Constraint Solving on Bounded String Variables

Joseph D. Scott, Pierre Flener, and Justin Pearson

Uppsala University, Uppsala, Sweden
first.last@it.uu.se

Abstract Constraints on strings of unknown length occur in a wide variety of real-world problems, such as test case generation, program analysis, model checking, and web security. We describe a set of constraints sufficient to model many standard benchmark problems from these fields. For strings of an unknown length bounded by an integer, we describe propagators for these constraints. Finally, we provide an experimental comparison between a state-of-the-art dedicated string solver, CP approaches utilising fixed-length string solving, and our implementation extending an off-the-shelf CP solver.

1 Introduction

Constraints on strings occur in a wide variety of problems, such as test case generation [8], program analysis [6], model checking [11], and web security [5].

As a motivating example, we consider the *symbolic execution* [7,20] of string-manipulating programs. Symbolic execution is a semantics for a programming language, wherein program variables are represented by symbols, and language operators are redefined to accept symbolic inputs and produce symbolic expressions. In symbolic execution, a program P is represented by a *control flow graph* [2]: a directed graph with nodes representing the basic blocks in P, and arcs representing possible branchings. A *path* on a control flow graph is a finite sequence of arc-connected nodes. A *symbolic state* for a path π on a program P consists of a mapping, μ, from the program variables of P to symbolic expressions, and a *path constraint*, PC, which is associated with the path π, over the symbols used in μ. Solving the path constraint results in either a set of concrete inputs that yields an execution following the path π, or, when PC is unsatisfiable, a proof that π is an infeasible path.

Example 1. In Fig. 1 is some JavaScript-like code (it is uninteresting, but small enough to illustrate our points), with a corresponding control flow graph. For the path $\pi = 1\text{-}2\text{-}4\text{-}5\text{-}6$, a corresponding path constraint may be as follows:

$$PC_\pi : y = |s| \land y \bmod 2 = 0 \land s \in \mathrm{L}((\mathrm{a}^*\mathrm{b})\mathrm{c}\backslash 1) \land x = s_{1\,:\,y/2} \tag{1}$$

(Notation will be introduced in Section 2 but should not be an obstacle here.) Note that PC_π is unsatisfiable: only a string of odd length can match the expression on line 4, due to the required symbol ‘c’ in the middle, so if the condition on that line is true, then the condition on line 2 must also be true, and therefore π is infeasible, as node 3 should be visited at least once between nodes 2 and 4.

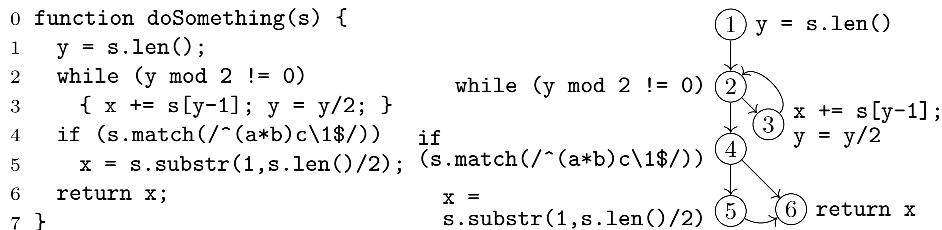

Figure 1. JavaScript-like code with a corresponding control flow graph

Example 1 helps to illuminate the type of string constraints needed for software verification purposes: the constraint language should be rich enough to model the kinds of string operators typically found in programming languages. For example, the `substr()` operation on line 5 of Fig. 1 suggests that a constraint stating "string y is a substring of x, starting from index i" would be useful. Similarly, the `match()` operation on line 4 suggests the utility of a constraint stating "string x is a member of the regular language $\mathcal{L}$". However, this second constraint is somewhat misleading, as the '`\1`' in the pattern on line 4 is a *back-reference*: the parentheses delineate a subexpression (in this case, '`a*b`'), and the '`\1`' indicates that the *value* matched by that subexpression is repeated in the same string. In the absence of a bound on string length, the languages defined by expressions with back-references are not regular, but rather context sensitive.[1] Nevertheless, back-references are a common feature of regular expressions as implemented in modern programming languages, and hence are a feature we would like to model. To avoid confusion, we will write *regular expression* to refer *only* to the formal language concept, while we will write *regex* to refer to a possibly non-regular pattern allowed by a programming language.

String constraint solving has been the focus of a large amount of research in recent years. Current string constraint solving methods may be broadly classified by their treatment of string length. At one extreme are solvers for string variables of *unbounded* length, such as [1,9,11,15,17,18,23,33,36]. These solvers define the set of all satisfying strings intensionally, typically by formal languages. Constraint reasoning in these solvers generally reduces to a question of language intersection; research centres on the question of how to avoid the exponential blowup of these intersection operations. At the opposite end of the spectrum, *fixed*-length string solvers, such as [19,24], are extensional in the sense that they generate solutions individually. Fixed-length solvers are generally superior to unbounded-length solvers for producing a single solution, but suffer comparatively when producing the set of all solutions.

[1] In fact, for strings of *bounded* length, expressions with back-references *do* correspond to regular languages; however the size of a finite-automaton encoding grows exponentially in the size of the bound, so even in the bounded case back-references tend to result in inefficient encodings.

In this paper, we address a problem between these two extremes: we consider string variables of *bounded* length. While bounded-length solvers exist in other fields, such as [6, 29], to this point *constraint programming* (CP) models have handled bounded-length strings either by iterating over a series of fixed string lengths, or by representing a string variable as an array of variables long enough to accommodate the maximum considered string length, while allowing occurrences of a padding symbol at the end of the string (e. g., [16]). We focus instead on encoding the string length and contents directly; nevertheless, our implementation (Sect. 6) can be seen as an encapsulation of padding into a variable type. Using fixed-length strings with a padding character is appealingly simple in theory, but in practice it leads to complicated and error-prone models. Our approach allows simpler modelling, without requiring an extension to the solver's modelling language, by introducing a new structured variable type for strings. In this framework, the choice to use a padding character and the consequences of that choice are implementation details, which may be ignored during modelling.[2] We also note that bounded-length string variables ease the design of string-specific branching heuristics.

The contributions and organisation of this paper are as follows, after defining notation and terminology (Sect. 2) and outlining related work (Sect. 3):

- a formalisation of string variables and a specification of several interesting string constraints, all applicable to strings of fixed, bounded, or unbounded length (Sect. 4);
- a definition of a bounded-length string variable representation, called the *open-sequence representation*, which is directly implementable for any existing finite-domain CP solver, and propagator descriptions for the specified string constraints (Sect. 5);
- an implementation of our bounded-length string variable representation and a principled derivation of actual propagators for the specified constraints, all for the CP solver GECODE [12] (Sect. 6);
- an experimental evaluation of our implementation: despite being only a prototype, it already outperforms not only off-the-shelf fixed-length CP approaches [16], but also, by orders of magnitude, the state-of-the-art dedicated string solvers SUSHI [9] and KALUZA [29], on their benchmarks (Sect. 7).

Finally, we conclude in Sect. 8.

2 Notation and Terminology for Strings and Languages

An *alphabet* Σ is a finite set of *symbols*. A *string* s of length $|s| = n$ over an alphabet Σ is a finite sequence of n symbols of Σ, denoted $s_1 s_2 \cdots s_n$, where $s_i \in \Sigma$ for all $1 \leq i \leq n$. For a given string s, we denote its ith symbol by s_i. We denote the *empty string*, of length 0, by ϵ. We denote the *concatenation* of

[2] As noted elsewhere (e. g., [13]), the case for structured variable types is similar to that for global constraints: both capture commonly recurring combinatorial substructure.

strings x and y by $x \cdot y$. We say that a string y is a *substring* of a string s if there exist strings x and z such that $s = x \cdot y \cdot z$. For $1 \leq i, j \leq |s|$, we define $s_{i:\,j}$ as the substring $s_i \cdots s_j$ from the ith to the jth symbol of s; if $i > j$, then $s_{i:\,j} = \epsilon$. The *reverse* of a string $s = s_1 \cdots s_n$ is the string $s^{\text{rev}} = s_n \cdots s_1$.

We denote by Σ^n the set of strings over Σ of length n. The infinite set of all strings over Σ, including ϵ, is denoted by Σ^*. A *language* over Σ is a possibly infinite subset of Σ^*. The language of a regex r is denoted by $\mathrm{L}(r)$.

A *constraint* C of *arity* k is a pair $\langle R, S \rangle$ where R is the underlying relation on ground instances of the variable tuple $S = \langle X_1, \ldots, X_k \rangle$, called the *scope* of C. We denote the *domain* of a variable X by $\mathcal{D}(X)$.

We denote scalar variables in uppercase (e. g., N, N_1, etc. for integers, and A for a symbol of a finite alphabet) and string variables (to be introduced in Sect. 4) in boldface uppercase (e. g., $\mathbf{S}$). We denote sets in script (e. g., $\mathcal{A}$, $\mathcal{B}$, etc.), and write $|\mathcal{A}|$ for the cardinality of a set $\mathcal{A}$. We refer to the set of integers $\{\ell, \ell+1, \ldots, u-1, u\}$, which is the empty set $\emptyset$ if $\ell > u$, using the notation $[\ell, u]$. We use angled brackets to denote an ordered sequence, or tuple, $\langle a_1, \ldots, a_n \rangle$.

3 Related Work

We distinguish between string variables of fixed, unbounded, and bounded length.

Fixed-Length String Variables. In CP, a fixed-length string variable has a natural representation as an array of scalar variables that may be acted upon by a wide variety of constraints. Of particular interest for solving string problems are constraints for membership in regular [4, 26] and context-free [27, 32] languages. For example, the propagator in [26] for regular language membership works by maintaining a *layered graph*: each layer replicates the nodes of a finite automaton representing the regular language, but with each transition connecting to a node in the next layer. The labels of the arcs between nodes in two consecutive layers determine the feasible values for the corresponding variable in the string, and propagation works by removing arcs not on a path between the start node in the first layer and any accepting node in the last layer. Fixed-length bit-vector variables have also been explored [24].

HAMPI [19] provides a theory of fixed-length strings for *satisfaction modulo theories* (SMT) solvers, using the bit-vector solver STP [10]. HAMPI handles constraints of membership in both regular and context-free languages. For a set of such constraints, on a *single* fixed-length string variable, HAMPI either returns one satisfying string, or reports that the constraints are unsatisfiable.

Unbounded-Length String Variables. At the other extreme are solvers for string variables of unbounded length. An example of this approach in CP is [15], in which the *regular domain* of a string variable is defined by a regular language. A regular domain is represented as a finite automaton accepting that language, and propagation of a constraint over string variables is achieved by computing a series of automaton operations, such as intersection or negation. The expressivity

of regular domains is balanced out by relatively expensive propagation: several of the presented propagators take time quadratic in the size of the automata, and the size of the automata themselves may grow exponentially with the number of constraints. It is not surprising that performing propagation on string variables of unbounded length by computing on a set of strings is expensive. The equivalent for integer domains would be propagation over multiple integer variables through computation on value tuples, which is not generally reasonable. Constraints of regular language membership are, of course, trivially enforceable on the regular domain, although extension to context-free languages is impractical.

A decision procedure for Boolean combinations of equalities on unbounded-length string variables, called word equations, is provided in [23]. A *word equation* [21] is a constraint such as $x \oplus y = z$, where $\oplus$ is a string operator and x, y, z are string variables. Word equations are not decidable in the general case, and their decidability in conjunction with other constraints, including length constraints, remains open [6]. Nonetheless, for fragments of the logic of word equations with constraints on length or regular or context-free language membership, there exist several decision procedures. For example, SUSHI [9] handles a restricted fragment of word equations called *simple linear string equations* (SLSE); in essence, these are word equations in which no string variable appears more than once, and string variables occur only on the left-hand side. SUSHI allows concatenation, substring, regular membership, and regular replacement. Other solvers handling weak fragments include the stand-alone solver DPRLE [17], which handles only language subset and language concatenation constraints for regular languages, and Z3-STR [36], an extension for the SMT solver Z3 that provides a theory of word equations with length constraints, but does not include language membership. The algorithm in NORN [1] is sound for the complete logic of word equations with both length and regular language membership constraints, and is a decision procedure for a restricted fragment (strictly stronger than that of SUSHI). Also, S3 [33] improves the Z3-STR solver and adds a procedure for unfolding unbounded repetitions in regular expressions.

Another line of work has focused on avoiding the exponential blowup encountered in language intersection. Both REVENANT [11] and NORN utilise interpolation, albeit in different contexts, while StrSolve [18] handles automaton intersection operations by lazily constructing cross-products.

Also worth noting is that automaton-based approaches do not allow a natural handling of length constraints. The latter may be directly encoded as automata (e. g., [35]); however, this results in only a weak connection between string lengths and other numerical constraints. Solvers that combine automata and numerical reasoning [1, 14, 28] strengthen this connection to varying degrees.

Bounded-Length String Variables. Less work has been done on bounded-length string solvers. Probably the best known solver in this category is KALUZA [29], which solves constraints in two stages. First, an SMT solver is used to find possible lengths for strings that satisfy explicit length constraints, length constraints implied by the string constraints of the problem, and any other integer constraints present in the model. Second, these lengths are applied to create a

fixed-length bit-vector problem, solved with STP [10] in the same manner as by HAMPI. If the problem in the second stage is unsatisfiable, then the first stage is repeated, with the addition of new constraints to avoid previously tried lengths. Further, a stand-alone bit-vector solver for bounded-length strings is described in [6]; however, it does not handle regular language membership, and no information is propagated from numerical constraints to the string variables.

In CP, propagation of constraints for bounded-length sequences of variables is described in [22], which treats *open* global constraints. A constraint is *global* if the cardinality of its scope is not determined 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 process progresses [3]. In [22], the scope of an open global constraint is a sequence of scalar variables with a length that that has an upper *bound* that is an integer variable. During the solving process, scalar variables are *added* to the end (never the beginning) of the sequence, in connection with changes to the bounds of the length. We here take inspiration from that work, particularly in regard to propagation for constraints of regular and context-free language membership; nevertheless, the two approaches are essentially orthogonal.

In [31], we introduced a representation for bounded-length string variables by prefix-suffix pairs, and we designed propagators for this representation in an *ad hoc* way, testing them only on a *home-made* benchmark. We here introduce a *much simpler* representation, leading to propagators that are *different* and achieve an *incomparable* level of consistency, show how to derive such propagators in a *principled* way, and test them on *third-party* standard benchmarks.

4 String Variables and String Constraints

In a model of a constraint problem, we refer to unknown strings over a finite alphabet Σ as *string variables.* The most precise representation of the domain of a string variable is a subset of Σ^*; in other words, such a domain of a string variable is the language of all strings that are not (yet) known to be infeasible. Operations on this representation are expensive [15], making the representation unsuitable for propagation. We use this representation as an ideal *starting* point, suitable for strings of fixed, bounded, or unbounded length. We introduce in Sect. 5 a representation more suited to propagation.

We divide constraints involving string variables into three groups: pure string constraints, mixed string constraints, and language membership constraints.

Pure String Constraints. We refer to constraints involving only string variables as *pure string constraints.*

The constraint $\text{EQ}(\mathbf{X}, \mathbf{Y})$ holds if string variables $\mathbf{X}$ and $\mathbf{Y}$ are equal, where equality for strings means that they have equal length and the same symbol at each index. The underlying relation $\mathcal{E}$ is $\{\langle x, y\rangle \mid |x| = |y| \land \forall i \in [1, |x|] : x_i = y_i\}$.

The constraint $\text{NEQ}(\mathbf{X}, \mathbf{Y})$ holds if string variables $\mathbf{X}$ and $\mathbf{Y}$ are not equal, where inequality for two strings holds if the strings have different lengths, or

if there exists an index for which the two strings have a different symbol. The underlying relation is $\{\langle x, y\rangle \mid |x| \neq |y| \vee \exists i \in [1, |x|] : x_i \neq y_i\}$.

The constraint $\textsc{Rev}(\mathbf{X}, \mathbf{Y})$, for string variables $\mathbf{X}$ and $\mathbf{Y}$, holds if $\mathbf{X}$ is equal to the reverse of $\mathbf{Y}$. The underlying relation is $\{\langle x, y\rangle \mid x = y^{\mathrm{rev}}\}$.

The constraint $\textsc{Cat}(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ holds if string variable $\mathbf{Z}$ is the concatenation of string variables $\mathbf{X}$ and $\mathbf{Y}$. The underlying relation is $\{\langle x, y, z\rangle \mid z = x \cdot y\}$.

Mixed String Constraints. We refer to constraints involving at least one string variable and at least one non-string variable as *mixed string constraints.*

The constraint $\textsc{Sub}(\mathbf{X}, \mathbf{Y}, N)$ holds if string variable $\mathbf{Y}$ is a contiguous substring of string variable $\mathbf{X}$, starting at the index given by the integer variable N. The underlying relation is $\{\langle x, y, n\rangle \mid y = x_{n:\, n+|y|-1}\}$.

For the special case of $\textsc{Sub}$ in which $\mathbf{Y}$ has a fixed length of one (i. e., where $\mathbf{Y}$ can be replaced by a scalar variable), we instead propose $\textsc{Char}(\mathbf{X}, A, N)$, whose underlying relation is $\{\langle x, a, n\rangle \mid x_n = a\}$.

The constraint $\textsc{Len}(\mathbf{X}, N)$ holds if the string variable $\mathbf{X}$ has a length equal to the integer variable N. The underlying relation is $\{\langle x, n\rangle \mid n = |x|\}$.

Language Membership Constraints. Conceptually, a constraint that holds if a string variable $\mathbf{X}$ is a member of a given formal language $\mathcal{L}$ may be viewed as a unary constraint on $\mathbf{X}$, parameterised by $\mathcal{L}$, irrespective of the class of $\mathcal{L}$. In practice, propagators for such a *language membership constraint* are specific to the language class, so it is common to name such constraints by the language class. For a language $\mathcal{L}$, we have the constraints $\textsc{Regular}(\mathbf{X}, \mathcal{L})$, if $\mathcal{L}$ is regular, and $\textsc{ContextFree}(\mathbf{X}, \mathcal{L})$, if $\mathcal{L}$ is context-free, with $\mathcal{L}$ as the underlying relation.

Example 2. Consider once again the path π = 1-2-4-5-6 of Example 1. We can now express its path constraint PC_π in (1) using the string constraints defined in this section, along with some primitive numerical constraints:

$$\textsc{Len}(\mathbf{S}, Y) \wedge \textsc{Mod}(Y, 2, 0) \wedge \textsc{Regular}(\mathbf{S}_1, \mathrm{L}(\mathrm{a}^*\mathrm{b})) \wedge \\ \textsc{Cat}(\text{“c”}, \mathbf{S}_1, \mathbf{S}_2) \wedge \textsc{Cat}(\mathbf{S}_1, \mathbf{S}_2, \mathbf{S}) \wedge \textsc{Div}(Y, 2, Z) \wedge \textsc{Len}(\mathbf{X}, Z) \wedge \textsc{Sub}(\mathbf{S}, \mathbf{X}, 1)$$

Note the use of Cat in eliminating the back-reference in the regex `/^(a*b)c\1$/`.

5 Open-Sequence Representation and Propagation

As previously noted, a language is a natural representation of the domain of a string variable, but the complexity of computation over languages makes this representation unsuitable for propagation. As a more practical representation, we consider an over-approximation of a finite set of strings, upon which we describe propagators for the constraints defined in Sect. 4.

Open-Sequence Representation. Inspired by [22], we now introduce a string variable representation, called the *open-sequence representation*: $\langle \boldsymbol{\mathcal{A}}, \mathcal{N} \rangle$ consists of a sequence $\boldsymbol{\mathcal{A}} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_m \rangle$ of sets over the same alphabet, and a set $\mathcal{N}$ of natural numbers, representing the possible lengths of the string, with $\max(\mathcal{N}) \leq m$. An $\langle \boldsymbol{\mathcal{A}}, \mathcal{N} \rangle$ pair corresponds to the set of all strings that have a length $\ell \in \mathcal{N}$ and are constructed by selecting a symbol from $\mathcal{A}_i$ at each index $i \in [1, \ell]$; in other words, the domain of a string variable represented by $\langle \boldsymbol{\mathcal{A}}, \mathcal{N} \rangle$ is given by:

$$\mathcal{D}(\langle \boldsymbol{\mathcal{A}}, \mathcal{N} \rangle) = \bigcup_{\ell \in \mathcal{N}} \left\{ s \in \Sigma^\ell \;\middle|\; \forall i \in [1, \ell] : s_i \in \mathcal{A}_i \right\} \tag{2}$$

Intuitively, (2) shows that if any $\mathcal{A}_i$ is empty, then $\mathcal{D}(\langle \boldsymbol{\mathcal{A}}, \mathcal{N} \rangle)$ contains no strings of length at least i. This insight leads to the following representation invariant:

$$\forall i \in [1, m] : \mathcal{A}_i = \emptyset \iff \max(\mathcal{N}) < i \tag{3}$$

Note that we are purposefully general in this section: in Sect. 6, we discuss *a* possible implementation of the open-sequence representation, namely by treating the value sets as the domains of scalar variables. However, the open-sequence representation could *also* be implemented for a CP solver as a new first-class string variable type. In this section, we consider an $\langle \boldsymbol{\mathcal{A}}, \mathcal{N} \rangle$ pair as the representation of a string variable, without regard to the choice of implementation.

Open-Sequence Propagators. We now describe, for some representative constraints specified in Sect. 4, the strongest pruning that may be achieved by a propagator implementing that constraint for string variables that are *all* in the open-sequence representation. In Sect. 6, we will use these propagator descriptions as the basis for automatically generating an implementation of the pure and mixed string constraints.

We give the following propagation descriptions, which specify exactly what pruning can be achieved in the open sequence representation. It is possible to derive the propagator descriptions in a principled manner, using a methodology that has been omitted from this paper for reasons of space and orthogonality.

The propagator for EQ performs set intersections between $\boldsymbol{\mathcal{A}}^x$ and $\boldsymbol{\mathcal{A}}^y$:

$$\begin{aligned} &\mathrm{EQ}^{\mathrm{P}} \left(\langle \boldsymbol{\mathcal{A}}^x, \mathcal{N}^x \rangle, \langle \boldsymbol{\mathcal{A}}^y, \mathcal{N}^y \rangle \right) \\ &= \left\langle \begin{array}{l} \left\langle \left\langle \mathcal{A}^x_1 \cap \mathcal{A}^y_1, \ldots, \mathcal{A}^x_{\max(\mathcal{N}^x \cap \mathcal{N}^y)} \cap \mathcal{A}^y_{\max(\mathcal{N}^x \cap \mathcal{N}^y)}, \emptyset, \ldots, \emptyset \right\rangle, \mathcal{N}^x \cap \mathcal{N}^y \right\rangle, \\ \quad \left\langle \left\langle \mathcal{A}^x_1 \cap \mathcal{A}^y_1, \ldots, \mathcal{A}^x_{\max(\mathcal{N}^x \cap \mathcal{N}^y)} \cap \mathcal{A}^y_{\max(\mathcal{N}^x \cap \mathcal{N}^y)}, \emptyset, \ldots, \emptyset \right\rangle, \mathcal{N}^x \cap \mathcal{N}^y \right\rangle \end{array} \right\rangle \end{aligned}$$

Note that EQ^{P} does not enforce the representation invariant (3): a separation between the invariant and the propagators presented in this section significantly simplifies design, at the level of both theory and implementation.

Example 3. If $\mathbf{X}$ and $\mathbf{Y}$ are string variables with open-sequence representations $\mathbf{X} = \langle \langle [1,3], \{3\}, [1,3], \emptyset, \ldots \rangle, [2,3] \rangle$ and $\mathbf{Y} = \langle \langle [1,2], [1,2], [1,2], \emptyset, \ldots \rangle, [1,3] \rangle$, then propagation by EQ^{P} yields $\mathbf{X}' = \mathbf{Y}' = \langle \langle [1,2], \emptyset, [1,2], \emptyset, \ldots \rangle, [2,3] \rangle$. The invariant (3) reveals that neither string has a feasible length, resulting in failure.

The propagator for Cat is similar to Eq^{P} in regards to the relationship between $\boldsymbol{\mathcal{A}}^x$ and $\boldsymbol{\mathcal{A}}^z$, but the relationship between $\boldsymbol{\mathcal{A}}^y$ and $\boldsymbol{\mathcal{A}}^z$ is complicated by a dependency on $\mathcal{N}^x$:

$$\begin{aligned}&\text{Cat}^{\text{P}}\left(\langle\boldsymbol{\mathcal{A}}^x,\mathcal{N}^x\rangle,\langle\boldsymbol{\mathcal{A}}^y,\mathcal{N}^y\rangle,\langle\boldsymbol{\mathcal{A}}^z,\mathcal{N}^z\rangle\right)\\&\quad=\left\langle\,\langle\langle\mathcal{A}_1^{x\prime},\ldots,\mathcal{A}_m^{x\,\prime}\rangle,\mathcal{N}^{x\prime}\rangle,\langle\langle\mathcal{A}_1^{y\prime},\ldots,\mathcal{A}_m^{y\,\prime}\rangle,\mathcal{N}^{y\prime}\rangle,\langle\langle\mathcal{A}_1^{z\prime},\ldots,\mathcal{A}_m^{z\,\prime}\rangle,\mathcal{N}^{z\prime}\rangle\right\rangle\end{aligned}$$

where

$$\begin{aligned}\mathcal{N}^{x\prime}&=\mathcal{N}^x\cap[\min(\mathcal{N}^z)-\max(\mathcal{N}^y),\,\max(\mathcal{N}^z)-\min(\mathcal{N}^y)]\\\mathcal{N}^{y\prime}&=\mathcal{N}^y\cap[\min(\mathcal{N}^z)-\max(\mathcal{N}^x),\,\max(\mathcal{N}^z)-\min(\mathcal{N}^y)]\\\mathcal{N}^{z\prime}&=\mathcal{N}^z\cap[\min(\mathcal{N}^x)+\min(\mathcal{N}^y),\,\max(\mathcal{N}^x)+\max(\mathcal{N}^y)]\end{aligned}$$

and

$$\mathcal{A}_i^{x\prime}=\begin{cases}\mathcal{A}_i^x\cap\mathcal{A}_i^z & \text{if } i<\max(\mathcal{N}^{x\prime})\\\emptyset & \text{otherwise}\end{cases}\qquad\mathcal{A}_i^{y\prime}=\begin{cases}\mathcal{A}_i^y\cap\bigcup\limits_{j\in\mathcal{N}^x\cap[1,\max(\mathcal{N}^{z\prime})-i]}\mathcal{A}_{i+j}^z & \text{if } i<\max(\mathcal{N}^{y\prime})\\\emptyset & \text{otherwise}\end{cases}$$

$$\mathcal{A}_i^{z\prime}=\begin{cases}\mathcal{A}_i^z\cap\mathcal{A}_i^x & \text{if } i<\min(\mathcal{N}^{x\prime})\\\mathcal{A}_i^z\cap\mathcal{A}_i^x\cap\bigcup\limits_{j\in\mathcal{N}^{x\prime}\cap[i-\max(\mathcal{N}^{y\prime}),i]}\mathcal{A}_{i-j}^y & \text{if } \min(\mathcal{N}^{x\prime})\le i<\max(\mathcal{N}^{x\prime})\\\mathcal{A}_i^z\cap\bigcup\limits_{j\in\mathcal{N}^{x\prime}\cap[i-\max(\mathcal{N}^{y\prime}),i]}\mathcal{A}_{i-j}^y & \text{if } \max(\mathcal{N}^{x\prime})\le i<\max(\mathcal{N}^{z\prime})\\\emptyset & \text{if } \max(\mathcal{N}^{z\prime})\le i\end{cases}$$

Propagation of Len on the open-sequence representation is trivial:

$$\text{Len}^{\text{P}}\left(\langle\boldsymbol{\mathcal{A}},\mathcal{N}\rangle,\mathcal{S}\right)=\left\langle\left\langle\left\langle\mathcal{A}_1,\ldots,\mathcal{A}_{\max(\mathcal{N}\cap\mathcal{S})},\emptyset,\ldots,\emptyset\right\rangle,\mathcal{N}\cap\mathcal{S}\right\rangle,\mathcal{N}\cap\mathcal{S}\right\rangle$$

It is also easy to express the desired propagation for the Regular constraint, although the description in this case is of little help in regards to efficient implementation (see Sect. 6).

$$\text{Regular}^{\text{P}}\left(\langle\boldsymbol{\mathcal{A}},\mathcal{N}\rangle,\mathcal{L}\right)=\left\langle\begin{aligned}&\Big\langle\{s_1'\in\mathcal{A}_1\mid s'\in\mathcal{L}\},\ldots,\\&\quad\{s'_{\max(\mathcal{N})}\in\mathcal{A}_{\max(\mathcal{N})}\mid s'\in\mathcal{L}\},\emptyset,\ldots,\emptyset\Big\rangle,\{\ell\in\mathcal{N}\}\end{aligned}\right\rangle$$

Propagators for the remaining constraints from Sect. 4 are omitted for reasons of space; all may be described similarly to the propagators detailed above.

6 Implementation

While the open-sequence representation described in the previous section could be implemented as a new variable type for a CP solver, the correspondence between sets of feasible values and the domains of scalar variables suggests another method of implementing the open-sequence representation, namely as an

aggregation of two components: an array of scalar variables over the alphabet of the string, and an integer variable for the length of the string. Without loss of generality, we focus on strings of integers.

This implementation is similar to [22], which also involves a sequence of scalar variables that may be extended at the end but not at the beginning, and an integer variable that determines the length of that sequence. Beyond that similarity, our treatment diverges significantly. The open constraints described in [22] rely on the existence of a meta-programming framework to dynamically add variables to the model during search. In contrast, we extend a (closed) CP solver by adding a variable type representing a bounded-length sequence, eliminating the need for meta-programming and maintaining the declarative nature of CP solving. Unlike [22], we have no concept of *adding* a variable to the sequence: our implementation uses a fixed sequence of scalar variables, each of which may or may not participate in a solution as determined by the length variable. Additionally, we choose to treat each sequence-length pair as a single bounded-length sequence variable; whereas in [22] OpenRegular is defined as a global constraint of bounded arity, in our treatment Regular is a unary (non-global) constraint on a string variable of bounded length. This choice allows us to define constraints conventionally as relations over tuples (constraint semantics in [22] are described using formal languages), and eases the presentation of n-ary constraints on sequence variables (constraints in [22] involve only one sequence).

After discussing the technical challenges to such an aggregate implementation, we show how to derive actual propagators in a principled way, both from the underlying relations of the constraints in Sect. 4 and from the propagator descriptions in Sect. 5.

Aggregate Implementation. The open-sequence representation $\langle \boldsymbol{\mathcal{A}}, \mathcal{N} \rangle$ for a string of integers is here implemented as an array of integer variables $\boldsymbol{N} = \langle N_1, \ldots, N_m \rangle$ representing $\boldsymbol{\mathcal{A}} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_m \rangle$, and an integer variable N representing $\mathcal{N}$.

In regards to consistency level, the length variable and the sequence variables seem to have different requirements. For the length the most interesting values are the bounds (i. e., the lengths of the shortest and longest feasible strings). It seems unlikely, however, that maintaining bounds consistency on the variables of $\boldsymbol{N}$ is useful, as the set of feasible symbols at any index will rarely form a meaningful interval. We therefore choose to maintain a mixed consistency level, which considers the upper and lower bounds of N, and all domain values of the variables of $\boldsymbol{N}$; other choices are certainly possible.

For correctness, the representation invariant (3) must be enforced for each $\langle \boldsymbol{N}, N \rangle$ pair. Some care needs to be taken in the interpretation of (3), however: while $\mathcal{A}_i = \emptyset$ merely means $\max(\mathcal{N}) < i$, we have that $\mathcal{D}(N_i) = \emptyset$ leads to a failed search node. One way to avoid this is to include a reserved character, Null, in the domains of all variables of $\boldsymbol{N}$. The representation invariant may then be enforced by propagating the following conjunction of reified constraints:

$$\forall i \in [1, m] : N_i = \text{Null} \iff N < i \tag{4}$$

```
1 def EQ(vint[] X, vint LenX, vint[] Y, vint LenY){
2   checker{
3     (val(LenX) == val(LenY)) and
4     and(i in (min(rng(X)) .. ((min(rng(X)) + val(LenX)) + -1)))
5        (val(X[i]) == val(Y[i]))
6   }
```

Figure 2. Checker for the string equality constraint $\text{EQ}(\mathbf{X}, \mathbf{Y})$

Pure and Mixed String Constraints. An interesting feature of the propagator descriptions in Sect. 5 for pure and mixed string constraints is that they consist solely of a conjunction of range restriction operations. When applied to a variable domain, such an operation is called an *indexical* [34]: it is of the form $X \in \sigma$ and restricts the domain of the variable X to its intersection with the interval σ. An *indexical language* is a high-level solver-independent language for writing a propagator description with indexicals. The extended indexical language of [25] includes arrays and n-ary operations, and its system includes the following two automated transformations:

- A solver-independent *synthesis* of an indexical description of a propagator from a ground checker of its constraint.[3]
- A solver-specific *code generation* of an actual propagator from an indexical description thereof.

Following our ideas in [30], applied there to our more complex representation of bounded-length string variables in [31], we use this system to generate automatically a prototype implementation of the pure and mixed string constraints of Sect. 4 for GECODE [12].

We illustrate this process using the string equality constraint $\text{EQ}(\mathbf{X}, \mathbf{Y})$. Its underlying relation $\mathcal{E}$ from Sect. 4

$$\mathcal{E} = \{\langle x, y\rangle \mid x = y\} = \{\langle x, y\rangle \mid |x| = |y| \wedge \forall i \in [1, |x|] : x_i = y_i\} \tag{5}$$

can be seen as a ground checker for the constraint. We first replace the string variables $\mathbf{X}$ and $\mathbf{Y}$ with the pairs $\langle \boldsymbol{\mathcal{A}}^x, \mathcal{N}^x\rangle$ and $\langle \boldsymbol{\mathcal{A}}^y, \mathcal{N}^y\rangle$, respectively, as in Sect. 5. We then manually translate $\mathcal{E}$ into the checker sub-language of the extended indexical language, yielding Fig. 2. For our purposes, it suffices to illuminate a few less obvious features of the syntax. The aggregate variable $\langle \boldsymbol{\mathcal{A}}^x, \mathcal{N}^x\rangle$ is represented by two variables: an integer variable for the length (`vint LenX`) and an array of integer variables for the string (`vint[] X`). One constraint is on the lengths of the two strings (line 3). Another constraint is on the contents of the arrays (lines 4 and 5): it is expressed as an n-ary conjunction of equality constraints, corresponding to the universal quantification in (5).

From this checker, automatic synthesis yields an indexical description of a propagator for EQ, given in Fig. 3. Compared with the hand-derived propagator

[3] This synthesiser is not mentioned in [25], but described in a paper under preparation.

```
1 def EQ(vint[] X, vint LenX, vint[] Y, vint LenY){
2   propagator(gen)::DR{
3     LenX in dom(LenY);
4     LenY in dom(LenX);
5     forall(i in (min(rng(X)) .. ((min(rng(X)) + min(LenX)) + -1))){
6       X[i] in dom(Y[i]);
7       Y[i] in dom(X[i]);
8 }}}
```

Figure 3. Synthesised indexical description of a propagator for $\text{EQ}(\mathbf{X}, \mathbf{Y})$

description EQ^{P} in Sect. 5, we note that while the synthesised propagator correctly filters values in the arrays `X` and `Y` at indices below the current minimum length, it misses some propagation on `LenX` and `LenY`. Intuitively, if the intersection of the domains of `X[i]` and `Y[i]` is empty, then all feasible strings in $\mathbf{X}$ and $\mathbf{Y}$ must be shorter than `i`. This additional reasoning is expressed with the following `forall` construct that can be added to Fig. 3:

```
forall(i in ((min(rng(X)) + min(LenX)) .. (min(rng(X)) + max(LenX))))
  {(dom(X[i]) inter dom(Y[i])) == emptyset -> LenX in inf .. (i - 1);}
```

Automatic code generation from the extended version of Fig. 3 results in a C++ implementation of an EQ propagator for GECODE.

Alternatively, one can hand-code an EQ propagator for GECODE directly from the mathematical description that was derived in Sect. 5. This is much more labour-intensive and error-prone than the tool-assisted approach. Hence, the implementation we evaluate in Sect. 7 started as code generated by the indexical compiler; however, portions have been modified for efficiency reasons.

Language Membership Constraints. Indexicals are no help when it comes to language membership constraints, because propagators for those constraints rely on internal data structures. However, there are propagators for open constraints of language membership.

The REGULAR propagator of [26] is extended in [22] to handle bounded-length sequences. Propagation proceeds by dynamically increasing the number of layers in the layered automaton as the minimum feasible length of the string increases. We implemented this bounded-length extension of REGULAR in GECODE. Bounded-length propagators for the GCC and CONTEXTFREE constraints are also described in [22]; the addition of these constraints to our implementation has been left to future work.

7 Experimental Results

We compare our *bounded*-length CP implementation of the open representation,[4] called 'open' below, against *fixed*-length CP models and against state-of-the-art string solvers, on benchmarks provided by the latter. It outperforms the

[4] It is available at `https://github.com/jossco/gecode-string`.

Table 1. Runtimes in seconds (fastest in **bold**) for SUSHI word equations

	$n = 37$			$n = 50$			$n = 100$		
	open	pad	SUSHI	open	pad	SUSHI	open	pad	SUSHI
Eq. 1	**0.02**	0.05	0.30	**0.02**	0.07	1.11	**0.09**	0.24	2.56
Eq. 2	**<0.01**	**<0.01**	0.37	**<0.01**	**<0.01**	0.88	**0.01**	0.02	19.24
Eq. 3	**0.01**	0.03	0.29	**0.01**	0.03	0.64	**0.02**	0.09	1.14
Eq. 4	**<0.01**	0.01	42.16	**<0.01**	0.03	>300	**0.06**	0.07	>300
Eq. 5	**<0.01**	**<0.01**	1.56	**<0.01**	**<0.01**	2.93	**<0.01**	0.02	6.37

implementation of our previous representation of bounded string variables [31]. In each experiment, all CP models used the same upper bound for string length.

Benchmark of SUSHI. SUSHI [9] is a word equation solver for *unbounded*-length string variables (see Sect. 3). Being automaton-based, it computes the entire solution set in one go, rather than seeking solutions one by one. Nevertheless, the applicability of SUSHI as a satisfiability solver for string constraints is considered in [9]: SUSHI is compared to the *bounded*-length string solver KALUZA [29] (see Sect. 3) on a benchmark of five satisfiable word equations, each parameterised by a natural number n. To solve an equation with KALUZA, a bounded-length version of the equation is created for each n.

Example 4. We can model SUSHI word equation 1, namely $x \cdot \mathrm{a}^n = (\mathrm{a}|\mathrm{b})^{2n}$, as follows: $\textsc{Cat}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) \wedge \textsc{Regular}(\mathbf{Y}, \mathrm{L}(\mathrm{a}^n)) \wedge \textsc{Regular}(\mathbf{Z}, \mathrm{L}((\mathrm{a}|\mathrm{b})^{2n}))$.

Using the CP models in [16] for the five word equations, we also test against two *fixed*-length CP approaches. In the first, the string lengths are fixed at a pessimistically large upper bound [16], and multiple occurrences of a padding symbol are allowed at the end of each string [16]. In the second (not tried in [16]), the string lengths are initially fixed to a lower bound and a set of satisfying strings is sought; upon unsatisfiability, the lengths are increased lexicographically and search is restarted. These models use the REGULAR constraint, and concatenation is modelled with reified channelling constraints [16]. The padding approach, called 'pad' below, is always faster, so we omit results on the iterative approach. Our *bounded*-length CP models use the OPENREGULAR propagator [22] and our indexical-based CAT propagator of Sect. 6. For all CP models, we use the same deterministic first-fail search heuristic, and stop at the first solution, with a time-out of 300 seconds. The tests were run on a 2.66 GHz Intel Core 2 Duo with 4 GB of RAM, on VirtualBox 4.3.10 (the recommended way to run SUSHI) running Ubuntu 10.04 on 1 GB of RAM, using SUSHI 2.0 and GECODE 4.3.2.

In Table 1 we give runtimes for all five word equations. We compare the CP approaches only against SUSHI; experimental results reported in [9] (and replicated in [16]) show that SUSHI typically outperforms KALUZA, often significantly, and we do not attempt to replicate those results here. Our smallest instance size, $n = 37$, is the largest size tried in [9,16]. Even for $n \in \{50, 100\}$ the benchmark

Table 2. Runtimes (in seconds) and backtracks (best in **bold**) for KALUZA instances

	GECODE(open)		GECODE(pad)		KALUZA
instance name	runtime	backtracks	runtime	backtracks	runtime
concat	**0.003**	**0**	0.008	**0**	0.088
indexof	**0.003**	**0**	0.010	**0**	1.560
bettermatch1	**0.002**	**0**	0.005	**0**	0.223
bettermatch2	**0.003**	**0**	**0.003**	**0**	0.192
streq	**0.003**	**0**	0.006	**0**	0.077
replace	**0.006**	**30**	0.019	**30**	0.364

turns out to be trivial for *all* CP approaches, outperforming the state-of-the-art SUSHI solver by up to three orders of magnitude, as already observed in [16] for $n = 37$. On all instances, our bounded-length prototype implementation results in the same search tree as the fixed-length padding CP approach of [16], but with a lower runtime.

It turns out that all instances run *without* backtracks in both CP approaches! The reason is that the underlying constraint graph (with variables as vertices and constraints as hyper-arcs) is Berge-acyclic, so that domain consistency on the entire model is achieved by maintaining domain consistency on each constraint: this follows from the definition of the SLSE fragment, as one can observe in Example 4. We argue that a CP model preserves problem structure that is lost by KALUZA when translating to a bit-vector representation, and that knowledge of the complexity results of CP applicable to high-level models could have prevented the creation of the SUSHI word equation benchmark in the first place.

We thus look now at another third-party benchmark (which we did not try in [31]), also in order to see if our bounded-length prototype implementation can outperform the fixed-length CP padding approach of [16] by a larger margin.

Benchmark of KALUZA. The bounded-length string solver KALUZA [29] (see Sect. 3) includes over 50,000 instances that were generated for the symbolic execution of JavaScript, based on real-world Ajax web applications. Unfortunately, they all turn out to be trivial for CP approaches, with runtimes below 0.01 seconds, and even the KALUZA runtimes are below half a second. Hence this extensive benchmark is also not particularly interesting. In order not to be biased by hand-picking among the 50,000 instances, we pick all the 14 instances that are in the KALUZA code. It turns out that KALUZA gives erroneous results or crashes on several of these instances, as reported also by [36]. The results on the remaining instances are in Table 2; note that KALUZA does not report backtracks (incomparable in any case to those of CP approaches), and that Z3-STR [36] can only be applied to REGULAR-free versions of the actual instances.

Once again, the state-of-the-art solver is beaten, but the difference between the CP models with bounded-length string variables (open) and padded fixed-length string variables (pad) is small: we address this issue in the conclusion. We are not aware of a hard third-party benchmark for string variables.

8 Conclusion

We have formalised string variables and specified several interesting string constraints, all applicable to strings of fixed, bounded, or unbounded length. We have defined a bounded-length string variable representation, called the *open-sequence representation*, which is directly implementable for any existing CP solver, and we have given propagator descriptions for the specified string constraints. We have implemented the open-sequence representation and derived in a principled way actual propagators for the specified constraints, for the CP solver GECODE. Despite being only a prototype, our implementation already outperforms not only off-the-shelf fixed-length CP approaches, but also, by orders of magnitude, state-of-the-art dedicated string solvers, on their own benchmarks.

The experimental *time* comparison of our advocated CP approach of bounded-length string variables against the existing CP approach of padded fixed-length string variables has shown only minor speed-ups on the third-party benchmarks. In retrospect, this is not so surprising, as propagation is similar, witness the *backtrack* counts in Table 2 and the zero backtracks behind Table 1, and as those benchmarks seem not to exercise the string length reasoning that could give an advantage to our approach. The invariant (4) connecting the length of strings N_i and the length variable N can be seen as an implementation, via reification, of padding, thus it is unlikely that the bounded-length representation will perform more propagation than using padding symbols, unless non-trivial reasoning is required on string lengths. Also, at the *modelling* level, we argue that it is much easier to model a bounded-length string problem without using padding symbols, since encoding such a problem as a fixed-length one is both labour-intensive and error-prone: by designing the required propagators once and for all, we allow modellers to save the encoding effort and risk. Indeed, in [16] the automaton representation had to be modified to include the padding symbol, adding an extra level of complexity to the modelling. Since our bounded-length approach subsumes the fixed-length one, it suffices to fix the length instead of bounding it when one has a fixed-length string variable. Future work consists of strengthening our length reasoning, implementing our open-sequence representation as a first-class string variable type, and adapting our propagators.

We argue that CP is well-suited for string variables and constraints: unlike for many non-CP solvers mentioned here, there is no difficulty in upgrading from a single string variable to multiple ones, possibly with shared element variables, in having both string and numeric variables in a model, or in handling numeric variables and constraints without unnatural encodings. Indeed, it suffices to extend any CP solver, coming with existing numeric variables and numeric or symbolic constraints, in plug-and-play fashion, by adding the new type of string variables and providing propagators for the new constraints. This may result in high-level models that preserve problem structure and are amenable to faster solving than by lower-level encodings in ad hoc solvers.

Acknowledgements. This work is supported by grants 2009-4384, 2011-6133, and 2012-4908 of VR, the Swedish Research Council. We thank J.-N. Monette.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) Computer-Aided Verification (CAV 2014). LNCS, vol. 8559, pp. 150–166. Springer (2014)
2. Allen, F.E.: Control flow analysis. ACM Sigplan Notices 5(7), 1–19 (1970)
3. Barták, R.: Dynamic global constraints in backtracking based environments. Annals of Operations Research 118(1), 101–119 (2003)
4. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer (2004)
5. 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)
6. 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). LNCS, vol. 5505, pp. 307–321. Springer (2009)
7. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering 2(3), 215–222 (1976)
8. 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)
9. Fu, X., Powell, M.C., Bantegui, M., Li, C.C.: Simple linear string constraints. Formal Aspects of Computing 25, 847–891 (November 2013), SUSHI is available from http://people.hofstra.edu/Xiang_Fu/XiangFu/projects/SAFELI/SUSHI.php
10. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) Computer-Aided Verification (CAV 2007). LNCS, vol. 4590, pp. 519–531. Springer (2007), STP is available from https://sites.google.com/site/stpfastprover/
11. 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). LNCS, vol. 7795, pp. 277–291. Springer (2013)
12. Gecode Team: Gecode: A generic constraint development environment (2006), http://www.gecode.org
13. Gervet, C.: Constraints over structured domains. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 17, pp. 605–638. Elsevier (2006)
14. Ghosh, I., Shafiei, N., Li, G., Chiang, W.F.: JST: an automatic test generation tool for industrial Java applications with strings. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) International Conference on Software Engineering (ICSE 2013). pp. 992–1001. IEEE / ACM (2013)
15. Golden, K., Pang, W.: Constraint reasoning over strings. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 377–391. Springer (2003)
16. He, J., Flener, P., Pearson, J.: Solving string constraints: The case for constraint programming. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 381–397. Springer (2013)

17. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Hind, M., Diwan, A. (eds.) Programming Language Design and Implementation (PLDI 2009). pp. 188–198. ACM (2009)
18. Hooimeijer, P., Weimer, W.: StrSolve: solving string constraints lazily. Automated Software Engineering 19(4), 531–559 (2012)
19. Kieżun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: Rothermel, G., Dillon, L.K. (eds.) International Symposium on Software Testing and Analysis (ISSTA 2009). pp. 105–116. ACM (2009), HAMPI is available from http://people.csail.mit.edu/akiezun/hampi/
20. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
21. Lothaire, M.: Combinatorics on words. Cambridge Mathematical Library, Cambridge University Press (1997)
22. Maher, M.J.: Open constraints in a boundable world. In: van Hoeve, W.J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 163–177. Springer (2009)
23. Makanin, G.: The problem of solvability of equations in a free semigroup. Sbornik: Mathematics 32(2), 129–198 (1977)
24. Michel, L.D., Van Hentenryck, P.: Constraint satisfaction over bit-vectors. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 527–543. Springer (2012)
25. Monette, J.N., Flener, P., Pearson, J.: Towards solver-independent propagators. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 544–560. Springer (2012), the indexical compiler is available from http://www.it.uu.se/research/group/astra/software#indexicals
26. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer (2004)
27. Quimper, C.G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer (2006)
28. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: Kroeze, J.H., de Villiers, R. (eds.) South African Institute of Computer Scientists and Information Technologists Conference (SAICSIT 2012). pp. 139–148. ACM (2012)
29. 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), KALUZA is available from http://webblaze.cs.berkeley.edu/2010/kaluza/
30. Scott, J.D.: Rapid prototyping of a structured domain through indexical compilation. In: Schaus, P., Monette, J.N. (eds.) Domain Specific Languages in Combinatorial Optimization (CoSpeL workshop at CP 2013) (2013), http://cp2013.a4cp.org/workshops/cospel
31. Scott, J.D., Flener, P., Pearson, J.: Bounded strings for constraint programming. In: Tools with Artificial Intelligence (ICTAI 2013). pp. 1036–1043. IEEE Computer Society (2013)
32. Sellmann, M.: The theory of grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 530–544. Springer (2006)
33. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Computer and Communications Security (CCS 2014) (2014)
34. Van Hentenryck, P., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). techreport CS-93-02, Brown University,

Providence, USA (January 1993), revised version: Journal of Logic Programming, 37(1–3):293–316 (1998)
35. Yu, F., Bultan, T., Ibarra, O.H.: Symbolic string verification: Combining string analysis and size analysis. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009). LNCS, vol. 5505, pp. 322–336. Springer (2009)
36. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A Z3-based string solver for web application analysis. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Foundations of Software Engineering (FSE 2013). pp. 114–124. ACM (2013)