Bounded Strings for Constraint Programming

Joseph D. Scott, Pierre Flener, and Justin Pearson
Department of Information Technology
Uppsala University, Sweden
Email: first.last@it.uu.se

Abstract—We present a domain for string decision variables of bounded length, combining features from fixed-length and unbounded-length string solvers to reason on an interval defined by languages of prefixes and suffixes. We provide a theoretical groundwork for constraint solving on this domain and describe propagation techniques for several common constraints.

Index Terms—constraint programming; string constraints

I. INTRODUCTION

Constraints over strings occur in a wide variety of real-world problems, especially in fields such as test case generation [1], program analysis [2], model checking [3], and web security [4]. In recent years several methods for solving string constraints have appeared, which may be broadly classified based on their treatment of string length.

For constraints on unbounded-length strings, solvers typically work by reasoning on automaton representations; see for example the regular domains of [5]. To avoid exponential blowup, these solvers are frequently restricted to a small set of constraints, such as the tools SUSHI [6] and DPRLE [7]. Recent work also includes tools such as REVENANT [3], which uses SAT and interpolation to check intersections of regular languages defined as symbolic automata; and STRSOLVE [8], a dedicated solver that handles expensive automata intersection operations by lazily constructing cross-products. These tools rely exclusively on automata operations; constraints that would be handled using simpler methods in other solvers (such as integer arithmetic constraints on string lengths) are encoded into automata, resulting in increased computational complexity.

Another approach is to solve constraints on strings of a fixed length. For example, the HAMPI tool provides a theory of fixed-length strings for satisfaction modulo theories (SMT) solvers, using the bit-vector solver STP to solve problems of a single string variable [9]. The bit-vector approach has also been explored in constraint programming (CP) [10]. More commonly, however, fixed-length approaches in CP reason at the level of symbols, providing constraints for regular [11], [12] or context-free [13], [14] languages that operate on a sequence of symbol variables. In this approach, string constraints are implemented in existing CP frameworks, allowing solvers to easily combine constraints on strings with constraints on individual symbols and other non-string variables.

A possibility lying between these two extremes is to handle strings with lengths that are not fixed, but bounded by an integer interval. Probably the best known bounded-length solver is KALUZA [15], which solves constraints in two stages. In the first, an SMT solver is used to find possible lengths for strings which satisfy explicit length constraints, length constraints implied by the string constraints of the problem, and any other integer constraints present in the problem. In the second stage, these lengths are applied to create a fixed-length bit-vector problem, solved with STP in the same manner as HAMPI. If the problem in the second stage is unsatisfiable, the first stage is repeated, with the addition of new constraints to avoid previously tried lengths. Other work, such as [16], [17], handles bounded-length strings by adding a string length reasoning component to unbounded automata approaches. In CP, [18] illustrates propagation for bounded-length versions of the fixed-length REGULAR and CFG constraints.

We propose a new bounded-length approach in CP. We begin by defining variables and constraints for bounded-length strings. We then describe the affix domain for bounded-length strings, which is based on languages of prefixes and suffixes. This domain allows us to reason about the content of string suffixes, even when the length of the string is unknown. We then describe propagation techniques for affix domains over constraints of string equality, concatenation, indexing, substring, reversal, and regular language membership.

This new domain fits into the framework of existing CP solvers, thereby making available the large number of constraints over scalar and set variables which already exist. Combined with these pre-existing non-string constraints, this new domain and its attendant constraints yield a solver at least as expressive as that handled by other bounded-length solvers. Indeed, it is generally more expressive, as it freely combines string and numeric reasoning, without restrictions on the number of variables or types of arithmetic constraint.

II. PRELIMINARIES

A constraint satisfaction problem (CSP) [19] is a tuple \((X, D, C)\) of decision variables \(X = \langle X_1, \ldots, X_n \rangle\), domains \(D = \langle D_1, \ldots, D_n \rangle\) such that \(X_i \in D_i\), and constraints \(C\). A constraint \(C_j\) is a pair \((R_j, S_j)\) where \(R_j\) is a relation on the variables \(S_j \subseteq X\), called the scope of \(C_j\). An assignment is a tuple \(A = \langle a_1, \ldots, a_n \rangle\); \(A\) is a solution of the CSP if all \(a_i \in D_i\), and every \(C_j\) is satisfied, i.e., the projection of \(A\) onto \(S_j\) is in \(R_j\). \(A(X_i)\) denotes the projection of \(A\) onto \(X_i\).

In sequel, we denote decision variables in uppercase (\(N\)) to distinguish them from mathematical variables in lowercase (\(n\)). We refer to the current domain of a decision variable \(N\) as \(D(N)\), and, for an integer decision variable \(N\), to the lower and upper bounds of the domain as \(N\) and \(\overline{N}\), respectively, and denote the original upper bound as \(N_{\text{orig}}\). We indicate
arrays of mathematical variables or scalar decision variables with vector notation \((\vec{c}, \vec{A})\), and string decision variables with bold-face \((X)\). We denote an element of an array with the index in brackets \((\vec{A}[i])\). We use the notation \([t, u]\) to refer to the set of integers \(\{t, t+1, \ldots, u-1, u\}\).

An alphabet \(\Sigma\) is a finite set of symbols. A string \(w\) of length \(|w| = n\) over the alphabet \(\Sigma\) is a finite sequence of elements of \(\Sigma\), denoted \(w_1w_2 \cdots w_n\), where \(w_i \in \Sigma\) for all \(1 \leq i \leq n\). The set of all strings over \(\Sigma\) is denoted by \(\Sigma^*\). A possibly infinite subset of \(L \subseteq \Sigma^*\) is called a language over \(\Sigma\). Given a language \(L\) over \(\Sigma\), for a string \(w \in L\) we say \(p\) is a prefix of \(w\) if there exists a string \(x \in \Sigma^*\) such that \(w = px\) (where \(px\) denotes the concatenation of \(p\) and \(x\)). Similarly, \(s\) is a suffix of \(w\) if there exists a string \(y \in \Sigma^*\) such that \(w = ys\); we refer to prefixes and suffixes collectively as affixes. The reversal of a word \(w\) is the word \(w^{rev} = w_n \cdots w_2w_1\). The reversal of a language \(L\) is the set \(L^{rev} = \{w^{rev} | w \in L\}\).

### III. BOUNDED-LENGTH STRINGS

In this section, we describe a representation for string decision variables of bounded length, and discuss constraints on them. In Section IV we introduce a representation which we argue is more useful for propagation; however, it is convenient to specify constraints on bounded-length string decision variables in terms of the simpler representation introduced here.

#### A. Decision Variables

We will refer to unknown strings, over an arbitrary alphabet \(\Sigma\), which occur in the model of a constraint satisfaction problem, as string decision variables. Abstractly, the domain of a string decision variable is a subset of \(\Sigma^*\); however, we require a concrete representation. In [5], automata serve as a concrete representation, allowing for domains that are possibly infinite sets of strings; propagation is defined in terms of automata operations. Another choice of representation is a fixed-length sequence of scalar decision variables over \(\Sigma\) [11], [12]. The latter representation has been more widely adopted, primarily because of tighter integration with existing finite-domain solvers; additional constraints are stated over individual decision variables from the sequence in a natural way. In contrast, a value of a decision variable in [5] is a string, which does not readily facilitate interaction between individual symbols and other types of decision variables.

As we propose to treat bounded-length strings, it is natural to consider a representation based on the one used for bounded open constraints [18]. A constraint is global [20] if the cardinality of its scope is not determined \textit{a priori}. In a closed global constraint, the cardinality of the scope is determined by the model, and remains constant throughout the solution process; however, in an open global constraint, the cardinality of the scope is determined as the solution procedure progresses [21].

In [18], the scope of an open global constraint is a sequence of scalar decision variables with a length that is bounded by an integer decision variable. This formulation leads to a string representation we will call the open sequence representation: \((\vec{A}, N)\) consists of an array \(\vec{A}\) of scalar decision variables over \(\Sigma\), and an integer decision variable \(N\) representing the length of the string. In contrast to [18], we treat the sequence and length together as a representation of a single, bounded-length string decision variable, a syntactical difference that will be convenient when we define constraints over multiple variables in the next subsection. The domain of \((\vec{A}, N)\) is defined in terms of the domains of the pair:

\[
D((\vec{A}, N)) = \bigcup_{n \in \mathbb{N}} \{a_1 \ldots a_n | \forall i \in [1, n]: a_i \in D(\vec{A}[i])\}
\]

#### B. Simple Constraints

Semantically, the relation of a constraint may be defined as a set of tuples over the scope of the constraint. We define string constraints in the open sequence representation.

The constraint \(\text{EQUAL}(X, Y)\) enforces string equality for string decision variables \(X\) and \(Y\), defined as strings having equal length and the same symbol at each index:

\[
\text{EQUAL}\left((\vec{A}_x, N_x), (\vec{A}_y, N_y)\right) = \left\{ (\vec{b}, p, \{c, q\}) \mid p \in D(N_x) \land q \in D(N_y) \land p = q \land \forall i \in [1, p]: \begin{align*}
\vec{b}[i] &\in D(\vec{A}_x[i]) \land c[i] \in D(\vec{A}_y[i]) \land e[i] = c[i]\end{align*}\right\}
\]

The constraint \(\text{REVERSE}(X, Y)\), for string decision variables \(X\) and \(Y\), states that \(X\) is equal to the reverse of \(Y\) (the specification differs from \(\text{EQUAL}\) only in the indexing of \(Y\)):

\[
\text{REVERSE}\left((\vec{A}_x, N_x), (\vec{A}_y, N_y)\right) = \left\{ (\vec{b}, p, \{c, q\}) \mid p \in D(N_x) \land q \in D(N_y) \land p = q \land \forall i \in [1, p]: \begin{align*}
\vec{b}[i] &\in D(\vec{A}_x[i]) \land c[i] \in D(\vec{A}_y[i]) \land e[i] = \vec{c}[p-i+1]\end{align*}\right\}
\]

The constraint \(\text{CONCAT}(X, Y, Z)\) states that string decision variable \(Z\) equals the concatenation of string decision variables \(X\) and \(Y\):

\[
\text{CONCAT}\left((\vec{A}_x, N_x), (\vec{A}_y, N_y), (\vec{A}_z, N_z)\right) = \left\{ (\vec{b}, p, \{c, q, d, r\}) \mid p \in D(N_x) \land q \in D(N_y) \land r \in D(N_z) \land p + q + r \in \mathbb{N} \land \forall i \in [1, p]: \begin{align*}
\vec{b}[i] &= d[i] \land \vec{b}[i] \in D(\vec{A}_z[i]) \\
\vec{c}[j] &= d[j+p-1] \land \vec{c}[j] \in D(\vec{A}_y[j]) \\
\forall j \in [1, q]: \begin{align*}
\vec{d}[j] &= \vec{c}[j]\end{align*}\right\}
\]

The constraint \(\text{SUBSTRING}(X, Y, I)\) states that string decision variable \(Y\) is a contiguous substring of string decision variable \(X\), starting at the index given by the integer decision variable \(I\). The special case when \(Y\) has a length of one (i.e., where the second decision variable is scalar) is defined as \(\text{CHARACTER}(X, C, I)\):

\[
\text{SUBSTRING}\left((\vec{A}_x, N_x), (\vec{A}_y, N_y), I\right) = \left\{ (\vec{b}, p, \{c, q, r\}) \mid p \in D(N_x) \land q \in D(N_y) \land r \in D(I) \land p \geq q + r - 1 \land \forall i \in [1, p]: \begin{align*}
\vec{b}[i] &\in D(\vec{A}_x[i]) \land \\
\vec{c}[j] &\in D(\vec{A}_y[j]) \land r + j - 1 = c[j]\end{align*}\right\}
\]
The affix domain of \( X \) is the concatenation of the affix languages:

\[
\mathcal{D} \left( \langle \vec{P}, P, \vec{S}, S \rangle \right) = \left\{ ps \mid p \in \mathcal{L}(\vec{P}, P), s \in \mathcal{L}(\vec{S}, S) \right\}
\]

As strings in \( \mathcal{L}(\vec{S}, S) \) are constructed from \( \vec{S} \) by decreasing index, the first symbol in \( \vec{S} \) is the last symbol of any word in \( \mathcal{D}(X) \). Each assignment of the affix representation is mapped to a unique open sequence representation assignment via the semantic function:

\[
\left\langle \langle \vec{P}, P, \vec{S}, S \rangle \right\rangle = \left\langle \langle \vec{P}[1], \ldots, \vec{P}[P], \vec{S}[s], \ldots, \vec{S}[1] \rangle, P + S \right\rangle
\]

In the examples which follow, we take as our alphabet the set of symbols corresponding to the lowercase, English characters, denoted \( \mathcal{A} \), etc. We write \( [p-s] \) to represent the lexicographic interval \( \{ p, q, r, s \} \). A singleton lexicographic interval \( \{ p \} \) or integer interval \( \{ 1 \} \) we denote as a bare element, i.e., \( p \) or \( 1 \). We denote a concrete string as ‘example’.

Example 1 Consider a concrete affix representation:

\[\alpha = \langle \langle [p-s], [a-z], [a-z], [f-x] \rangle, [1, 3], \langle x, i, f, [f-x] \rangle, [2, 4] \rangle\]

Let \( A \) be a string decision variable, where the current domains of the components of the affix representation are the components of \( \alpha \). Here \( \vec{P}^\alpha \) has 4 elements, corresponding to the original upper bound on the length of strings in the prefix language, as opposed to \( \vec{P}^\alpha = 3 \), which is the current upper bound on that length. The strings ‘prefix’ and ‘suffix’ are both in \( \mathcal{D}(A) \), while ‘affix’ is not (since \( \not\in [p-s] \)). Also, the prefix language of \( A \) does not include the string ‘post’, because its length exceeds \( \vec{P}^\alpha = 3 \); nevertheless, ‘postfix’ is in \( \mathcal{D}(A) \), as a concatenation of ‘pos’ and ‘tfix’.

The semantic function is surjective, but not injective: a string defined in the open sequence representation may have several equivalent definitions in the affix representation. This is, however, sufficient for the purposes at hand; namely, propagators which act on the underlying affix representation may, by means of the semantic function, implement constraints which are defined in terms of the open sequence representation.

The affix representation \( \langle \vec{P}, P, \vec{S}, S \rangle \) of a string decision variable which is not yet fixed may be mapped to an open sequence representation \( \langle \vec{A}, N \rangle \), where \( N = P + S \) and for every index \( i \) in the array \( \vec{A} \):

\[
\mathcal{D}(\vec{A}[i]) = \begin{cases} 
\mathcal{D}(\vec{P}[i]) & \text{if } i \in [1, P] \\
\mathcal{D}(\vec{P}[i]) \cup \bigcup_{j=\max(P+S-i+1, 1)}^{P+s-i+1} \mathcal{D}(\vec{S}[j]) & \text{if } i \in [P+1, P+S] \\
\bigcup_{k=\max(P+S-i+1, 1)}^{P+S-i+1} \mathcal{D}(\vec{S}[k]) & \text{if } i \in [P+1, P+S] 
\end{cases}
\]

Example 2 The affix representation

\[\langle \langle [p-s], [a-z], [a-z], [f-x] \rangle, [1, 4], \langle x, i, f, [f-x] \rangle, [2, 3] \rangle\]
and the representation $\alpha$ from Example 1 both map to the open sequence representation
$$\langle \langle [p-s], [a-z], [a-z], [f-x], \{ f, i, x \}, \{ i, x \}, x \rangle, [3, 7] \rangle$$

\[ \vec{P} = [1, P] \quad \vec{P}_{\text{opt}} = [P + 1, P] \quad \vec{P}_{\text{dis}} = [P + 1, P_{\text{org}}] \]

The prefix language may be strengthened by tightening the bounds on $P$, thereby increasing the required and/or discarded ranges, and decreasing the optional range. In addition, the prefix language may be strengthened by removing values from the domains of the scalar decision variables with indices in the required or optional range.

**Example 3** Let the components of the affix representations of string decision variables $B$ and $C$ be the components of, respectively:
$$\beta = \langle \langle [p, r], [a-z], [a-z], \rangle, [1, 3], \langle x, i, f, [f-x], \rangle, [2, 4] \rangle$$
$$\gamma = \langle \langle [p-s], [a-z], [a-z], \rangle, [1, 3], \langle x, i, f, [f-x], \rangle, [4] \rangle$$

Comparing with $A$ from Example 1, we have $D(P^b) = D(P^a)$, and $D(P^b[i]) \subseteq D(P^a[i])$ for all $i \in [1, P]$; hence, $L(P^b) \subseteq L(P^a)$. Since $L(\vec{S}^b, \vec{S}^a) = L(S^b, S^a)$, this results in $D(B) \subseteq D(A)$. On the other hand, we have $\vec{D}(\vec{P}^b[i]) \subseteq \vec{D}(\vec{P}^a[i])$ for all $i \in [1, \vec{P}]$; yet we still have $L(S^b, S^a) \subseteq L(S^b, S^a)$ due to $D(S^c) \subseteq D(S^a)$. Since $L(\vec{P}^b, P^c) = L(\vec{P}^a, P^c)$, $D(B)$ and $D(C)$ are not comparable, as illustrated by the strings ‘prefix’ and ‘suffix’; while both are elements of $D(A)$, ‘prefix’ is an element of $D(B)$ but not $D(C)$, and ‘suffix’ is an element of $D(C)$ but not $D(B)$.

Any propagator which only removes values from domains of the elements of the affix arrays, or tightens the integer bounds on an affix length, must be contracting. We further note that each of these operations is finitely bounded: $P$ and $S$ are finitely bounded integers, which cannot be tightened indefinitely, while the finity of $\Sigma$ guarantees that $\vec{P}_{\text{req}}$ and $\vec{S}_{\text{req}}$ must eventually converge to concrete strings.

**V. PROPAGATION**

Next, we describe propagators for constraints over affix domains. The constraints given in Section III-B state relationships between the components of the open sequence representations of bounded-length string decision variables, but do not imply the same relationships among the components of their affix representations. For example, in the open sequence representation, $EQUAL(\langle \vec{A}^x, N^x, \rangle, \langle \vec{A}^y, N^y, \rangle)$ implies both $\vec{A}^x = \vec{A}^y$ and $N^x = N^y$, but it does not imply either $\vec{P}^x = \vec{P}^y$ or $P^x = P^y$ in a corresponding affix representation.

Propagation of $EQUAL$ over affix domains must account for cases where a region of $\vec{P}^x$ must be equal to a region of $\vec{P}^y$, but also cases where a region of $\vec{P}^x$ must be equal to the reverse of a region of $\vec{S}^b$; for $\vec{S}^b$ there exist symmetric cases.

**Example 4** Continuing from Example 3, consider the constraint $EQUAL(B, C)$. In any satisfying assignment, the symbols represented by $\vec{P}^b[1]$ and $\vec{P}^c[1]$ must be the same; no similar deduction is possible for $\vec{P}^b[2]$ and $\vec{P}^c[2]$, as $\vec{P}^b = \vec{P}^c = 1$. Pruning the domain of $C$ may be accomplished by giving it the affix representation:
$$\gamma' = \langle \langle [p, [a-z], [a-z], [a-z]], [1, 3], \langle x, i, f, [f-t], \rangle, 4 \rangle \rangle$$
which is strictly stronger than $\gamma$. The domain of $B$ may also be pruned, but only by reducing the domains of the affix lengths to reflect $P^b + S^b \in [5,7]$; i.e., the current interval of $P^b + S^b$. Pruning of affix lengths is discussed in Section V-B1.

**Example 5** If $B$ has an affix representation stronger than $\beta$, however, then an additional region of alignment may be reasoned upon. For example:

$$\beta' = (\langle p, r, [a\text{-}z], [a\text{-}z] \rangle, \langle x, i, f, [f\text{-}t] \rangle, 2)$$

If the components of the affix representations of $B$ and $C$ are $\beta'$ and $\gamma'$ (of Example 4), respectively, then the length of any satisfying string must be 5 (being both the minimum of $P^b + S^b$ and the maximum of $P^c + S^c$); furthermore, in any solution the region $\vec{R}^b[2]$ through $\vec{R}^b[3]$ must align with the region $\vec{S}^c[2]$ through $\vec{S}^c[3]$. As $D(\vec{P}^b[3]) \cap D(\vec{S}^c[3]) = \emptyset$, there can be no satisfying solution.

In Section V-A we define a pair of primitive pruning operations that filter the domains of symbols in overlapping regions of two affixes at a time. Then, in Section V-B we use these operations to construct propagators for the constraints specified in Section III-B. In Section V-C we discuss propagating a regular language membership constraint.

**A. Pruning Regions**

A large portion of the filtering required for the propagation of the constraints in Section III-B may be generally described as a sequence of successive equalities between the symbols occurring in a range of indices in an affix of one variable, and a same-sized range in an affix of another variable. We define this operation as $\text{EQREGION}_U(Q, Q, \vec{R}, R, D)$, where $Q$ and $\vec{R}$ are arrays of scalar decision variables, and $Q$, $R$, and $D$ are integer decision variables, where $\mathcal{L}(Q, Q)$ and $\mathcal{L}(\vec{R}, \vec{R})$ are affine languages of two distinct string variables. The integer $D$ represents the distance (in number of symbols) between the position of $Q[1]$ and $\vec{R}[1]$; in other words, when $D$ is fixed, $D(\vec{Q}[D + 1])$ must be equal to $D(\vec{R}[1])$. In some cases, this distance will be a fixed quantity; however, for the sake of generality, we need to treat the distance as an integer decision variable. The domain of the distance defines the possible relative alignments of the two affix bounds.

To cover all the cases of aligned regions between the arguments of the constraints, two versions of $\text{EQREGION}$ are required. The versions differ only in the relative direction of these two affixes: a pair of prefixes or pair of suffixes is said to be unidirectional, while a mixed prefix-suffix pair is said to be bidirectional.

1) Unidirectional Region: The unidirectional pruning operation $\text{EQREGION}_U(Q, Q, \vec{R}, R, D)$ makes two types of inferences. The first considers elements of $Q$ with an index in $Q_{\text{req}}$ that align with some element of $\vec{R}$ with an index in $\vec{R}_{\text{req}}$, for every distance in $D(D)$. Each value in the domains of these elements of $Q$ requires a witness in the union of the domains of the elements of $\vec{R}$ with which it aligns. There is a symmetric requirement for elements of $\vec{R}$ with index in $\vec{R}_{\text{req}}$ that align with some element of $Q$ with an index in $Q_{\text{req}}$.

The second type of inference concerns the possible values of $D$: for each distance, the domains of the elements of $\vec{Q}$ and $\vec{R}$ that would be aligned must have some value in common, or the distance can be removed from $D(D)$.

$$\text{EQREGION}_U(Q, Q, \vec{R}, R, D) \equiv$$

$$\forall i \in Q_{\text{req}} \cap [\overline{D}, R + D - 1]: (\vec{Q}[i] \in \bigcup_{k \in D(D)} D(\vec{R}[i - k])$$

$$\land D \in \{ k \mid D(\vec{R}[i - k]) \cap D(\vec{Q}[i]) \neq \emptyset \} )$$

$$\land \forall j \in \vec{R}_{\text{req}} \cap [-\overline{D}, Q - D - 1]: \vec{R}[j] \in \bigcup_{k \in D(D)} D(\vec{Q}[k + j])$$

$$\land D \in \{ k \mid D(\vec{Q}[k + j]) \cap D(\vec{R}[j]) \neq \emptyset \} )$$

2) Bidirectional Region: For prefix-suffix or suffix-prefix alignments, the two affixes have opposite directions. So while $\vec{R}[1]$ and $\vec{Q}[D + 1]$ align in both cases, the alignments between subsequent elements will differ. Otherwise, the reasoning performed is similar to that of $\text{EQREGION}_U$: elements of $Q$ that always align with some element of $\vec{R}$ have their domains filtered, and vice versa; while the domain of possible distances is filtered based on the domains of the elements that would become equal in that alignment.

$$\text{EQREGION}_B(Q, Q, \vec{R}, R, D) \equiv$$

$$\forall i \in Q_{\text{req}} \cap [\overline{D}, R - D - 1]: (\vec{Q}[i] \in \bigcup_{k \in D(D)} D(\vec{R}[k - i - 1])$$

$$\land D \in \{ k \mid D(\vec{R}[k - i - 1]) \cap D(\vec{Q}[i]) \neq \emptyset \} )$$

$$\land \forall j \in \vec{R}_{\text{req}} \cap [-\overline{D}, Q - D - 1]: \vec{R}[j] \in \bigcup_{k \in D(D)} D(\vec{Q}[k - j - 1])$$

$$\land D \in \{ k \mid D(\vec{Q}[k - j - 1]) \cap D(\vec{R}[j]) \neq \emptyset \} )$$

**B. Propagators**

We proceed to describe propagators for the constraints of Section III-B in terms of the two primitive pruning operations defined in the previous subsection, with the addition of a few integer equalities on affix length bounds.

1) Equality: It seems likely that unifying $X$ and $Y$ in the model prior to constraint posting will generally be preferable to maintaining $\text{EQUAL}(X, Y)$; however, this is not always possible, for example if one of the decision variables is used in a refined context. Propagation of equality over affix domains is significantly more complicated than for more familiar domains, as we must account for interactions between $\vec{P}^x$ and $\vec{S}^y$, and between $\vec{S}^x$ and $\vec{P}^y$. It is, nevertheless, simpler than propagation for the other constraints specified here; we therefore describe propagation of $\text{EQUAL}$ in some detail, to serve as an illustrative example of how propagation over affix representations functions.

Figure 1 provides a propagator for the $\text{EQUAL}$ constraint. Lines 2 to 12 correspond to $N^x = N^y$ in the open sequence.
equal((\(P^x, P^y, S^x, S^y\), (\(P^x, P^y, S^x, S^y\))):
  if \(P^x + S^x < P^y + S^y\) then
    \(P^x \leftarrow \min(P^x, \min(P^y, P^x + S^y - S^x))\)
    \(S^x \leftarrow \max(S^x, \min(S^y, P^x + S^y - P^x))\)
  else
    \(P^y \leftarrow \max(P^y, \min(P^x, P^y + S^x - S^y))\)
    \(S^y \leftarrow \max(S^y, \min(S^x, P^y + S^x - P^y))\)
  \(n \leftarrow \min(P^x + S^x, P^y + S^y)\)
  \(P^x \leftarrow \min(P^x, n - S^x)\)
  \(S^x \leftarrow \min(S^x, n - P^x)\)
  \(P^y \leftarrow \min(P^y, n - S^y)\)
  \(S^y \leftarrow \min(S^y, n - P^y)\)
  \(\text{eqregion}^U(P^x, P^y, P^y, P^y, 0)\)
  \(\text{eqregion}^B(P^x, P^y, S^x, S^y, P^y, P^y + S^x)\)
  \(\text{eqregion}^B(S^x, S^x, S^y, S^y, P^y, P^y + S^x)\)
  \(\text{eqregion}^U(S^x, S^x, S^y, S^y, 0)\)

Fig. 1. Propagation of the constraint \text{equal}.

representation. The naive translation into \(P^x + S^x = P^y + S^y\) would fail to take advantage of all available information. Consider that an increase to \(N^x\) due to propagation could be achieved with an increase to \(P^x, S^x\), or both; if, however, the required region of the prefix of \(Y\) is larger than that of \(X\), then increasing \(P^x\) is likely to result in filtering of domains for symbols in \(P^x\). In contrast, increasing \(S^x\) under those same circumstances will yield little, if any, filtering of symbol domains, as shown in the following example.

Example 6 Let the components of the affix representations of string decision variables \(X\) and \(Y\) be, respectively:

\[
\langle [p-s], [a-z], [a-z], f, \rangle, [1, 3], \langle x, i, [a-z], [a-z] \rangle, [2, 4] \rangle
\]

\[
\langle [p-s], [r-u], [f-s], [f-x], \rangle, 3, \langle x, i, f, [a-f] \rangle, [2, 4] \rangle
\]

The minimum length of any string in \(D(Y)\) is \(P^y + S^y = 5\), so clearly one or both of \(P^y = 1\) and \(S^y = 2\) should be increased accordingly. Choosing \(S^x \leftarrow 4\) satisfies the length constraints, but results in rather limited pruning; i.e., it would not be valid to prune \(D(S^x[3]) \rightarrow \{ f \}\) based on \(D(S^y[3])\), as depending on the value of \(P^x, S^x[3]\) might also align with \(P^y[2]\) or \(P^y[3]\). In contrast, the choice of \(P^x \leftarrow 3\) ensures that in all satisfying assignments \(P^x[2]\) aligns with \(P^x[2]\).

Every domain of \(N\) may be represented by several \(P\) and \(S\) combinations, so we choose the most advantageous of these combinations. The best choice can be determined in constant time (see line 2) by comparing the length bounds of the two affix representations. For the pruning of the lower bounds of lengths of strings in the affixes (following the conditional at line 2), this means that the required region of each prefix (resp. suffix) may be enlarged to take advantage of information from the other prefix (resp. suffix), but only so long as the operation would not cause the minimum length of either string to exceed the minimum length of the other string. Adjustment to the maximum lengths of the affixes is then made based on the combination of these new affix minimum lengths, and the prior maximum bound of the string lengths (computed at line 8).

As each affix is anchored to one or the other end of its string, the prefix-prefix and suffix-suffix pairs each have a fixed distance of zero; as a result, the \(\text{eqregion}^U\) operations at lines 13 and 16 perform quite strong pruning. For the bidirectional pairs, the distance is the length of the two strings; as long as this length is not fixed, \(\text{eqregion}^B\) will be relatively weaker. This is exactly as we would expect: with an unknown string length, the precise alignment between prefix symbols of one string and suffix symbols of the other may only be known very late in the solution process, and it is only the domains of those symbols that must align with some symbol from the opposing affix for every feasible value of the string length which might be pruned.

The \(\text{eqregion}^B\) propagator in Figure 1 is contracting; it is also checking, as any assignment must have a fixed length, at which point the relative positions of the symbols in the two strings are known. However, the reasoning on the required regions of \(X\) and \(Y\) is not quite sufficient to maintain PSL-consistency. This is due to a gap between the affix regions which are covered by the two primitive pruning operations described in the previous subsection. For example, when \(P^x < P^y < P^y\), then there may be elements of \(P^x\) that align with either one element from \(P^y\) or a region of elements from \(S^y\). The \(\text{eqregion}^U\) propagator in Figure 1 is correct despite ignoring this additional case, as the affix elements affected are not covered by either the \(\text{eqregion}^U\) or \(\text{eqregion}^B\) cases. Furthermore, as long as the minimum affix lengths are pruned in the maximally advantageous method, as described above, the additional case seems to occur infrequently. Nevertheless, it should be possible to extend the propagator to cover these cases by adding a third primitive pruning operation which takes into account the optional regions of the affixes, using similar reasoning to that for required regions presented here. This should allow not only for a PSL-consistent propagator, but also for extension to one that is fully PS-consistent.

2) \(\text{Reverse}\): The propagator for \(\text{reverse}\) follows directly from that of \(\text{equal}\). The only required modification is the swapping of the parameters corresponding to the prefixes and suffixes of \(Y\) on lines 13 through 16 of Figure 1.

3) \(\text{Concatenation}\): The propagator for \(\text{concatenation}\) given in Figure 2 has many more cases than \(\text{equal}\); the prefix of \(Z\) could have regions aligned with each of the four affixes of \(X\) and \(Y\), and there are another four possible alignments for the suffix of \(Z\). Nevertheless, each of these eight possible aligned regions may be expressed as an instance of one of the two defined pruning operations, with the calculation of a suitable distance value.

Updating minimum length of the affix bounds is also similar to \(\text{equal}\), although in the case of \(\text{concatenation}\) there is less information available. Specifically, \(S^x\) and \(P^y\) are difficult to increase, because of their position inside \(Z\). The adjustment of \(P^x\) must consider not only \(S^x\), but also the upper bounds of the lengths of the affixes of \(Y\), in order to avoid unjustified pruning on \(N^y\); adjustment of \(S^y\) is symmetric.
The **CONCAT** propagator also does not maintain PSL-consistency. The cases missed by the **EQUAL** propagator described in the last section are missed here, as well. Furthermore, the possible alignments encountered while propagating **CONCAT** allow for cases in which an element from an affix array of one string decision variable might align with ranges of elements from more than one affix array of another string decision variable. These cases are somewhat more difficult to generalize than the additional case discussed in **EQUAL**, and further study is required to determine a method for achieving PSL-consistency, and ideally PS-consistency, for **CONCAT**.

4) **Character At Position and Substring:** the **CONCAT** propagator described here does not maintain PSL-consistency, for the same reasons described for **CONCAT**.

**Character At Position and Substring:**

4) **Character At Position and Substring:**

**CONCAT** propagator described here does not maintain PSL-consistency, for the same reasons described for **CONCAT**.

**Character At Position and Substring:**

4) **Character At Position and Substring:**

**CONCAT** propagator described here does not maintain PSL-consistency, for the same reasons described for **CONCAT**.

C. **Regular Language Membership**

Given a finite automaton $M$ that specifies $L$, an automaton $M^{pre}$ accepting the bounded-length prefixes of strings in $L$ is constructed by unrolling $M$ to length $P$, and making all states with distance at least $P$ from the start state be accepting states in $M^{pre}$ (as long as they fall on a path leading to an accepting state in $M$). Similarly, an automaton $M^{sup}$ accepting bounded-length suffixes is constructed from $M^{pre}$. We then make use of the regular constraint for sequences of bounded length described in [18], which we refer to as **OPENREGULAR**. From this we get the decomposition into **OPENREGULAR**($M^{pre}, \vec{P}, P$) and **OPENREGULAR**($M^{sup}, \vec{S}, S$).

VI. **Preliminary Results**

This theoretical paper aims at laying a self-contained foundation for bounded-length string decision variables in CP. A full experimental evaluation is orthogonal to this purpose, and would have to be omitted for space reasons. Nevertheless, we implemented a prototype of the propagators described here using the extended indexical language described in [22]. An indexical [23] is an expression of the form $x \in \sigma$ restricting the domain of the decision variable $x$ to the intersection of its current domain and the interval $\sigma$. An indexical language is a high-level language for propagator description; [22] provides a compiler which generates propagator descriptions from checkers, and also has several solver-specific back-ends which allow for compilation of the propagator into source code.

We used this compiler to generate prototypes, in Gecode 3.7.3 [24], of propagators for the constraints given here (exclusive of **REGULAR**), using both the affix representation and the open sequence representation. We compared performance on several randomly generated concatenation problems. In each instance, a concrete string of length 100 to 200, over an alphabet of size ten, was generated. **CONCAT** constraints were used to constrain seven bounded-length string decision variables to be equal to the concrete string, with the variables occurring in a fixed sequence that included a single repeated variable. Branch and bound search was used to find the solution with the maximum length for the repeated variable (i.e., the longest repeated symbol sequence in the concrete string).
Preliminary results were encouraging. Despite the increased complexity inherent in propagating the affix representation, over several hundred instances we observed very little variation in runtime. It is possible that the constraints in the randomly generated problems were too loose to force the execution of the more computationally expensive portions of the propagators; it seems likely that hand-crafted test cases will be required to properly evaluate the impact of these portions on performance. More strongly constrained test cases should also better demonstrate the impact of the affix representation on search tree size; in the random instances we observed only a moderate improvement (approx. 2% on average) in the number of search tree nodes.

Full experimental analysis will, of course, require an affix representation implementation of Regular.

VII. FUTURE WORK

Immediate work is focused on extending the propagators to achieve PS-consistency, and on improving the efficiency of the initial prototype. We also plan to investigate branching heuristics, as our intuition is that the early stages of search must rely heavily on intelligent branching over string length. A CP angle is promising here, as for fixed-length strings (implemented as arrays of scalar decision variables), CP solvers have already been shown [25] to outperform systematically HAMPI [9], KALUZA [15], and SUSHI [6] by orders of magnitude, on their own benchmarks. A promising direction of research is alternative affix representations, such as using regular languages to state the affix languages, allowing for stronger domains than the array-based representation we describe here. Supplementing the domain with a data structure such as layered graphs [11], or MDD constraint stores [26], would allow propagation directly on the data structure, similar to the approach for unbounded regular domains in [5].

Acknowledgements: Work supported by grant 2009-4384 of the Swedish Research Council (VR). The authors wish to thank the anonymous reviewers, of this and a prior version, for their valuable comments and suggestions; J.-N. Monette for assistance with the indexable compiler; and F. Hassani Bijarbooneh and C. Schulte for Gecode assistance.

REFERENCES