsf{Concat}\frac{\textsf{Refl}\dfrac{}{\mathcal{L}((\mathsf{range}_{0,9})^*) \subseteq \mathcal{L}((\mathsf{range}_{0,9})^*)} \quad \textsf{All}\dfrac{}{\mathcal{L}(\mathsf{rcon}(\Sigma^*, \mathsf{re}(\texttt{"b"}), \Sigma^*)) \subseteq \mathcal{L}(\Sigma^*)}}{\mathcal{L}(\mathsf{rcon}((\mathsf{range}_{0,9})^*, \Sigma^*, \mathsf{re}(\texttt{"b"}), \Sigma^*)) \subseteq \mathcal{L}(\mathsf{rcon}((\mathsf{range}_{0,9})^*, \Sigma^*))}$$

This allows us to derive a conflict, since the regular expression of the negative membership constraint in Eq. (2) includes the regular expression in the positive regular membership constraint in Eq. (1).

6 An Extended Strategy for Strings in CDCL(T)

In this section, we summarize our overall strategy for solving string constraints that leverages the aforementioned techniques. This strategy integrates the techniques presented in this paper with existing techniques used in modern string solvers. In general, the techniques presented in this work are applicable to a wide range of solvers. The techniques from Sect. 3 can be combined with any string solver that computes the congruence closure of the constraints. Model-based reductions are applicable to string solvers that can compute models and have the infrastructure to selectively refine/ignore certain constraints. Regular expression inclusion can be used in all string solvers.

Recall that in a CDCL(T)-based SMT solver, the theory solvers produce conflict clauses or lemmas based on the content of the current context, the truth assignment incrementally constructed by the SAT solver. In the following, we split the discussion between checks that are performed on partial assignments and checks that are performed on full assignments from the SAT solver.

Checking Partial Assignments. Recall that M is the assignment to literals chosen by the SAT solver. In our implementation, whenever the SAT solver adds a literal $(\neg)t \approx s$ to M, that literal is immediately added to the congruence closure data structure of the appropriate theory.[3] This means that in a typical configuration,

[3] In our implementation, each theory locally maintains its own congruence closure data structure.

checkFull(S)

1 Let F = getRefineExt(S); **if** $F = \varnothing$ **return** SAT **else return** F

getRefineExt(S)

1 $C, E, E_m := \varnothing$
2 **for all** ext. terms and r.e. memberships $t \in \mathcal{T}(\mathsf{S})$ where $t = f(t_1, \ldots, t_n)$ **do**
3 **if** $\exists s_1, \ldots s_n$ s.t. $\mathsf{S} \models t_1 \approx s_1 \wedge \ldots \wedge t_n \approx s_n$ and $f(s_1, \ldots, s_n)\downarrow = c$ **then**
4 **if** $\mathsf{S} \not\models t \approx c$ **then** add t to C
5 **else if** t is $x \in R$ **then**
6 Let b be the Boolean value such that $\mathsf{S} \models t \approx b$.
7 **if** $b = \bot$ and $\mathsf{S} \models x \approx x' \wedge (x' \in R')$ and $\mathcal{L}(R') \subseteq \mathcal{L}(R)$ **then**
8 **return** CONFLICT, $\{(x \not\approx x' \vee x \in R \vee \neg x' \in R')\}$
9 **else if** $\mathsf{S} \models x \approx x' \wedge (x' \in R') \approx b$ and
 $((\mathcal{L}(R') \subseteq \mathcal{L}(R)$ and $b = \top)$ or $(\mathcal{L}(R) \subseteq \mathcal{L}(R')$ and $b = \bot))$ **then**
10 **continue**
11 **end if**
12 Add t to E_m if b is false, and E otherwise
13 **else**
14 Add t to E
15 **end if**
16 **end for**
17 **if** C is non-empty **then return** $\{\mathsf{cd\_simplify}(\mathsf{S}, t) \mid t \in C\}$
18 $F :=$ getRefine(S)
19 **if** F is non-empty **then return** F
20 **if** E is non-empty **then return** $\{\mathsf{reduce}(t) \mid t \in E\}$
21 Construct model $\mathcal{M}$ for $\alpha(\mathsf{S})$ and **return** $\{\mathsf{reduce}(t) \mid t \in E_m, \mathsf{S} \not\models t \approx t^{\mathcal{M}}\}$

Fig. 4. Strings theory solver using context-dependent simplification, regular expression inclusion, and model-based reductions.

conflicts that are based purely on equality reasoning may be raised the moment M becomes unsatisfiable in the theory. This behavior makes the SMT solver faster, as it may backtrack without having to generate any further extension to M. The techniques in Sects. 3.1 and 3.2 increase the likelihood that such conflicts may be discovered eagerly based on evaluation, arithmetic approximations, and tracking prefixes and suffixes for string terms. Given that those techniques are executed every time the SAT solver assigns a value, it is imperative that they are inexpensive.

Checking Full Assignments. When a full assignment is generated by the SAT solver, each theory solver is called upon to do a *full effort* consistency check on the assignment M. We describe the strategy used for strings that incorporates reasoning about context-dependent simplification, regular expression inclusion, and model-based reductions.

Our approach checkFull is sketched in Fig. 4, which summarizes the behavior of our (extended) theory solver for strings to be used in the CDCL(T) loop.

The method takes as input a set of string constraints S, which is the subset of the literals assigned by the SAT solver that belongs to the theory of strings. We assume the method is called when S is satisfiable in the empty theory, and is such that the techniques from Sect. 3 did not raise a conflict. It calls the subprocedure $\mathsf{getRefineExt}$, which returns a set of formulas F. This set may contain a *conflict clause*, that is, a disjunction of literals that are false in S. If F is non-empty, these formulas are returned to the SAT solver. Otherwise, if F is empty, then the method returns SAT, indicating that S is satisfiable.

In the subprocedure $\mathsf{getRefineExt}$, we first classify the extended terms t from S by adding them to (at most) one of three sets: the set of terms C to simplify based on the context, the set of terms E to reduce, and the set of terms E_m to reduce if necessary based on a candidate model. This is done as follows. We first check if term t can be simplified based on the context, that is, if we can infer that its arguments are equivalent to terms $s_1, \ldots, s_n$ such that $f(s_1, \ldots, s_n)$ can be simplified to a constant c. In this case, t is added to C if it is not already entailed in S to be equal to c. Otherwise, if t is a regular expression membership $x \in R$, then we check whether t is otherwise directly in conflict with another membership or can be discarded. The former holds when it is the case that $x \in R$ holds with negative polarity, there exists a term x' that is entailed to be equal to x such that $x' \in R'$ is entailed to hold with positive polarity, and our regular expression inclusion test can prove that the language of R includes that of R'. In this case, we know that we are in conflict since x cannot be both in R' and not in R, and a conflict clause is returned. Otherwise, we may avoid reducing t if it is entailed by another membership $x' \in R'$ with the same polarity again where x' is entailed equal to x. This may occur if the language of R includes R' and the polarity of both memberships are positive, or if R' includes R and the polarity of both memberships are negative. If none of these cases hold, then we add t to E if it is a positive membership, and E_m otherwise. Here, the intuition is that *negative* memberships are both more expensive to reason about via reductions, and more likely to be satisfied by candidate models. All other extended terms are added to E, marking them to be reduced. Although not shown in the figure, if t is an application of string containment, then it is handled analogously to regular expression membership, noting that $\mathsf{ctn}(x, y)$ is equivalent to $x \in \mathsf{rcon}(\Sigma^*, \mathsf{re}(y), \Sigma^*)$.

Assuming the above classification, we run four steps in decreasing order of priority. First, if C is non-empty, we add the simplification formula for each $t \in C$, where we write $\mathsf{cd\_simplify}(\mathsf{S}, t)$ to denote the formula corresponding to the context-dependent simplification of t in S. Second, we run the core theory solver for strings, denoted by method $\mathsf{getRefine}$, which we assume runs the rule-based procedure from [21]. For our purposes, we assume this method returns a (possibly empty) set of refinement lemmas or conflict clauses, which we denote F and return this set if it is non-empty. Otherwise, if our set E of terms to reduce is non-empty, we return the set of reduction formulas $\mathsf{reduce}(t)$ for all $t \in E$. If none of these cases generated lemmas, then we construct a candidate model $\mathcal{M}$ for the abstraction of S, denoted $\alpha(\mathsf{S})$, which denotes a formula where all

Table 1. Number of solved problems per benchmark set for different configurations. Best results are in **bold**. All benchmarks ran with a timeout of 1200 s.

Set	cvc5	cvc5-v	cvc5-e	cvc5-m	cvc5-r	cvc5-vemr	z3
Industry (62)	**58**	57	**58**	56	57	55	31
Slog (17)	**17**	**17**	**17**	**17**	**17**	**17**	10
QGen (159)	158	158	**159**	**159**	158	153	**159**
Norn (175)	85	84	81	**98**	85	88	47
Kepler (436)	**89**	**89**	**89**	**89**	**89**	**89**	85
Kaluza (225)	**225**	**225**	**225**	**225**	**225**	**225**	65
PyEx (6,948)	6,927	6,902	**6,931**	6,767	6,926	6,716	5,949
Slent (105)	**93**	82	69	**93**	**93**	41	39
Leetcode (13)	**13**	**13**	**13**	**13**	**13**	**13**	11
FullStrInt (2,718)	**2,630**	2,608	**2,630**	2,629	2,628	2,611	2,461
SmallRw (73)	**52**	**52**	**52**	51	**52**	51	6
Total (10,931)	**10,347**	10,287	10,324	10,197	10,343	10,059	8,863

extended terms in S are replaced by fresh variables. Then, for each $t \in E_m$ we check whether the constraint for t holds in the candidate model $\mathcal{M}$. In particular, this is the case if $\mathsf{S} \models t \approx t^{\mathcal{M}}$. We return $\mathsf{reduce}(t)$ only for terms t for which this does not hold.

Notice that the model $\mathcal{M}$ serves only as a way of filtering our reductions. We do not apply context-dependent simplification based on the model, e.g., adding the lemma $(t_1 \approx t_1^{\mathcal{M}} \wedge \ldots \wedge t_n \approx t_n^{\mathcal{M}}) \Rightarrow t \approx f(t_1^{\mathcal{M}}, \ldots, t_n^{\mathcal{M}})\downarrow$, as this would introduce an unbounded number of new literals $t_i \approx t_i^{\mathcal{M}}$ to the search.

7 Evaluation

We have implemented the strategy from Sect. 6 by extending CVC5, a CDCL(T)-based state-of-the-art SMT solver that implements context-dependent simplifications [29], aggressive rewriting [27], and efficient reductions [28]. To evaluate our extension, we measure its performance on the 69,907 SMT-LIB benchmarks [9] that include the theory of strings[4] and on a set of 74 benchmarks which we have obtained from an industrial partner but are not allowed to make public. In this section, we present and discuss the results of that evaluation.

We test the performance impact of the four techniques presented in this paper: enhanced congruence closure (**v**), eager conflicts based on properties of equivalence classes (**e**), model-based reductions (**m**), and regular expression inclusion (**r**). We compare a configuration with all techniques enabled (**cvc5**) with configurations that disable individual techniques (prefixed with **cvc5-***). To measure the combined impact, we additionally include a configuration that disables all

[4] We excluded one benchmark with a quantifier in the quantifier-free logic `QF_SLIA`.

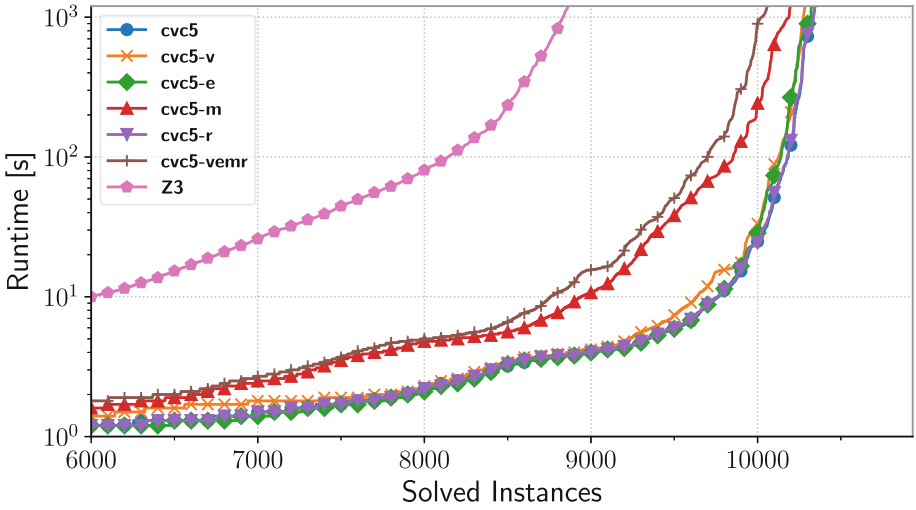

Fig. 5. Cactus plot of the number of solved benchmarks. All benchmarks ran with a timeout of 1200 s.

techniques presented in this paper, but otherwise uses all of CVC5's advanced techniques for strings (**cvc5-vmre**). Finally, as an additional reference point, we compare with another state-of-the-art solver, Z3 Version 4.8.14 [15]. In our experience, Z3 is the most stable, feature-complete competitor to CVC5's string solver. We omit a comparison with Z3STR4 [23] because it returned wrong answers at SMT-COMP 2021 [2] and there has not been a new release. Similarly, we omit a comparison with Z3-TRAU 1.1 [1] (the successor of TRAU [4]), because we found it to be unsound in earlier work [28]. Finally, OSTRICH 1.1 [14] requires inputs to be in the straight-line fragment [22], which is not the case for some of the benchmarks.

We ran all experiments on a cluster equipped with Intel Xeon E5-2620 v4 CPUs. We allocated one physical CPU core and 8 GB of RAM for each solver-benchmark pair and used a time limit of 1200 s, which is the same time limit used at SMT-COMP 2021. In the following presentation of the results, we omit the 59,050 benchmarks that are solved in less than a second by all solvers to emphasize non-trivial benchmarks. Table 1 lists the number of solved benchmarks for each benchmark family and configuration. Figure 5 shows a cactus plot of the number of solved instances for each configuration. The scatter plots in Fig. 6 compare the performance of **cvc5** with the other CVC5 configurations and Z3. Each scatter plot shows the solving times of the two solvers for each benchmark and differentiates between satisfiable and unsatisfiable inputs.

Overall, all configurations of CVC5 significantly outperform Z3, which is reflected in Fig. 5. The scatter plot Fig. 6f shows that while CVC5 outperforms Z3, they also complement each other to a certain extent, which is not surprising given the complexity of the problem and the fact that the two code bases

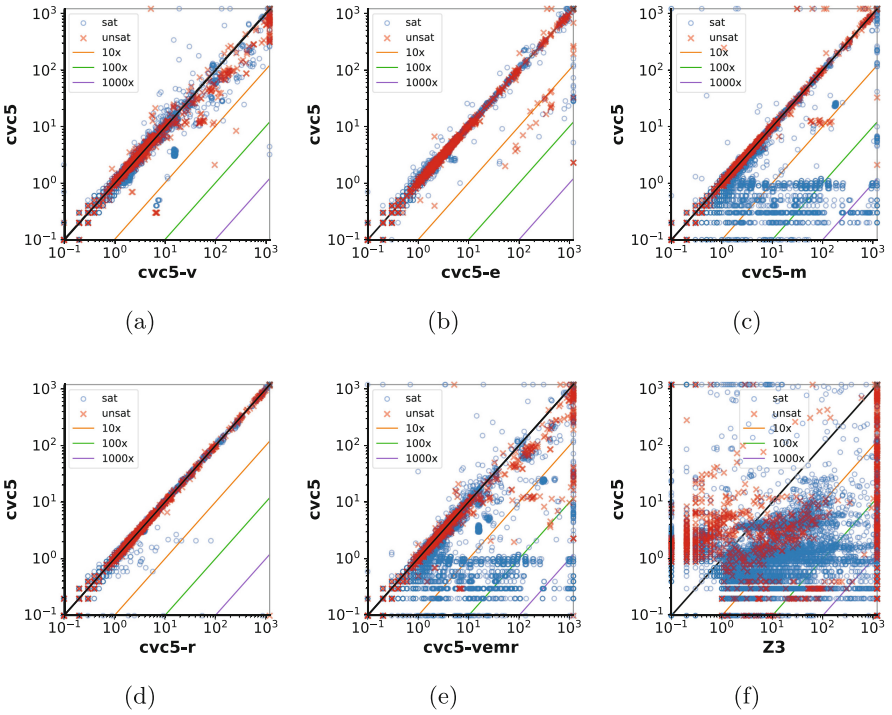

Fig. 6. Scatter plots that compare the performance of CVC5 with the other configurations. The scatter plots differentiate between satisfiable and unsatisfiable benchmarks.

differ significantly. Overall, Z3 solves 270 benchmarks that **cvc5-vmre** does not solve and 171 benchmarks that **cvc5** does not solve. Conversely, **cvc5** solves 1645 benchmarks that Z3 does not solve. Between **cvc5** and **cvc5-vmre**, **cvc5** uniquely solves 309 benchmarks and **cvc5-vmre** 15 benchmarks. This suggests that our techniques help CVC5 solve some of the benchmarks that previously only Z3 could solve, but that they also have a significant impact on benchmarks that Z3 could not solve. Thus, adapting those techniques in Z3 may be beneficial.

The PyEx benchmarks show the biggest difference in number of solved benchmarks across the techniques, with model-based reductions (**m**) solving 160 more benchmarks, significantly increasing the success rate for **cvc5**. Figure 6c indicates that primarily satisfiable benchmarks benefit from **m**. This is expected because the technique allows the solver to skip reductions if it guesses a correct model. Nevertheless, some unsatisfiable benchmarks are also solved noticeably faster due to **m**. This is possibly due to the technique resulting in a search that prioritizes reducing operators that are more likely to participate in conflicts.

Both the enhanced congruence closure (**v**) and the more eager conflicts (**e**) have a relatively low impact on the number of solved benchmarks. However, Figs. 6a and 6b show they significantly improve solving times on several benchmarks. This is expected because they allow the solver to detect conflicts more

eagerly, but the same or similar conflicts would have been found (later on) with existing techniques. Since the solving procedure does not fundamentally change, roughly the same benchmarks should be solved when adding these techniques, but potentially much faster.

Finally, the regular expression inclusion technique (**r**) has a low impact overall, since it is restricted to a specific fragment, but Fig. 6d shows it significantly improves solving time for a few benchmarks. The benchmarks come from the set of industrial problems and from the QGen set of benchmarks. While the technique does not always apply, we have found it to be very important for certain industrial problems. Moreover, the scatter plot shows that having the technique available has no negative effect, which allows such a specialized procedure to be always active in a modular solver.

8 Conclusion

We have presented new techniques that make conflict detection more eager and reductions lazier in CDCL(T)-based string solvers. Our evaluation shows that both classes of techniques significantly improve performance in the state-of-the-art SMT solver CVC5 on SMT-LIB and industrial problems. As future work, we plan to generalize our eager equality-based conflict detection to leverage more sophisticated properties. We also plan to apply similar techniques to other congruence-closure-based theory solvers, such as those for the theory of finite sets and relations. The set of rules for proving regular expression inclusion was driven by empirical work on industrial benchmarks, but it could be expanded. We also plan to investigate further strategies for lazy reductions of other extended string terms that lead to bottlenecks in real-world applications.

References

1. Z3-Trau (2019). https://github.com/guluchen/z3/tree/new_trau
2. SMT-COMP 2021 (2021). https://smt-comp.github.io/2021/
3. Abdulla, P.A., et al.: Flatten and conquer: a framework for efficient analysis of string constraints. In: PLDI, pp. 602–617. ACM (2017)
4. Abdulla, P.A., et al.: TRAU: SMT solver for string constraints. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, 30 October–2 November, 2018, pp. 1–5. IEEE (2018)
5. Abdulla, P.A., et al.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_10
6. Backes, J., et al.: Stratified abstraction of access control policies. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 165–176. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_9
7. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: FMCAD, pp. 1–9. IEEE (2018)

8. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2022. LNCS, vol. 13243. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
9. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). https://www.smt-lib.org/
10. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
11. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: FMCAD, pp. 55–59. IEEE (2017)
12. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_27
13. Bouchet, M., et al.: Block public access: trust safety verification of access control policies. In: ESEC/SIGSOFT FSE, pp. 281–291. ACM (2020)
14. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. PACMPL **3**(POPL), 49:1–49:30 (2019)
15. 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). https://doi.org/10.1007/978-3-540-78800-3_24
16. Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Academic Press, Cambridge (2001)
17. Fu, X., Li, C.: A string constraint solver for detecting web application vulnerability. In: SEKE, pp. 535–542. Knowledge Systems Institute Graduate School (2010)
18. 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). https://doi.org/10.1007/978-3-642-18275-4_18
19. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: 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)
20. 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). https://doi.org/10.1007/978-3-319-03077-7_2
21. 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). https://doi.org/10.1007/978-3-319-08867-9_43
22. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: POPL, pp. 123–136. ACM (2016)
23. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: a multi-armed string solver. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 389–406. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_21
24. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980)
25. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 453–468. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_33

26. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
27. Reynolds, A., Nötzli, A., Barrett, C., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 23–42. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_2
28. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: Reductions for strings and regular expressions revisited. In: FMCAD, pp. 225–235. IEEE (2020)
29. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 453–474. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_24
30. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, Berleley/Oakland, California, USA, 16–19 May 2010, pp. 513–528. IEEE Computer Society (2010)
31. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), pp. 13–22. IEEE (2007)
32. Trinh, M., Chu, D., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: CCS, pp. 1232–1243. ACM (2014)
33. Veanes, M., Bjørner, N., de Moura, L.: Symbolic automata constraint solving. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6397, pp. 640–654. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16242-8_45
34. Veanes, M., Tillmann, N., de Halleux, J.: Qex: symbolic SQL query explorer. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 425–446. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_24
35. Yu, F., Alkhalaf, M., Bultan, T.: 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, 20–28 March 2010. Proceedings, pp. 154–157 (2010)
36. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, Saint Petersburg, Russian Federation, 18–26 August 2013, pp. 114–124. ACM (2013)