Propagating regular membership with dashed strings

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

[1] University of Melbourne, Victoria, Australia
[2] Monash University, Melbourne, Victoria, Australia

Abstract. Using dashed strings is an approach recently introduced in Constraint Programming (CP) to represent the domain of string variables, when solving combinatorial problems with string constraints. One of the most important string constraints is that of regular membership: REGULAR(x, R) imposes string x to be a member of the regular language defined by automaton R. The REGULAR constraint is useful for specifying complex constraints on fixed length finite sequences, and regularly appears in CP models. Dealing with REGULAR is also desirable in software testing and verification, because regular expressions are often used in modern programming languages for pattern matching. In this paper, we define a REGULAR propagator for dashed string solvers. We show that this propagator, implemented in the G-STRINGS solver, is substantially better than the current state-of-the-art. We also demonstrate that many REGULAR constraints appearing in string solving benchmarks can actually be tackled by dashed strings solvers without explicitly using REGULAR.

1 Introduction

String constraint solving is an emerging topic that bases its motivation in fields like web security and software analysis and verification. Suitable solvers have been introduced over the last years for solving combinatorial problems involving string variables and constraints [1, 9, 12, 14, 18–21].

Recent works [6] introduced the *dashed-string* representation for string variables in constraint programming (CP), and described propagation algorithms for equality and related constraints [5], lexicographic ordering and find/replace [4]. A key advantage of this representation is the ability to efficiently represent strings of uncertain – but possibly very large – length by dividing similarly behaved regions of a partially specified string into a sequence of concatenated *blocks*.

A common element of string constraint problems which has not yet been considered by dashed string solvers is the regular language membership constraint REGULAR(x, R), whose semantics is $x \in L(R)$ where x is a string variable and $L(R)$ is the regular language denoted by the finite state automaton R.

Constraint programming treatments of REGULAR typically act on a *fixed-length* sequence of integer variables, and require that running the automaton on this sequence finishes in an accepting state. Sequences of non-fixed (but *bounded*) length are typically padded with a special character to a maximum length in order to use the fixed-length REGULAR propagator. We do not want to adopt this strategy for dashed-strings, as it would totally defeat the advantage of dashed strings that operate effectively on strings whose length bound is large. Instead, we must develop a new propagation algorithm, which operates at the level of *blocks* of characters, rather than individual characters.

In this paper we present such an algorithm and integrate it into G-STRINGS, a constraint programming solver using the dashed-string representation. We evaluate its effectiveness on a range of real-world benchmarks containing regular language constraints, and find that it significantly outperforms existing CP and SMT approaches. We also identify a frequently-occurring subclass of regular languages which can be reformulated as basic string constraints, and evaluate the effect of this substitution.

2 Preliminaries

In this Section we give background notions about dashed strings representation, G-STRINGS solver, automata and regular expressions.

2.1 Dashed Strings

We assume a finite alphabet of symbols Σ. A *string* $w \in \Sigma^*$ is either the empty string ε or of the form cw' where $c \in \Sigma$ is a symbol and $w' \in \Sigma^*$ is a string. Typewriter font is used to denote constant characters $\texttt{c} \in \Sigma$. The *length* $|w|$ of string w is the number of symbols appearing in w. We use array notation to lookup the symbols in a string: $w[i]$ is the i^{th} symbol of string w, with $1 \leq i \leq |w|$.

Let us fix a maximum string length $\lambda \in \mathbb{N}$ and a universe $\mathbb{S} = \bigcup_{i=0}^{\lambda} \Sigma^i$. A *dashed string* of length k is defined by a concatenation of $k > 0$ *blocks* $S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$, where $S_i \subseteq \Sigma$ and $0 \leq l_i \leq u_i \leq \lambda$ for $i = 1, \dots, k$ and $\Sigma_{i=1}^{k} l_i \leq \lambda$. Note that the latter condition does not pose any upper bound to the dashed string length: we might have both $\Sigma_{i=1}^{k} l_i \leq \lambda$ and $k > \lambda$.

For each block $S^{l,u}$, we call S the *base* and (l, u) the *cardinality*. The i-th block of a dashed string X is denoted by $X[i]$, and $|X|$ is the length of X. We do not distinguish blocks from dashed strings of unary length and we consider only *normalised* dashed strings, where the adjacent blocks have distinct bases and the *null block* $\emptyset^{0,0}$ occurs only to denote the empty string. In this way we provide an unique representation for each concrete string $w \in \Sigma^*$.

Let $\gamma(S^{l,u}) = \{x \in S^* \mid l \leq |x| \leq u\}$ be the *language denoted* by block $S^{l,u}$. We extend γ to dashed strings: $\gamma(S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}) = (\gamma(S_1^{l_1,u_1}) \cdots \gamma(S_k^{l_k,u_k})) \cap \mathbb{S}$ (intersection with $\mathbb{S}$ excludes the strings with length greater than λ). A dashed string X is *known* if it denotes a single string: $|\gamma(X)| = 1$. Normalisation entails that each string $w \in \mathbb{S}$ has a unique known dashed string X such that $w = \gamma(X)$.

A block of the form $S^{0,u}$ is called *nullable*, i.e. $\varepsilon \in \gamma(S^{0,u})$. There is no upper bound on the length of a dashed string since an arbitrary number of nullable blocks may occur.

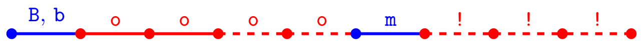

Fig. 1: Graphical representation of $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$.

The *size* $\|S^{l,u}\|$ of a block is the number of concrete strings it denotes, i.e., $\|S^{l,u}\| = |\gamma(S^{l,u})|$. The size of dashed string $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ is an overestimate of $|\gamma(X)|$, given by $\|X\| = \Pi_{i=1}^{k} \|S_i^{l_i,u_i}\|$.

Given dashed strings X and Y we define the relation $X \sqsubseteq Y \Leftrightarrow \gamma(X) \subseteq \gamma(Y)$. Intuitively, $\sqsubseteq$ denotes the relation *"is more precise than"* between dashed strings. Unfortunately, the set of dashed strings does not form a lattice according to $\sqsubseteq$ [6]. For example, there is not a "best" dashed string denoting $\{\texttt{ab}, \texttt{ba}\} \subseteq \Sigma^*$. This implies that some workarounds have to be used to preserve the soundness of propagation. For more details, we refer the reader to [5,6].

Intuitively, we can imagine each block $S_i^{l_i,u_i}$ of $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ as a continuous segment of length l_i followed by a dashed segment of length $u_i - l_i$. The continuous segment indicates that exactly l_i characters of S_i *must* occur in each concrete string of $\gamma(X)$, and defines the *mandatory part* S^{l_i,l_i}. The dashed segment indicates that n characters of S_i, with $0 \leq n \leq u_i - l_i$, *may* occur and defines the *optional part* S_i^{0,u_i-l_i}. Consider, for example, the graphical representation of dashed string $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$ in Fig. 1. Each string of $\gamma(X)$ starts with `B` or `b`, followed by 2 to 4 `o`s, one `m`, then 0 to 3 `!`s.

2.2 G-Strings Solver

Dashed string solving is implemented in G-Strings, an extension of Gecode solver [11]. It implements the domain $D(x)$ of every string variable x with a dashed string, and defines a *propagator* for each string constraint.

Most of the propagators refine the domains of the string variables based on the notion of dashed string *equation*. Equating dashed string X and Y means determining two dashed strings X' and Y' such that: *(i)* $X' \sqsubseteq X$, $Y' \sqsubseteq Y$; and *(ii)* $\gamma(X') \cap \gamma(Y') = \gamma(X) \cap \gamma(Y)$. Informally, we can see this problem as a semantic unification where we want to find a refinement of X and Y including all the strings of $\gamma(X) \cap \gamma(Y)$ and removing the most values not belonging to $\gamma(X) \cap \gamma(Y)$ (there may not exist a greatest lower bound for X, Y according to $\sqsubseteq$). G-Strings uses the sweep-based algorithm of [5] to propagate dashed strings equality. For example, string equality $x = y$ is simply propagated by equating the domains $D(x)$ and $D(y)$; the propagator for $z = x \cdot y$ is implemented by equating $D(z)$ and the concatenation of blocks $D(x) \cdot D(y)$.

G-Strings implements string (dis-)equality, (half-)reified equality, (iterated) concatenation, string domain, length, reverse, substring selection, global cardinality, channeling with integers, lexicographic ordering, find and replace. Since propagation is in general not complete, G-Strings also defines strategies for branching on variables (e.g., the one with smallest domain size or having the domain with the minimum number of blocks) and domain values (by heuristically selecting first a block, and then a character of its base).

2.3 Automata and regular expressions

A finite-state automaton (or simply automaton) is a tuple $R = \langle Q, \Sigma, \delta, q_0, F\rangle$ where Σ is the *alphabet*; Q is a finite set of *states* including the *initial state* q_0 and a set F of *accepting states*; and $\delta \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. A transition $(q, c, q') \in \delta$ from state q to q' is also written as $q \rightarrow q'$. A *computation* of length l for string w is a sequence of l transitions $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_l$ where $\delta(s_{i-1}, w[i]) = s_i$ for $i = 1, \ldots, l$. The string w is *accepted* by automaton R when $|w| = l$, $s_0 = q_0$ and $s_l \in F$. The *language* $L(R)$ of automaton R is the regular language consisting of all the strings of Σ^* accepted by R. If it does not exist an automaton $R' = \langle Q', \Sigma', \delta', q'_0, F'\rangle$ such that $L(R') = L(R)$ and $|Q'| < |Q|$, then R is *minimal*.

An automaton is *deterministic* (DFA) if δ is a (partial) function $Q \times \Sigma \rightarrow Q$. In this case, we can use the notation $\delta(q, c) = q'$ if $(q, c, q') \in \delta$; otherwise, $\delta(q, c) = \perp$ if undefined. A DFA is *trim* if for each $q \in Q$ there exists a computation $q_0 \rightarrow \cdots \rightarrow q$ and a computation $q \rightarrow \cdots \rightarrow q' \in F$.

If δ is a total function, then the DFA is *complete*. If a DFA $\langle Q, \Sigma, \delta, q_0, F\rangle$ is not complete, we can extend Q to $Q' = Q \cup \{q_\perp\}$, and δ to $\delta' = \delta \cup \{(q, c, q_\perp) \mid q \in Q', c \in \Sigma, \delta(q, c) = \perp\}$ in order to have a complete DFA $\langle Q', \Sigma, \delta', q_0, F\rangle$. Given an automata R, the *complement automaton* $\overline{R}$ is an automata such that $L(\overline{R}) = \Sigma^* - L(R)$. Given a complete DFA $R = \langle Q, \Sigma, \delta, q_0, F\rangle$, we can easily compute $\overline{R} = \langle Q, \Sigma, \delta, q_0, Q - F\rangle$ by complementing the final states.

Given a complete automaton $R = \langle Q, \Sigma, \delta, q_0, F\rangle$, a state $q \in F$ is said *universally accepting* if all computations from q bring to a state $q' \in F$ (i.e., any computation reaching q will be accepted). Dually, a state $q \in Q - F$ is said *universally rejecting* if all computations from q bring to a state $q' \in Q - F$ (i.e., any computation reaching q will be rejected).

Let $acc(R)$ and $rej(R)$ be the set of universally accepting and rejecting states of R respectively. If R is minimal, then $|acc(R)|, |rej(R)| \leq 1$. We can efficiently compute the minimum and the maximum length string accepted by R, $minl(R) = \min\{|w| \mid w \in L(R)\}$ and $maxl(R) = \max\{|w| \mid w \in L(R)\}$[1]. Note that $maxl(R)$ may be $+\infty$ (if loops occur) but for our purposes can be at most λ (the maximum allowed string length).

An alternative yet equivalent way to denote a regular language is by means of *regular expressions*. We define inductively the set $\mathcal{RE}$ of regular expressions (over alphabet Σ), as well as the language $L(r)$ denoted by each $r \in \mathcal{RE}$, as: *(i)* $\varnothing \in \mathcal{RE}$, denoting $L(\varnothing) = \emptyset$; *(ii)* if $c \in \Sigma \cup \{\epsilon\}$, then $c \in \mathcal{RE}$ denoting $L(c) = \{c\}$; *(iii)* if $r, r' \in \mathcal{RE}$, then $r \cdot r' \in \mathcal{RE}$ denoting $L(r \cdot r') = L(r)L(r')$, and $r|r' \in \mathcal{RE}$ denoting $L(r|r') = L(r) \cup L(r')$; *(iv)* if $r \in \mathcal{RE}$ then $r^* \in \mathcal{RE}$ denoting $L(r^*) = L(r)^*$; *(v)* nothing else belongs to $\mathcal{RE}$.

Given a regular expression $r \in \mathcal{RE}$, we indicate with DFA the function such that $R = \text{DFA}(r)$ is the minimal automaton such that $L(r) = L(R)$. Note that in our case we actually consider the finite language $L(r) \cap \mathbb{S}$, having strings with length smaller or equal to λ.

[1]Note that $maxl(R) \neq \max\{|w| \mid w \in L(R), |w| \leq \lambda\}$, which is less easy to compute. If $maxl(R) > \lambda$, we set $maxl(R) = \lambda$: this is a correct but not optimal upper bound.

3 Propagating regular on dashed strings

The REGULAR constraint arises fairly frequently in constraint programming problems. The usual CP constraint REGULAR(x, R) constraint takes a fixed description of an automata R, a fixed length sequence x of integer variables, and constrains $x \in L(R)$. This form of REGULAR was introduced in [7] and several propagation algorithms have been developed [10,16,17]. These algorithms *unfold* the regular automaton into a *layered graph* – creating a copy of each automaton state for each variable – which is incrementally updated during search; propagation occurs when there is no longer a viable edge for some value k at some level.

For unfolding-based string constraint solvers like [18], these REGULAR propagators may be used directly. Given automaton $\langle Q, \Sigma, q_0, \delta, F\rangle$, if ϵ is the null symbol used to pad strings of length smaller than λ, it is enough to introduce a fresh state q_ϵ, such that $\delta(q_\epsilon, c) = q_\epsilon$, for each $c \in \Sigma, q \in F \cup \{q_\epsilon\}$. But dashed strings solvers represent a much richer sequence variable x, where crucially we do not know the length of various components. While we could use the unfolding approach for dashed strings, this would defeat their main purpose which is to reason about potentially long strings efficiently by means of a *lazy* approach.

In this paper we also consider the *reified* form of the regular constraint, which is rare in CP but frequent in SMT benchmarks derived, e.g. from security analysis and model checking. This is not surprising since we want also to express more complex constraints like $x \notin L(R)$ or `if` $x \in L(R)$ `then` $A(x)$ `else` $B(x)$.

We then implemented the reified constraint $b \Leftrightarrow$ REGULAR(x, R), where b is a Boolean variable. While in the general case we restrict R to be a complete DFA, if $b = \mathit{true}$, i.e. we just have the positive constraint REGULAR(x, R), the propagation algorithms work for any non-deterministic and non-complete automaton.

3.1 Propagation

Let us propagate $b \Leftrightarrow$ REGULAR(x, R), where $R = \langle Q, \Sigma, q_0, \delta, F\rangle$ is a complete DFA and x is a string variable. We take inspiration from the propagation of the regular global constraint for integer variables.

Before posting the REGULAR constraint itself, we post some bound constraints on the length of x by taking advantage of *minl* and *maxl* functions (see Fig. 2). These constraints may detect early failures or provide additional information. For example, consider $D(x) = \{\mathtt{a}\}^{0,1}\{\mathtt{b}\}^{0,1}$ and $L(R) = \{\mathtt{a}, \mathtt{b}\}$. If $b = \mathit{true}$, then we get $|x| = 1$; otherwise, nothing can be inferred: $0 \leq |x| \leq 2$.

The reified regular constraint propagator, summarised in Fig. 3, takes the current domain $B = D(b)$ of Boolean variable b, the current domain $X = D(x)$ of string variable x, the automata R, and returns a triple $\langle B', X', s\rangle$, where B' (resp., X') is an updated domain for b (resp., x) and s is a Boolean value which determines if the constraint is *subsumed* (i.e, we cannot propagate further). Although we assume R is complete, the pseudo code is also correct for arbitrary automata if we omit the greyed out parts.

The propagation algorithm essentially works in two steps: *(i)* a *forward* pass, where we compute a set of reachable states, potentially detecting inconsistency;

```
function POST-REIFIED-REGULAR($b$, $x$, $R$)
    if $D(b) = \{true\}$ then                                   ▷ positive regular constraint
        POST($minl(R) \leq |x| \leq maxl(R)$)
        POST($true \Leftrightarrow$ REGULAR($x, R$))
    else if $D(b) = \{false\}$ then                             ▷ complemented regular constraint
        POST($minl(\overline{R}) \leq |x| \leq maxl(\overline{R})$)
        POST($true \Leftrightarrow$ REGULAR($x, \overline{R}$))
    else                                                        ▷ general form
        POST($b \Leftrightarrow$ REGULAR($x, R$))
```

Fig. 2: Pre-checks before actually posting $b \Leftrightarrow x \in L(R)$.

```
function PROP-REIFIED-REGULAR($B$, $X = S_1^{l_1,u_1}, \ldots, S_n^{l_n,u_n}$, $R = \langle \Sigma, Q, q_0, \delta, F \rangle$)
    $F_0 \leftarrow [\{q_0\}]$
    for $i \in 1, 2, \ldots, n$ do                                  ▷ forward pass.
        $F_i \leftarrow$ REACH-FWD($B, Q, \delta,$ LAST($F_{i-1}$)$, S_i^{l_i,u_i}$)
        if LAST($F_i$) $\subseteq rej(R)$ then                      ▷ $x$ surely rejected
            return $B \cap \{false\}, X, true$
        if LAST($F_i$) $\subseteq acc(R)$ then                      ▷ $x$ surely accepted
            return $B \cap \{true\}, X, true$
    if LAST($F_n$) $\subseteq F$ then                               ▷ $x$ surely accepted
        return $B \cap \{true\}, X, true$
    if LAST($F_n$) $\subseteq Q - F$ then                           ▷ $x$ surely rejected
        return $B \cap \{false\}, X, true$
    if $B = \{true\}$ then                                          ▷ positive regular constraint
        $E \leftarrow$ LAST($F_n$) $\cap F$
    else if $B = \{false\}$ then                                    ▷ complemented regular constraint
        $E \leftarrow$ LAST($F_n$) $\cap (Q - F)$
    else                                                            ▷ nothing to propagate
        return $B, X, false$
    if $E = \emptyset$ then                                         ▷ $E$ is the set of feasible ending states.
        return $\emptyset, \emptyset, true$
    for $i \in n, n-1, \ldots, 1$ do                                ▷ backward pass.
        $E, X'_i \leftarrow$ REACH-BWD($Q, \delta, F_i, E, S_i^{l_i,u_i}$)
    return $B,$ NORM($[X'_1, \ldots, X'_n]$)$, false$
```

Fig. 3: Propagation algorithm for $b \Leftrightarrow x \in L(R)$.

(ii) a *backward* pass, where only feasible end-states are considered and the domains of the variables are possibly pruned.

The forward pass keeps track of the sets of states F_i reachable by any concrete string in $\gamma(X)$ after consuming block $X[i]$. REACH-FWD returns in particular a sequence $F_i = [Q_{i,0}, \ldots, Q_{i,l_i}, Q_{i,l_i+1}]$ of sets of states where, for $j = 0, \ldots, l_i$, $Q_{i,j}$ is the set of states reachable after consuming *exactly* j characters of $X[i]$ (corresponding to the mandatory part of the block), while Q_{i,l_i+1} is the set of states reachable after consuming an arbitrary number $k \in [l_i, u_i]$ of characters of $X[i]$ (corresponding to the optional part of the block). The last set of states $\textsc{last}(F_i)$ (we assume that LAST is a function returning the last element of a sequence) is used to possibly detect when the constraint is subsumed.

After the loop, if all the states of $\textsc{last}(F_n)$ are accepting, then the constraint must hold (i.e., it is subsumed) and we can propagate $b = true$ (similarly, if they are all rejecting we can propagate $b = false$). We then calculate the set of accepting final states E, and if b is not fixed we return since no propagation is possible. If E is empty we detect unsatisfiability, and return. Otherwise, we iterate backward over the blocks of X and we use REACH-BWD to compute the sets of states that are both reachable and may lead to an accepting state.

At the end of the function, we return the possibly refined domains for variables b and x. Note that, for the latter, we use the NORM function to make the sequence of blocks $[X'_1, \dots, X'_n]$ a normalised dashed string. For example, $\textsc{norm}([\{a,b\}^{0,2}, \emptyset^{0,0}, \{a,b\}^{1,1}]) = \{a,b\}^{1,3}$. In the following we shall explain the forward and backward phases in more details.

```
function REACH-FWD($B$, $\delta$, $Q$, $Q_F$, $S^{l,u}$)
    $\delta_{fwd} \leftarrow \{q \mapsto \{\delta(q,c) \mid c \in S\} \mid q \in Q\}$        ▷ Feasible forward transitions.
    $Q_0 \leftarrow Q_F$
    for $i \in 1, 2, \dots, l$ do                                              ▷ Mandatory region
        $Q_i \leftarrow \bigcup_{q \in Q_{i-1}} \delta_{fwd}(q)$
        if $Q_i = Q_{i-1}$ then                                                ▷ Fixpoint
            $Q_l \leftarrow \cdots \leftarrow Q_{i+1} \leftarrow Q_i$
            return $[Q_0, \dots, Q_l, Q_l]$
        if $Q_i \subseteq rej(R) \vee Q_i \subseteq acc(R)$ then                ▷ Constraint subsumed
            return $[Q_i]$
    $Q_{bfs} \leftarrow \textsc{queue}(Q_l)$
    $dist \leftarrow \left\{ q \mapsto \begin{array}{ll} l & \textbf{if } q \in Q_l \\ +\infty & \textbf{if } q \in Q - Q_l \end{array} \right\}$
    while $Q_{bfs} \neq [\,]$ do                                               ▷ BFS over optional region.
        $q \leftarrow \textsc{pop}(Q_{bfs})$
        $d \leftarrow dist[q] + 1$
        if $d \leq u$ then
            for $q' \in \delta_{fwd}(q)$ where $dist[q'] > d$ do
                $\textsc{push}(Q_{bfs}, q')$
                $dist[q'] = d$
    return $[Q_0, \dots, Q_l, \{q \in Q \mid dist[q] \leq u\}]$
```

Fig. 4: Forward pass of the algorithm. Returns reachable end-states, plus intermediate states needed for the backward pass.

Forward pass The forward pass is implemented by REACH-FWD (see Fig. 4). It computes a sequence of sets of states $[Q_0, Q_1, \dots, Q_l, Q_{l+1}]$ that are reachable after consuming characters in the block $S^{l,u}$. In particular, for $0 \leq i \leq l$, sets

Q_i are those after consuming exactly i characters, while Q_{l+1} collects any states possible after consuming a number of characters between l and u inclusive.

The mandatory part is straightforward. If we find a set of states always rejecting (or accepting in the positive case) we can return since the constraint is subsumed. If instead a set of states Q_i is identical to the previous set Q_{i-1}, then we have reached a *fixpoint*: we are finished since $Q_j = Q_{j-1}$ for $j = i, \dots, l$.

The optional part proceeds by breadth first search (BFS) finding new states reachable in at most $u-l$ characters. We store in *dist* dictionary the least distance to reach any state starting from the states in Q_0. The queue of states Q_{bfs} to expand consists initially of the states of Q_l, i.e. all the states reachable after consuming all the l characters of the mandatory part of $S^{l,u}$.

Note that QUEUE is a function returning a queue containing all the elements of a given set (the order of the element does not matter here). Functions PUSH and POP have the usual semantics. We pop states from Q_{bfs} and if they are less than u distance we push their neighbors onto the queue as long we have found a shorter route to them, updating their distance.

The complexity of REACH-FWD is $O(|\delta| \times (l+1))$, that for a complete DFA corresponds to $O(|Q| \times |\Sigma| \times (l+1))$, since we consider each transition at most once in each iteration of the mandatory region, and at most once in the BFS over the optional region. However, in the case where $b = true$ we could consider a trim DFA δ' with $|\delta'|$ typically far smaller than $|Q| \times |\Sigma|$.

Backward pass The backward pass of the algorithm calculates the states which can both reach a final state, and be reached from the start state. The approach of REACH-BWD (see Fig. 5) is analogous to a reversed forward pass, but uses the stored reachability vectors F_i to compute the intersection.

The first step simulates characters in the optional part of the block. It considers all possible ending states Q_E and adds them to a queue. It maintains the least distance to reach each state in *dist*. When it pops a state in $q \in Q_l$ it updates the least possible distance l' required to reach q. If such a distance is at most $u-l$ we collect the characters of the usable transitions into S_{opt}.

If we have reached a state for the first time, we push it onto the BFS queue Q_{bfs}, and update its distance. This creates the optional block with length at least l' and at most $u-l$, with all characters met in S_{opt}.

The remainder is simpler. Given set of states E we could reach after exactly l characters (and can reach a final state) we collect characters c that might reach these states in S_{man}, and the states q' that we could reach this state from, to initialise E for the next iteration. For this step we create a block of unary length with characters in S_{man}. Finally we return the (normalised) dashed string of these $l+1$ blocks.

The complexity of REACH-BWD is similarly $O(|\delta| \times (l+1))$, so the overall worst-case complexity of PROP-REIFIED-REGULAR is $O(|\delta| \times \Sigma_{i=1}^{n} l_i)$. This means that, apart from $|\delta|$, the complexity of the propagation asymptotically depends on the characters that *must* occur in the string, and not on those that *may* appear. This makes a big difference when λ is big.

```
function REACH-BWD($\delta$, $Q$, $[Q_0, \ldots, Q_l, Q_{l+1}]$, $Q_E$, $S^{l,u}$)
    $\delta_{bwd} \leftarrow \{q \leftarrow \{(c, q') \mid (q', c, q) \in \delta, c \in S\} \mid q \in Q\}$        ▷ Backward transitions
    $S_{opt} \leftarrow \emptyset$
    $Q_{bfs} \leftarrow$ QUEUE($Q_E$)
    $l' \leftarrow +\infty$
    $dist \leftarrow \left\{ q \mapsto \begin{matrix} 0 & \textbf{if } q \in Q_E \\ +\infty & \textbf{if } q \in Q - Q_E \end{matrix} \right\}$
    while $Q_{bfs} \neq [\,]$ do        ▷ BFS over optional region.
        $q \leftarrow$ POP($Q_{bfs}$)
        if $q \in Q_l$ then
            $l' \leftarrow \min(l', dist[q])$
        $d \leftarrow dist[q] + 1$
        if $d \leq u - l$ then
            for $(c, q') \in \delta_{bwd}(q)$ where $q' \in Q_{l+1}$ do
                $S_{opt} \leftarrow S_{opt} \cup \{c\}$
                if $dist[q'] > d$ then
                    PUSH($Q_{bfs}, q'$)
                    $Q_E \leftarrow Q_E \cup \{q'\}$
                    $dist[q'] \leftarrow d$
    $X_{l+1} = S_{opt}^{l', u-l}$
    $E \leftarrow Q_E \cap Q_l$
    for $i \in l, l-1, \ldots, 1$ do        ▷ Mandatory region
        $E' \leftarrow S_{man} \leftarrow \emptyset$
        for $q \in E$ do
            for $(c, q') \in \delta_{bwd}(q), q' \in Q_{i-1}$ do
                $S_{man} \leftarrow S_{man} \cup \{c\}$
                $E' \leftarrow E' \cup \{q'\}$
        $E \leftarrow E'$
        $X_i \leftarrow S_{man}^{1,1}$
    return $E$, NORM($[X_1, \ldots, X_{l+1}]$)
```

Fig. 5: Backward pass of the algorithm. Returns the feasible starting states, and a dashed string corresponding to the refined block.

As mentioned in Section 2, for some set of strings we cannot define a best dashed string representation. It is therefore unlikely to have propagators maintaining consistency notions like, e.g., Generalised Arc Consistency.

If the domain of x has no optional parts, i.e., $D(x) = S_1^{l_1,l_1} \cdots S_n^{l_n,l_n}$, then our approach is equivalent to the "standard" CP propagation of [17], where x corresponds to a vector of $l_1 + \ldots + l_n$ integer variables $x_{i,j}$ such that $D(x_{i,j}) = S_i$ for $i = 1, \ldots, n$ and $j = 1, \ldots, l_i$. This is however not very interesting for string solving, where typically lengths are unknown and potentially very long.

Reverse propagation We can run the regular propagator in reverse, assuming we know $b = true$. In practice, we run PROP-REIFIED-REGULAR($true, X^{-1}, R^{-1}$) by reversing the dashed string $X^{-1} = S_n^{l_n,u_n} \cdots S_1^{l_1,u_1}$ and the automaton $R^{-1} =$

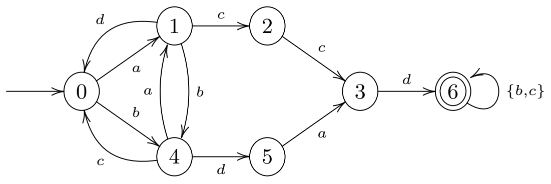

Fig. 6: Example automata R for propagating in Example 1.

$\langle \Sigma, Q, F, \{(q', c, q) \mid (q, c, q') \in \delta\}, \{q_0\}\rangle$. This has a set of initial states F rather than a single state q_0, but this only requires to initialize F_0 with $[F]$.

The reversed automaton is not a DFA hence we must omit the greyed out code. The advantage of the reversed automaton is that because the propagator is directional it may propagate where the other direction does not. Using the reversed automaton effectively doubles the time for propagation, but if it generates more propagation this can substantially reduce the total solving time, hence we leave it on by default in G-Strings (however the user can override this option).

Example 1. Consider the propagation of the positive constraint ($b = \mathit{true}$) when $D(x) = \{\mathtt{a}, \mathtt{b}\}^{0,4}\{\mathtt{c}, \mathtt{d}\}^{2,5}\{\mathtt{a}, \mathtt{b}\}^{0,5}$ and automata given by the DFA shown in Figure 6. The forward propagation determines sets of states $F_1 = [\{0\}, \{0, 1, 4\}]$, $F_2 = [\{0, 1, 4\}, \{2, 5\}, \{3\}, \{3, 6\}]$, and $F_3 = [\{3, 6\}, \{3, 6\}]$.

The backwards pass for block 3 starts from state $\{6\}$ and determines it can reach only set of states $\{6\}$ using $\{\mathtt{b}\}$. It returns $\{6\}$, $\{\mathtt{b}\}^{0,5}$.

The backwards pass for block 2 starts from state $\{6\}$, It sets $\mathit{dist}[6] = 0$ and the rest to infinity. It then determines it can reach 3 at distance 1, 2 at distance 2, and 1 at distance 3 (node 0 at distance 4 is not considered since $4 > 5-2$). Since $3 \in Q_2 = \{3\}$ is only reachable at distance 1, we find $l' = 1$. This creates the optional block $\{\mathtt{c}, \mathtt{d}\}^{1,3}$. We then consider one step backwards from $\{3\}$ which reaches $\{2\}$ and creates block $\{\mathtt{c}\}^{1,1}$, then one step backwards from $\{2\}$, which reaches $\{1\}$ and creates block $\{\mathtt{c}\}^{1,1}$. The function returns $\{1\}, \{\mathtt{c}\}^{2,2}\{\mathtt{c}, \mathtt{d}\}^{1,3}$.

The backwards pass for block 1 starts from state $\{1\}$ and determines it can reach $\{0, 4\}$ at distance 1 and $\{1\}$ at distance 2. We find $l' = 1$ and the function returns $\{0\}, \{\mathtt{a}, \mathtt{b}\}^{1,4}$. So, $D(x)$ becomes $\{\mathtt{a}, \mathtt{b}\}^{1,4}\{\mathtt{c}\}^{2,2}\{\mathtt{c}, \mathtt{d}\}^{1,3}\{b\}^{0,5}$.

Note the propagator is *not idempotent*. Running it again will determine the finer domain $\{\mathtt{a}, \mathtt{b}\}^{1,4}\{\mathtt{c}\}^{2,2}\{\mathtt{d}\}^{1,1}\{\mathtt{c}\}^{0,2}\{\mathtt{b}\}^{0,5}$. Running the reverse propagator will split the first block into $\{\mathtt{a}, \mathtt{b}\}^{0,3}\{\mathtt{a}\}^{1,1}$. We can thus infer that substring `accd` must occur in x. $\square$

4 Regular expressions decomposition

A natural way to express a constraint of the form $x \in L(R)$, where R is an automaton, is to give an equivalent formulation $x \in L(r)$ in terms of an equivalent regular expression $r \in \mathcal{RE}$. We observed that these kind of constraints often occur in SMTLIB instances derived from real-world program analysis.

$$x \in L(\varnothing) \models \mathit{false} \qquad\qquad x \in L(c) \models x = c \quad \text{if } c \in \Sigma \cup \{\varepsilon\} \tag{1}$$

$$x \in L(r_1 \cdot r_2) \models x_1 \in L(r_1) \wedge x_2 \in L(r_2) \wedge x = x_1 \cdot x_2 \tag{2}$$

$$x \in L(r_1 \mid r_2) \models x_1 \in L(r_1) \wedge x_2 \in L(r_2) \wedge n \in \{1,2\} \wedge x = [x_1, x_2][n] \tag{3}$$

$$x \in L((r_1|\dots|r_k)^*) \models x :: \{c_1, \dots, c_k\}^{0,\lambda} \quad \text{if } L(r_i) = \{c_i\} \subseteq \Sigma \text{ for } i = 1, \dots, k \tag{4}$$

$$x \in L(r^*) \models n \in [0, \lambda] \wedge x = w^n \quad \text{if } L(r) = \{w\} \subseteq \Sigma^* \tag{5}$$

Fig. 7: Decomposition rules. Variables x_1, x_2 are new string variables, n a new integer variable, and $r, r_1, r_2, \dots, r_k$ are regular expressions.

We can easily deal with constraints of the form $b \Leftrightarrow x \in L(r)$ by simply propagating $b = \text{REGULAR}(x, \text{DFA}(r))$, where DFA is the function introduced in Section 2.3 for converting a given regular expression into a DFA. However, if $b = \mathit{true}$, we could avoid instantiating a propagator entirely.

First, we observe that dashed strings are themselves a particular class of regular expressions: the language $\gamma(X)$ denoted by $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$ actually corresponds to $L(r) \cap \mathbb{S}$ where $r = r_1^{l_1,u_1} \cdots r_k^{l_k,u_k}$ with $r_i = (c_{i,1}|\dots|c_{i,n_i})$, $S_i = \{c_{i,1}, \dots, c_{i,n_i}\}$, and $r_i^{l,u}$ is a shorthand for $\overbrace{(r_i \cdots r_i)}^{l \text{ times}}\overbrace{((r_i|\epsilon) \cdots (r_i|\epsilon))}^{u-l \text{ times}}$ for $i = 1, \dots, k$. Hence we can directly encode some classes of REGULAR constraints as domain constraints on dashed strings (e.g., the occurrence of characters or substrings in a given string). Moreover, with the help of auxiliary string constraints we can also encode more complex regular expressions. For example, $x \in L((\texttt{fee}|\texttt{foo})\texttt{bar})$ can be reformulated into $x = yz \wedge y \in \{\texttt{fee}, \texttt{foo}\} \wedge z = \texttt{bar}$.

Unfortunately, not all regular expressions are easily decomposable into basic string constraints. For example, we can easily map $x \in L((\texttt{a}|\texttt{b}|\texttt{c})^*)$ into the domain constraint $x :: \{\texttt{a}, \texttt{b}, \texttt{c}\}^{0,\lambda}$ but we cannot do the same for the constraint $x \in L((\texttt{a}|\texttt{bc})^*)$ because this would require to have a propagator for the iterated *concatenation of sets* of strings $\{\texttt{a}, \texttt{bc}\}^n, n \geq 0$.[1]

Hence, we use simple syntactic pre-checks to identify opportunities for decomposing a constraint $\mathit{true} \Leftrightarrow x \in L(r)$ into a conjunction $C_1 \wedge \dots \wedge C_k$ of basic string constraints (if $D(b) \neq \{\mathit{true}\}$, we simply propagate $b = \text{REGULAR}(x, \text{DFA}(r))$). We indicate with $x \in L(r) \models C_1 \wedge \dots \wedge C_k$ such a decomposition.

Fig. 7 summarises the decomposition rules we implemented. Rules 1–2 are straight rewritings into equality/concatenation. Rule 3 encodes the construct $x \in \{x_1, x_2\}$ by means of the ELEMENT global constraint [8]. Rule 4 decomposes into a domain constraint when a regular expression r_i denotes a single character c_i, while rule 5 takes advantage of iterated concatenation when r denotes a single string w. In addition to these rules, we also implemented a number of

[1]Note this is different from the iterated concatenation of strings. For example, encoding $x \in L((\texttt{a}|\texttt{bc})^*)$ with $x = y^n \wedge n \geq 0 \wedge y \in \{\texttt{a}, \texttt{bc}\}$ is unsound because this actually encodes the constraint $x \in L(\texttt{a}^*|(\texttt{bc})^*)$ (e.g., $\texttt{abc} \in \texttt{L}((\texttt{a}|\texttt{bc})^*) - \texttt{L}(\texttt{a}^*|(\texttt{bc})^*)$).

other rules to cope with SMTLIB syntax (e.g., we also decompose $x \in L([a, b])$ and $x \in L([a, b]^*)$, where $[a, b]$ is the range of characters denoted by $(a| \dots |b)$.

Note that this reformulation does not detect all opportunities for reformulation. Indeed, deciding DFA *primality* – that is, whether there exist non-trivial L_1, L_2 such that $L(R) = L_1 L_2$ for a given DFA R – is PSPACE-hard [15], so an efficient complete method is vanishingly unlikely. Nevertheless, devising a 'good enough' decomposition method remains an interesting challenge.

As we shall see in Section 5, decomposing a regular expression r can be advantageous since it can significantly reduce the number of states of $\textsc{dfa}(r)$, especially when dealing with expressions involving very long fixed strings, and the number of propagations performed. Let us clarify this by providing an example extracted from the empirical evaluation of Section 5.

Example 2. Consider the constraint $x \in L(\mathtt{a}^*\mathtt{bb}^*) \wedge x \in L((\mathtt{a}|\mathtt{b})^*\mathtt{ba}(\mathtt{a}|\mathtt{b})^*)$, which is clearly unsatisfiable since the `ba` sub-string in the second expression conflicts with the first expression, imposing each character `b` to be followed by only `b`'s.

If we do not decompose the regular expressions, the propagation algorithm is only able to infer that $x :: \{\mathtt{a}, \mathtt{b}\}^{2,\lambda-1}\{\mathtt{b}\}^{1,1}$ from the corresponding DFAs. This entails that we have to branch on x and explore all the possible alternatives to detect the unsatisfiability: the solving time clearly depends on λ.

Conversely, by decomposing the expressions into basic string constraints we can infer that $x :: \{\mathtt{a}\}^{0,\lambda-1}\{\mathtt{b}\}^{1,\lambda} \wedge x :: \{\mathtt{a}, \mathtt{b}\}^{0,\lambda-2}\{\mathtt{b}\}^{1,1}\{\mathtt{a}\}^{1,1}\{\mathtt{a}, \mathtt{b}\}^{0,\lambda-2}$. In this case, no search is performed because the sweep-based equate algorithm [5] implemented in G-Strings instantaneously triggers a failure since the two dashed strings are not equatable. $\square$

5 Evaluation

We implemented the REGULAR propagator in the G-Strings solver, and we also implemented a MiniZinc/FlatZinc interface for it [3]. The user can either specify a (reified) REGULAR constraint in terms of a regular expression or a finite state automaton. We tested G-Strings on three well-known SMTLIB string benchmarks containing regular expressions:

- AppScan: 8 satisfiable instances derived from security analysis performed by IBM AppScan tool [13]. We discarded two of them (namely, `t01` and `t06`) since they do not contain regular. The remaining 6 instances contain only decomposable and non-reified regular expressions.
- Stranger: 3392 instances derived by Stranger [22] tool from real-world PhP programs. These instances are not publicly available and the authors sent them to us privately, but 56 of them were malformed. We thus ended up with 3336 SMTLIB instances, containing only decomposable and non-reified regular expressions.
- Norn: 1027 instances generated by a model checker based on CEGAR refinement [2], from which we discarded 24 instances not containing regular

Table 1: AppScan results. Times are in seconds.

Instances	`t02`	`t03`	`t04`	`t05`	`t07`	`t08`
G-Strings	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
Z3str3	0.10	0.29	0.38	0.97	2.09	0.02
CVC4	0.01	6.12	T	5.75	**0.00**	0.27

Table 2: Stranger results. Times are in seconds.

	SAT		*UNS*		*TOT*	
	Solved	Runtime	Solved	Runtime	Solved	Runtime
G-Decomp	**1254**	**0.20**	**2082**	**0.00**	**3336**	**0.08**
G-NotDec	**1254**	0.65	**2082**	0.01	**3336**	0.25
CVC4	1247	2.19	**2082**	0.01	3329	0.88
Z3str3	936	76.27	**2082**	0.09	3018	28.68

expressions. In the remaining 1003 instances, 942 contain at least a non-decomposable expression and 870 contain at least a reified regular expressions (all of these are negated expressions of the form $\mathit{false} \Leftrightarrow x \in L(r)$).

We compared G-Decomp (the version of G-Strings that always unfolds decomposable, not-reified regular expressions using the method of Section 4) and G-NotDec (the version that never unfolds them) against three state-of-the-art string solvers: two SMT solvers supporting the theory of strings (namely, Z3str3 [9] and CVC4 [14]), and a CP solver that extends Gecode for supporting bounded-length string variables, i.e., Gecode+$\mathbb{S}$ [18].[1]

Note that Gecode+$\mathbb{S}$ is no longer actively developed. So, since its FlatZinc support is incomplete, we used a compiler from SMTLIB to C++ that the authors used for their previous experiments (here we call it `smt2cpp`). For G-Decomp and G-NotDec we instead took advantage of the SMTLIB to MiniZinc compiler introduced in [4]. We run all the experiments on a Ubuntu 15.10 machine with 16 GB of RAM and 2.60 GHz Intel® i7 CPU by setting a solving timeout of $T = 300$ seconds and varying $\lambda \in \{500, 1000, 10000\}$.

5.1 AppScan and Stranger benchmarks

Results on AppScan benchmark are shown in Table 1. Gecode+$\mathbb{S}$ is not included in the comparison since `smt2cpp` could not process these instances (statements like logical implication and `if-then-else` are not supported).

AppScan is not a challenging benchmark: SMT solvers can solve most of the problems in a short time, while G-Strings resolution is instantaneous. Here we

[1]We used Z3str3 4.6.2 and CVC4 1.5. The source code of the experiments is publicly available at: `https://bitbucket.org/robama/exp_cp_2018`.

do not discriminate between G-Decomp and G-NotDec since they both find a solution in 0 seconds, regardless of maximum length λ.

Results on the Stranger benchmark are shown in Table 2. Solving time is set to T when a solver can not solve an instance. Note that we are considering the simplified instances used also in [12], where the `str.replaceall` operation is replaced by `str.replace`. Gecode+$\mathbb{S}$ is not included in the results because of unsupported constraints (e.g., string replacement) and characters (all the Stranger instances contain extended ASCII characters, while the alphabet size of Gecode+$\mathbb{S}$ is limited to 64 characters).

For G-Decomp and G-NotDec, only the results with $\lambda = 10000$ are presented: here strings can be very long, so with $\lambda = 500$ we have 107 unsound results: we instantaneously detect the unsatisfiability because at least one string has length greater than 500. With $\lambda = 1000$, we have 50 unsound results. For both G-Decomp and G-NotDec, we branched on binary variables first.

The two versions of G-Strings outperform the SMT solvers, especially on the satisfiable instances (the unsatisfiable ones appear very easy to solve). We can also observe the benefits of decomposition (all the regular constraints of this benchmark can be rewritten into concatenation constraints). On average, G-Decomp is more than three times faster than G-NotDec.

5.2 Norn benchmark

Results on the 1003 instances of the Norn benchmark are shown in Table 3. Note that `smt2cpp` can process only 150 of them, mainly because Gecode+$\mathbb{S}$ does not support negated regular expressions. CVC4 and Z3str3 work on unbounded strings, so they do not use a maximum string size.

The results clearly show that G-Strings is faster and more powerful than alternative approaches. The advantages of decomposition are also illustrated, although the performance of G-NotDec and G-Decomp is not so different (in particular on satisfiable instances they are equivalent).

Conversely to the Stranger benchmark here satisfiable instances are trivial, while unsatisfiable ones are harder to solve. This is in general not surprising for CP solvers – especially those like Gecode not employing nogood learning – and this in particular holds for G-Strings, which is based on Gecode and for which the resolution is obviously influenced by λ size. As an example, let us consider the only instance that no solver can solve within the time limit T.[1] This unsatisfiable instance is hard to solve since it contains a pattern of the form:

$$x, y :: \{\mathtt{b}\}^{1,\lambda} \ \wedge \ x \cdot \mathtt{z} \cdot y \in L((\mathtt{bz}^*\mathtt{b})^*) \ \wedge \ x \cdot \mathtt{z} \cdot y \cdot \mathtt{b} \notin L((\mathtt{bz}^*\mathtt{b})^*\mathtt{b})$$

with x, y string variables and $\mathtt{b}, \mathtt{z}$ characters of Σ. Unfortunately, our propagation algorithms cannot further narrow the domains of x and y, and thus we have to rely on branching. This means that $O(|D(x) \times D(y)|) = O(\lambda^2)$ nodes must be explored to detect the inconsistency.

[1] Precisely, this is the instance 489 of the `HammingDistance` class. G-Decomp with $\lambda = 500$ takes 454.6 seconds to detect the unsatisfiability. Clearly, we can only prove that it does not exist a solution if all the string variables x have length $|x| \leq \lambda$.

Table 3: Norn results. Times are in seconds.

	Solver	G-Decomp			G-NotDec			CVC4	Z3str3	Gecode+$\mathbb{S}$		
	λ	500	1000	10000	500	1000	10000			500	1000	10000
SAT	Solved	**688**	**688**	**688**	**688**	**688**	**688**	627	178	75	75	43
SAT	Runtime	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	29.82	223.10	267.65	268.43	283.90
UNS	Solved	**314**	**314**	312	312	309	307	182	89	41	41	25
UNS	Runtime	**0.96**	1.00	3.81	4.82	5.75	7.62	123.64	214.02	260.3	261.54	276.88
TOT	Solved	**1002**	**1002**	1000	1000	997	995	809	267	116	116	68
TOT	Runtime	**0.30**	0.31	1.20	1.51	1.81	2.39	58.72	265.38	266.31	248.31	281.74

Table 4: Norn results on the 116 instances that Gecode+$\mathbb{S}$ can solve.

Solver	G-Decomp			G-NotDec			Gecode+$\mathbb{S}$			CVC4	Z3str3
λ	500	1000	10000	500	1000	10000	500	1000	10000		
Solved	**116**	**116**	**116**	**116**	115	115	**116**	**116**	68	104	46
Runtime	**0.00**	**0.00**	**0.00**	1.20	2.59	2.59	0.69	8.69	142.09	32.72	181.41

Gecode+$\mathbb{S}$ is the closest approach to G-Strings. It implements the CP approach of [17] with a termination character for strings with length less than λ. If we consider only the 116 instances that Gecode+$\mathbb{S}$ can correctly solve (see Table 4) G-Strings is on average still faster and, as already observed in [5, 6], its performance decay is less pronounced as λ grows.

6 Conclusion

We have presented a propagation algorithm for enforcing regular constraints over dashed strings. Unlike existing propagators for regular, the algorithm runs in time independent of the upper bound on the string length. We have also identified a sub-class of regular expressions which may be translated directly into dashed string domain constraints.

We have demonstrated the effectiveness of the propagator on three sets of existing string constraint problems. On these benchmarks, G-Strings is considerably faster and more robust than existing solvers.

We also defined the first propagator we are aware of for reified regular. The same modifications we use here could be adapted to create a reified regular propagator on fixed length arrays, as is standard in CP.

Acknowledgments

This work is supported by the Australian Research Council (ARC) through Linkage Project Grant LP140100437 and Discovery Early Career Researcher Award DE160100568.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: 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, June 18-23, 2017. pp. 602–617 (2017)
2. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: CAV. LNCS, vol. 9206, pp. 462–469. Springer (2015)
3. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: Minizinc with strings. In: Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2016 (2016), https://arxiv.org/abs/1608.03650
4. Amadini, R., Gange, G., Stuckey, P.J.: Propagating lex, find and replace with dashed strings. In: To appear in Fifteenth International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming. LNCS, Springer (2018)
5. Amadini, R., Gange, G., Stuckey, P.J.: Sweep-based propagation for string constraint solving. In: To appear in AAAI 2018 (2018)
6. Amadini, R., Gange, G., Stuckey, P.J., Tack, G.: A novel approach to string constraint solving. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10416, pp. 3–20. Springer (2017)
7. Barták, R.: Modelling resource transitions in constraint-based scheduling. In: SOFSEM 2002: Theory and Practice of Informatics, 29th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 22-29, 2002, Proceedings. pp. 186–194 (2002)
8. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present and future. Constraints 12(1), 21–62 (March 2007), the catalogue is available at `http://sofdem.github.io/gccat/`
9. Berzish, M., Zheng, Y., Ganesh, V.: Z3str3: A string solver with theory-aware branching. CoRR abs/1704.07935 (2017), `http://arxiv.org/abs/1704.07935`
10. Cheng, K., Yap, R.: Maintaining generalized arc consistency on ad hoc r-ary constraints. In: 14th International Conference on Principles and Process of Constraint Programming. LNCS, vol. 5202, pp. 509–523 (2008)
11. Gecode Team: Gecode: Generic constraint development environment (2016), available at `http://www.gecode.org`
12. 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)
13. IBM: Security AppScan (2018), available at `https://www.ibm.com/security/application-security/appscan`
14. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: CAV. LNCS, vol. 8559, pp. 646–662. Springer (2014)
15. Martens, W., Niewerth, M., Schwentick, T.: Schema design for XML repositories: complexity and tractability. In: Paredaens, J., Gucht, D.V. (eds.) Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA. pp. 239–250. ACM (2010)

16. Perez, G., Régin, J.: Improving GAC-4 for table and MDD constraints. In: O'Sullivan, B. (ed.) Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8656, pp. 606–621. Springer (2014)
17. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 3258, pp. 482–495. Springer-Verlag (2004)
18. Scott, J.D., Flener, P., Pearson, J., Schulte, C.: Design and implementation of bounded-length sequence variables. In: Fourteenth International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming. LNCS, Springer (2017)
19. Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. ACM Transactions on Software Engineering Methodology 22(4), 33 (2013)
20. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: Search-driven string constraint solving for vulnerability detection. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. pp. 198–208 (2017)
21. Trinh, M., Chu, D., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: SIGSAC. pp. 1232–1243. ACM (2014)
22. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for PHP. In: TACAS. LNCS, vol. 6015, pp. 154–157. Springer (2010)