# MiniZinc with Strings

Roberto Amadini[1], Pierre Flener[2], Justin Pearson[2],
Joseph D. Scott[2], Peter J. Stuckey[1], and Guido Tack[3]

[1] University of Melbourne, Victoria, Australia
[2] Uppsala University, Uppsala, Sweden
[3] Monash University, Australia

**Abstract.** Strings are extensively used in modern programming languages and constraints over strings of unknown length occur in a wide range of real-world applications such as software analysis and verification, testing, model checking, and web security. Nevertheless, practically no constraint programming solver natively supports string constraints. We introduce string variables and a suitable set of string constraints as builtin features of the MiniZinc modelling language. Furthermore, we define an interpreter for converting a MiniZinc model with strings into a FlatZinc instance relying only on integer variables. This conversion is obtained via rewrite rules, and does not require any extension of the existing FlatZinc specification. This provides a user-friendly interface for modelling combinatorial problems with strings, and enables both string and non-string solvers to actually solve such problems.

## 1 Introduction

Strings are widely adopted in modern programming languages for representing input/output data as well as actual commands to be executed dynamically. The latter is particularly critical for security reasons: consider, e.g., the dynamic execution of a malicious SQL query that might dump a database or delete entire tables. Apart from security issues, tracking (an approximation of) the possible values of a string variable can also help in bug detection and code optimisation.

String analysis — needed in real-life applications such as test-case generation [13], program analysis [8], model checking [17], web security [5] — is an active and growing field, [11, 25, 28], and requires the processing of string constraints such as string (in-)equality, concatenation, and so on. Nevertheless, in constraint programming (CP), practically no solver natively supports string constraints. To our knowledge, the only exception is a new extension [33, 36] with bounded-length string variables of the Gecode solver [18], here called Gecode+$\mathbb{S}$ for convenience, which will become part of the official Gecode release. Empirical results show that Gecode+$\mathbb{S}$ is usually better than dedicated string solvers such as Hampi [23], Kaluza [32], and Sushi [14].

In this paper we take a further step towards the definition and solving of string constraints. The three contributions of this paper are as follows.

First, an extension of the MiniZinc [30] modelling language by string variables of possibly unknown length. MiniZinc enables the specification of constraint problems over (sets of) integers and real numbers, but currently does not allow models containing string variables. Thanks to the extension we describe, a MiniZinc user can now naturally define and solve a MiniZinc model containing string variables and constraints, as well as other constraints on other variable types.

Second, we provide a solver independent conversion of MiniZinc models with strings into equivalent FlatZinc instances containing only integer variables. Thus, every solver supporting FlatZinc can now solve a MiniZinc model with strings. This conversion follows the padding representation advocated in [21] and implemented in [35]. However, we underline that our contribution is orthogonal to [35] and generalises its work (see Section 4.2): our MiniZinc formulation does not impose restrictions on the string length (enabling us to express unbounded-length strings), and further allows any solver to use its preferred string representation (e.g., bit vectors or automata), and handles a superset of the constraints of [35].

Third, we provide an experimental evaluation on the Norn string benchmark [1] used in Gecode+$\mathbb{S}$ [33, 36] and the state-of-the-art constraint solvers Chuffed [10], Gecode [18], iZplus [15], Picat-SAT [43], MZN/Gurobi [4], MZN/Yices2 [9] and MZN/OscaR.cbls [7]. Results indicate that native support for string variables usually pays off, but not always, in which case the technology of the best solver varies. Indeed, we show that — despite longer flattening times — sometimes our conversion is more beneficial than using a dedicated string solver.

**Paper Structure.** Section 2 gives some background notions about string variables, MiniZinc and FlatZinc. Sections 3 and 4 describe the string extensions we implemented for MiniZinc and FlatZinc. Section 5 presents the experimental results before we discuss related work in Section 6 and conclude in Section 7.

## 2 Background

MiniZinc [30] is a flexible and user-friendly modelling language for representing constraint problems. The motto is *model once, solve anywhere*: each MiniZinc model is solver-independent, although it may contain annotations to communicate with the underlying solver.

MiniZinc supports the most common global constraints (constraints defined over an arbitrary number of variables [3]) and allows the separation between model and data: a MiniZinc model can be defined as a generic template to be instantiated by different data.

As an example, consider the $n$-queens problem, where $n \geq 4$ queens have to be placed on an $n \times n$ chessboard in such a way that they do not attack each other. This problem can be modelled in MiniZinc in terms of an unspecified number $n$ of queens, and then instantiated by providing the value of parameter $n$.

FlatZinc is a solver-specific target language for MiniZinc. Each MiniZinc model (together with corresponding data, if any) is converted into FlatZinc in the form required by a solver. In other terms, from the same MiniZinc model different FlatZinc instances can be derived according to solver-specific redefinitions.

For example, the $n$-queens problem can be modelled with the well-known `alldifferent`$([x_1, \ldots, x_n])$ global constraint, which holds if and only if all variables $x_i$ take different values. In this case a solver can decide to keep the constraint as is or to unfold it into the logical conjunction $\bigwedge_{1 \leq i < j \leq n} x_i \neq x_j$.

Following the approach of [23, 32, 33, 35, 36] we focus in this work on constraint solving over *bounded* string variables, i.e., string variables $x$ having a bounded length $\ell$, with $|x| \leq \ell \in \mathbb{N}$. We point out that our MiniZinc language extension allows us to express problems with unbounded string variables. Note that, while problems over bounded-length string variables are trivially decidable, satisfiability with unbounded-length strings is not decidable in general [16].

*Notation.* Given a fixed alphabet $\Sigma$, a string $x \in \Sigma^*$ is a finite sequence of $|x| \geq 0$ characters of $\Sigma$, where $|x|$ is the length of $x$. Let $\mathsf{ASC}$ denote the set of the ASCII symbols: we define the function $\mathcal{I} \colon \mathsf{ASC} \to [1, 128]$ such that $\mathcal{I}(a) = k$ if and only if $a$ is the $k$-th ASCII symbol.

The symbols $=$, $\neq$, and $\preceq$ respectively denote string equality, inequality, and lexicographical order on $\Sigma^*$. The concatenation of $x$ and $y$ is denoted by $x \cdot y$, while $x^n$ denotes the iterated concatenation of $x$ for $n$ times; $x^0$ denotes the empty string $\epsilon$, while $x^{-1}$ denotes the reverse of $x$.

If $x$ is a string (resp., an array), then we denote by $x[i]$ its $i$-th character (resp., element) and by $x[i..j]$ the subsequence $x[i]x[i+1]\cdots x[j]$; indices start from 1 in both cases. The symbol $\in$ is used for both set membership and character occurrence within a string.

## 3 MiniZinc with Strings

MiniZinc supports plenty of builtins (e.g., comparisons, basic and advanced numeric operations, set operations, logical operators, ...) and global constraints. It currently permits four types of variables (i.e., Booleans, integers, floats, and sets of integers) while strings can only be fixed literals, used for formatting output or defining model annotations.

Our first contribution is introducing *string variables*, i.e., variables $x \in \Sigma^*$, where $\Sigma$ is a given alphabet. As a first step, we assume that the alphabet $\Sigma$ is always the set $\mathsf{ASC}$ of ASCII characters. Although we focus on bounded-length strings, we do not impose any limitation on the maximum string length $\ell$.

Figure 1 shows three string variable declarations in a MiniZinc model. Variable `x` belongs to $\mathsf{ASC}^*$ but its maximum length is not specified: a solver can choose the preferred upper bound $\ell$ for its length or consider it unbounded. For example, a solver using automata for representing strings does not need to set a maximum length since it can represent strings of arbitrary length. Conversely, a bounded-length string solver such as Gecode+$\mathbb{S}$ has to fix a maximum string

```
int: N;
var string: x;
var string(N): y;
var string(500) of {"a", "b", "c"}: z;
```

**Fig. 1.** Examples of string variable declarations.

**Table 1.** MiniZinc string constraints, for each $x, y, z \in \mathsf{ASC}^*$, $a, b \in \mathsf{ASC}$, $n, m, q, q_0 \in \mathbb{N}$, $S \subseteq \mathsf{ASC}$, $F \subseteq \mathbb{N}$, $D \in \mathbb{N}^{q \times |S|}$, and $N \in \mathcal{P}(\mathbb{N})^{q \times |S|}$.

| Constraint | MiniZinc Syntax | Description |
|---|---|---|
| $x = y,\ x \neq y$ | `x = y, x != y` | (in-)equality |
| $x \prec y,\ x \preceq y,\ x \succeq y,\ x \succ y$ | `x < y, x <= y, x >= y, x > y` | lexicographic order |
| $x \in S^*$ | `x in S` | character set |
| $x \overline{\in} S^*$ | `str_alphabet(x, S)` | alphabet |
| $x \in [a, b]^*$ | `str_range(x, a, b)` | character range |
| $z = x \cdot y$ | `z = x ++ y` | concatenation |
| $a = x[n]$ | `a = x[n]` | character access |
| $y = x[n..m]$ | `y = str_sub(x, n, m)` | sub-string |
| $y = x^n$ | `y = str_pow(x, n)` | iterated concatenation |
| $y = x^{-1}$ | `y = str_rev(x)` | reverse |
| $n = \|x\|$ | `n = str_len(x)` | length |
| $x \in \mathcal{L}_{\mathrm{D}}(q, S, D, q_0, F)$ | `str_dfa(x, q, S, D, q0, F)` | DFA membership |
| $x \in \mathcal{L}_{\mathrm{N}}(q, S, N, q_0, F)$ | `str_nfa(x, q, S, N, q0, F)` | NFA membership |
| $\mathcal{GCC}(x, A, N)$ | `str_gcc(x, A, N)` | global cardinality |

length $\ell$. This tricky part is analogous to a MiniZinc declaration of the form "`var int: i`" for an integer variable `i`: a finite-domain solver assumes the domain of `i` to be finite and chooses its preferred bounds, while for a MIP solver `i` is unbounded. The length of `y` in Figure 1 can be at most `N`, where `N` is an integer parameter to be initialised within the model or in a separate data file. Variable `z` even has a constrained alphabet: $\mathtt{z} \in \{w \in \{\mathtt{"a"}, \mathtt{"b"}, \mathtt{"c"}\}^* \mid |w| \leq 500\}$.

Given that we now have string variables, inspired by [33,35,36], we introduce the string constraints specified in Table 1. A constraint for membership in a context-free language could be added; it was considered in [33,35,36] for inclusion in Gecode+$\mathbb{S}$, but not implemented for time-reasons as the state-of-the-art propagator of [21] for fixed-length string variables needs work to be generalised to bounded-length string variables.

The constraints $=, \neq, \prec, \preceq, \succeq, \succ$ have the semantics of their standard definitions. Given $S \subseteq \mathsf{ASC}$, the semantics of $x \in S^*$ is $\forall a : a \in x \implies a \in S$, while $x \overline{\in} S$ also enforces the reverse implication, i.e., $\forall a : a \in x \iff a \in S$.

The constraint `str_range` offers a shortcut for defining a set of strings over a range of characters: $[a, b]^* = \{c \in \mathsf{ASC} \mid a \leq c \leq b\}^*$, so for instance $[\mathtt{"a"}, \mathtt{"d"}]^* = \{\mathtt{"a"}, \mathtt{"b"}, \mathtt{"c"}, \mathtt{"d"}\}^*$. The function $x[i..j]$ returns the substring $x[n]x[n+1]\cdots x[m]$, where $n = \max(1, i)$ and $m = \min(j, |x|)$. In particular, $i > j$ implies $x[i..j] = \epsilon$.

```
int: m;
var int: n;
var string(m): x;
constraint x = str_rev(x);
constraint str_range(x, "a", "z");
constraint str_len(x) mod 2 = 1;
constraint str_gcc(x, ["a", "b", "c"], [n, n, n]);
constraint n > 0;
solve minimize str_len(x);
```

**Fig. 2.** A model for finding minimum-odd-length palindromes with the same, positive number of a's, b's, and c's. An optimal solution must have $n = 2 \wedge |x| = 7$.

The constraint $x \in \mathcal{L}_{\mathrm{D}}(q, S, D, q_0, F)$ constrains $x$ to be accepted by the deterministic finite automaton (DFA) $\langle Q, S, \delta, q_0, F\rangle$ where: $Q = \{1, \ldots, q\}$ is the state set, $S = \{a_1, \ldots, a_{|S|}\}$ is the alphabet, $\delta : Q \times S \to Q$ is the transition function such that $D[i, j] = k \iff \delta(i, a_j) = k$, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. The same applies to the non-deterministic finite automaton (NFA) constraint $x \in \mathcal{L}_{\mathrm{N}}(q, S, N, q_0, F)$, with the only difference that, while $D[i, j] \in Q$, in this case $N[i, j] \subseteq Q$.

Finally, we add a global cardinality constraint $\mathcal{GCC}(x, A, N)$ for strings, stating that each character $A[i] \in \mathsf{ASC}$ must occur exactly $N[i]$ times in string $x$.

The constraints in Table 1 express all those used in existing string solvers [1, 14, 23, 24, 32, 41] and reflect the most used string operations in modern programming languages. We are not aware of string solvers supporting constraints like lexicographic ordering and global cardinality, but these are natural for a CP solver.

Some constraints are redundant. For example we have that $x[i] = x[i..i]$ and $y = x[i..j] \iff (\exists y_1, y_2 \in \mathsf{ASC}^*)\ x = y_1 \cdot y \cdot y_2 \wedge |y_1| = i - 1 \wedge |y_1 \cdot y| = j$. The rationale behind such redundancy is to ease the model writing and to allow solvers to define a specialised treatment for each constraint in order to optimise the solving process.

The constraint set we added to MiniZinc is intended to be an extensible interface for the definition of string problems to be solved by fixed, bounded, and unbounded-length string solvers.

Consider the MiniZinc model in Figure 2, encoding the problem of finding a minimum-length palindrome string belonging to $\{\texttt{"a"}, \ldots, \texttt{"z"}\}^*$, having an odd length, and containing the same, positive number of occurrences of `"a"`, `"b"`, and `"c"`. We can see in this example the potential of MiniZinc with strings: the model is succinct and readable, it allows the specification of optimisation problems and not just of satisfaction problems, it accepts constraints over different types than just strings, it does not impose any bounds on the lengths of the strings, and it enables expressing the membership of a string variable to a context-*sensitive* language.

A more interesting example is provided in Figure 3, where we show a simplified way to detect a potential SQL injection attack in a script. An *SQL injection* is a technique where a malicious SQL statement is injected into a regular SQL

```
1 string sql;
2 var int: m; var int: n;
3 var string: pref; var string: suff; var string expr;
4 constraint sql = pref ++ expr ++ str_pow(" ", m) ++ "=" ++ str_pow(" ", n) ++
      expr ++ suff;
5 constraint str_len(expr) > 0;
6 solve satisfy;
```

**Fig. 3.** A model for detecting a possible SQL injection.

command. A well-known example is the injection of the condition `" OR 1=1 "` into the `WHERE` clause of an SQL query. Since every Boolean expression containing such a condition evaluates to `true`, an SQL injection of this type may cause the deletion or communication of tables of a database. The model in Figure 3 is actually more general, by detecting an injection into the parametric string `sql` of a substring of the form $\texttt{expr} \cdot b^{\texttt{m}} = b^{\texttt{n}} \cdot \texttt{expr}$, where `expr` can be any non-empty string while $b^{\texttt{m}}$ and $b^{\texttt{n}}$ are arbitrary sequences of `m` and `n` blanks respectively, where `m` and `n` are non-negative integer variables. The prefix `pref` and the suffix `suff` of `sql` can be any string. Clearly, this simplified example is not general enough to cover all the possible SQL injections. Nonetheless, this MiniZinc model is strictly more powerful than when using only regular expressions: the constraint in line 4 cannot be replaced by an equivalent `str_dfa` or `str_nfa` constraint, but could alternatively be modelled using the mentioned constraint for membership in a context-free language, which is not considered in this paper.

## 4 FlatZinc with(out) Strings

MiniZinc is a solver-independent modelling language. In practice, this is achieved by the MiniZinc compiler, which can translate any MiniZinc model into a specialised FlatZinc instance for a particular solver, using a solver-specific library of suitable redefinitions for basic and global constraints.

In order to extend MiniZinc with support for string variables, our second contribution consists of two redefinition libraries to perform different conversions:

- a string-to-string conversion $\mathcal{F}^{\text{str}}$ that flattens a model $M$ with string constraints into a FlatZinc instance $\mathcal{F}^{\text{str}}(M)$ with all such constraints preserved;
- a string-to-integers conversion $\mathcal{F}^{\text{int}}$ that flattens a model $M$ with string constraints into a FlatZinc instance $\mathcal{F}^{\text{int}}(M)$ with string constraints transformed into integer constraints.

We now discuss these two conversions in turn.

### 4.1 The $\mathcal{F}^{\text{str}}$ Conversion

The conversion $\mathcal{F}^{\text{str}}$ is straightforward and we omit its technical details. Each string predicate is preserved in the resulting FlatZinc instance, with a few exceptions in order to be consistent with the FlatZinc syntax. For example, the

```
array [1..3] of string: X_INTRODUCED_3 = ["a","b","c"];
var int: n :: output_var;
var string(100): x :: output_var;
var string: X_INTRODUCED_0 :: var_is_introduced :: is_defined_var;
var int: X_INTRODUCED_1 :: var_is_introduced :: is_defined_var;
constraint str_eq(x,X_INTRODUCED_0);
constraint str_range(x,"a","z");
constraint int_mod(X_INTRODUCED_1,2,1);
constraint str_gcc(x,X_INTRODUCED_3,[n,n,n]);
constraint int_le(1,n);
constraint str_rev(x,X_INTRODUCED_0) :: defines_var(X_INTRODUCED_0);
constraint str_len(x,X_INTRODUCED_1) :: defines_var(X_INTRODUCED_1);
solve minimize X_INTRODUCED_1;
```

**Fig. 4.** FlatZinc instance resulting from $\mathcal{F}^{\text{str}}$ applied to the MiniZinc model in Figure 2.

constraints `x = y` and `x != y` are rewritten into `str_eq(x, y)` and `str_neq(x, y)` respectively. Similarly, a string function is rewritten into a corresponding FlatZinc predicate; e.g., `n = str_len(x)` is translated into `str_len(x, n)`, while `z = x ++ y` translates into `str_concat(x, y, z)`.

Figure 4 gives the FlatZinc instance obtained by the $\mathcal{F}^{\text{str}}$ conversion of the MiniZinc model in Figure 2, assuming that the length-bound parameter `m` is instantiated with value `100` (see line 3).

$\mathcal{F}^{\text{str}}$ is a straightforward and fast conversion aimed at solvers supporting (some of) the constraints of Table 1. At present, to the best of our knowledge, the only CP solver with such a capability is the new Gecode+S [33, 36].

### 4.2 The $\mathcal{F}^{\text{int}}$ Conversion

When extending MiniZinc with new features, the goal is to be always conservative: the compiler should produce FlatZinc code executable by any current FlatZinc solver, albeit less efficiently than by a solver with native support for the new features. Hence we also develop the $\mathcal{F}^{\text{int}}$ conversion.

The underlying idea of $\mathcal{F}^{\text{int}}$ is to map each string variable $x$ to an integer variable $\ell_x \in [0, n]$ representing the string length $|x|$ and an array $X \in [0, 128]^n$ of $n$ integer variables representing the string itself; we choose $n = \min\left(\overline{|x|}, \ell\right)$, where $\overline{|x|}$ denotes the upper bound on $|x|$ if it is specified in the model and $\overline{|x|} = \ell$ otherwise, as we cannot exceed the maximum string length $\ell$. For $i = 1, \ldots, n$ the invariant $i > \ell_x \iff X[i] = 0$ enforces that the end $X[|x| + 1] \cdots X[n]$ of the array $X$ is padded with trailing zeros. The notation $(\forall_{i=1,\ldots,|x|})\ P(i)$ is actually a shortcut for the constraint $(\forall_{i \in [1, \overline{\ell_x}]})\ i \leq |x| \rightarrow P(i)$, and similarly for existential quantification, where $\overline{\ell_x}$ denotes the current upper bound of the domain of $\ell_x$.

The main issue of $\mathcal{F}^{\text{int}}$ is the maximum size $\ell$, since FlatZinc does not allow dynamic-length arrays. We set $\ell = 1000$ by default and issue a warning to the user if an unbounded string variable is artificially restricted by this transformation. The user (and in fact each solver) can override this parameter.

$$\mathcal{V}_{\text{str}}(x,n,S) \mapsto \{\mathcal{A}(x)\} \tag{1}$$

$$\mathcal{A}(x) \mapsto \langle X \rangle \begin{Bmatrix} n = \min(\overline{|x|},\ell),\ \mathcal{V}_{\text{arr}}(X,n,0..\mathcal{I}(\mathcal{D}(x))), \\ \mathcal{V}_{\text{int}}(\ell_x,0..n), \ell_x = |x|, (\forall_{i\in[1,n]})\ i > \ell_x \iff X[i] = 0 \end{Bmatrix} \tag{2}$$

$$x \in S^* \mapsto \left\{(\forall_{i\in[1,|x|]})\ \mathcal{A}(x)[i] \in \{0\} \cup \mathcal{I}(S)\right\} \tag{3}$$

$$x \mathrel{\overline{\in}} S^* \mapsto \left\{x \in S, (\forall_{i\in\mathcal{I}(S)})(\exists_{j\in[1,|x|]})\ \mathcal{A}(x)[j] = i\right\} \tag{4}$$

$$x \in [a,b]^* \mapsto \left\{(\forall_{i\in[1,|x|]})\ \mathcal{A}(x)[i] \in \{0\} \cup [\mathcal{I}(a),\mathcal{I}(b)]\right\} \tag{5}$$

$$x = y \mapsto \{|x| = |y|,\ (\forall_{i\in[1,|x|]})\ \mathcal{A}(x)[i] = \mathcal{A}(y)[i]\} \tag{6}$$

$$x \neq y \mapsto \{|x| = |y| \to (\exists_{i\in[1,|x|]})\ \mathcal{A}(x)[i] \neq \mathcal{A}(y)[i]\} \tag{7}$$

$$x \preceq y \mapsto \{\texttt{lex\_lesseq}(\mathcal{A}(x),\mathcal{A}(y))\} \tag{8}$$

$$\mathcal{GCC}(x,A,N) \mapsto \{\texttt{global\_cardinality}(\mathcal{A}(x),[\mathcal{I}(a) \mid a \in A],N)\} \tag{9}$$

$$|x| \mapsto \langle n \rangle \{\mathcal{V}_{\text{int}}(n,0..\ell)\} \tag{10}$$

$$x^{-1} \mapsto \langle y \rangle \left\{\mathcal{V}_{\text{str}}(y), |x| = |y|, (\forall_{i\in[1,|x|]})\ \mathcal{A}(y)[i] = \mathcal{A}(x)[|x| - i + 1]\right\} \tag{11}$$

$$x \cdot y \mapsto \langle z \rangle \begin{Bmatrix} \mathcal{V}_{\text{str}}(z), |z| = |x| + |y|, (\forall_{i\in[1,|x|]})\ \mathcal{A}(z)[i] = \mathcal{A}(x)[i], \\ (\forall_{j\in[1,|y|]})\ \mathcal{A}(z)[j + |x|] = \mathcal{A}(y)[j] \end{Bmatrix} \tag{12}$$

$$x^n \mapsto \langle y \rangle \begin{Bmatrix} \mathcal{V}_{\text{str}}(y), |y| = n|x|, \\ (\forall_{i\in[1,|x|],j\in[1,|y|]})\ \mathcal{A}(x)[i] = \mathcal{A}(y)[|x|(j-1) + i] \end{Bmatrix} \tag{13}$$

$$x[i..j] \mapsto \langle y \rangle \begin{Bmatrix} n = \max(1,i),\ m = \min(|x|,j), \\ \mathcal{V}_{\text{str}}(y),\ |y| = \max(0, m - n + 1), \\ (\forall_{k\in[1,|y|]})\ \mathcal{A}(y)[k] = \mathcal{A}(x)[k + n - 1] \end{Bmatrix} \tag{14}$$

$$x[i] \mapsto \langle y \rangle \begin{Bmatrix} \mathcal{V}_{\text{str}}(y),\ |y| \leq 1, \\ (i \in [0,|x|] \wedge \mathcal{A}(y)[1] = \mathcal{A}(x)[i]) \vee (i \notin [0,|x|] \wedge y = \epsilon) \end{Bmatrix} \tag{15}$$

$$x \in \mathcal{L}_{\text{D}}(q,S,D,q_0,F) \mapsto \begin{Bmatrix} s = |S| + 1,\ D' \in [1,q]^{q\times s},\ T = \texttt{sort}(\mathcal{I}(S)), \\ (\forall_{i\in[1,q],j\in[1,s]})\ D'[i,j] = \begin{cases} 0 & \text{if } j = 1 \wedge D[i,j] \notin F \\ D[i,j] & \text{otherwise} \end{cases} \\ \mathcal{V}_{\text{arr}}(X,|x|,0..|x|),\ \texttt{regular}(X,q,s,D',q_0,F), \\ (\forall_{i\in[1,|x|]})\ \mathcal{A}(x)[i] = \begin{cases} T[X[i]-1] & \text{if } X[i] > 1 \\ 0 & \text{otherwise} \end{cases} \end{Bmatrix} \tag{16}$$

**Fig. 5.** Rewrite rules of $\mathcal{F}^{\text{int}}$.

The $\mathcal{F}^{\text{int}}$ conversion follows the padding representation advocated in [21] and implemented in [35]: it works through the rewrite rules listed in Figure 5. This conversion is specified as a library containing the rewrite rules expressed in the MiniZinc language itself and does not require any extension of the existing FlatZinc specification.[1] Each rewrite rule has one of the following forms:

- $P \mapsto \{C_1, \ldots, C_n\}$, meaning that predicate $P$ is rewritten into the constraint conjunction $C_1 \wedge \cdots \wedge C_n$; or
- $F(x_1, \ldots, x_k) \mapsto \langle E \rangle \{C_1, \ldots, C_n\}$, meaning that function $F$ is rewritten into expression $E$ subject to constraint $C_1 \wedge \cdots \wedge C_n$.

We use a more readable meta-syntax instead of using MiniZinc/FlatZinc directly. We denote by $\mathcal{D}(x) \subseteq \mathsf{ASC}$ the auxiliary function that returns the set of characters that may occur in $x$, and by $\mathcal{I}(S)$ the set $\{\mathcal{I}(a) \mid a \in S\}$ of the ASCII codes for each character of $S$. Given $D \subseteq \mathbb{N}$ and $S \subseteq \mathsf{ASC}$, the constructs $\mathcal{V}_{\text{int}}(n, D)$, $\mathcal{V}_{\text{str}}(x, m, S)$, and $\mathcal{V}_{\text{arr}}(X, m, D)$ denote respectively: an integer variable declaration `var D: n`, a string variable declaration `var string(m) of S: x`, and an array of integer variables declaration `array[1..m] of var D: X`. If a parameter is omitted, then we assume $D = [0, 128]$, $m = \ell$, and $S = \mathsf{ASC}$.

Rule (1) of Figure 5 transforms a declaration of a string variable $x$ into the corresponding declaration of an array $X$ of integer variables via the $\mathcal{A}(x)$ function of Rule (2), which enforces the properties of $X$ described above. It is important to note that this transformation relies on the *same* array of integer variables being returned by $\mathcal{A}(x)$ for a variable $x$, even if the function is called multiple times. This is achieved through the common subexpression elimination mechanism built into MiniZinc functions [37].

Rules (3) to (9) are examples of predicate rewriting. In particular, the latter two rules take advantage of MiniZinc expressiveness by rewriting $x \preceq y$ and $\mathcal{GCC}(x, A, N)$ in terms of the `lex_lesseq` and the `global_cardinality` global constraints over integers. The rewrite rules for predicates $\in$, $\overline{\in}$ , $=$, and $\neq$ are intuitive.

Rules (10) to (15) are examples of function rewriting: a string variable is created, constrained, and then returned. We can see that dealing with special cases enables us to reduce the number of generated constraints; e.g., see Rules (14) and (15).

Rule (16) for `str_dfa` predicate is tricky. Indeed, the `regular` global constraint cannot straightforwardly encode $x \in \mathcal{L}_{\text{D}}(q, S, D, q_0, F)$ since the "empty character" 0 might occur in $\mathcal{A}(x)$. In order to agree with the semantics of `regular`, it is necessary to increment the number $s$ of its symbols (so, the $i$-th character of $S$ becomes the $(i+1)$-st symbol of the DFA encoded by `regular`), and to add a column at the head of $D$ for dealing with the 0 character (matrix $D'$ is the result of this addition — note that the state 0 is always a failing state).[2] If `regular` is satisfiable, then the accepted sequence $X$ is re-mapped to

[1] This library, called `nostrings.mzn`, is publicly available at `https://bitbucket.org/jossco/gecode-string`.

[2] Details at `http://www.minizinc.org/doc-lib/doc-globals-extensional.html`

a corresponding string thanks to the auxiliary array $T$. The rule for `str_nfa` is analogous.

We remark that the $\mathcal{F}^{\text{int}}$ converter enables the solving of string problems by *any* solver. Clearly, this is achieved at the expense of efficiency. Indeed, several new constraints and reifications are introduced.

Consider for example the model $M$ of Figure 2: the $\mathcal{F}^{\text{str}}(M)$ conversion is instantaneous and produces a FlatZinc instance of only 13 lines, regardless of the maximum length `m` of string variable `x` (see Figure 4). Conversely, the $\mathcal{F}^{\text{int}}(M)$ conversion can be considerably less efficient depending on the `m` parameter. For example, if `m` = 100, then $\mathcal{F}^{\text{int}}(M)$ consists of 4,511 lines; if `m` = 1000, then a FlatZinc instance of 45,011 lines is produced.

## 5 Evaluation

Our third contribution is an evaluation of our framework with different solvers. We compared the string CP solver Gecode+𝕊 [33, 36] against various state-of-the-art constraint solvers, namely:

- Chuffed [10] is a CP solver with lazy clause generation [31];
- Gecode [18] is a CP solver;
- iZplus [15] is a CP solver that also exploits local search;
- Picat-SAT [43] translates a CP problem into a Boolean satisfiability (SAT) problem, solved by Lingeling;
- MZN/Gurobi [4] translates a MiniZinc (MZN) model into a mixed-integer linear program, solved by Gurobi Optimizer [20];
- MZN/Yices2 [9] translates a MiniZinc model into a SAT modulo theories (SMT) model without string variables, solved by Yices2;
- MZN/OscaR.cbls [7] translates a MiniZinc model in a constraint-based local search model and a black-box search procedure, run by OscaR.cbls [12].

There is a lack of standardised and challenging string benchmarks [21, 33, 35, 36]. However, we stress that the goal of this paper *is not* an evaluation of solver performance, but the introduction of a framework for modelling string problems easily, with solving by both string and non-string solvers. Moreover, one of the benefits of introducing string variables and constraints in MiniZinc is the possibility of designing and comparing challenging and standard benchmarks.

We picked five problems from the Norn benchmark [1]: $a^n b^n$, ChunkSplit, HammingDistance, Levenshtein, and StringReplace (we use the same names as in [1]). We also used our Palindrome problem of Figure 2 and our SQL injection problem of Figure 3. All these problems have no parameters, except for the maximum string length $\ell$. For each problem, we:

1. wrote a MiniZinc model $M$ with parametric bound $\ell$ on string length;
2. obtained FlatZinc instances $F_M(f, \ell)$ by flattening $M$ with $f \in \{\mathcal{F}^{\text{str}}, \mathcal{F}^{\text{int}}\}$ and $\ell \in \{250, 500, 1000\}$;

**Table 2.** Runtimes of the solvers. Bold font indicates the best performance for each problem instance.

| | Chuffed | | | Gecode | | | iZplus | | | MZN/Gurobi | | | Picat-SAT | | | Gecode+$\mathbb{S}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\ell$ | 250 | 500 | 1000 | 250 | 500 | 1000 | 250 | 500 | 1000 | 250 | 500 | 1000 | 250 | 500 | 1000 | 250 | 500 | 1000 |
| $a^n b^n$ | 0.9 | **2** | **4.5** | 2.6 | 16.8 | 145.2 | 2.2 | 6.8 | 22.7 | 9.7 | 20.7 | 54.7 | 2.1 | 3.9 | 7.2 | **0.4** | 2.7 | 28.2 |
| Chunk. | 4.7 | 14.9 | n/a | 3.5 | **8** | 26 | 7.2 | 22.2 | **24.8** | t/o | t/o | t/o | 46.8 | 152 | 291.1 | **1.4** | 14.2 | 187.9 |
| Hamm. | 25.7 | 283.6 | n/a | 84.6 | t/o | t/o | t/o | t/o | t/o | 363.6 | t/o | t/o | 46.8 | 454 | t/o | **0.6** | **3.8** | **37.4** |
| Leven. | 1.3 | 2.6 | 6 | 1.2 | 2.3 | 5.4 | 3.7 | 19.5 | 8.1 | 91 | 345.7 | t/o | 1.7 | 3.8 | 26.8 | **0.1** | **0.1** | **0.1** |
| Str.Rep. | 2.4 | 6.8 | 23.2 | t/o | t/o | t/o | 3.1 | 9.7 | 44.2 | 264.2 | t/o | t/o | 28.3 | 148.1 | t/o | **0.2** | **0.8** | **4.7** |
| Palind. | 1.6 | 23.4 | 90 | t/o | t/o | t/o | **0.8** | **2.3** | **7.1** | 119.5 | t/o | t/o | 16.6 | 93.7 | 504.5 | n/a | n/a | n/a |
| SQLInj. | 17.9 | 399.8 | n/a | 4.6 | 10.2 | 396.3 | 108.9 | 431.1 | 617.9 | t/o | t/o | t/o | 83.3 | 148.7 | 502.6 | **0.5** | **0.1** | **0.1** |

3. solved each $F_M(\mathcal{F}^{\text{str}}, \ell)$ with Gecode+$\mathbb{S}$ (we extended the FlatZinc interpreter of Gecode for handling $\mathcal{F}^{\text{str}}$ builtins) and each $F_M(\mathcal{F}^{\text{int}}, \ell)$ with the other solvers.

We ran the experiments on Ubuntu 15.10 machines with 16 GB of RAM and 2.60 GHz Intel® i7 CPU. The source code for Gecode+$\mathbb{S}$ and the used MiniZinc models are available at `https://bitbucket.org/jossco/gecode-string`. The versions of the solvers with results in Table 2 are those used by the `sunny-cp` portfolio solver [2], version 2.2, in the MiniZinc Challenge 2016.[3] We do not compare with the Norn solver, as our results are incomparable with those of an unbounded-length solver such as Norn, which generates the language of all satisfying assignments for each string variable.

Table 2 shows the runtimes, in seconds, to conclude the search, i.e., the time needed by a solver to prove the (un-)satisfiability of a problem (for satisfaction problems) or to find and prove an optimal solution (for Palindrome, the only optimisation problem). The 't/o' abbreviation means that the time-out of 600 seconds was reached, while 'n/a' means that a solver failed prematurely (e.g., due to a segmentation fault) or is not applicable. For instance, Gecode+$\mathbb{S}$ is not applicable to the Palindrome problem since it does not implement the $\mathcal{GCC}$ constraint, which, to the best of our knowledge, has not been proposed before in the literature. Our MiniZinc extension (see Table 1) covers all the constraints implemented by Gecode+$\mathbb{S}$.

The chosen solvers whose results are not listed in Table 2 were not competitive on the chosen problems. Local search, performed by MZN/OscaR.cbls, is by design unable to prove unsatisfiability and thus always times out on the unsatisfiable $a^n b^n$, Hamming, and StringReplace problems. Further, the black-box local search performed by MZN/OscaR.cbls unfortunately meanders on some of the chosen satisfiable problems and optimisation problems upon flattening by the $\mathcal{F}^{\text{int}}$ conversion: our future work includes integrating the extension [6] for string variables and constraints of OscaR.cbls [12] into MZN/OscaR.cbls, so that the $\mathcal{F}^{\text{str}}$ conversion can be used instead. Similarly, MZN/Yices2 makes the state-of-the-art SMT solver Yices2 suffer from the result of the composi-

[3] `sunny-cp` is available at `https://github.com/CP-Unibo/sunny-cp`. We actually took advantage of its architecture for running and evaluating the solvers in Table 2.

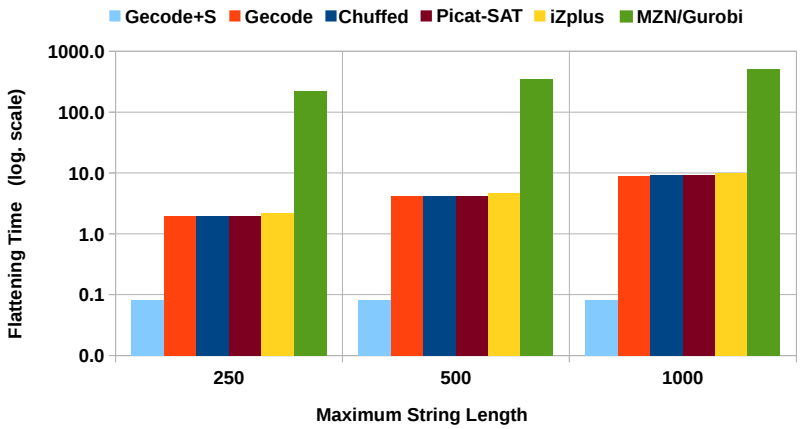


**Fig. 6.** Average time (in seconds) taken by $\mathcal{F}^{\text{int}}$ or $\mathcal{F}^{\text{str}}$.

tion of the $\mathcal{F}^{\text{int}}$ conversion with the FlatZinc-to-SMT-LIB-format conversion [9], which has not been modernised for a while. We hope that somebody will enable the use of the $\mathcal{F}^{\text{str}}$ conversion so that SMT solvers with a string theory — such as CVC4 [27], S3 [39], and Z3STR2 [41] — can be used instead, though not for optimisation problems.

All the runtimes in Table 2 include the FlatZinc flattening time. As explained at the end of Section 4, this time is far greater when the $\mathcal{F}^{\text{int}}$ conversion is used. This is clearly noticeable in Figure 6, where the average flattening time (in seconds) taken by $\mathcal{F}^{\text{int}}$ (for all the solvers except GECODE+$\mathbb{S}$) or $\mathcal{F}^{\text{str}}$ (for GECODE+$\mathbb{S}$) is shown.[4] As mentioned at the end of Section 4, this time is proportional to the maximum string length $\ell$.

While GECODE, CHUFFED, PICAT-SAT, and IZPLUS have comparable performance, the flattening time for MZN/GUROBI is remarkably higher. This is due to the fact that the complex reified expressions created by $\mathcal{F}^{\text{str}}$ must be linearized for use with MZN/GUROBI and hence this further expands the resulting FlatZinc. The average percentage of the total solving time (when a problem is solved) taken by $\mathcal{F}^{\text{int}}$ is 42.41% for IZPLUS, 47.10% for CHUFFED, 55.97% for GECODE, and 62.36% for MZN/GUROBI. Conversely, the average percentage of the total solving time taken by $\mathcal{F}^{\text{str}}$ for GECODE+$\mathbb{S}$ is only 6.95%.

The message of this evaluation is twofold. On the one hand, the GECODE+$\mathbb{S}$ CP solver is by far the best solver overall, due to its native string support and the short flattening times via $\mathcal{F}^{\text{str}}$ to FlatZinc. On the other hand, solvers without native string support sometimes benefit from $\mathcal{F}^{\text{int}}$ for being faster than

[4] We assume a flattening time of $T = 600$ seconds when the conversion time exceeded the time limit $T$. This happened only for MZN/GUROBI.

GECODE+𝕊 despite longer flattening times. This is interesting and should stimulate further development of native string support in CP solvers.

## 6 Related Work

GECODE+𝕊 [33,36] is currently the only CP solver that handles bounded-length string variables; its representation of string variables improves over the prefix-suffix pairs representation [34] and the open-sequence representation [35]. Fixed-length Boolean string variables, that is bit vectors, are handled in a CP fashion in [29]. Older CP approaches are surveyed in [33].

Apart from these systems, there are a number of string solvers, some custom-made and some others relying on existing solving technologies such as satisfiability modulo theories (SMT). We now discuss three approaches.

*Bit-vector* solvers map string constraints into bit-vector constraints. Examples of solvers using this approach are HAMPI [23, 24] and KALUZA [32]. The effectiveness of this approach appears to be limited when compared with other, more recent string solving techniques [22, 41].

*Automaton-based* solvers rely on regular expressions or (simplified) context-free grammars in order to represent strings and handle string constraints. Examples of these approaches are STRSOLVE [22], STRANGER [40], PASS [26], and PISA [26]. While they can naturally deal with unbounded-length strings, the main drawback of these solvers is their inability to capture other variable types, such as integers. For example, as observed in [41], the PISA solver can provide good performance but cannot model string lengths and symbolic arithmetic operations.

*Word-based* string solvers, according to [41], are SMT solvers that treat strings without abstractions or representation conversions. They take advantage of already defined theories, and enable a precise modelling of unbounded strings and length constraints. For instance, Z3STR [42], Z3STR2 [41], and Z3STRBV [38] extend the well-known SMT solver Z3. Other SMT-based string solvers are SUSHI [14], CVC4 [27], and NORN [1]. Although it is out of the scope of this paper to provide a comparison with all of them, we remark that GECODE+𝕊 provides a better performance than SUSHI in the evaluation reported in [33].

## 7 Conclusion

We presented an extension of the MiniZinc language that allows users to model and solve combinatorial problems with strings. The framework we propose is expressive enough to encode the most used string operations in modern programming languages, and — via proper FlatZinc translations — it also enables both string and non-string solvers to solve such problems. All the solvers having a FlatZinc interface can now solve string problems without manual intervention.

We took advantage of our framework for evaluating the state-of-the-art constraint solvers — CHUFFED, GECODE, IZPLUS PICAT-SAT, MZN/GUROBI,

MZN/Yices2, and MZN/OscaR.cbls — on problems with bounded-length strings. The results indicate that, despite longer flattening times, sometimes our FlatZinc decomposition can be more beneficial than using a dedicated string solver.

We are not aware of similar works in CP, and we see our work as a solid starting point for the handling of string variables and constraints with the MiniZinc toolchain. We hope our extension encourages the development of further CP solvers that can natively deal with strings. This will hopefully lead to the creation of new, challenging string benchmarks, and to the development of dedicated search heuristics (e.g., heuristics based on character frequencies in a string).

We are planning to enhance our framework by adding new search annotations, constraints, and features, as well extending the string domain from ASCII to other alphabets, such as Unicode. In particular, the useful missing constraint for membership in a context-free language should at least have a default handling under the $\mathcal{F}^{\text{int}}$ conversion, if not a propagator in Gecode+$\mathbb{S}$ used via the $\mathcal{F}^{\text{str}}$ conversion.

Finally, non-character alphabets could be useful, such as for the generation of protocol logs [19], where the natural model would use strings of timestamps.

**Acknowledgements.** The authors from the University of Melbourne are supported by the Australian Research Council (ARC) through Linkage Project Grant LP140100437. The authors in Sweden are supported by the Swedish Research Council (VR) through Project Grant 2015-04910. Many thanks to Gustav Björdal for having run the experiments on his local-search backend [7] for MiniZinc. Many thanks also to all the referees and to the audience of LOPSTR 2016 for their thoughtful feedback.

## References

1. P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. Norn: An SMT solver for string constraints. In *CAV*, volume 9206 of *LNCS*, pages 462–469. Springer, 2015.
2. R. Amadini, M. Gabbrielli, and J. Mauro. A multicore tool for constraint solving. In *IJCAI*, pages 232–238. AAAI Press, 2015.
3. N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, March 2007. The catalogue is available at `http://sofdem.github.io/gccat/`.
4. G. Belov, P. J. Stuckey, G. Tack, and M. Wallace. Improved linearization of constraint programming models. In *CP*, volume 9892 of *LNCS*, pages 49–65. Springer, 2016.
5. P. Bisht, T. L. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *CCS*, pages 575–586. ACM, 2011.
6. G. Björdal. String variables for constraint-based local search. Master's thesis, Department of Information Technology, Uppsala University, Sweden, August 2016. Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-301501`.

7. G. Björdal, J.-N. Monette, P. Flener, and J. Pearson. A constraint-based local search backend for MiniZinc. *Constraints*, 20(3):325–345, July 2015.
8. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, volume 5505 of *LNCS*, pages 307–321. Springer, 2009.
9. M. Bofill, J. Suy, and M. Villaret. A system for solving constraint satisfaction problems with SMT. In *SAT*, volume 6175 of *LNCS*, pages 300–305. Springer, 2010.
10. G. Chu. *Improving Combinatorial Optimization*. PhD thesis, Department of Computing and Information Systems, University of Melbourne, Australia, 2011.
11. G. Costantini, P. Ferrara, and A. Cortesi. A suite of abstract domains for static analysis of string values. *Software: Practice and Experience*, 45(2):245–287, 2015.
12. R. De Landtsheer and C. Ponsard. OscaR.cbls: An open source framework for constraint-based local search. In *ORBEL-27, the 27th annual conference of the Belgian Operational Research Society*, 2013. Available at `http://www.orbel.be/orbel27/pdf/abstract293.pdf`; the OscaR.cbls solver is available from `https://bitbucket.org/oscarlib/oscar/wiki/CBLS`.
13. M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162. ACM, 2007.
14. X. Fu, M. C. Powell, M. Bantegui, and C. Li. Simple linear string constraints. *Formal Aspects of Computing*, 25(6):847–891, 2013.
15. T. Fujiwara. iZplus description. `http://www.minizinc.org/challenge2016/description_izplus.txt`, 2016.
16. V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard. Word equations with length constraints: What's decidable? In *HVC*, volume 7857 of *LNCS*, pages 209–226. Springer, 2013.
17. G. Gange, J. A. Navas, P. J. Stuckey, H. Søndergaard, and P. Schachte. Unbounded model-checking with interpolation for regular language constraints. In *TACAS*, volume 7795 of *LNCS*, pages 277–291. Springer, 2013.
18. Gecode Team. Gecode: Generic constraint development environment, 2016. Available at `http://www.gecode.org`.
19. O. Grinchtein, M. Carlsson, and J. Pearson. A constraint optimisation model for analysis of telecommunication protocol logs. In *Tests and Proofs*, volume 9154 of *LNCS*, pages 137–154. Springer, 2015.
20. Gurobi Optimization, Inc. Gurobi Optimizer Reference Manual, 2016. Available at `http://www.gurobi.com`.
21. J. He, P. Flener, and J. Pearson. Solving string constraints: The case for constraint programming. In *CP*, volume 8124 of *LNCS*, pages 381–397. Springer, 2013.
22. P. Hooimeijer and W. Weimer. StrSolve: Solving string constraints lazily. *Automated Software Engineering*, 19(4):531–559, 2012.
23. A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering and Methodology*, 21(4):article 25, 2012.
24. A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *ISSTA 2009*, pages 105–116. ACM, 2009.
25. S. Kim, W. Chin, J. Park, J. Kim, and S. Ryu. Inferring grammatical summaries of string values. In *APLAS*, volume 8858 of *LNCS*, pages 372–391. Springer, 2014.
26. G. Li and I. Ghosh. PASS: String solving with parameterized array and interval automaton. In *HVC*, volume 8244 of *LNCS*, pages 15–31. Springer, 2013.

27. T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*, volume 8559 of *LNCS*, pages 646–662. Springer, 2014.
28. M. Madsen and E. Andreasen. String analysis for dynamic field access. In *CC*, volume 8409 of *LNCS*, pages 197–217. Springer, 2014.
29. L. D. Michel and P. Van Hentenryck. Constraint satisfaction over bit-vectors. In *CP*, volume 7514 of *LNCS*, pages 527–543. Springer, 2012.
30. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
31. O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
32. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *S&P*, pages 513–528. IEEE Computer Society, 2010.
33. J. D. Scott. *Other Things Besides Number: Abstraction, Constraint Propagation, and String Variable Types*. PhD thesis, Department of Information Technology, Uppsala University, Sweden, 2016. Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-273311`.
34. J. D. Scott, P. Flener, and J. Pearson. Bounded strings for constraint programming. In *ICTAI*, pages 1036–1043. IEEE Computer Society, 2013.
35. J. D. Scott, P. Flener, and J. Pearson. Constraint solving on bounded string variables. In *CPAIOR*, volume 9075 of *LNCS*, pages 375–392. Springer, 2015.
36. J. D. Scott, P. Flener, J. Pearson, and C. Schulte. Design and implementation of bounded-length sequence variables. In *CPAIOR*, LNCS. Springer, 2017.
37. P. J. Stuckey and G. Tack. MiniZinc with functions. In *CPAIOR*, volume 7874 of *LNCS*, pages 268–283. Springer, 2013.
38. S. Subramanian, M. Berzish, Y. Zheng, O. Tripp, and V. Ganesh. A solver for a theory of strings and bit-vectors. *CoRR*, abs/1605.09446, 2016.
39. M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *SIGSAC*, pages 1232–1243. ACM, 2014.
40. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS*, volume 6015 of *LNCS*, pages 154–157. Springer, 2010.
41. Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *CAV*, volume 9206 of *LNCS*, pages 235–254. Springer, 2015.
42. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *SIGSOFT*, pages 114–124. ACM, 2013.
43. N. Zhou and H. Kjellerstrand. The Picat-SAT compiler. In *PADL*, volume 9585 of *LNCS*, pages 48–62. Springer, 2016.