Design and Implementation of Bounded-Length Sequence Variables

Joseph D. Scott[1], Pierre Flener[1], Justin Pearson[1], and Christian Schulte[2]

[1] Department of Information Technology, Uppsala University, Sweden
{Joseph.Scott, Pierre.Flener, Justin.Pearson}@it.uu.se
[2] KTH Royal Institute of Technology, Sweden
cschulte@kth.se

Abstract We present the design and implementation of bounded-length sequence (BLS) variables for a CP solver. The domain of a BLS variable is represented as the combination of a set of candidate lengths and a sequence of sets of candidate characters. We show how this representation, together with requirements imposed by propagators, affects the implementation of BLS variables for a copying CP solver, most importantly the closely related decisions of data structure, domain restriction operations, and propagation events. The resulting implementation outperforms traditional bounded-length string representations for CP solvers, which use a fixed-length array of candidate characters and a padding symbol.

1 Introduction

String variables are useful for expressing a wide variety of real-world problems, such as test generation [7], program analysis [4], model checking [10], security [3], and data mining [16]. Despite this usefulness, string variables have never received an optimized implementation in a modern constraint programming (CP) solver.

We describe the design and implementation of a string variable type for a copying CP solver. Properly designed, a string variable can be much more efficient, in both time and space complexity, than the decompositions commonly used to model strings in CP. Additionally, string variables greatly simplify the modeling of problems including strings, with benefits to both readability and correctness. We choose to implement a *bounded*-length sequence (BLS) variable type, as it provides much more flexibility [9, 18] than fixed-length approaches, but avoids the blow-ups that plague unbounded-length representations.

We make contributions in three dimensions. First, we select a data structure for the BLS variable domain, namely a dynamic list of bitsets, to represent domains in reasonable space. The data structure is designed for efficient implementation of domain restriction operations, which BLS variables expose to propagators; we develop a correct and minimal set of these operations. Variables in turn notify propagators of domain changes using propagation events; we design a monotonic set of propagation events that are useful for propagators on BLS variables. Second, the BLS variable is implemented on top of Gecode [25], a mature CP toolkit with state-of-the-art performance and many deployments

in industry. Third, we show that BLS variables outperform other string solving methods in CP, as demonstrated upon both a string problem in the CSPlib repository and a representative benchmark from the field of software verification.

In this paper, we summarize [22, Chapters 10 and 11]: this is an improvement of our [23], as discussed in the experiments of Section 7. We forego discussion in this paper of several interesting and important theoretical aspects of the BLS representation, as previous publications cover these theoretical aspects in detail: [22] provides an extended discussion of the logical properties of the representation, a justification for the utility of BLS variables in string constraint problems, a comparison of different modeling strategies, a survey of related work in CP such as [11], and a discussion of intensional representations; these topics are also presented in briefer form in [23]. Hence we choose to deal specifically with the implementation of an efficient BLS variable, rather than repeating material that has been presented elsewhere.

Plan of the Paper Section 2 discusses the design of variable implementations in general. Section 3 defines a domain representation appropriate for BLS variables and motivates the design choices of subsequent sections. Section 4 considers several representations for each of the components of a BLS variable and motivates why particular choices have been made. Section 5 defines restriction operations, which allow propagators to modify the domain of such a variable. Section 6 describes BLS-specific systems of propagation events, which describe restrictions of the domain of a variable. Section 7 justifies these design choices by an experimental evaluation. Section 8 concludes the paper.

2 Variables and Propagators

We summarize key concepts relating to propagators as implementations of constraints and three fundamental choices of how to implement a variable type.

Propagators A *propagator* implements a constraint and is executed by a *propagation loop*. The execution of a propagator results in the restriction of some variable domains pruning values for variables that are in conflict with the implemented constraint. The propagation loop executes the propagators until no propagator can prune any values, the propagator is said to be at *fixpoint*. For a more detailled discussion of propagators see for example [19].

Variables The central decision in implementing a variable type is how to represent the domain of a variable. With this representation decided, the implementation of the variable type must take into account three primary choices: how a domain is stored, how a propagator prunes a domain, and how a propagator is informed of a restriction of a domain. We consider each of these choices in turn.

No *data structure* exists in a vacuum, and every choice of data structure represents a tradeoff between memory requirements and computational complexity. Evaluating the choices made in designing a data structure, therefore, requires

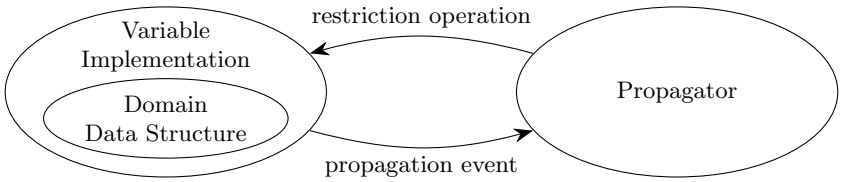

Figure 1. Interactions of a variable implementation and a propagator: a propagator requests a change to the domain of a variable via a *restriction operation,* and a variable notifies a propagator via a *propagation event* that its domain has been restricted.

considering the environment in which the data structure will function. For a variable implementation, that means considering how it interacts with the CP solver, and most importantly how it interacts with propagators. A simplistic view of this interaction is shown in Figure 1.

In an implementation, propagators work by modifying the domains of variables. These domains are not exposed directly to propagators; rather the variable implementation encapsulates the domain representation and exposes some *restriction operations* by which propagators can request that the domain be modified. Restriction operations provide an interface to the domain that allows the removal of candidate values; no other modifications are allowed, as a propagator is required to be contracting on domains.

When the domain of a variable is restricted by a propagator, any other propagator that implements a constraint on the corresponding variable must be notified, as it is possible that the latter propagator is no longer at fixpoint. Most CP solvers make use of more fine-grained information than simply which variable has a newly restricted domain. This more detailed information is called a *propagation event* (for an extended discussion of events, see [20]); for an integer variable X, for example, a propagation event might indicate that the upper bound of $\mathrm{dom}(X)$ has shrunk, or that $\mathrm{dom}(X)$ has become a singleton.

3 BLS Variables

The domain of a string variable is a set of strings. Even in the bounded-length case, sets of strings are difficult to represent in extension, due to high space complexity; on the other hand, intensional representations, such as finite automata or regular expressions, generally have a higher time complexity for operations. However, both the time complexity and space complexity may be reduced if the representation of the set of strings is not required to be exact. For example, an integer interval is a compact and efficient representation for a set of integers. Of course, many sets of integers cannot be exactly represented by an interval; however, every set of integers may be *over-approximated* by some interval.

We now define an over-approximation that is appropriate for representing the domain of a bounded-length set of strings. We then give an example of how a propagator for a string constraint would interact with domains thus represented.

$$\Big\langle \boldsymbol{\mathcal{A}}^{\,b}, \mathcal{N} \Big\rangle = \Big\langle \big\langle \underbrace{\mathcal{A}[1], \ldots, \mathcal{A}[5]}_{\text{mandatory}}, \underbrace{\mathcal{A}[6], \ldots, \mathcal{A}[10]}_{\text{optional}}, \underbrace{\overset{\varnothing}{\mathcal{A}[11]}, \ldots, \overset{\varnothing}{\mathcal{A}[15]}}_{\text{forbidden}} \big\rangle, \big\{5, 7, 10\big\} \Big\rangle$$

Figure 2. Example of a b-length sequence $\langle \boldsymbol{\mathcal{A}}^{\,b}, \mathcal{N} \rangle$ with maximum length $b = 15$. Each set $\mathcal{A}[i]$ of candidate characters is the set of all symbols at index i for some string in the domain; the set $\mathcal{N}$ of candidate lengths is $\{5, 7, 10\}$. The lower bound, 5, and upper bound, 10, of the set of candidate lengths partition the sets of candidate characters into *mandatory*, *optional*, and *forbidden* regions, as indicated. A set of candidate characters is empty if and only if it is found in the forbidden region.

Representation of a String Domain For any finite set $\mathcal{W}$ of strings over an alphabet Σ there exists an upper bound on the lengths of the strings in $\mathcal{W}$, i.e., some number $b \in \mathbb{N}$ such that for every string $w \in \mathcal{W}$ the length of w is not greater than b. Such a set $\mathcal{W}$ may be over-approximated by a pair $\langle \boldsymbol{\mathcal{A}}^{\,b}, \mathcal{N} \rangle$, consisting of a sequence $\boldsymbol{\mathcal{A}}^{\,b}$ of sets $\langle \mathcal{A}[1], \ldots, \mathcal{A}[b] \rangle$ over Σ, where every $\mathcal{A}[i] \subseteq \Sigma$ is the set of all *candidate characters* occurring at index $i \in [1, b]$ for some string in $\mathcal{W}$, and a set $\mathcal{N} \subseteq [0, b]$ of *candidate lengths* of the strings in $\mathcal{W}$. This over-approximation, illustrated in Figure 2, is called the *bounded-length sequence representation*. A pair $\langle \boldsymbol{\mathcal{A}}^{\,b}, \mathcal{N} \rangle$ is referred to as a *b-length sequence*, and is an abstract representation of the set of all strings that have a length $\ell \in \mathcal{N}$ and a character at each index $i \in [1, \ell]$ taken from the set $\mathcal{A}[i]$ of symbols.

The domain of a string variable X_j, written $\mathrm{dom}(X_j)$, is a finite set of strings; thus, the domain of any string variable X_j can be over-approximated by a b-length sequence $\langle \boldsymbol{\mathcal{A}}_j^b, \mathcal{N}_j \rangle$ for some appropriately large value of b. In this context, we will sometimes write $\mathrm{dom}(\boldsymbol{\mathcal{A}}_j^b, \mathcal{N}_j)$ to mean the domain of a string variable X_j that is represented by the b-length sequence $\langle \boldsymbol{\mathcal{A}}_j^b, \mathcal{N}_j \rangle$; thus we have:

$$\mathrm{dom}\big(\boldsymbol{\mathcal{A}}_j^b, \mathcal{N}_j\big) = \bigcup_{\ell \in \mathcal{N}_j} \big\{ w \in \Sigma^\ell \;\big|\; \forall i \in [1, \ell] : w[i] \in \mathcal{A}_j[i] \big\} \tag{1}$$

Bounded-length sequences are not unique: if the sequences $\langle \boldsymbol{\mathcal{A}}_1^b, \mathcal{N} \rangle$ and $\langle \boldsymbol{\mathcal{A}}_2^b, \mathcal{N} \rangle$ differ only in one or more pair of sets $\mathcal{A}_1[i]$ and $\mathcal{A}_2[i]$ where $i > \max(\mathcal{N})$, then $\langle \boldsymbol{\mathcal{A}}_1^b, \mathcal{N} \rangle$ and $\langle \boldsymbol{\mathcal{A}}_2^b, \mathcal{N} \rangle$ represent the same set of strings. Uniqueness may be imposed by a *representation invariant* that only allows bounded-length sequences of a canonical form, in which a set $\mathcal{A}[i]$ of candidate characters is the empty set if and only if the index i is greater than $\max(\mathcal{N})$. As illustrated in Figure 2, the set $\mathcal{N}$ of candidate lengths divides the sequence $\boldsymbol{\mathcal{A}}^b$ into three regions: the *mandatory* candidate characters, with an index at most $\min(\mathcal{N})$; the *optional* candidate characters, with an index greater than $\min(\mathcal{N})$ but at most $\max(\mathcal{N})$; and the *forbidden* candidate characters, with an index greater than $\max(\mathcal{N})$.

Propagation A properly designed variable implementation must take into account the operation of propagators implementing constraints for the corresponding variable type. For string variables, expected constraints include regular

$$
\begin{aligned}
&\left\langle \mathcal{A}_3[1], \mathcal{A}_3[2], \mathcal{A}_3[3], \mathcal{A}_3[4], \mathcal{A}_3[5], \mathcal{A}_3[6], \ldots \right\rangle \\
\mathcal{N}_1 = \{3\} : &\left\langle \mathcal{A}_1[1], \mathcal{A}_1[2], \mathcal{A}_1[3], \mathcal{A}_2[1], \mathcal{A}_2[2], \mathcal{A}_2[3], \ldots \right\rangle \\
\mathcal{N}_1 = \{4\} : &\left\langle \mathcal{A}_1[1], \mathcal{A}_1[2], \mathcal{A}_1[3], \mathcal{A}_1[4], \mathcal{A}_2[1], \mathcal{A}_2[2], \ldots \right\rangle \\
\mathcal{N}_1 = \{5\} : &\left\langle \mathcal{A}_1[1], \mathcal{A}_1[2], \mathcal{A}_1[3], \mathcal{A}_1[4], \mathcal{A}_1[5], \mathcal{A}_2[1], \ldots \right\rangle \\
\mathcal{N}_1 = \{6\} : &\left\langle \mathcal{A}_1[1], \mathcal{A}_1[2], \mathcal{A}_1[3], \mathcal{A}_1[4], \mathcal{A}_1[5], \mathcal{A}_1[6], \ldots \right\rangle
\end{aligned}
$$

Figure 3. The post-condition, implied by the constraint $\textsc{Cat}(X_1, X_2, X_3)$, on the set of candidate characters $\mathcal{A}_3[5]$, is determined by the four candidate lengths of X_1, each yielding a possible alignment between the three string variables.

language membership ($\textsc{Regular}_{\mathcal{L}}$), string length (Len), reversal (Rev), and concatenation (Cat); many others are possible (see, e.g., [22]). The following example provides a sample of the inference required by typical propagators for such constraints, and gives context to the design decisions that follow.

Example 1 (Concatenation). The constraint $\textsc{Cat}(X_1, X_2, X_3)$ holds if the concatenation of string variables X_1 and X_2 is equal to the string variable X_3. Consider string variables X_1, X_2, and X_3 with domains represented by the b-length sequences $\langle \boldsymbol{\mathcal{A}}_1^b, \mathcal{N}_1 \rangle$, $\langle \boldsymbol{\mathcal{A}}_2^b, \mathcal{N}_2 \rangle$, and $\langle \boldsymbol{\mathcal{A}}_3^b, \mathcal{N}_3 \rangle$, each with maximum length $b = 15$, and sets of candidate lengths $\mathcal{N}_1 = [3, 6]$, $\mathcal{N}_2 = [4, 7]$, and $\mathcal{N}_3 = [5, 14]$.

A propagator is a contracting function on tuples of variable domains. A propagator implementing the constraint Cat is a function of the following form:

$$
\begin{aligned}
&\textsc{Cat}\left(\mathrm{dom}\left(\boldsymbol{\mathcal{A}}_1^b, \mathcal{N}_1\right), \mathrm{dom}\left(\boldsymbol{\mathcal{A}}_2^b, \mathcal{N}_2\right), \mathrm{dom}\left(\boldsymbol{\mathcal{A}}_3^b, \mathcal{N}_3\right)\right) \\
&\qquad = \left\langle \mathrm{dom}\left(\boldsymbol{\mathcal{A}}_1'^b, \mathcal{N}_1'\right), \mathrm{dom}\left(\boldsymbol{\mathcal{A}}_2'^b, \mathcal{N}_2'\right), \mathrm{dom}\left(\boldsymbol{\mathcal{A}}_3'^b, \mathcal{N}_3'\right)\right\rangle
\end{aligned} \tag{2}
$$

The simplest inference relevant to the propagation of Cat is the initial arithmetic adjustment of the candidate lengths. For example, every candidate length for X_3 should be the sum of some candidate lengths for X_1 and X_2:

$$
\mathcal{N}_3' \coloneqq \{n_3 \in \mathcal{N}_3 \mid \exists n_1 \in \mathcal{N}_1, n_2 \in \mathcal{N}_2 : n_3 = n_1 + n_2\} = [7, 13] \tag{3}
$$

For sets of candidate characters, there are dependencies not only upon sets of candidate characters from the other string variables constrained by Cat, but also on the sets of candidate lengths. For example, Figure 3 illustrates the four possible alignments of X_1, X_2, and X_3, corresponding to the four candidate lengths in the set $\mathcal{N}_1 = [3, 6]$. If, on the one hand, the length of X_1 were fixed to either of the two smallest of these candidates, then any symbol in the set $\mathcal{A}_3'[5]$ of candidate characters would also have to be an element in either $\mathcal{A}_2[2]$ or $\mathcal{A}_2[1]$. If, on the other hand, the length of X_1 were fixed to either of the two largest of these candidates, then in either case any symbol in the set $\mathcal{A}_3'[5]$ of candidate characters would have to be an element of the set $\mathcal{A}_1[5]$. Hence:

$$
\mathcal{A}_3'[5] \coloneqq \mathcal{A}_3[5] \cap (\mathcal{A}_2[1] \cup \mathcal{A}_2[2] \cup \mathcal{A}_1[5]) \tag{4}
$$

The resulting sets of candidate lengths also depend upon the sets of candidate characters. For example, if the intersection of $\mathcal{A}_2[2]$ and $\mathcal{A}_3[5]$ were empty, then the uppermost alignment illustrated in Figure 3, where $\mathcal{N}_1 = \{3\}$ could be ruled out: for any combination of strings X_1, X_2, and X_3 satisfying $\textsc{Cat}(X_1, X_2, X_3)$ such that the length of X_1 was 3, the maximum length of X_3 would be $4 \notin [7, 13]$, since the set $\mathcal{A}_3[5]$ of candidate characters would be empty. □

Solver-Based Requirements In a CP solver, the propagation loop described in Section 2 is interleaved with a backtracking search. If that loop ends with some variables as yet unassigned, then the search tree is grown by partitioning the search space to create two or more subproblems, obtained by mutually exclusive *decisions* on the domain of an unassigned variable; the propagation loop is then executed on the chosen subproblem. On the other hand, if the propagation loop results in a failed domain, then some prior decision is undone: the search returns to a previously visited node, from which the tree is grown, if possible, by choosing another of the mutually exclusive decisions that were determined at that node.

There are two main CP techniques for restoring a previously visited node during backtracking [19]: *trailing*, in which changes to the variable domains during search are stored on a stack, and backtracking consists of reconstructing a previously visited node; and *copying*, in which the domains are copied before a choice is applied, and every new node of the search tree begins with a fresh copy.

The choice of a copying solver has two main impacts on the design of a variable type. First, under copying, the search procedure and the fixpoint algorithm are orthogonal, while under trailing all the solver components are affected by backtracking; thus, under copying, design choices can be made without reference to the details of the search procedure. Second, for a copying system, memory management is critical. The amount of memory required by a data structure used to represent the domain of a variable (as well as data structures used to store the states of other solver components) should therefore be minimal.

4 Data Structure

For a string variable X with a domain represented by a bounded-length sequence, there are three largely orthogonal structural choices to be considered: the representation of the set $\mathcal{N}$ of candidate lengths, the representation of the sets $\mathcal{A}[i]$ of candidate characters, and the construction of the sequence $\langle \mathcal{A}[1], \ldots, \mathcal{A}[b] \rangle$ itself. Lengths are natural numbers, and any finite alphabet may be mapped to the natural numbers (upon imposing an ordering over the alphabet); therefore the possible implementations for both the set $\mathcal{N}$ of candidate lengths and the sets $\mathcal{A}[i]$ of candidate characters at all indices i of X are the same. However, lengths and characters have different characteristics and benefit from different choices of representation. Each of these choices is now considered in turn.

The Set of Candidate Lengths The set $\mathcal{N}$ of candidate lengths may be exactly represented or over-approximated. An exact representation could utilize a data

structure such as those used in CP solvers to exactly represent the domain of an integer variable (e.g., a range sequence or a bitset). Alternately, the set of candidate lengths can be over-approximated as an interval: a pair of integers representing the lower and upper bounds of the set.

Recall from Example 1 that a propagator implementing the constraint Cat must calculate sums of all candidate lengths for its string variables. This computation is quadratic in the size of the set representation (i.e., in the number of elements, ranges, or bounds used to represent the set). For the interval representation, the representation size is constant, but for exact set representations it is not. Example 1 also shows that the upper and lower bounds of $\mathcal{N}$ are slightly more useful during propagation than interior values of $\mathcal{N}$. The *lower* bound of the set of candidate lengths defines the smallest sequence of sets of candidate characters that an implementation should represent explicitly, as *every* candidate solution must have some symbol at those *mandatory* indices. In contrast, all, some, or none of the other sets of candidate characters might be explicitly represented; the *upper* bound of the set of candidate lengths divides these sets of candidate characters into those at *optional* indices that participate in some of the candidate solutions, and those at *forbidden* indices that participate in none.

An exact representation would allow some extra inference during the propagation of string constraints, and could be sensible given enough string constraints with efficient propagators performing non-trivial reasoning on the interior values of $\mathcal{N}$. Otherwise, an interval representation appears to be most suitable.

Sets of Candidate Characters The exact composition of an alphabet is problem dependent; however, several properties generally apply. First, alphabet sizes are typically manageable; this is certainly true for many interesting classes of string constraint problems, such as occur in computational biology, natural-language processing, and the verification and validation of software. Second, the symbols of an alphabet often have a known total ordering, in which case the strings of any language on that alphabet have a corresponding lexicographic order; even when no meaningful order exists, such an order may be imposed as needed. Third, intervals of symbols often have little or no inherent meaning, as, in contrast to numeric types, there is generally no logical relationship between consecutive symbols in even a totally-ordered alphabet. Finally, as shown in Example 1, the propagation of constraints over string variables with bounded-length sequence domain representations depends heavily on standard set operations (i.e., union and intersection) between sets of candidate characters.

Upon considering these properties, we propose to implement each set of candidate characters as a bitset. A *bitset* for a finite set $\mathcal{S} \subset \mathbb{N}$ is a sequence of $\max(\mathcal{S})$ bits (i.e., values in $\{0, 1\}$) such that the i-th bit is equal to 1 if and only if $i \in \mathcal{S}$. With word size w, a bitset representation of $\mathcal{S}$ requires $k = \lceil \max(\mathcal{S})/w \rceil$ words, or a total of kw bits. Bitsets allow for very fast set operations (typically, a small constant number of operations per word is required): specifically, the complexity of the union and intersection operations is linear in k. Furthermore, as long as $|\Sigma|$ is not significantly larger than w, the memory requirement of a bitset representation is competitive with other common exact set representations.

Sequence The sequence $\langle \mathcal{A}[1], \ldots, \mathcal{A}[b] \rangle$ of sets of candidate characters could be implemented in two ways: as an array-based structure, requiring minimal memory overhead and affording direct access to the elements; or as a list-based structure, allowing for better memory management of dynamic-sized lists, but no direct access. A key observation is that the maximum length of any b-length sequence is already given, namely b. We propose a hybrid array-list implementation: a fixed-length array of n pointers to blocks of m bitsets each, where $n = \lceil b/m \rceil$. This design offers more efficient access to individual nodes than a traditional linked list, and easier memory (de)allocation than a static-sized array. Blocks are allocated as needed: at a minimum, a number of blocks sufficient to accommodate $\min(\mathcal{N})$ sets of candidate characters (i.e., the mandatory region) is allocated; additional blocks are allocated when $\min(\mathcal{N})$ increases, or when a restriction operation is applied to a set of candidate characters in the optional region. Deallocation is performed as needed upon a decrease of $\max(\mathcal{N})$. For an evaluation of alternate sequence implementations, see [22].

5 Domain Restriction Operations

As seen in Section 2, propagators interact with domains via restriction operations that are exposed by the variable implementation. A restriction operation should satisfy three important properties. First, the operation should be *useful* for the implementation of some propagator. Second, the operation should be *efficient* when executed on the data structure that implements the variable domain. Finally, the operation should be *correct*, meaning that the resulting domain is actually restricted according to the semantics of the operation.

We now describe restriction operations appropriate for a BLS variable type. The data structure chosen in Section 4 allows an efficient implementation of these operations. Interestingly, though, the correct semantics of BLS variable restriction operations is not obvious, so before proceeding we define the semantics of string equality and disequality that is most suited to string variables.

Restriction Operations on String Variables Table 1 lists several restriction operations that might be provided by a string variable implementation. The operations are divided into two categories, affecting either lengths or characters.

The first category of operations works on sets of candidate lengths: the equality (`leq`) operation restricts the domain of the variable to strings of a given length, and two inequality operations tighten the lower (`lgr`) or upper (`lle`) bound of the set of candidate lengths. The second category of operations works on sets of candidate characters: the equality and disequality operations restrict the domain of the variable to strings with a given symbol at the specified index (`ceq`) or to exclude all strings with a given symbol at the specified index (`cnq`), two inequality operations tighten the lower (`cgr`) or upper (`cle`) bound of the set of candidate characters, and the set intersection and set subtraction operations restrict the set of candidate characters to their intersection with a given set of symbols (`cin`) or to exclude a given set of symbols (`cmi`). Note that strict

Table 1. Restriction operations for bounded-length string variables.

restriction		operator
length	equality	$\texttt{leq}(\text{dom}(X), \ell) = \{x \in \text{dom}(X) \mid \lvert x\rvert = \ell\}$
	$\leq$	$\texttt{lle}(\text{dom}(X), \ell) = \{x \in \text{dom}(X) \mid \lvert x\rvert \leq \ell\}$
	$>$	$\texttt{lgr}(\text{dom}(X), \ell) = \{x \in \text{dom}(X) \mid \lvert x\rvert > \ell\}$
character	equality	$\texttt{ceq}(\text{dom}(X), i, c) = \{x \in \text{dom}(X) \mid x[i] = c\}$
	disequality	$\texttt{cnq}(\text{dom}(X), i, c) = \{x \in \text{dom}(X) \mid x[i] \neq c\}$
	$\leq$	$\texttt{cle}(\text{dom}(X), i, c) = \{x \in \text{dom}(X) \mid x[i] \leq c\}$
	$>$	$\texttt{cgr}(\text{dom}(X), i, c) = \{x \in \text{dom}(X) \mid x[i] > c\}$
	intersection	$\texttt{cin}(\text{dom}(X), i, \mathcal{C}) = \{x \in \text{dom}(X) \mid x[i] \in \mathcal{C}\}$
	subtraction	$\texttt{cmi}(\text{dom}(X), i, \mathcal{C}) = \{x \in \text{dom}(X) \mid x[i] \notin \mathcal{C}\}$

greater than ($>$) and non-strict less than ($\leq$) operations have been chosen for each category based solely on the complementarity of the operations; the pairs could just as well have been $<$ and $\geq$, $<$ and $>$, or $\leq$ and $\geq$.

Table 1 omits restriction operations of length disequality, intersection, and subtraction, as these would be incorrect here. For example, a length disequality operation that attempts to remove an interior value from the set of candidate lengths will result in no change to the domain. As sets of candidate characters are assumed to be explicitly represented, the character disequality and intersection operations are included.

Some of the restriction operations in Table 1 can be rewritten using the other operations. For example, the character intersection and subtraction operations are equivalent to a series of character disequality operations. In practice, an implementation using these decompositions may be inefficient; however, for the purpose of *defining* the restriction operations for domains represented by bounded-length sequences, a set of four *required* operations is sufficient: `lle`, `lgr`, `ceq`, and `cnq`, allowing a propagator to increase the lower bound of the string length (`lgr`), decrease the upper bound of the string length (`lle`), fix the character at an index i (`ceq`), or forbid a character at index i (`cnq`).

Restriction Operations as Update Rules Table 2 defines, for string variable domains represented by a b-length sequence, the four chosen restriction operations. They are given as a series of update rules on the components of the affected b-length sequence; for each restriction operation, only those components of the resulting b-length sequence $\langle \boldsymbol{\mathcal{A}}'^b, \mathcal{N}' \rangle$ that differ from the corresponding component of the initial b-length sequence are defined; all other components are unchanged.

Representation Invariant All of the restriction operations in Table 2 are designed to respect the representation invariant of the b-length sequence representation (see Section 3), which enforces the relationship between empty sets of candidate characters and the upper bound of the set of candidate lengths. Hence, the restriction operation $\texttt{lle}(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, \ell)$, which reduces the upper bound of $\mathcal{N}$,

Table 2. Restriction operations for a string variable with a domain represented by a b-length sequence, expressed as update rules. Primed set identifiers ($\mathcal{A}'[i]$ and $\mathcal{N}'$) indicate changed component values in the b-length sequence resulting from the restriction operation; all other components in the resulting b-length sequence are identical to the corresponding component in the original b-length sequence.

restriction operation	**update rule**
$\texttt{lle}(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, \ell)$	$\texttt{forall } k \in [\ell+1, b] : \mathcal{A}'[k] \coloneqq \varnothing;\ \mathcal{N}' \coloneqq \mathcal{N} \cap [0, \ell]$
$\texttt{lgr}(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, \ell)$	$\mathcal{N}' \coloneqq \mathcal{N} \cap [\ell+1, b]$
$\texttt{ceq}(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, i, c)$	$\mathcal{A}'[i] \coloneqq \mathcal{A}[i] \cap \{c\};\ \mathcal{N}' \coloneqq \mathcal{N} \cap [i, b]$
$\texttt{cnq}(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, i, c)$	$\mathcal{A}'[i] \coloneqq \mathcal{A}[i] \setminus \{c\}; \texttt{if } \mathcal{A}'[i] = \varnothing \texttt{ then}$ $(\mathcal{N}' \coloneqq \mathcal{N} \cap [0, i-1];\ \texttt{forall } k \in [i, b] : \mathcal{A}'[k] \coloneqq \varnothing)$

also updates all sets of candidate characters at indices greater than ℓ to be the empty set. The restriction operation $\texttt{lgr}(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, \ell)$ requires no corresponding update on sets of candidate characters with indices less than or equal to ℓ: if there existed an empty set $\mathcal{A}[i] = \varnothing$ of candidate characters such that i was less than or equal to ℓ, then the upper bound of $\mathcal{N}$ would already be at most i; hence $\mathcal{N} \cap [\ell+1, b]$ would also be the empty set, and the domain would be failed.

Equality Semantics `ceq` and `cnq` have very different effects on the considered set of candidate lengths: after `ceq` the set $\mathcal{N}'$ only contains lengths that are at least i; but if $\mathcal{A}'[i]$ is not empty after `cnq`, then the set $\mathcal{N}'$ may contain lengths greater than or equal to i. This appears to run counter to the intuition that equality and disequality should be complementary operations. However, a restriction operation is a function on the *domain* of a variable, not a function on a *component* of that domain; the operations `ceq` and `cnq` are complementary in that together they always define a partition of a string variable domain:

$$\mathrm{dom}\big(\texttt{ceq}\left(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, i, c\right)\big) \cup \mathrm{dom}\big(\texttt{cnq}\left(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, i, c\right)\big) = \mathrm{dom}\big(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle\big) \quad (5)$$

$$\mathrm{dom}\big(\texttt{ceq}\left(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, i, c\right)\big) \cap \mathrm{dom}\big(\texttt{cnq}\left(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, i, c\right)\big) = \varnothing \quad (6)$$

For full details on the equality semantics, see [22].

6 Propagation Events

As seen in Section 2, information about a domain restriction is communicated to a propagator via a *propagation event* [19]. In practice, a propagation event should be useful to some propagator; that is, the propagation event should distinguish between a change to the domain that leaves the propagator at fixpoint, versus one that does not. The set of propagation events exposed by a variable implementation is called a *propagation event system*. Larger propagation event systems come with a commensurate cost to efficiency; see [20] for a detailed analysis of event-based propagation.

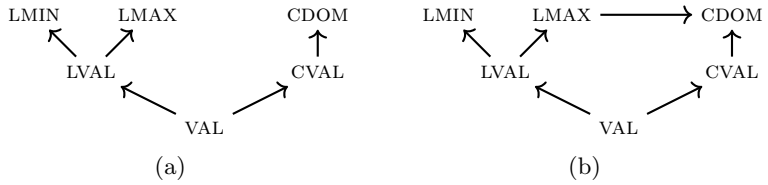

Figure 4. Implications in a minimal string variable event system (omitting transitive implications). If the changes to the set of candidate lengths and the sets of candidate characters are viewed independently (a), then the resulting propagation event system is not monotonic. The corrected propagation event system (b) is monotonic.

A minimal event system for a string variable X with a domain represented by a b-length sequence $\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle$ is shown in Figure 4. The VAL propagation event indicates that the string variable domain has been reduced to a single string. The next three propagation events indicate changes to the set $\mathcal{N}$ of candidate lengths: either the lower bound has increased (LMIN), or the upper bound has decreased (LMAX), or the set has become a singleton (LVAL). The remaining propagation events indicate changes to a set $\mathcal{A}[i]$ of candidate characters: either some symbol has been removed (CDOM) or the symbol of the character at index i has become known (CVAL). Note that CVAL indicates that the character at some index i is a symbol $c \in \Sigma$, and *not* that the set of candidate characters at i is a singleton. If i is a mandatory index, then the two conditions are equivalent: for every string x in the domain of X, the character $x[i]$ is c. However, when an optional set of candidate characters $\mathcal{A}[i]$ is a singleton, then there are additionally strings in the domain for which the character at index i is undefined.

Some propagation events are implied by others. Figure 4(a) shows an event system in which length and character propagation events are isolated, making it possible to report changes to the set of candidate lengths and changes to sets of candidate characters independently. Unfortunately, this design violates one of the properties required of propagation events [24]: the set of propagation events generated by a sequence of restriction operations must be *monotonic*, that is, it must not depend on the order of the restriction operations.

In a non-monotonic propagation event system, the set of events generated by a series of restriction operations is dependent on their order, as shown now:

Example 2. Let X be a string variable with a domain represented by the b-length sequence $\langle \boldsymbol{\mathcal{A}}^4, \mathcal{N} \rangle = \langle \langle \{1,2\}, \{1,2\}, \{5,6\}, \varnothing \rangle, [1,3] \rangle$, where $b = 4$. The following sequence of restriction operations restricts first the set of candidate characters at index 3, and then the set of candidate lengths:

$$\begin{aligned}
\langle \boldsymbol{\mathcal{A}}'^4, \mathcal{N}' \rangle &\coloneqq \texttt{lle}\left(\texttt{cnq}\left(\left\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \right\rangle, 3, 6\right), 2\right) \\
&\coloneqq \texttt{lle}\left(\langle \langle \{1,2\}, \{1,2\}, \{5,6\} \setminus \{6\}, \varnothing \rangle, [1,3] \rangle, 2\right) \\
&\coloneqq \texttt{lle}\left(\langle \langle \{1,2\}, \{1,2\}, \{5\}, \varnothing \rangle, [1,3] \rangle, 2\right) \\
&\coloneqq \langle \langle \{1,2\}, \{1,2\}, \{5\} \cap \varnothing, \varnothing \rangle, [1,3] \cap [0,2] \rangle = \langle \langle \{1,2\}, \{1,2\}, \varnothing, \varnothing \rangle, [1,2] \rangle
\end{aligned}$$

The resulting set of propagation events is $\{\text{CDOM}, \text{LMAX}\}$: the set of candidate characters at index 3 was restricted, followed by the set of candidate lengths. If the order of the two restriction operations is reversed, then the resulting domain representation is the same:

$$\begin{aligned}\langle \boldsymbol{\mathcal{A}}''^4, \mathcal{N}'' \rangle &\coloneqq \texttt{cnq}\left(\texttt{lle}\left(\langle \boldsymbol{\mathcal{A}}^b, \mathcal{N} \rangle, 2\right), 3, 6\right) \\ &\coloneqq \texttt{cnq}\left(\langle\langle\{1,2\},\{1,2\},\{5,6\} \cap \varnothing, \varnothing\rangle, [1,3] \cap [0,2]\rangle, 3, 6\right) \\ &\coloneqq \texttt{cnq}\left(\langle\langle\{1,2\},\{1,2\},\varnothing,\varnothing\rangle,[1,2]\rangle, 3, 6\right) \\ &\coloneqq \langle\langle\{1,2\},\{1,2\},\varnothing \setminus \{6\},\varnothing\rangle,[1,2]\rangle = \langle\langle\{1,2\},\{1,2\},\varnothing,\varnothing\rangle,[1,2]\rangle\end{aligned}$$

but the set of propagation events generated by this sequence of restriction operations is different, namely only $\{\text{LMAX}\}$: no CDOM propagation event is generated, because the `cnq` restriction operation did not change the domain. □

The monotonicity of event systems is important because propagation events are used in the fixpoint algorithm to schedule propagators for execution. A non-monotonic event system makes propagator scheduling non-monotonic in the sense that executing propagators in different orders can generate different propagation events possibly resulting in different amounts of pruning; see [21, 24] for a complete explanation of event systems and efficient propagator scheduling.

Figure 4(b) shows a monotonic version of our propagation event system, in which LMAX implies CDOM. Intuitively, this implication arises from the representation invariant: any change to the upper bound of the set $\mathcal{N}$of candidate lengths must empty at least one set $\mathcal{A}[i]$ of candidate characters. A proof of the monotonicity of this propagation event system is omitted for reasons of space: see [22] for further discussion.

7 Experimental Evaluation

Experimental Methodology Experiments were carried out on a VirtualBox 4.3.10 virtual client with 1,024 MB of RAM, running Xubuntu 14.04. The host machine was a 2.66 GHz Intel Core 2 Duo with 4 GB of RAM, running OpenSUSE 13.1. Code for implemented propagators was written in C++ for the Gecode 4.4.0 constraint solving library, using 64-bit bitsets, and compiled with GCC 4.8.4.

For each problem including strings of unknown length, the same initial maximum length b was used for every string variable, and the experiments were run for several possible values of b when possible. Timeout always was at 10 minutes.

Models and Implementations BLS variables are compared with two other bounded-length string representations for finite-domain CP solvers.

The first of these methods, the *de facto* standard for solving bounded-length string constraint problems using a CP solver, is the *padded-string method* [12], which requires no proper string variable type; instead, each string unknown in a problem is modeled as a pessimistically large array of integer variables, allowing multiple occurrences of a null or padding symbol at the end of each string. In

the padded-string method there are no propagators implementing string constraints. Instead, each string constraint is modeled as a decomposition consisting of a conjunction of reified constraints over the sequence of integer variables corresponding to each string in the scope of the constraint, which express the relationship between the length of the modeled string and the occurrences of the padding character in the corresponding sequence.

The second method, the *aggregate-string method* [23], is similar to the padded-string method, but an integer variable is added for each string unknown, modeling its set of candidate lengths. Thus, the model of a string unknown in the aggregate-string method is isomorphic to the bounded-length sequence representation in Section 3, modulo the inclusion of the padding character. The aggregate-string method also differs from the padded-string method in that each string constraint is implemented by a single propagator. This method is implemented with the aid of the indexical compiler [15].

All implementations, as well as the corresponding models for the benchmarks listed below, can be found at `https://bitbucket.org/jossco/gecode-string`. Note that these experiments do not use the machinery of the recent string extension [2] of the MiniZinc modeling language [17].

Search A serviceable, if not compelling, branching heuristic for string variables applies a value selection heuristic for integer variables to either a set of candidate characters or the set of candidate lengths. More interesting branching heuristics for string variables should be explored; however, a simple heuristic is sufficient for the purpose of comparing our string variable type with the two alternate methods described above. We used the following heuristic: at each choice point, the first unassigned string variable is selected; the sets of candidate characters at the mandatory indices are evaluated, and the set with the lowest cardinality is selected; if there exist no mandatory indices, then the minimum of the set of candidate lengths is increased instead. Character value selections are made by splitting the selected set at its median element, and taking the lower half first.

Benchmarks The well-known string benchmarks of HAMPI [9], KALUZA [18], and SUSHI [8] have previously been shown to be trivial for CP solvers even without sequence variables, see [12, 23] for instance, hence we do not revisit them here.

Word Design for DNA Computing on Surfaces This problem [6], with origins in bioinformatics and coding theory, is to find the largest set of strings $\mathcal{S}$, each of length 8 and with alphabet $\Sigma = \{\mathtt{A}, \mathtt{T}, \mathtt{C}, \mathtt{G}\}$, such that:

- Each string $s \in \mathcal{S}$ contains exactly four characters from the set $\{\mathtt{C}, \mathtt{G}\}$.
- For all $x, y \in \mathcal{S}$ such that $x \neq y$, x and y differ in at least four positions.
- For all $x, y \in \mathcal{S}$ (including when $x = y$), the strings x^{rev} and $\text{comp}(y)$ differ in at least four positions, where comp is the permutation $(\mathtt{AT})(\mathtt{CG})$.

Despite its name, this problem is actually rather a weak candidate for modeling with bounded-length string variables. Every word in $\mathcal{S}$ has the same fixed length, and most of the constraints are binary constraints on characters; hence, the

Table 3. Time, in seconds, either to find a solution if one exists, or to prove b-bounded unsatisfiability otherwise; `t` indicates that the instance timed out (> 10 minutes).

		padded			**aggregate**			**BLS**			**SAT /**
benchmark	inst \ b	256	512	1024	256	512	1024	256	512	1024	**UNSAT**
WordDesign	80		216.0			221.3			25.8		sat
WordDesign	85		290.0			t			32.9		sat
WordDesign	112		t			t			92.5		sat
ChunkSplit	16	t	t	t	t	t	t	0.2	2.0	21.9	sat
ChunkSplit	25	t	t	t	t	t	t	1.4	15.0	215.2	sat
ChunkSplit	29	t	t	t	50.8	t	t	0.3	2.0	21.7	sat
Levenshtein	2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	sat
Levenshtein	37	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	sat
Levenshtein	84	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	sat
$a^n b^n$	89	2.0	t	t	0.2	0.5	1.7	0.1	0.3	2.3	sat
$a^n b^n$	102	2.0	2.1	2.0	0.1	0.1	0.1	0.5	4.5	50.3	unsat
$a^n b^n$	154	0.1	0.1	0.4	0.1	0.2	0.5	0.4	3.1	34.0	unsat
StringReplace	20	t	t	t	t	t	t	t	t	t	---
StringReplace	136	0.9	3.5	t	0.9	3.6	t	0.1	0.2	1.6	unsat
StringReplace	142	21.0	t	t	0.1	0.4	1.5	0.1	0.3	1.8	sat
Hamming	389	t	t	t	t	t	t	0.4	3.9	59.5	unsat
Hamming	1005	8.5	t	t	7.9	t	t	0.7	5.8	61.8	unsat
Hamming	1168	t	t	t	t	t	t	0.4	3.7	55.9	unsat

problem is easily modeled as an $|\mathcal{S}| \times 8$ matrix of integer variables. In such a fixed-length string case, a proper string variable type seems to have relatively little to offer over a fixed-length array of integer variables.

Experimental results for the Word Design problem are shown in Table 3. We treat each value (shown in the instance column) of the cardinality of $\mathcal{S}$ as a satisfaction problem. No implementation times out for $|\mathcal{S}| \leq 80$, only the aggregate implementation times out for $80 < |\mathcal{S}| \leq 85$, only the BLS implementation does not time out for $85 < |\mathcal{S}| \leq 112$, and all implementations time out for $112 < |\mathcal{S}|$. As all strings in the problem are of fixed length, there is nothing to be gained by varying the maximum string length b for the problem; hence, we report only one run for $b = 8$ per cardinality for each implementation.

The model using BLS variables performs comparatively well. As all strings in the problem are of fixed length, the padded and aggregate methods collapse into a single method; the propagators provided by the aggregate model for the purpose of treating fixed-length arrays of variables as bounded-length strings are unhelpful for true fixed-length strings. In this fixed-length context, the superior performance of BLS variables must be attributed to the choice of a bitset representation for the alphabet.

Benchmark of NORN A set of approximately 1,000 string constraint problems were generated for the *unbounded*-length string solver NORN [1]. These instances do not require Unicode and have regular-language membership constraints, gen-

erated by a model checker, based on counter-example-guided abstraction refinement (CEGAR), for string-manipulating programs, as well as concatenation and length constraints on string variables, and linear constraints on integer variables.

Instances of the benchmark of NORN are written in the CVC4 dialect of the SMTLIB2 language [14]. These instances were translated into GECODE models using the three methods described above. Regular languages in the instances are specified as regular expressions; these were directly translated into GECODE's regular-expression language and modeled by $\text{REGULAR}_{\mathcal{L}}$ constraints, with the exception of expressions of the form $X \in \epsilon$, which were instead modeled by a constraint $\text{LEN}(X, 0)$. Approximately three quarters of all instances in the benchmark include a negated regular expression. As GECODE does not implement negation for regular expressions, these instances were omitted from the experiments as a matter of convenience; adding support for taking the complement of regular languages to GECODE is straightforward (e.g., [13]).

We solve the remaining 255 instances in a *bounded*-length context; our results are therefore incomparable with those of an unbounded-length solver such as NORN. For satisfiable instances, NORN generates a language of satisfying assignments for each string variable, whereas a CP-based method returns individual satisfying strings; furthermore, NORN can determine that an instance is unsatisfiable for strings of *any* length, whereas CP solvers are limited to determining b-bounded unsatisfiability, for some practical upper bound b on string length.

Nevertheless, the benchmark of NORN remains interesting in a *bounded*-length context, as there are several challenging instances to be found. Complete results for the 255 evaluated instances are omitted for space; for further discussion, see [22]. Table 3 shows results for three instances in each of the five categories of the benchmark of NORN; these instances were selected as they appear to be the hardest, in their respective categories, for solving in a bounded-length context.

As shown in Table 3, the BLS variable implementation is either significantly faster than the aggregate and padding implementations, or tied with them, except (for reasons we failed so far to understand) on the unsatisfiable instances of the a^nb^n benchmark. For large upper bounds on string sizes ($b > 256$), both of the decomposition-based implementations are prone to time outs, mostly likely as the search space is exhausting the available memory.

8 Conclusion

We have designed a new variable type, called bounded-length sequence (BLS) variables. Implemented for the copying CP solver GECODE, BLS variables ease the modeling of string constraint problems while simultaneously providing considerable performance improvements over the alternatives. The described extension is agreed to become official part of GECODE. It would be interesting to see how our ideas transpose to a trailing CP solver.

Bitsets are less appropriate for large alphabets, say when when full Unicode (16 bit) coverage is required: future work includes adapting the choice of

character-set representation based on alphabet size, possibly using BDDs, or a sparse-bitset representation similar to that of [5].

BLS variables, together with propagators for string constraints [22], are part of an emerging ecosystem of string solving in CP. The recent string extension [2] of the MiniZinc modeling language [17] — for which we have extended the FlatZinc interpreter of Gecode in order to support the BLS variable extension — can only serve to encourage further development.

Acknowledgements. We thank the anonymous referees for their helpful comments. The authors based at Uppsala University are supported by the Swedish Research Council (VR) under grant 2015-4910.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification (CAV 2015). pp. 462–469. No. 9206 in LNCS, Springer (2015), the benchmark of Norn is available at `http://user.it.uu.se/~jarst116/norn`
2. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: MiniZinc with strings. In: Hermenegildo, M., López-García, P. (eds.) Logic-Based Program Synthesis and Transformation (LOPSTR 2016), Pre-Proceedings. No. 1608.02534 in Computing Research Repository (2016), available at `https://arxiv.org/abs/1608.03650`
3. Bisht, P., Hinrichs, T., Skrupsky, N., Venkatakrishnan, V.N.: WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Computer and Communications Security (CCS 2011). pp. 575–586. ACM (2011)
4. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009). pp. 307–321. No. 5505 in LNCS, Springer (2009)
5. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J., Schaus, P.: Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In: Rueher, M. (ed.) Principles and Practice of Constraint Programming (CP 2016). LNCS, vol. 9892, pp. 207–223. Springer (2016), `http://dx.doi.org/10.1007/978-3-319-44953-1_14`
6. van Dongen, M.: CSPLib problem 033: Word design for DNA computing on surfaces. `http://www.csplib.org/Problems/prob033`
7. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Rosenblum, D.S., Elbaum, S.G. (eds.) Software Testing and Analysis (ISSTA 2007). pp. 151–162. ACM (2007)
8. Fu, X., Powell, M.C., Bantegui, M., Li, C.C.: Simple linear string constraints. Formal Aspects of Computing 25, 847–891 (November 2013)
9. Ganesh, V., Kieżun, A., Artzi, S., Guo, P., Hooimeijer, P., Ernst, M.: HAMPI: A string solver for testing, analysis and vulnerability detection. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification (CAV 2011). pp. 1–19. No. 6806 in LNCS, Springer (2011)

10. 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.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013). pp. 277–291. No. 7795 in LNCS, Springer (2013)
11. Golden, K., Pang, W.: A constraint-based planner applied to data processing domains. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming (CP 2004). p. 815. No. 3258 in LNCS, Springer (2004)
12. He, J., Flener, P., Pearson, J.: Solving string constraints: The case for constraint programming. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming (CP 2013). No. 8124 in LNCS, Springer (2013)
13. Hopcroft, J., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, 3rd edn. (2007)
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: Biere, A., Bloem, R. (eds.) Computer Aided Verification (CAV 2014). pp. 646–662. No. 8559 in LNCS, Springer (2014)
15. Monette, J.N., Flener, P., Pearson, J.: Towards solver-independent propagators. In: Milano, M. (ed.) Principles and Practice of Constraint Programming (CP 2012). pp. 544–560. No. 7514 in LNCS, Springer (2012), indexical compiler software available at `http://www.it.uu.se/research/group/astra/software/indexicals`
16. Négrevergne, B., Guns, T.: Constraint-based sequence mining using constraint programming. In: Michel, L. (ed.) Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015). pp. 288–305. No. 9075 in LNCS, Springer (2015)
17. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming (CP 2007). pp. 529–543. No. 4741 in LNCS, Springer (2007), the MiniZinc toolchain is available at `http://www.minizinc.org`
18. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Security and Privacy (S&P 2010). pp. 513–528. IEEE Computer Society (2010)
19. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 14, pp. 495–526. Elsevier, Amsterdam (2006)
20. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. Transactions on Programming Languages and Systems 31(1), 2:1–2:43 (December 2008)
21. Schulte, C., Tack, G.: Implementing efficient propagation control. In: Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a conference workshop of CP 2010. St Andrews, UK (09 2010)
22. Scott, J.: Other Things Besides Number: Abstraction, Constraint Propagation, and String Variable Types. Ph.D. thesis, Uppsala University, Sweden (2016), available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-273311`
23. Scott, J., Flener, P., Pearson, J.: Constraint solving on bounded string variables. In: Michel, L. (ed.) Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015). pp. 375–392. No. 9075 in LNCS, Springer (2015)
24. Tack, G.: Constraint Propagation: Models, Techniques, Implementation. Ph.D. thesis, Saarland University, Germany (2009)
25. The Gecode Team: Gecode: A generic constraint development environment (2006), `http://www.gecode.org`