Dashed Strings and the Replace(-all) Constraint

Roberto Amadini[1](✉), Graeme Gange[2], and Peter J. Stuckey[2]

[1] University of Bologna, Bologna, Italy
roberto.amadini@unibo.it
[2] Monash University, Melbourne, VIC, Australia
{graeme.gange,peter.stuckey}@monash.edu

Abstract. Dashed strings are a formalism for modelling the domain of string variables when solving combinatorial problems with string constraints. In this work we focus on (variants of) the Replace constraint, which aims to find the *first* occurrence of a query string in a target string, and (possibly) replaces it with a new string. We define a Replace propagator which can also handle Replace-Last (for replacing the *last* occurrence) and Replace-All (for replacing *all* the occurrences). Empirical results clearly show that string constraint solving can draw great benefit from this approach.

1 Introduction

In the past decade the interest in solving string constraints has considerably grown in disparate application domains such as test-case generation, software analysis and verification, model checking, web security, database query processing, bionformatics (see, e.g., [3,8,9,11–15,18,24,29]).

Among the most effective paradigms for string constraint solving we mention Satisfiability Modulo Theory (SMT), which relies on (extensions of) the theory of *word equations*, and Constraint Programming (CP), which basically uses an *unfolding*-based approach to eventually decompose a string variable into sequences of integer variables representing the string characters.

A promising CP approach, on which this paper focuses, is based on *dashed strings*. Given a fixed finite alphabet Σ and a maximum string length λ, dashed strings are a particular class of regular expressions denoting sets of concrete strings $W \subseteq \{w \in \Sigma^* \mid |w| \leq \lambda\}$ through the concatenation of distinct sets of characters called *blocks*. Dashed strings are used to represent in a compact way the domain of string variables without eagerly unfolding them into λ integer variables. This can make a huge difference, especially when λ is big.

Several algorithms for dashed strings have been defined to propagate string constraints like (in-)equality, concatenation, length, or regular expression membership. Most of them are based on the concept of dashed strings *equation* [7].

In this paper, we focus on a well-known constraint frequently occurring in problems derived from software verification and testing: the Replace constraint.

H. Simonis (Ed.): CP 2020, LNCS 12333, pp. 3–20, 2020.
https://doi.org/10.1007/978-3-030-58475-7_1

Despite being a construct heavily used in modern programming, very few solvers have implemented Replace, and its variants, due its non trivial semantics. For this reason, few benchmarks exist.

Given an input string x, a query string q and a new string q' we have that Replace(x, q, q') is the string obtained by replacing the *first* occurrence of q in x with q' (Replace$(x, q, q') = x$ if q does not occur in x). Common variants are Replace-Last (that replaces the *last* occurrence of the query string) and Replace-All (that replaces *all* the occurrences of the query string).

At present, no dashed string propagator has been defined for Replace (in [5] it is simply *decomposed* into other string constraints). This approach is sound but certainly improvable. In this work we define a unified propagation algorithm that *(i)* improves the precision and the efficiency of Replace propagation, and *(ii)* is general enough to propagate Replace-Last and Replace-All. The latter in particular cannot be decomposed into basic constraints because we do not know *a priori* how many times the query string occurs.

Empirical results show that the approach we propose significantly outperforms the performance of the decomposition approach and state-of-the-art SMT solvers.

Paper structure. In Sect. 2 we give preliminary notions. In Sect. 3 we explain how we propagate Replace and related constraints. In Sect. 4 we report the empirical results, before discussing the related works in Sect. 5 and concluding in Sect. 6.

2 Dashed Strings

Let Σ be a finite alphabet and Σ^* the set of all the strings over Σ. We denote with ϵ the empty string and with $|w|$ the length of $w \in \Sigma^*$. We use 1-based notation for (dashed) strings: $w[i]$ is the i-th symbol of w for $i = 1, \dots, |w|$.

The concatenation of $v, w \in \Sigma^*$ is denoted by $v \cdot w$ (or simply vw) while $w^n = ww^{n-1}$ is the concatenation of w for $n > 0$ times ($w^0 = \epsilon$). The concatenation of $V, W \subseteq \Sigma^*$ is denoted with $V \cdot W = \{vw \mid v \in V, w \in W\}$ (or simply VW) while $W^n = WW^{n-1}$ is the concatenation of W for n times ($W^0 = \{\epsilon\}$). In this work we focus on *bounded-length* strings, i.e., we fix an upper bound $\lambda \in \mathbb{N}$ and only consider strings in $\mathbb{S}_\Sigma^\lambda = \{w \in \Sigma^* \mid |w| \leq \lambda\}$.

A *dashed string* $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$ is a concatenation of $k = |X| > 0$ blocks such that, for $i = 1, \dots, k$: *(i)* $S_i \subseteq \Sigma$; *(ii)* $0 \leq l_i \leq u_i \leq \lambda$; *(iii)* $u_i + \sum_{j \neq i} l_j \leq \lambda$. For each block $X[i] = S_i^{l_i,u_i}$ we call S_i the *base* and (l_i, u_i) the *cardinality*, where $l_i = lb(S_i^{l_i,u_i})$ is the *lower bound* while $u_i = ub(S_i^{l_i,u_i})$ is the *upper bound* of $X[i]$. We extend this notation to dashed strings: $lb(X) = \sum_{i=1}^n lb(X[i])$ and $ub(X) = \sum_{i=1}^n ub(X[i])$. Intuitively, $lb(X)$ and $ub(X)$ are respectively the lower and the upper bound on the length of each concrete string denoted by X.

The primary goal of dashed strings is to compactly represent sets of strings, so we define a function γ such that $\gamma(S^{l,u}) = \{w \in S^* \mid l \leq |w| \leq u\} \subseteq \mathbb{S}_\Sigma^\lambda$ is the *language denoted* by block $S^{l,u}$ (in particular $\gamma(\emptyset^{0,0}) = \{\epsilon\}$ for the *null block* $\emptyset^{0,0}$). We extend γ to dashed strings: $\gamma(S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}) =$

$(\gamma(S_1^{l_1,u_1}) \dots \gamma(S_k^{l_k,u_k})) \cap \mathbb{S}_\Sigma^\lambda$. Blocks of the form $S^{0,u}$ are called *nullable* because $\epsilon \in \gamma(S^{0,u})$. A dashed string X is *known* if $\gamma(X) = \{w\}$, i.e., it represents a single, "concrete" string $w \in \mathbb{S}_\Sigma^\lambda$.

We call $\mathbb{DS}_\Sigma^\lambda$ the set of all the dashed strings. Note that, while $\mathbb{S}_\Sigma^\lambda$ is finite, $\mathbb{DS}_\Sigma^\lambda$ is *countable*: λ bounds the cardinalities of the blocks, but not the number of blocks that may appear in a dashed string. For example, given distinct characters $a, b \in \Sigma$ we can generate an infinite sequence of dashed strings in $\mathbb{DS}_\Sigma^\lambda$: $\{a\}^{0,1}\{b\}^{0,1}$, $\{a\}^{0,1}\{b\}^{0,1}\{a\}^{0,1}$, $\{a\}^{0,1}\{b\}^{0,1}\{a\}^{0,1}\{b\}^{0,1}, \dots$.

A dashed string $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ is *normalised* if: *(i)* $S_i \neq S_{i+1}$, *(ii)* $S_i = \emptyset \Leftrightarrow l_i = u_i = 0$, *(iii)* $X = \emptyset^{0,0} \vee S_i \neq \emptyset$ for $i = 1, \dots, k$. The operator $\textsc{Norm}$ normalises a dashed string (note $\gamma(X) = \gamma(\textsc{Norm}(X))$). We denote the set of normalised dashed strings as $\overline{\mathbb{DS}_\Sigma^\lambda} = \{\textsc{Norm}(X) \mid X \in \mathbb{DS}_\Sigma^\lambda\}$. Normalisation is fundamental to remove a large number of "spurious" dashed strings from $\mathbb{DS}_\Sigma^\lambda$, i.e., distinct dashed strings denoting the same language. If not otherwise specified, we will always refer to normalised dashed strings.

We define the partial order $\sqsubseteq$ such that $X \sqsubseteq Y \Leftrightarrow (X = Y \vee \gamma(X) \subset \gamma(Y))$ to model the relation *"is more precise than or equal to"* between dashed strings, i.e., we consider X more precise than Y if $\gamma(X) \subset \gamma(Y)$.[1] Note that the poset $(\overline{\mathbb{DS}_\Sigma^\lambda}, \sqsubseteq)$ is well-founded but *not a lattice*: in general, we may not be able to determine the dashed string which best represents two or more concrete strings. For example, the set $\{ab, ba\}$ has two minimal representations $\{a\}^{0,1}\{b\}^{1,1}\{a\}^{0,1}$ and $\{b\}^{0,1}\{a\}^{1,1}\{b\}^{0,1}$ which are not comparable according to $\sqsubseteq$.

From a graphical perspective, we can see a block $S^{l,u}$ as a continuous segment of length l followed by a dashed segment of length $u - l$. The continuous one indicates that exactly l characters of S *must* occur in each concrete string of $\gamma(X)$; the dashed one indicates that $k \leq u - l$ characters of S *may* occur. Consider dashed string $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$ of Fig. 1. Each string of $\gamma(X)$ must start with '`B`' or '`b`', followed by 2 to 4 '`o`'s, one '`m`', and 0 to 3 '`!`'s.

2.1 Positions and Equation

A convenient way to refer a dashed string is through its *positions*. Given $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$, a positions is a pair (i, j) where *index* $i \in \{1, \dots, k\}$ refers to block $X[i]$ and *offset* j indicates how many characters from the beginning of $X[i]$ we are considering. As shown in Fig. 1, indexes are 1-based while offsets are 0-based: position $(i, 0)$ refers to the beginning of $X[i]$ and can be equivalently identified with the end of $X[i-1]$, i.e., with position $(i-1, u_{i-1})$. For convenience, we consider $(k+1, 0)$ equivalent to (k, u_k) and $(0, 0)$ to $(1, 0)$. Given $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$ and positions $(i, j), (i', j')$ we denote with $X[(i, j), (i', j')]$

[1] $\sqsubseteq$ is not defined as $X \sqsubseteq Y \iff \gamma(X) \subseteq \gamma(Y)$ because otherwise $\sqsubseteq$ would be a *pre-order* but not a partial order in general, e.g., if $\lambda = 2$ then $X = \{a, b, c\}^{0,2}$ and $Y = \{a, b\}^{0,2}\{b, c\}^{0,2}\{a, c\}^{0,2}$ are such that $\gamma(X) \subseteq \gamma(Y)$ and $\gamma(Y) \subseteq \gamma(X)$ so $X \sqsubseteq Y$ and $Y \sqsubseteq X$ but $X \neq Y$.

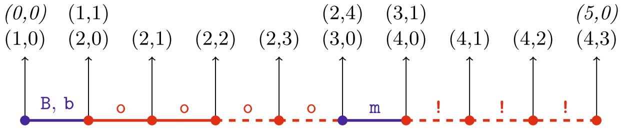

Fig. 1. Representation of $\{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$.

the *region* of X between (i,j) and (i',j'), defined by:

$$\begin{cases} \emptyset^{0,0} & \text{if } (i,j) \succeq (i',j') \\ S_i^{\max(0,l_i-j),j'-j} & \text{else if } i = i' \\ S_i^{\max(0,l_i-j),u_i-j} S_{i+1}^{l_{i+1},u_{i+1}} \cdots S_{i'-1}^{l_{i'-1},u_{i'-1}} S_{i'}^{\min(l_{i'},j'),j'} & \text{otherwise} \end{cases}$$

For brevity, we also define $X[..., P] = X[(0,0), P]$ and $X[P, ...] = X[P, (k, u_k)]$.

For example, in Fig. 1 we have that region $X[(2,1),(2,4)]$ is $\{\texttt{o}\}^{1,3}$ while $X[(2,3),(4,2)] = \{\texttt{o}\}^{0,1}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,2}$. Region $X[(2,2), ...]$ is $\{\texttt{o}\}^{0,2}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$ while $X[..., (4,1)]$ is $\{\texttt{B,b}\}^{2,4}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,1}$.

Positions are used to implement a fundamental operation between dashed strings, on which the propagator Replace propagator relies: the dashed string *equation*. Equating X and Y means looking for a *refinement* of X and Y including *all* the strings of $\gamma(X) \cap \gamma(Y)$ and removing *the most* strings not belonging to $\gamma(X) \cap \gamma(Y)$. In other words, we look for a minimal—or at least small enough—dashed string denoting all the concrete strings of $\gamma(X)$ and $\gamma(Y)$ (the smallest dashed string does not always exist because $(\overline{\mathbb{DS}_\Sigma^\lambda}, \sqsubseteq)$ is not a lattice).

In [7] the authors introduced a sweep-based algorithm, that here we will call Equate[2], such that: *(i)* if $\textsc{Equate}(X,Y) = \bot$, then $\gamma(X) \cap \gamma(Y) = \emptyset$; *(ii)* if $\textsc{Equate}(X,Y) = (X',Y') \neq \bot$, then $X' \sqsubseteq X$, $Y' \sqsubseteq Y$ and $\gamma(X) \cap \gamma(Y) = \gamma(X') \cap \gamma(Y')$. In a nutshell, $\textsc{Equate}(X,Y)$ matches each block $X[i]$ against each block of Y to determine its earliest/latest start/end positions in Y, and uses this information to possibly refine $X[i]$ into more precise block(s) $X'[i] \sqsubseteq X[i]$.

For example, consider matching block $B = \{\texttt{o},\texttt{m},\texttt{g}\}^{2,6}$ against dashed string X of Fig. 1 starting from position $(0,0)$. The earliest start of B is $(2,0)$ (it cannot match $X[1] = \{\texttt{B},\texttt{b}\}^{1,1}$ because its base does not contain any B or b) and the earliest end is $(2,2)$ (B contains at least 2 characters); the latest start is $(2,3)$ because B cannot finish after latest end position $(4,0)$ (it cannot match $\{!\}^{0,3}$). This means that if B is the i-th block of a dashed string Y that we are equating with X, then the prefix $Y[..., i-1]$ has to match $\{\texttt{B},\texttt{b}\}^{1,1}$ and the suffix $Y[i+1, ...]$ has to match $\{!\}^{0,3}$ (otherwise, $\textsc{Equate}(X,Y) = \bot$).

The region between earliest start and latest end is called *feasible region*, while the region between latest start and earliest end is the *mandatory region*. In the above example, the feasible region is $X[(2,0),(4,0)] = \{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}$ while mandatory region is null (latest start comes after earliest end). This information enables the refinement of B into the more precise blocks $\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}$.

[2] In [7] it was called Sweep to distinguish it the Cover equation algorithm.

```
function PUSH$^+$($S^{l,u}$, $Y$, $(i,j)$)
   $(i_0, j_0) \leftarrow (i, j)$
   $k \leftarrow l$                    ▷ $k$ is the number of "consumed" characters for $S^{l,u}$
   while $k > 0$ do
      if $i > |Y|$ then
         return $(i,j),(i,j)$
      $S'^{l',u'} \leftarrow Y[i]$
      if $S \cap S' = \emptyset$ then              ▷ Incompatible block, move to next one
         $(i,j) \leftarrow (i+1, 0)$
         if $l' > 0$ then
            $(i_0, j_0) \leftarrow (i, j)$             ▷ Updating earliest start position
            $k \leftarrow l$
      else if $k \leq u' - j$ then       ▷ We can consume all the characters of $S^{l,u}$
         return $(i_0,j_0),(i,j+k)$
      else
         $k \leftarrow k - (u' - j)$              ▷ Consuming $u' - j$ characters of $S^{l,u}$
         $(i,j) \leftarrow (i+1, 0)$
   return $(i_0,j_0),(i,j)$
```

Fig. 2. PUSH$^+$ algorithm.

```
function STRETCH$^-$($S^{l,u}$, $Y$, $(i,j)$)
   $k \leftarrow u$
   if $j > lb(Y[i])$ then
      $j \leftarrow lb(Y[i])$                    ▷ Skip the "optional part" of $Y[i]$
   while $i > 0$ do
      $S'^{l',u'} \leftarrow Y[i]$
      if $l' = 0$ then                                   ▷ Skip block $Y[i]$
         if $i > 1$ then
            $(i,j) \leftarrow (i-1, lb(Y[i-1]))$
         else
            return $(0,0)$
      else if $S \cap S' = \emptyset$ then                  ▷ We cannot go further
         return $(i,j)$
      else if $k < l'$ then
         return $(i,j-k)$
      else
         $k \leftarrow k - l'$              ▷ Consuming the least characters of $S^{l,u}$
         if $i > 1$ then
            $(i,j) \leftarrow (i-1, lb(Y[i-1]))$
         else
            return $(0,0)$
   return $(i,j)$
```

Fig. 3. STRETCH$^-$ algorithm

EQUATE(X, Y) uses auxiliary functions PUSH$^-$ and PUSH$^+$ to detect the "earliest" positions. Given block $X[i]$, dashed string Y and initial position P_0 of Y, PUSH$^+(X[i], Y, P_0)$ returns a pair (P_s, P_e) where P_s (resp., P_e) is the earliest start (resp. end) of $X[i]$ in $Y[P_0, ...]$ (see pseudo-code in Fig. 2). If $X[i]$ cannot match any block of $Y[P_0, ...]$, then $P_s = P_e = (|Y|+1, 0)$. Dually, Push$^-$ works "backwards" to find (P_s, P_e) in $Y[..., P_0]$ from right to left.

In Fig. 1, if $P_0 \preceq (3,0)$ then $\textsc{Push}^+(\{\mathtt{m}\}^{1,2}, X, P_0) = ((3,0),(4,0))$ and $\textsc{Push}^-(\{\mathtt{m}\}^{1,2}, X, P_0) = ((0,0),(0,0))$; otherwise, if $P_0 \succ (3,0)$, then we have $\textsc{Push}^+(\{\mathtt{m}\}^{1,2}, X, P_0) = ((5,0),(5,0))$ and $\textsc{Push}^-(\{\mathtt{m}\}^{1,2}, X, P_0) = ((4,0),(3,0))$.

Analogously, a pair of Stretch functions are used for "latest" positions: $\textsc{Stretch}^+(X[i], Y, P_0)$ is the latest position (left-to-right) where $X[i]$ can match Y starting from P_0 (while Stretch$^-$ works right-to-left, see Fig. 3).[3]

In Fig. 1, for example, if $P_0 = (2,1)$ then $\textsc{Stretch}^+(\{\mathtt{o,m,g}\}^{1,8}, X, P_0) = (3,1)$ and $\textsc{Stretch}^-(\{\mathtt{o,m,g}\}^{1,8}, X, P_0) = (1,1)$. If instead $P_0 = (0,0)$ then both functions return P_0 because Stretch$^-$ cannot go further left and Stretch$^+$ cannot go further right since $\{\mathtt{o,m,g}\}^{1,8}$ does not match $\{\mathtt{B,b}\}^{1,1}$.

3 Propagating Replace

If $y = \textsc{Replace}(x, q, q')$, then y is the string obtained by replacing the first occurrence of q in x with q' (if q does not occur in x, then $x = y$). The only implementation of Replace with dashed strings, defined in [5], simply rewrites $y = \textsc{Replace}(x, q, q')$ into:[4]

$$\exists\, n, p, s. \begin{pmatrix} n = \textsc{Find}(q,x) \wedge \textsc{Find}(q,p) = [\![|q| = 0]\!] \\ x = p \cdot q^{[\![n>0]\!]} \cdot s \wedge y = p \cdot q'^{[\![n>0]\!]} \cdot s \wedge \\ |p| = \max(0, n-1) \end{pmatrix} \qquad (1)$$

where $\textsc{Find}(q,x)$ returns the index of the first occurrence of q in x ($\textsc{Find}(\epsilon, x) = 1$ and $\textsc{Find}(q,x) = 0$ if q does not occur in x) and $[\![b]\!] \in \{0,1\}$ is the integer value of Boolean variable b, i.e., $[\![b]\!] = 1 \iff b$.

This rewriting is sound but, as we shall see, has three main disadvantages: *(i)* for each Replace constraint, all the constraints of decomposition (1) are added; *(ii)* it may lose precision; *(iii)* it cannot be generalised to Replace-All.

To overcome the above drawbacks we defined a unified Replace propagator which can also handle Replace-Last and Replace-All. The pseudo-code of the propagator is shown in Fig. 4, where x, q, q', y are the variables involved in the constraint and the Boolean flag *LST* (resp. *ALL*) is *true* if we are propagating Replace-Last (resp. Replace-All). Replace* denotes all three of Replace. The Prop-Replace function returns: *(i)* UNS, if Replace* is *infeasible*; *(ii)* OK, if Replace* *may* be feasible (in this case the domains are possibly refined); *(iii)* Rewrite(C) if Replace* is *rewritten* into constraint C.

[3] Push and Stretch are not dual. For example, when incompatible blocks are found, Push moves on (Fig. 2, line 9) while Stretch returns (Fig. 3, line 13).

[4] In [5] the order of Replace arguments is different. In this paper we change the notation to be consistent with SMT-LIB specifications.

```
1:  function Prop-Replace($x$, $q$, $q'$, $y$, $LST$, $ALL$)
2:     $(X, Q, Q', Y) \leftarrow dom(x, q, q', y)$,    $m \leftarrow 0$
3:     if $\gamma(Q) = \{\mathbf{q}\}$ then                                        ▷ $q$ is fixed
4:        if $\mathbf{q} = \epsilon$ then
5:           if $ALL$ then
6:              if $\gamma(X) = \{\mathbf{x}\}$ then                               ▷ $x$ is fixed
7:                 return rewrite( RepAllFixed($\mathbf{x}, \epsilon, q', y$) )
8:           else
9:              if $LST$ then
10:                return rewrite( $y = x \cdot q'$ )
11:             return rewrite( $y = q' \cdot x$ )
12:       if $\gamma(Q') = \{\mathbf{q}'\} \ \wedge\ \mathbf{q} = \mathbf{q}'$ then           ▷ $q'$ is fixed
13:          return Rewrite( $y = x$ )
14:       if $\gamma(X) = \{\mathbf{x}\}$ then                                     ▷ $x$ is fixed
15:          if $ALL$ then
16:             return Rewrite( RepAllFixed($\mathbf{x}, \mathbf{q}, q', y$) )
17:          $n \leftarrow LST$ ? FindLast($\mathbf{q}, \mathbf{x}$) : Find($\mathbf{q}, \mathbf{x}$)
18:          if $n = 0$ then
19:             return Rewrite( $y = \mathbf{x}$ )
20:          else
21:             return Rewrite( $y = \mathbf{x}[..., n-1] \cdot q' \cdot \mathbf{x}[n + |\mathbf{q}|, ...]$ )
22:       $m \leftarrow$ CountMin($\mathbf{q}, X, ALL$)
23:    if $m = 0 \ \wedge\ \neg$Check-Equate($X, Y$) then                          ▷ $x \neq y$
24:       $m \leftarrow 1$
25:    if $m > 0 \ \wedge\ \neg ALL$ then
26:       $C \leftarrow x = p \cdot q \cdot s \ \wedge\ y = p \cdot q' \cdot s \ \wedge\ LST$ ? Find($q, s$) $= 0$ : Find($q, p$) $= 0$
27:       return Rewrite( $C$ )
28: ────────────────────────────────────────────────────────────────────────────
29:    $(es, le) \leftarrow$ Check-Find($Q, X$)
30:    if $es = \bot$ then                                          ▷ $q$ not occurs in $x$
31:       if $m > 0$ then
32:          return UNS
33:       return Rewrite( $x = y$ )
34:    else                                                        ▷ $q$ may occur in $x$
35:       $B \leftarrow (\bigcup X \cup \bigcup Q')^{0, ub(Y)}$
36:       $Y' \leftarrow$ Norm($X[..., es] \cdot (B \cdot Q')^m \cdot B \cdot X[le, ...]$)
37:       if Equate($Y, Y'$) $= \bot$ then
38:          return UNS
39:    $(es, le) \leftarrow$ Check-Find($Q', Y$)
40:    if $es = \bot$ then                                          ▷ $q'$ not occurs in $y$
41:       if $m > 0$ then
42:          return UNS
43:       return Rewrite( Find($q, x$) $= 0 \ \wedge\ x = y$ )
44:    else                                                        ▷ $q'$ may occur in $y$
45:       $B \leftarrow (\bigcup Q \cup \bigcup Y)^{0, ub(X)}$
46:       $X' \leftarrow$ Norm($Y[..., es] \cdot (B \cdot Q)^m \cdot B \cdot Y[le, ...]$)
47:       if Equate($X, X'$) $= \bot$ then
48:          return UNS
49:    $dom(x) \leftarrow X$,    $dom(y) \leftarrow Y$
50:    return OK
```

Fig. 4. Generalised propagator for Replace*. Line 28 separates the particular cases from the general case.

Prop-Replace first retrieves the dashed strings X, Q, Q', Y corresponding to the domains of variables x, q, q', y respectively (line 2). From now on we indicate in bold font the *fixed* variables, i.e., those representing a single, concrete string. We can divide the pseudo-code of Prop-Replace in two main blocks. The first one (until line 28) refers to the *particular* cases, where either q is fixed or $x \neq y$. Here we do not refine domains, but we possibly rewrite Replace*. The second one (after line 28) models instead the *general* case. Here we possibly refine the variables' domain with a sweep-based approach relying on the Equate algorithm.

The "if" statement between lines 3 and 22 handles the cases where q is fixed by rewriting Replace* into basic constraints without any loss of precision. In particular, as shown in Fig. 5, RepAllFixed is used to break down Replace-All(x, q, q') into concatenations when both x and q are fixed. Note that Replace-All assumes *no overlaps*, e.g., Replace-All$(aaaaa, aa, b) = bba$. We assume that ϵ occurs in each position of a string, e.g., Replace$(ab, \epsilon, c) = cab$, while Replace-Last$(ab, \epsilon, c) = abc$, and Replace-All$(ab, \epsilon, c) = cacbc$ (see lines 4–11).

```
function RepAllFixed($\mathbf{x}$, $\mathbf{q}$, $q'$, $y$)
    if $\mathbf{q} = \epsilon$ then
        return $y = q' \cdot \mathbf{x}[1] \cdot q' \cdot \mathbf{x}[2] \cdot q' \cdots q' \cdot \mathbf{x}[\,|\mathbf{x}|\,] \cdot q'$
    $i \leftarrow$ Find$(\mathbf{q}, \mathbf{x})$
    if $i = 0$ then
        return $y = x$
    $j \leftarrow 1$
    $Y \leftarrow [\,]$
    while $i \neq 0$ do
        push-back$(Y, \mathbf{x}[j..i-1])$
        $j \leftarrow i + |\mathbf{q}|$
        $i \leftarrow$ Find$(\mathbf{q}, \mathbf{x}[j, ...])$
    return $y = Y[1] \cdot q' \cdot Y[2] \cdot q' \cdots q' \cdot Y[\,|Y|\,] \cdot q' \cdot \mathbf{x}[j, ...]$
```

Fig. 5. RepAllFixed algorithm.

If $q = q'$ then $x = y$ (line 13), and if x is fixed then we can safely rewrite Replace* into simpler constraints (lines 14–21, note that FindLast(q, x) returns the *last* occurrence of q in x).

In line 22, CountMin (see Fig. 6) is used to count the occurrences of q in x. Precisely, CountMin$(q, X, ALL) = m$ if q surely occurs in each string of $\gamma(X)$ at least m times. If $ALL = true$, we want to maximize the returned value m (for Replace-All); otherwise, CountMin returns 1 if q must occur in $\gamma(X)$ and 0 otherwise (for Replace and Replace-Last). For example, with X as in Fig. 1, we have that: CountMin$(\mathtt{b}, X, true)$ = CountMin$(\mathtt{b}, X, false) = 0$, CountMin$(\mathtt{o}, X, true) = 2$, and CountMin$(\mathtt{o}, X, false) = 1$.

```
function CountMin($\mathbf{q}$, $X$, $ALL$)
    $curr \leftarrow \epsilon$
    $k \leftarrow 0$
    for $i \leftarrow 1, \dots, |X|$ do
        $S^{l,u} \leftarrow X[i]$
        if $S = \{a\}$ then
            $curr \leftarrow curr \cdot a^l$
            $j \leftarrow$ Find($\mathbf{q}, curr$)
            while $j \neq 0$ do
                $k \leftarrow k + 1$
                if $\neg ALL$ then
                    return 1
                $curr \leftarrow curr[j + |\mathbf{q}| + 1, ...]$
                $j \leftarrow$ Find($\mathbf{q}, curr$)
            if $l < u$ then
                $curr \leftarrow a^l$
        else
            $curr \leftarrow \epsilon$
    return $k$
```

Fig. 6. CountMin algorithm.

In line 24, if $m = 0$, we call Check-Equate(X, Y) to check if X and Y are equatable. Check-Equate is a lightweight version of Equate that does not refine X and Y; it returns *false* if X and Y are surely not equatable (i.e., $\gamma(X) \cap \gamma(Y) = \emptyset$), otherwise *true* is returned. Hence, $\neg$Check-Equate(X, Y) means that $x \neq y$ and thus q must occur at least once in x (otherwise we would have $x = y$) so m is set to 1.

If q surely occurs in x and we are not propagating Replace-All, then we can safely rewrite as done in lines 25–27. This also implies a better precision. Consider for example $y =$ Replace(x, bb, ϵ) where the domains of x and y are respectively $X = \{a\}^{2,\lambda}\{b\}^{0,\lambda}$ and $Y = \{a\}^{1,1}\{b\}^{0,\lambda}$ with $a, b \in \Sigma$. We have that $\neg$Check-Equate(X, Y) because X has at least two a's and Y has at most one, so Replace is rewritten in $x = p \cdot bb \cdot s \wedge y = p \cdot s \wedge$ Find(bb, p) $= 0$ with p, s fresh string variables. From $x = p \cdot bb \cdot s$ the concatenation propagator infers that p must start with aa, and then $y = p \cdot s$ fails because y has at most one a.

Note that the decomposition approach of [5] cannot do this because from $n =$ Find(bb, x) one can only infer that $n \notin \{1, 2\}$ so $[\![n > 0]\!]$ stays unknown. This dramatically affects the resolution which, in this case, has to prove unsatisfiability via systematic search. In general, the decomposition approach tends to lose precision when we do not know if $n =$ Find(q, x) > 0: if the Boolean variable $[\![n > 0]\!]$ is unknown, we can say little about $x = p \cdot q^{[\![n>0]\!]} \cdot s$ and $y = p \cdot q'^{[\![n>0]\!]} \cdot s$.

Lines 29–50 handle the general case. Check-Find(X, Y) (see Fig. 7) returns, if feasible, the earliest and the latest position where X can match Y (otherwise $(\perp, \perp)$ is returned). To do so, it uses a pair of vectors ESP, LEP such that, for each $i = 1, \dots, |X|$, $ESP[i]$ and $LEP[i]$ are respectively the earliest start and

```
function CHECK-FIND($X$, $Y$)
    $n \leftarrow |X|$
    $LEP[0] \leftarrow (0, 0)$
    $ESP[n+1] \leftarrow (|Y|+1, 0)$
    for $i \leftarrow 1, \dots, n$ do                                ▷ Initialising positions.
        $ESP[i] \leftarrow (0, 0)$
        $LEP[i] \leftarrow (|Y|+1, 0)$
    $ESP \leftarrow$ PUSH-ESP$(X, Y, ESP)$
    if $ESP = \bot$ then
        return $(\bot, \bot)$
    $LEP \leftarrow$ PUSH-LEP$(X, Y, LEP)$
    if $LEP = \bot$ then
        return $(\bot, \bot)$
    for $i \leftarrow 1, \dots, n$ do
        if $LEP[i-1] \prec ESP[i] \vee ESP[i+1] \prec LEP[i]$ then
            return $(\bot, \bot)$
    return $(ESP[1], LEP[n])$
```

Fig. 7. CHECK-FIND algorithm.

```
function PUSH-ESP($X$, $Y$, $ESP$)
    $gap \leftarrow false$
    $end \leftarrow (0, 0)$
    repeat
        for $i \leftarrow 1, \dots, |Y|$ do
            if $gap$ then
                $end \leftarrow ESP[i]$
            $(ESP[i], end) \leftarrow$ PUSH$^+(X[i], Y, end)$
            if $end = (|Y|+1, 0)$ then
                return $\bot$
        $gap \leftarrow false$
        for $i \leftarrow n, \dots, 1$ do
            $end \leftarrow$ STRETCH$^-(X[i], Y, end)$
            if $ESP[i] \prec end$ then
                $ESP[i] \leftarrow end$
                $gap \leftarrow true$
    until $gap$
```

Fig. 8. PUSH-ESP algorithm.

the latest end position for block $X[i]$ in Y. After ESP and LEP are initialised CHECK-FIND calls PUSH-ESP to possibly improve the ESP positions.

PUSH-ESP (see Fig. 8) uses PUSH$^+$ to possibly push forward the earliest start positions. Variable *end* keeps track of the earliest ends. If *end* for a block $X[i]$ cannot be stretched backward until its earliest start $ESP[i]$ then we have a "gap" between $ESP[i]$ and *end*; in this case we update $ESP[i]$ with *end* and we repeat the procedure until there are no more gaps. PUSH-LEP is symmetrical to PUSH-ESP (it uses PUSH$^-$ for latest end positions).

For example, if $X = \{\mathtt{o},\mathtt{p}\}^{1,3}\{\mathtt{q}\}^{0,1}\{\mathtt{m}\}^{1,2}$ and Y as in Fig. 4, we have Check-Find $(X, Y) = ((2,0),(4,0))$ because X cannot match $\{\mathtt{B},\mathtt{b}\}^{1,1}$ and $\{!\}^{0,3}$.

Assuming that set operations are performed in constant time, the worst-case complexity of Push-ESP is quadratic in the number of blocks because function Push$^+$, costing $O(|X| + |Y|)$ in the worst case, may be called $\max(|X|, |Y|)$ times. Note that, as done e.g. in [7], assuming that set operations cost $O(1)$ is reasonable for string solving because: (1) Σ is constant, and (2) the base of each block it is typically either a small set or a large range. So, with a proper interval representation, the time complexity of set operations can be safely considered $O(1)$ in our context—we experimentally verified in all the empirical evaluations conducted so far that it is highly unlikely to have blocks with a large number of disjoint intervals. Furthermore, even if in principle the worst-case complexity of Push-ESP is $O(|X| \max(|X|, |Y|))$, in practice we experimentally verified that it is almost always $O(\max(|X|, |Y|))$. In other words, gaps are very rare.

If Check-Find(Q, X) returns $(\bot, \bot)$ then q cannot be a substring of x, so it must be that $x = y$ (lines 30–33). Otherwise, in lines 34–38 we try to refine the domain of y. Precisely, we replace the portion of X that can be matched by Q with a sequence of blocks approximating the replacement of Q with Q'. Note that at this stage we cannot know in general if and where Q occurs in X so we have to be conservative by adding a "buffer" block B containing all the characters of X and Q'. The resulting dashed string Y' is the normalised alternate repetition of $m+1$ blocks B and m dashed strings Q'. Once built Y', we call Equate(Y, Y') to possibly refine Y (or we return UNS if Y and Y' are not equatable).[5]

The above reasoning is applied symmetrically to possibly refine X, then OK is returned after refining the domain of x and y with the corresponding dashed strings (lines 49). Unfortunately, there is little room for incremental propagation because—unless fixed—dashed strings can both shrink and expand due to normalization. Also, it is hard to define filtering properties or consistency notions for the dashed string propagators because, as seen in Sect. 2, dashed strings do not form a lattice w.r.t. the $\sqsubseteq$ relation.

Example 1. Consider propagating Replace-All with X as in Fig. 1, $Y = \Sigma^{0,3}$, $Q = \{\mathtt{o},\mathtt{p}\}^{1,3}\{\mathtt{q}\}^{0,1}\{\mathtt{m}\}^{1,2}$, and $Q' = \{\mathtt{r}\}^{2,5}$. Because X cannot match Y, we have $\neg$Check-Equate(X, Y), so $m = 1$, and Check-Find$(Q, Y) = ((2,0),(4,0))$. Once defined the "buffer" block $B = \{\mathtt{B},\mathtt{b},\mathtt{o},\mathtt{m},!,\mathtt{r}\}^{0,3}$ we equate Y and $\{\mathtt{B},\mathtt{b}\}^{1,1}\, B\, \{\mathtt{r}\}^{2,5}\, B\, \{!\}^{0,3}$. The Equate function will thus refine Y into $\{\mathtt{B},\mathtt{b}\}^{1,1}\{\mathtt{r}\}^{2,2}$ (because Y can have at most 3 characters). By reapplying the same logic, X will be refined into $\{\mathtt{B},\mathtt{b}\}^{1,1}\{\mathtt{o}\}^{2,4}\{\mathtt{m}\}^{1,1}$.

Under the reasonable assumption that set operations cost $O(1)$, the overall worst-case complexity of Prop-Replace is dominated by Push-ESP, i.e., $O(|X||Y|)$. However, our experience is that that for almost all the problems we encounter the cost of propagation is linear in the number of blocks.

[5] For simplicity, in Fig. 4 we assume that, when Equate$(X, Y) \neq \bot$, Equate(X, Y) modifies its arguments instead of returning a pair of refined dashed strings (X', Y').

Table 1. Results on Stranger benchmarks.

	MIN	MAX	AVG	PSI
G-Strings	**0.00**	0.11	**0.01**	**100**
G-Decomp	**0.00**	17.16	0.23	**100**
CVC4	**0.00**	**0.06**	**0.01**	**100**
Z3	0.01	T	66.74	78.48
Z3str3	0.00	T	197.60	17.72

4 Evaluation

We extended G-Strings [26], a state-of-the-art string solver extending CP solver Gecode [17] with (dashed) strings solving capabilities, with the Replace propagator defined earlier. We browsed the literature to look for known benchmarks containing Replace, but unfortunately we found that only some of the Stranger benchmarks used in [6] contain it—precisely, only 79 problems (77 satisfiable, 2 unsatisfiable).

We compared G-Strings against state-of-the-art SMT solvers supporting the theory of strings, i.e., CVC4 [23] and Z3 [25] (its default version using the theory of sequences, and the one using the Z3str3 [10] string solver). Note that SMT solvers do not support Replace-Last while Replace-All is only supported by CVC4. We are not aware of how Replace* is handled by these SMT solvers.

We also include in the evaluation G-Decomp, the version of G-Strings decomposing Replace into basic constraints as in Sect. 3. For both G-Strings and G-Decomp we used the default maximum string length $\lambda = 2^{16} - 1$.

Results are shown in Table 1, where we report the minimum (MIN), maximum (MAX) and average (AVG) runtime in seconds to solve an instance (timeout $T = 300$ s is assigned for unsolved instances) and the percentage of solved instances (PSI).[6] As can be seen, these problems are not challenging for CVC4 and the 2 versions of G-Strings: they solve almost all of them instantaneously. G-Strings outperforms G-Decomp on average and for one problem it does considerably better (about 17 s faster) thanks to its tighter propagation.

Conversely, Z3 and Z3str3 are slower. However, the performance of solvers almost certainly depends on the overall problem structure rather than on Replace itself, that always come in the form Replace$(x, \mathbf{q}, \mathbf{q}', y)$ with $\mathbf{q}, \mathbf{q}'$ fixed.

We also evaluated $2 \times 79 = 158$ more instances by replacing all the occurrences of Replace with Replace-All and Replace-Last respectively, but we did not notice any performance difference.

[6] We used a Ubuntu 15.10 machine with 16 GB of RAM and 2.60 GHz Intel® i7 CPU. The benchmarks and the source code of the experiments is available at https://bitbucket.org/robama/exp_cp_2020.

Table 2. Results on encoding benchmarks.

	SAT.				UNS.			
	MIN	MAX	AVG	PSI	MIN	MAX	AVG	PSI
G-Strings	**0.00**	**0.50**	**0.14**	**100**	**0.00**	**0.25**	**0.09**	**100**
G-Decomp	**0.00**	0.87	0.27	**100**	**0.00**	0.53	0.20	**100**
CVC4	0.24	T	56.85	84.00	T	T	T	0.00

To have more insights on Replace we defined the following problem: given a word $\omega \in \Sigma^*$ and a sequence of bits $\beta \in \{0,1\}^*$, find if there exists a non-ambiguous encoding $f : \Sigma \to \{0,1\}^*$ such that $f(\omega[1] \cdots \omega[n]) = \beta$ where $n = |\omega|$. We first modelled this problem as:

$$\begin{aligned}
&X[0] = \omega \;\wedge\; X[n] = \beta \;\wedge \\
&\forall\, i = 1, \ldots, n : X[i] = \text{Replace}(X[i-1], \omega[i], F[i]) \\
&\forall\, 1 \le i < j \le n : \omega[i] = \omega[j] \iff F[i] = F[j]
\end{aligned}$$

where X is an array of string variables in $(\Sigma \cup \{0,1\})^*$ and F an array of string variables in $\{0,1\}^*$ s.t. $F[i] = f(\omega[i])$ and $X[i] = f(\omega[1] \cdots \omega[i])\omega[i+1] \cdots \omega[n]$. We then generated 50 problems (25 satisfiable and 25 unsatisfiable) with random bits β and random words ω having alphabet $\Sigma = \{a, \ldots, z, A, \ldots, Z\}$.

For simplicity, we had to impose a fixed-length 8-bits encoding, i.e., $|F[i]| = 8$ and $|X[i]| = |X[i-1]| + 7$ for $i = 1, \ldots, 25$ and a limited word length $|\omega| = 5i$ to be able to include the SMT solvers in the comparison (otherwise the problem would have been too difficult for them). Unsatisfiable instances were generated with reverse engineering starting from satisfiable instances.

As Table 2 shows, G-Strings clearly achieves the best performance (we omit the results of Z3, solving only 1 instance, and Z3str3, which cannot solve any instance). The performance of G-Decomp is rather close. CVC4 can solve almost half of the satisfiable instances, but it fails to detect unsatisfiability.

To have a more significant comparison between G-Strings and G-Decomp, we generated 150 more instances by varying $|\omega|$ in $\{130, 135, \ldots, 500\}$. Figure 9 shows the cumulative runtime distributions of the two approaches for $|\omega| = 5, 10, \ldots, 500$ (we use cumulative runtimes to better display the distributions). As we can see, as $|\omega|$ grows the performance gap between G-Decomp and G-Strings increases accordingly.

To evaluate Replace-All, we modelled the "most frequent pattern" problem: given a string $x \in \Sigma^*$ and an integer $k > 0$, find the substring q of x with $|q| \ge k$ occurring the most times in x. This generalises the problem of finding the most frequent character (where $k = 1$). We modelled this problem by replacing all the occurrences of q in x with a string of length $|q| + 1$, and then by

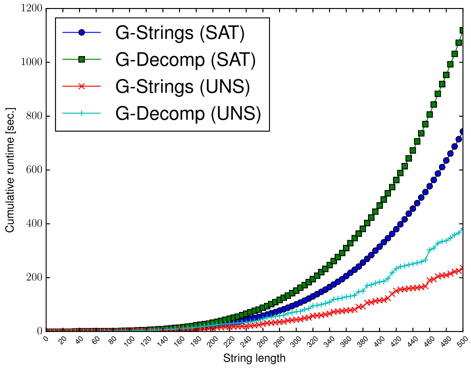

Fig. 9. Cumulative runtimes for G-Strings variants.

Table 3. Results on most frequent pattern benchmarks.

Solver	OPT	SAT	TTF	TIME	SCORE
CVC4	0	**100**	0.09	300.00	49.00
G-Strings	22	**100**	0.06	248.44	80.50
G-Strings$^+$	**72**	**100**	**0.02**	**141.31**	**93.00**

maximising the number of occurrences $|y| - |x|$ of q in x.

$$\begin{aligned} \max(|y| - |x|) \ s.t. \\ k \leq |q| \ \wedge \ |q'| = |q| + 1 \ \wedge \\ y = \textsc{Replace-All}(x, q, q') \ \wedge \ \textsc{Find}(q, x) > 0. \end{aligned}$$

We generated 100 problems by varying $|x| \in \{50i \mid i \in [1, 25]\}$ and $k \in \{1, 2, 4, 8\}$ and we added to the evaluation G-Strings$^+$, a variant of G-Strings which search strategy assigns the minimum value to $\textsc{Find}(q, x)$ integer variable, i.e., it looks for the first occurrence of q in x. We recall that G-Decomp cannot be evaluated because Replace-All is not decomposable into basic constraints.

Table 3 shows for each solver the number of problems where an optimal solution is proven (OPT) or at least one is found (SAT), the average time to find the first solution (TTF) and to prove optimality (TIME), where timeout $T = 300$ s is set if optimality is not proven.

The SCORE metric evaluates the solution quality. For each problem, a solver scores 0 for finding no solution, 0.25 for the worst known solution, 0.75 for the best known solution, a linear scale value in (0.25, 0.75) for other solutions, and 1 for proving the optimal solution (in Table 3 we report the sum of all the scores).

As we can see, the dashed string approach of G-Strings clearly outperforms CVC4, especially when it comes to proving optimality or, in other words, in detecting unsatisfiability.[7] The performance of G-Strings$^+$ confirms the importance of defining good search heuristics for CP solving. In this particular case, we argue that G-Strings$^+$ achieves a better performance w.r.t. G-Strings because its search heuristics somehow mimics a left-to-right search of q in x.

5 Related Work

Although string solvers are still in their infancy, a large number of string solving approaches have been proposed. We can classify them into three rough families: automata-based, word-based, and unfolding-based.

Automata-based approaches use automata to handle string variables and related operations. Examples of automata-based solvers are Stranger [32], PASS [22], StrSolve [20]. The advantage of automata is that they can handle unbounded-length strings and precisely represent infinite sets of strings. However, automata face performance issues due to state explosion and the integration with other domains (e.g., integers).

Word-based solvers are basically SMT solvers treating strings without abstractions or representation conversions. Among the most known word-based solvers, mainly based on the DPLL(T) paradigm [16], we mention: CVC4 [23], the family of solvers Z3str [34], Z3str2 [33], and Z3str3 [10] based on the Z3 solver [25], S3 [30] and its evolutions S3P [30] and S3# [31], Norn [2]. More recent proposals are Sloth [19] and Trau [1]. These solvers can reason about unbounded strings and take advantage of already defined theories. However, most of them are incomplete and face scalability issues due to disjunctive case-splitting.

Unfolding-based approaches basically select a length bound k, and consider each string variable as a vector of k elements. String constraints can be compiled down to bit-vector constraints (e.g., [21,27] solvers) or integer constraints (e.g., [4]). Gecode+$\mathbb{S}$ [28] instead defines dedicated propagators for string constraints. The unfolding approach adds flexibility but may sacrifice high-level relationships between strings, cannot deal with unbounded-length strings and may be very inefficient when the length bound is large—even if the generated solutions have short length. The dashed string approach implemented in G-Strings [7] can be seen as a *lazy unfolding* performing an higher level reasoning over the blocks of the corresponding dashed string domain. The well-known *regular* global constraint [6] is impractical for string solving because it assumes *fixed-length* strings.

6 Conclusions

String solving is of growing interest, since its a basis of many important program analyses. It has already been demonstrated that the G-strings solver is

[7] Since optimization is not a native feature for CVC4 we implemented it by iteratively solving decision problems with refined objective bounds.

highly competitive in solving problems involving strings arising from program analysis [3].

In this paper[8], we devise and implement a unified propagator for handling (variants of) the Replace constraint with dashed strings. Empirical results confirm the effectiveness of such propagator when compared to the decomposition approach and state-of-the-art SMT approaches. While on simple examples the propagator is often not that much better than the decomposition, it scales much better, and allows us to handle Replace-All constraints which cannot be defined by decomposition.

String solving is by no means a closed question, and their is much scope for further work in this direction, in particular: handling more exotic string constraints such as extended regular expression matching, or transducers. An important direction for investigation is to extend a dashed string solver to use nogood learning, which can exponentially reduce the amount of work required. This raises significant questions for nogood learning both in practice and theory.

References

1. Abdulla, P.A., et al.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 602–617 (2017)
2. Abdulla, P.A., et al.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_29
3. Amadini, R., Andrlon, M., Gange, G., Schachte, P., Søndergaard, H., Stuckey, P.J.: Constraint programming for dynamic symbolic execution of JavaScript. In: Rousseau, L.-M., Stergiou, K. (eds.) CPAIOR 2019. LNCS, vol. 11494, pp. 1–19. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-19212-9_1
4. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: MiniZinc with strings. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 59–75. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_4
5. Amadini, R., Gange, G., Stuckey, P.J.: Propagating lex, find and replace with dashed strings. In: van Hoeve, W.-J. (ed.) CPAIOR 2018. LNCS, vol. 10848, pp. 18–34. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93031-2_2
6. Amadini, R., Gange, G., Stuckey, P.J.: Propagating regular membership with dashed strings. In: Hooker, J. (ed.) CP 2018. LNCS, vol. 11008, pp. 13–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98334-9_2
7. Amadini, R., Gange, G., Stuckey, P.J.: Sweep-based propagation for string constraint solving. In: Proceedings 32nd AAAI Conference Artificial Intelligence, pp. 6557–6564. AAAI Press (2018)
8. Amadini, R., et al.: Combining string abstract domains for JavaScript analysis: an evaluation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 41–57. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_3

[8] This work is supported by Australian Research Council through Linkage Project Grant LP140100437 and Discovery Early Career Researcher Award DE160100568.

9. Barahona, P., Krippahl, L.: Constraint programming in structural bioinformatics. Constraints **13**(1–2), 3–20 (2008)
10. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: Stewart, D., Weissenbacher, G. (eds.) Proceedings of 17th Conference Formal Methods in Computer-Aided Design, pp. 55–59. FMCAD Inc. (2017)
11. Bisht, P., Hinrichs, T.L., Skrupsky, N., Venkatakrishnan, V.N.: WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction. In: Proceedings of ACM Conference on Computer and Communications Security, pp. 575–586. ACM (2011)
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. Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. Softw.: Pract. Exp. **45**(2), 245–287 (2015)
14. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 151–162. ACM (2007)
15. Gange, G., Navas, J.A., Stuckey, P.J., Søndergaard, H., Schachte, P.: Unbounded model-checking with interpolation for regular language constraints. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 277–291. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_20
16. 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). https://doi.org/10.1007/978-3-540-27813-9_14
17. Gecode Team: Gecode: generic constraint development environment (2016). http://www.gecode.org
18. Hojjat, H., Rümmer, P., Shamakhi, A.: On strings in software model checking. In: Lin, A.W. (ed.) APLAS 2019. LNCS, vol. 11893, pp. 19–30. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34175-6_2
19. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 4:1–4:32 (2018)
20. Hooimeijer, P., Weimer, W.: StrSolve: solving string constraints lazily. Autom. Softw. Eng. **19**(4), 531–559 (2012). https://doi.org/10.1007/s10515-012-0111-x
21. Kieżun, 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), article 25 (2012)
22. 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
23. 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
24. Loring, B., Mitchell, D., Kinder, J.: ExpoSE: practical symbolic execution of standalone JavaScript. In: Proceedings of 24th ACM SIGSOFT International SPIN Symposium Model Checking of Software (SPIN'17), pp. 196–199. ACM Press (2017)

25. 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
26. Amadini, R.: G-strings: gecode with string variables (2020). https://bitbucket.org/robama/g-strings
27. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proceedings of 2010 IEEE Symposium Security and Privacy, pp. 513–528. IEEE Computer Society (2010)
28. Scott, J.D., Flener, P., Pearson, J., Schulte, C.: Design and implementation of bounded-length sequence variables. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. LNCS, vol. 10335, pp. 51–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59776-8_5
29. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: Search-driven string constraint solving for vulnerability detection. In: ICSE 2017, Buenos Aires, Argentina, 20–28 May 2017, pp. 198–208 (2017)
30. Trinh, M., Chu, D., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: SIGSAC, pp. 1232–1243. ACM (2014)
31. Trinh, M.-T., Chu, D.-H., Jaffar, J.: Model counting for recursively-defined strings. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 399–418. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_21
32. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: an automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_13
33. 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). https://doi.org/10.1007/978-3-319-21690-4_14
34. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a Z3-based string solver for web application analysis. In: Proceedings of 9th Joint Meeting on Foundations of Software Engineering, pp. 114–124. ACM (2013)