Symmetry Declarations for MiniZinc

Nathaniel Baxter
Department of Computing and Information Systems
University of Melbourne, Australia.
baxtern@student.unimelb.edu.au

Geoffrey Chu
Department of Computing and Information Systems
University of Melbourne, Australia.
National ICT Australia, Victoria Laboratory.
geoffrey.chu@unimelb.edu.au

Peter J. Stuckey
Department of Computing and Information Systems
University of Melbourne, Australia.
National ICT Australia, Victoria Laboratory.
pstuckey@unimelb.edu.au

ABSTRACT

Symmetry is a common property of man-made systems, appearing frequently in interesting problems in research and industry. When modelling constraint satisfaction and optimisation problems, underlying symmetries can make the search for solutions or optimal solutions much harder. In contrast, when symmetries are known, they can be used to speed up solving by avoiding considering symmetric parts of the solution space. This can be achieved by using static or dynamic symmetry breaking approaches. Unfortunately symmetry breaking approaches are hard to compare. Each method is typically only implemented in one or two systems, and symmetry papers use different problems to compare and illustrate their ideas. In this paper we add symmetry declarations to the constraint modelling language MiniZinc, which is supported by a variety of solvers. These symmetries can then either be handled using static symmetry breaking constraints, or passed to a dynamic symmetry breaking method if the underlying solver supports it. The addition of symmetry declarations to a modelling language allows for specification of generic symmetries by the modeller as well as lower level handling by the solver.

CCS Concepts

•**Mathematics of computing** → **Combinatorial optimization;** *Permutations and combinations;* •**Software and its engineering** → **Constraints;** *Constraint and logic languages;*

Keywords

MiniZinc, Constraint programming, Symmetry breaking

ACE '16 Canberra, ACT Australia

http://dx.doi.org/10.1145/2843043.2843058

1. INTRODUCTION

Many interesting real-world problems in areas such as scheduling, packing and resource management can be modelled as constraint satisfaction or optimisation problems; having a clearly defined set of decision variables, constraints on those variables and possibly an objective function on the solutions. Constraint satisfaction and optimisation solvers search through the space of possible solutions to find solutions and prove optimality. *Solution symmetries* are permutations of the pairs of decision variables and possible values that do not alter the set of solutions [5]. Underlying symmetries can make combinatorial problems much harder to solve, with solvers performing redundant search through symmetrically equivalent areas of the search tree.

In contrast, when symmetries of a problem are known, problem solving can be dramatically improved. Static symmetry breaking methods transform the problem model before search, adding constraints that exclude symmetrically equivalent solutions [6]. There is significant work in the literature on exploiting symmetry using static constraints that are built for specific problems or problem instances. Some generic static symmetry breaking methods exist [6, 4], however the addition of symmetry breaking constraints remains a manual process. Unlike static methods, dynamic symmetry breaking alters the search so that symmetrically equivalent sections of the solution space are not explored [9]. Dynamic methods range from those that attempt to completely break symmetries such as Symmetry Breaking During Search [9] and Symmetry Breaking by Dominance Detection [7], to incomplete but often more efficient methods such as Lightweight Dynamic Symmetry Breaking [11].

While many methods exist for exploiting symmetries, comparison of them is non-trivial as models must be implemented for various different systems. Some solvers such as the ECLiPSe constraint programming system [1] have libraries for most dynamic symmetry breaking methods, however there is no standard set of symmetry declarations between the methods and model implementation is still tied heavily into the method used. We propose that Symmetry Declarations should be a standard part of modelling. This paper implements a set of generic symmetry declarations in the constraint modelling language MiniZinc [12] which is supported by a number of solvers with dynamic symmetry breaking methods such as Chuffed [2], Gecode [8] and ECLiPSe [1] . If the modeller understands that symmetries exist, then they can be declared so that any solver support-

ing these declarations can take advantage of them. Similarly a standard notation for symmetries gives a target for tools that automatically detect symmetries [3, 10], so these tools can become generic and system independent. Additionally a library of models with declared symmetries in a widely supported modelling language allows for easier comparison of the strengths and weaknesses of different symmetry breaking approaches, and helps to evaluate new symmetry breaking approaches.

Unfortunately, most constraint solvers do not support such symmetry breaking methods. An effective way of breaking symmetries is to convert the symmetry declarations into a set of lex-leader symmetry breaking constraints [6]. This has the advantage that such constraints will be supported by all solvers which support MiniZinc.

In this paper we discuss the addition of symmetry declarations to models in MiniZinc. The contributions are as follows:

- A complete set of symmetry declarations that allow any *solution symmetry* to be declared as part of a MiniZinc model.
- MiniZinc decomposition definitions which convert a symmetry declaration into a set of static symmetry breaking constraints. This allows symmetries to be exploited on any system supporting MiniZinc without any special extra requirement for handling symmetries (although solver writers and modellers can implement the proposed declarations via their own methods for better performance).
- Symmetry declarations handled by MiniZinc decomposition are automatically decomposed in a manner to ensure that the symmetry breaking constraints generated from different symmetry declarations are mutually compatible. This is typically very hard for a human modeller to do.

The remainder of the paper is organized as follows: in the next section we given definitions enough to define solutions symmetries. In Section 3 we define and discuss the symmetry declarations. In Section 4 we discuss our default static decompositions of the declarations. In Section 5 we discuss solver specific handling of the symmetry declarations. In Section 6 we perform experimental analysis of our implementation and static decompositions.

2. PRELIMINARIES

2.1 Constraint Satisfaction Problems

Defining symmetry first requires a formal definition of CSPs and their solutions.

Definition 1. *A* Constraint Satisfaction Problem *(CSP) P is a triple $P \equiv (V, D, C)$, where V is a set of variables, D is a set of domains, and C is a set of constraints. A* pair *of $P \equiv (V, D, C)$ is of the form x/d where $x \in V$ and $d \in D(x)$. The set of all* pairs of *P is denoted by pairs(P). A* pairset *of P is a subset of pairs(P), and a* valuation of *P over $X \subseteq V$ is one that contains exactly one pair x/d for each variable in $x \in X$. A constraint $c \in C$ over a set $X \subseteq V$ of variables is a set of valuations over X, and we say that scope$(c) = X$. Valuation θ over scope(c) is* allowed *by c if $\theta \in c$. Valuation θ over $X \subseteq V$* satisfies *constraint c if scope$(c) \subseteq X$ and the projection of θ over scope(c) — defined as $\{x/d | (x/d) \in \theta, x \in \text{scope}(c)\}$ — is allowed by c. A* solution *of P is a valuation over V that satisfies every $c \in C$.*

We reiterate the definition of a *solution symmetry* [5].

Definition 2. *For any CSP instance $P \equiv (V, D, C)$, a* solution symmetry *of P is a permutation σ of the set pairs(P) that preserves the set of solutions to P.*

2.2 MiniZinc

This paper contains a significant number of code snippets and examples written in the MiniZinc language. MiniZinc is a medium-level declarative modelling language for expressing satisfaction and optimisation constraint programming problems. Solving a model requires flattening it and any data into a lower level constraint language FlatZinc, running a solver which supports FlatZinc, and displaying any solutions found. Decision variables can be represented as single or array variables, and are modelled using Boolean, integer, float and set types. Constraints are Boolean statements which can be represented using standard operators on the above types, syntactic sugar such as 'forall', 'exists' and 'sum', various builtin functions such as sorting, as well as included or user defined predicates and functions. Search and solver strategies can be specified, with solver writers able to provide a global predicate library which can pass information through to the solver.

Example 1. A simple MiniZinc model. n and p are parameters which are calculated during flattening. x is a size n array of integer decision variables "var" with values from $1..n$. The first constraint contains a *let* clause which defines new parameters or variables within the scope of the statement following the *in*. p defines a parameter array of $1..n$ which is a permutation used to reorder x in the *forall* statement and put inequality constraints on the variables. The second constraint shows how the solver decomposes the first. The third constraint uses a predicate provided by MiniZinc to say that all the variables in x are different which is redundant here.

```
int: n = 4;
array[1..n] of var 1..n: x;

constraint let {
  array[int] of int: p = [ n - i + 1 | i in 1..n ]
  } in
  forall (i in 1..n-1) ( x[p[i]] < x[p[i + 1]] );

constraint
  x[4] < x[3] /\ x[3] < x[2] /\ x[2] < x[1];

constraint all_different(x);

solve satisfy;
```

In this paper, some syntax such as 'predicate' and 'constraint' is left for conciseness. Additionally the normally required 'include' or 'solve' items are not included. Familiarity with MiniZinc itself is not expected, however some understanding of list comprehensions and declarative programming is required.

3. SYMMETRY DECLARATIONS

We introduce six new symmetry breaking predicates into MiniZinc for declaring variable symmetries, value symmetries, variable sequence symmetries, value sequence symmetries, variable permutation symmetries and value permutation symmetries. Every solution symmetry can be expressed as a variable permutation symmetry as integers in MiniZinc have finite domains (although perhaps not very concisely), so these declarations are complete in terms of their ability to express symmetries. The following lists the integer (`int`) versions only, which naturally extend to Booleans (`bool`) in MiniZinc. Analogous versions of the variable symmetries also exist for real numbers (`floats`).

```
var_sym(array[int] of var int: x)
```

Requires: x is an array of distinct integer decision variables.

Means: Every variable in the set x is interchangeable, i.e., for any i, j, if the values of $x[i]$ and $x[j]$ are swapped, then solutions are preserved.

```
val_sym(array[int] of var int: x,
        array[int] of int: s)
```

Requires: x is an array of distinct integer decision variables. s is an array of distinct integers.

Means: Any pair of values in s can be interchanged over all the variables in x, i.e., for any i, j, if every $x[k]$ which was set to $s[i]$ is changed to $s[j]$ and every $x[k]$ which was set to $s[j]$ is changed to $s[i]$, then solutions are preserved.

```
var_seq_sym(array[int,int] of var int: x)
```

Requires: x is a two-dimensional array of distinct integer decision variables.

Means: Any pair of variable sequences in x can be interchanged, i.e., for any i, j, if for every k, $x[i, k]$ is swapped with $x[j, k]$, then solutions are preserved.

```
val_seq_sym(array[int] of var int: x,
            array[int,int] of int: s)
```

Requires: x is an array of distinct integer decision variables. s is a two-dimensional array of distinct integers.

Means: Any pair of value sequences in x can be interchanged, i.e., for any i, j, if for every m, every $x[k]$ which was set to $s[i, m]$ is changed to $s[j, m]$ and every $x[k]$ which was set to $s[j, m]$ is changed to $s[i, m]$, then solutions are preserved.

```
var_perm_sym(array[int] of var int: x,
             array[int,int] of int: p)
```

Requires: x is an array of distinct integer decision variables. p is a two-dimensional array of integers, where each row is a permutation of the integers $1..length(x)$ and the ith row represents the variable sequence $[x[p[i, k]] \mid k \in 1..length(x)]$.

Means: Any variable sequence represented in p can be mapped to another variable sequence represented in p, i.e., for any i, j, if for every k, $x[p[i, k]]$ is set to the value of $x[p[j, k]]$, then solutions are preserved.

```
val_perm_sym(array[int] of var int: x,
             array[int,int] of int: s);
```

Requires: x is an array of distinct integer decision variables. s is a two-dimensional array of integers where each row covers the same set of values.

Means: Any value sequence in s can be mapped to another value sequence in s, i.e., for any i, j, if every $x[k]$ which was set to $s[i, m]$ is changed to $s[j, m]$, then solutions are preserved.

Note that {`var`,`val`}`_perm_sym` are different from {`var`,`val`}`_seq_sym`. They allow arbitrary permutations of variables/values, rather than just swaps. Note that any solution symmetry can be declared as a `var_perm_sym`. For any solution symmetry, Boolean variables can be introduced to represent the literals in the problem, i.e., $x = v$ for each variable x and value v, and use `var_perm_sym` to declare which permutations preserves solutions. `var_perm_sym` is not a very efficient way to handle a symmetry, so if the symmetry fits into the other 5 categories, it is generally more concise and efficient to use those.

Example 2. Consider the problem of generating a Latin Square[1] of size n. A MiniZinc model for this is:

```
int: n;
array[1..n,1..n] of var 1..n: x;

forall (i in 1..n) (
      all_different([x[i,j] | j in 1..n])
  /\  all_different([x[j,i] | j in 1..n]) );
```

This model takes an integer parameter n which determines the size of the Latin Square. It declares a two-dimensional array of integer decision variables x of size $n \times n$ with the domain $1..n$. Each row and column is then constrained using the 'forall' statement so that the values of their variables must all be different. This model has a number of symmetries. The value symmetries, row symmetries and column symmetries of the Latin Squares problem are declared as follows:

```
val_sym([x[i,j] | i, j in 1..n], [i | i in 1..n]);
var_seq_sym(
  array2d(1..n, 1..n, [x[i,j] | i, j in 1..n]) );
var_seq_sym(
  array2d(1..n, 1..n, [x[j,i] | i, j in 1..n]) );
```

The 'val_sym' constraint declares that there is a value symmetry on all of the variables in x for each of the values $1..n$. Meaning that, given a solution of the model, another solution can be found by swapping any two values in $1..n$ across the entire array x. The first 'var_seq_sym' constraint declares that there is a variable sequence symmetry for the rows of x. Meaning that, given a solution of the model, another solution can be found by swapping the values of the variables of two distinct rows of x. Similarly the second 'var_seq_sym' constraint declares a variable sequence symmetry, although on the columns of x rather than the rows.

New symmetry declarations which are special cases of the above predicates can be generated using MiniZinc predicates.

[1] see e.g. http://en.wikipedia.org/wiki/Latin_square

Example 3. A common symmetry for problems with sequences is that the sequence can be reversed. We create a new symmetry predicate `rev_seq_sym` which captures this using `var_perm_sym`, as follows:

```
rev_seq_sym(array[int] of var int: x) =
  let {
    int: l = length(x),
    array[1..2,1..l] of 1..l:
      y = array2d(1..2, 1..l,
        [ if i == 1 then j
          else l - j + 1 endif
        | i in 1..2, j in 1..l ] )
  } in var_perm_sym(x,y);
```

Given an x array of integer decision variables, this declaration decomposes into a variable permutation symmetry on x, where the array of permutations y is a two-dimensional array with two rows and the same number of columns as the size (or length) l of x. The first permutation being the identity $1..l$ and the second the reverse $l..1$.

Example 4. Another common symmetry for problems of square matrices is that of rotation symmetry. We define a new symmetry predicate `rot_sqr_sym` which captures rotational symmetries as follows:

```
rot_sqr_sym(array[int,int] of var int: x) =
  let { int: n = card(index_set_1of2(x)),
        int: n2 = card(index_set_2of2(x)),
        int: l = n * n,
        array[1..l] of var int:
          y = [ x[i,j] | i in index_set_1of2(x),
                         j in index_set_2of2(x) ],
        array[1..4,1..l] of 1..l:
          p = array2d(1..4,1..l,
            [ if k == 1 then i*n + j - n else
              if k == 2 then (n - j)*n + i else
              if k == 3 then (n - i)*n + (n - j)+1
              else           i*n + (n - j) - n + 1
                endif endif endif
            | k in 1..4, i,j in 1..n]) }
  in
  var_perm_sym(y,p);
```

Given a two-dimensional array x of integer decision variables, this declaration decomposes into a variable permutation symmetry on the one-dimensional array y, which is a flattened version of x. Permutations on y then have size $l \times l$ where l is the size (or length) of x, and correspond directly with permutations on the square matrix. The four permutations in p which can be swapped across y represent anticlockwise rotations of a square matrix by $0°$, $90°$, $180°$ and $270°$ respectively.

The rotational symmetries of the Latin Squares problem of Example 2 can now be declarated by adding the following to our model:

```
rot_sqr_sym(x);
```

4. STATIC DECOMPOSITION

In this section we discuss our decomposition of the above symmetry declarations into static symmetry breaking constraints within MiniZinc. These allow our symmetry declarations to be supported by any solver that supports MiniZinc.

4.1 Single Declaration

If there is only one symmetry declaration in the program, it is relatively straight forward to convert the symmetry declaration into a lex-leader symmetry breaking constraint by using a MiniZinc decomposition. In general, a lexicographical order is picked (typically the order in which variables appear in the main argument x), and then constraints are added to prune any assignment which can be mapped to a lexicographically better assignment using any of the symmetries given in the declaration (since that shows it is not the lex-leader in its equivalence class).

The symmetry breaking constraints for variable symmetries, value symmetries and variable sequence symmetries are simple and standard. Variable symmetries are broken by applying the same order chosen for the variables to the values of those variables. Value symmetries are broken by ordering the first occurrence of each value across the variables. Variable sequence symmetries are broken by lexicographically ordering the rows of the two-dimensional variable array. For brevity the MiniZinc decomposition of these predicates is not shown. Their definitions can be found online [2] along with all the code and examples used in this paper. The symmetry breaking constraints for value sequence symmetries, variable permutation symmetries and value permutation symmetries are more complicated, we are not aware of any definitions in the literature.

4.1.1 Value Sequence Symmetries

The following lex-leader symmetry breaking constraint for value sequence symmetries works as follows. Let A consist of the smallest values from each column of s, and B be the remaining values in s. If $x[1] \in B$, then there exists a permutation which will take $x[1]$ to a lower value, hence the assignment is not the lex-leader and can be pruned. If the value of $x[1]$ is not in s and $x[2] \in B$, then $x[1]$ is invariant under any of the permutations and there exists a permutation which maps $x[2]$ to a lower value, so again, the assignment is not the lex-leader and can be pruned. In general, if the first k variables are not in s, then the $k+1$th must not be in B. This requires a new local decision variable array y to express for each variable in x the set A, B, or neither that it is in, with values 0, 2 and 1 respectively. Then the symmetry breaking predicate is decomposed into a *lex_lesseq* (or 'lexicographically less than or equal to') constraint comparing $y[i]$ with an array of ones to ensure that there is an $x[i] \in A$ where $i < j$ $\forall j$ s.t. $x[j] \in B$ as required.

```
predicate val_seq_sym(array[int] of var int: x,
                      array[int,int] of int: s) =
  let { int: l1 = min(index_set_1of2(s)),
        int: u1 = max(index_set_1of2(s)),
        int: l2 = min(index_set_2of2(s)),
        int: u2 = max(index_set_2of2(s)),
        set of int:
          A = { min([s[i,j] | i in l1..u1])
                            | j in l2..u2 },
        set of int:
          B = { s[i,j] | i in l1..u1, j in l2..u2 }
              diff A,
        int: l = min(index_set(x)),
        int: u = max(index_set(x)),
        array[l..u] of var 0..2: y }
```

[2] http://github.com/nathanielbaxter/minizinc-sym-paper-code

```
  in
  forall (i in index_set(x)) (
       (y[i] = 0 <-> (x[i] in A))
    /\ (y[i] = 2 <-> (x[i] in B))
  )
  /\
  lex_lesseq(y, [ 1 | i in index_set(x) ]);
```

4.1.2 Variable Permutation Symmetries

The following lex-leader symmetry breaking constraint for variable permutation constraints works as follows. Let ρ_i be the permutation defined by row i of the matrix p. It ensures that x is lexicographically less than or equal to $\rho_i(\rho_j^{-1}(x))$ for all permutations i, j in p. This means that no permutation swap will improve the lexicographic value of x, using the original order x for all constraints to ensure compatibility of the pairwise symmetry breaking constraints. This requires an additional predicate *var_perm_sym_pairwise*, which takes the original variable array x and two permutations p_1, p_2 on it. It generates the inverse of p_1 and decomposes to a lexicographical constraint ensuring that x is less than or equal to the application of p_1^{-1} then p_2 as required. The decomposition of *var_perm_sym* generates pairwise permutation constraints for all pairs of permutations in p, normalising the set x as the permutations in p are from $1..length(x)$.

```
predicate var_perm_sym(array[int] of var int: x,
                       array[int,int] of int: p) =
  let { int: l = min(index_set_1of2(p)),
        int: u = max(index_set_1of2(p)),
        array[1..length(x)] of var int:
          y = [ x[i] | i in index_set(x) ] }
  in
  forall (i, j in l..u where i != j) (
    var_perm_sym_pairwise(y,
      [ p[i,k] | k in index_set_2of2(p) ],
      [ p[j,k] | k in index_set_2of2(p) ])
  );

predicate var_perm_sym_pairwise(
    array[int] of var int: x,
    array[int] of int: p1,
    array[int] of int: p2) =
  let { int: n = length(x),
        array[1..n] of 1..n:
          invp1 = [ j | i, j in 1..n
                      where p1[j] = i ] }
  in
  lex_lesseq(x, [ x[p2[invp1[i]]] | i in 1..n ]);
```

4.1.3 Value Permutation Symmetries

The following lex-leader symmetry breaking constraint for value permutation symmetries works as follows. Similarly to *var_perm_sym*, *val_perm_sym* is decomposed into pairwise constraints for each of the pairs of permutations in s. Each pairwise predicate decomposes into a 'lex_lesseq' constraint on a new local variable y similarly to *val_seq_sym*. The new local decision variable array $y[i]$ has value 0, 1 and 2 for $x[i] \in A$, $x[i] \in B$ and $x[i] \in C$ when $x[i]$ is mapped to a worse, equal and better value, respectively. The 'lex_lesseq' constraint then enforces that $x[1] \in A \vee (x[1] \in B \wedge x[2] \in A) \vee (x[1] \in B \wedge x[2] \in B \wedge x[3] \in A) \ldots$ This constrains x so that solutions which can be mapped under the pairwise value permutation swap to lexigraphically worse solutions are excluded, as required.

```
predicate val_perm_sym(array[int] of var int: x,
                       array[int,int] of int: s) =
  let { int: l = min(index_set_1of2(s)),
        int: u = max(index_set_1of2(s)) }
  in
  forall (i in l..u, j in l..u where i != j) (
    val_perm_sym_pairwise(x,
      [ s[i,k] | k in index_set_2of2(s) ],
      [ s[j,k] | k in index_set_2of2(s) ])
  );

predicate val_perm_sym_pairwise(
    array[int] of var int: x,
    array[int] of int: s1,
    array[int] of int: s2) =
  let { int: l = min(index_set(x)),
        int: u = max(index_set(x)),
        set of int:
          A = { s1[i] | i in index_set(s1)
                      where s1[i] < s2[i] },
        set of int:
          B = { s1[i] | i in index_set(s1)
                      where s1[i] = s2[i] },
        set of int:
          C = { s1[i] | i in index_set(s1)
                      where s1[i] > s2[i] },
        array[l..u] of var 0..2: y }
  in
  forall (i in index_set(x)) (
       y[i] = 0 <-> (x[i] in A)
    /\ y[i] = 2 <-> (x[i] in C)
  )
  /\
  lex_lesseq(y, [1 | i in index_set(x)]);
```

4.2 Multiple Declarations

A difficulty arises when converting multiple symmetry declarations into lex-leader symmetry breaking constraints. It is well known that simply taking the conjunction of multiple lex-leader symmetry breaking constraints is not correct in general.

Example 5. Consider a simple problem with two variables $x[1]$ and $x[2]$ that can take the values 1 or 2, with the constraint that they must be different. This problem has two solutions $x = [1, 2]$ and $x = [2, 1]$, as well as a variable symmetry on the array x.

```
array[1..2] of var 1..2: x;
x[1] != x[2];
var_sym( [ x[1], x[2] ] );
var_sym( [ x[2], x[1] ] );
```

The two symmetry declarations in the model take different orderings for the variable array, and decompose into the constraints $x[1] \leq x[2]$ and $x[2] \leq x[1]$ respectively, which makes the problem unsatisfiable. Although this example is somewhat contrived, the naive decomposition of these symmetry predicates results in an incorrect model.

Example 6. Consider the following set of alternate symmetry declarations for the Latin Squares problem in Example 2. They are identical to the original ones, except that the values array in the value symmetry has been reversed.

```
val_sym([x[i,j] | i, j in 1..n],
        [n - i + 1 | i in 1..n]);
var_seq_sym(
  array2d(1..n, 1..n, [x[i,j] | i, j in 1..n]) );
var_seq_sym(
  array2d(1..n, 1..n, [x[j,i] | i, j in 1..n]) );
```

Consider a Latin Square of size 3 (the model is not made inconsistent for $n \leq 3$). When decomposed the value symmetry produces two constraints, that the minimum position of $s[1]$ (3) must be less than the minimum position of $s[2]$ (2), and that the minimum position of $s[2]$ (2) must be less than the minimum position of $s[3]$ (1). This applies to the flattened version of x and since the first three variables in x correspond to the first row, they each take a different value from $1, 2, 3$. So due to the value symmetry constraints $x[1,1] = 3$, $x[1,2] = 2$ and $x[1,3] = 1$. Then from the all different constraint $x[2,1] \in 1, 2$ so $x[2,1] < x[1,1]$. However the decomposition of the first variable sequence symmetry produces the lexicographical constraint $lex\_lesseq([x[1,1], x[1,2], x[1,3]], [x[2,1], x[2,2], x[2,3]])$ which implies that $x[2,1] \geq x[1,1]$. This contradicts $x[2,1] < x[1,1]$ so the model is inconsistent.

4.2.1 Consistency

In order to consistently decompose symmetry declarations, a set of compatible symmetry breaking constraints must be found. A sufficient condition for a set of lex-leader symmetry breaking constraints to be compatible is that they all follow the same lexicographical ordering.

Corollary 1. Given a finite domain CSP $P \equiv (V, D, C)$ and a lexicographic ordering O on variables V, then we can prune all assignments θ such that $\exists \theta'.\theta' \prec_O \theta$ where $\prec_O$ is the lexicographic ordering relation using O, without changing the satisfiability of P.

Proof. Follows from Theorem 1 of [4]. □

Since static symmetry breaking constraints can conflict with the search strategy the choice of global ordering is important. In order to maintain transparency the variable ordering can be extracted from the search annotation and symmetry declarations, or alternately provided by the modeller. Then we can convert the symmetry declarations into lex-leader symmetry breaking constraints which are consistent with "order".

In what follows we consider how to modify the symmetry constraints to ensure they only remove symmetric solutions that are not the least lexicographic solution given a global order `order`.

4.2.2 Variable Symmetries

For variable symmetries, we first sort the variables so they are ordered according to the global order `order`. This makes use of a new MiniZinc function `sort_by_var` which returns the variables in x sorted as they appear in *order*, so e.g. `sort_by_var`($[a, b, c, d, e]$, $[e, c, f, d, a, g, b]$) returns $[e, c, d, a, b]$.

```
array[int] of var int:
sort_by_var(array[int] of var int: x,
            array[int] of var int: order);
```

```
var_sym_ord(array[int] of var int: x,
            array[int] of var int: order) =
  let { array[1..length(x)] of var int:
          x2 = sort_by_var(x, order) }
  in
  var_sym(x2);
```

4.2.3 Value Symmetries

In order to apply value symmetries we must first sort the variables and the values, to make sure the symmetry breaking is compatible.

```
predicate val_sym_ord(
    array[int] of var int: x,
    array[int] of int: s,
    array[int] of var int: order) =
  let { array[1..length(x)] of var int:
          x2 = sort_by_var(x, order),
        array[1..length(s)] of int:
          s2 = sort(s) }
  in
  val_sym(x2, s2);
```

4.2.4 Variable Sequence Symmetries

Variable sequence symmetries can be decomposed into a set of equivalent variable permutation symmetries. The ordered version of variable sequence symmetry breaking is implemented by decomposing to the ordered version of variable permutation symmetry breaking. For any i, j, the symmetry swapping the ith and jth row in x is equivalent to a variable permutation symmetry mapping the concatenation of the ith and jth row to the concatenation of the jth and ith row.

```
predicate var_seq_sym_ord(
    array[int,int] of var int: x,
    array[int] of var int: order) =
  let { int: l = min(index_set_1of2(x)),
        int: u = max(index_set_1of2(x)),
        int: n = 2*card(index_set_2of2(x)) }
  in
  forall (i in l..u, j in i+1..u) (
    let { array[1..n] of var int:
          y =  [ x[i,k] | k in index_set_2of2(x) ]
            ++ [ x[j,k] | k in index_set_2of2(x) ],
          array[1..2,1..n] of int:
          p = array2d(1..2, 1..n,
                [ k | k in 1..n ]
              ++ [ k+(n div 2) | k in 1..(n div 2) ]
              ++ [ k | k in 1..(n div 2) ]) }
    in
    var_perm_sym_ord(y, p, order)
  );
```

4.2.5 Value Squence Symmetries

For value sequence symmetries the input array x is sorted, this is sufficient since the values are always treated in increasing order in any case.

```
predicate val_seq_sym_ord(
    array[int] of var int: x,
    array[int,int] of int: s,
    array[int] of var int: order) =
  let { array[1..length(x)] of var int:
          x2 = sort_by_var(x, order) }
```

```
  in
  val_seq_sym(x2, s);
```

4.2.6 Variable Permutation Symmetries

The decomposition for variable permutation symmetries already makes use of an order to make each pairwise symmetry breaking constraint compatible. For multiple symmetries, it simply uses the global order rather than the default order of x. This requires relabeling the permutation matrix. This makes use of a new MiniZinc builtin function `index_sort` which takes as input two variable sequences covering the same set of variables, and returns a permutation `new_index` of `index_set(x)` such that `x[i]` is the same variable as `y[new_index[i]]`. E.g.,
`index_sort`$([a, b, c, d, e], [e, c, d, a, b])$ returns [4, 5, 2, 3, 1].

```
array[int] of int:
index_sort(array[int] of var int: x,
           array[int] of var int: y);
```

```
var_perm_sym_ord(array[int] of var int: x,
                 array[int,int] of int: p,
                 array[int] of var int: order) =
  let { int: n = length(x),
        int: np = card(index_set_1of2(p)),
        array[1..n] of 1..n:
          r = index_sort(x, order),
        array[1..np,1..n] of 1..n:
          pr = array2d(1..np,1..n,
          [ p[i,r[j]] | i in index_set_1of2(p),
                        j in 1..n ]) }
  in
  var_perm_sym(x, pr);
```

4.2.7 Value Permutation Symmetries

Finally for value permutations we again simply need to sort the input variable array x, and use the simple form.

```
predicate val_perm_sym_ord(
    array[int] of var int: x,
    array[int,int] of int: s,
    array[int] of var int: order) =
  let { array[1..length(x)] of var int:
          x2 = sort_by_var(x, order) }
  in
  val_perm_sym(x2, s);
```

4.3 Global Variable Ordering

In order to support multiple symmetry declarations by default decomposition we make the following changes to MiniZinc (in particular the tool `mzn2fzn` which flattens MiniZinc models to FlatZinc models:

- *Detect when multiple symmetry breaking constraints appear in a model.* The predicate definitions for the unordered symmetry declarations are given a new `symmetry` annotation. The translator then counts the occurrences of symmetry predicates, while leaving them unflattened in the first (usual) flattening stage.
- *Derive a global variable order.* We extend MiniZinc with an order annotation:

  `global_order(array[int] of var int: order)`

  If this annotation is present on the search item for the model then it is used as the global order for the variables. If no annotation appears, but some variable sequences appear in the search annotation then the concatenation of the sequences of variables appearing (removing duplicates) are considered the global order. In any case the global order is extended with the all remaining variables that appear in symmetry declarations in the final model to ensure that no variable does not appear in the order.

- *Modify the translation to make use of ordered predicates.* When the translator detects that there are two or more symmetry predicates the ordered predicates must be used. Each symmetry predicate is unrolled in a later phase, where the predicates are replaced by ordered versions with an '_ord' suffix, and the global order argument is automatically added. This makes use of the newly implemented `index_sort` and `sort_by_var` builtins. Alternately, if only a single symmetry predicate is used, the simple "unordered" default decomposition can be used.

5. SOLVER SYMMETRY BREAKING

While default static symmetry breaking constraints are provided for all of the declarations, there are a number of ways that a solver supporting MiniZinc can handle the symmetries.

- they can ignore them;
- they can apply static symmetry breaking using the decompositions described here;
- they can provide their own decomposition to static symmetry breaking constraints;
- they can provide their own internal global static symmetry breaking constraints; or
- they can provide dynamic symmetry breaking using the definitions.

They can also mix the approaches but there are some caveats in doing so to ensure that the symmetry breaking approaches are compatible.

5.1 Solver Specific Static Decomposition

Any solver can choose to implement the proposed declarations using their own decomposition in MiniZinc, or pass them through as a global constraint to be handled natively. There are significant performance advantages to be gained by this. The decompositions defined above can be extremely large, resulting in significant time and memory overhead for the flattening and solving processes. In particular the variable permutation symmetries produce large numbers of lexicographical inequality constraints, for example when variable sequence symmetries on large arrays are decomposed by the ordered predicates. Ignoring these declarations, partially breaking them, or only specifing some of the permutations could improve performance for a given model. Unfortunately it is a difficult and manual process to determine exactly which static symmetry breaking constraints are the most effective to use for a particular model and solver.

Example 7. By defining a specific MiniZinc global library a solver writer can specialize how the symmetry predicate is handled. Suppose the solver natively supports `val_sym`, then the solver writer simply adds a file `val_sym.mzn` to their global library containing:

```
predicate val_sym(array[int] of var int: x,
                  array[int] of int: s);
```

This tells the MiniZinc system to pass these predicates directly to the FlatZinc sent to the solver. There is a small complication when dealing with globals with two dimensional arrays since FlatZinc only allows one dimensional arrays. To handle this some simple rewriting is required. For example for native support of `val_perm_sym` the solver writer would add a file `val_perm_sym.mzn` containing:

```
predicate val_perm_sym(array[int] of var int: x,
                       array[int,int] of int: s) =
  let { int: n = card(index_set_1of2(s)) }
  in
  val_perm_sym_fzn(x, n,
    [s[i,j] i in index_set_1of2(s),
            j in index_set_2of2(s)]);

predicate val_perm_sym_fzn(
  array[int] of var int: x, int: n,
  array[int] of int: s);
```

This rewrites the 2d array to 1d and passes in the size of the first dimension (so the solver can reconstruct the 2d array). To be compatible with multiple symmetry declarations the solver writer should implement decompositions or globals that implement the "ordered" versions of the symmetry breaking predicates, making use of the global order argument that will be passed in by the MiniZinc translation.

5.2 Dynamic Symmetry Breaking

Finally, if the solver supports some form of dynamic symmetry breaking then the solver writer can modify the globals library for the solver as above, but rather than treat the symmetry predicates as constraints, can record them for use by the dynamic symmetry breaking approach.

If they only support some of the forms of symmetry constraints then they can either:

- add decompositions to translate to the forms of symmetry predicates that the dynamic approach does support, if this is possible; or
- ensure that the dynamic symmetry breaking approach makes use of the "order" given in the ordered versions, and let the remaining symmetry constraints be handled by static decomposition.

A dynamic method such as LDSB [11] which ignores permutation symmetries and breaks the other symmetries in a consistent and efficient manner may often be preferable to producing large numbers of static constraints.

6. EXPERIMENTAL ANALYSIS

This section evaluates the consistency and effectiveness of the new MiniZinc symmetry declarations and the current methods for utilising them. We compare several main ways of handling symmetry declarations using MiniZinc solvers: no symmetry breaking, i.e., ignoring the symmetry declarations, the default decomposition into static symmetry breaking constraints as described in this paper, and dynamic symmetry breaking available in solvers. We consider two solvers - Chuffed [2] and Gecode [8] - which have existing MiniZinc interfaces that have been extended to allow dynamic symmetry breaking - LDSB [11] - to be used on the symmetry

Problem	chuffed-none			chuffed-static			chuffed-ldsb			gecode-static			gecode-ldsb		
	Sols	Fails	Time	Sols	Fails	Time	Sols	Fails	Time	Sols	Fails	Time	Sols	Fails	Time
latin-5	161k	161k	4.52	31	42	0.39	50	104	0.16	31	177	0.24	50	207	0.16
latin-6	24.6M	24.6M	∞	4.93k	4.99k	1.36	8.86k	13.7k	0.93	4.93k	21.1k	0.73	9.06k	28.1k	0.71
nqueens-12	14.2k	90.1k	15.68	5.56k	43.8k	3.96	3.97k	35.6k	2.59	5.56k	137k	0.72	3.97k	110k	0.50
nqueens-13	73.7k	415k	151.80	29.3k	197k	67.45	20.0k	157k	42.61	29.3k	710k	3.47	20.0k	559k	2.25
nqueens-14	360k	2.14M	∞	141k	1.01M	413.84	99.8k	801k	319.94	141k	3.92M	18.65	99.8k	3.10M	12.28
nqueens-15	-	-	-	178k	1.64M	∞	195k	1.76M	∞	893k	23.1M	122.37	613k	18.1M	78.88
nqueens-16	-	-	-	164k	1.62M	∞	214k	1.96M	∞	5.72M	146M	780.81	3.98M	114M	514.25
nnqueens-7	20.1k	90.5k	17.30	4	188	0.70	1.72k	17.1k	2.51	4	1.29k	0.32	4	1.30k	0.15
nnqueens-8	0	297k	99.58	0	1.49k	1.20	0	152k	62.50	0	184k	6.79	0	184k	3.01
nnqueens-9	0	1.74M	∞	0	72.6k	40.67	0	1.39M	∞	0	23.9M	∞	0	53.9M	∞
bibd-7-3-1	151k	157k	21.07	1	37	0.69	2	28	0.10	1	41	0.23	2	55	0.09
bibd-7-3-2	4.06M	4.44M	∞	24	205	2.01	25	201	0.16	24	373	0.71	35	531	0.17
bibd-8-4-3	2.16M	4.04M	∞	92	640	2.60	164	1.11k	0.24	92	1.42k	0.92	164	2.44k	0.27
bibd-9-3-1	3.00M	3.21M	∞	8	182	2.35	40	305	0.23	8	263	0.81	40	641	0.22
bibd-11-5-2	2.15M	3.13M	∞	1	112	2.81	72	939	0.32	1	213	1.02	72	1.96k	0.34
bibd-13-3-1	826k	932k	∞	215k	240k	∞	441k	1.09M	645.94	502k	5.02M	∞	545k	5.11M	∞
bibd-13-4-1	1.61M	1.64M	∞	8	308	5.07	4.50k	18.1k	4.30	8	1.37k	1.92	5.04k	51.1k	5.26
bibd-15-3-1	482k	500k	∞	140k	152k	∞	129k	213k	274.61	357k	1.39M	∞	366k	1.44M	∞
bibd-15-7-3	682k	2.31M	∞	256	18.5k	14.73	324k	4.31M	∞	256	83.4k	8.08	286k	16.6M	∞
bibd-16-4-1	786k	857k	∞	2.43k	54.3k	107.81	306k	2.48M	∞	2.43k	1.55M	152.58	318k	9.38M	∞

Table 1: Comparison of symmetry breaking methods: finding all solutions.

declarations provided. Note that since not all of these methods are complete symmetry breaking methods they will not necessarily find the same number of solutions. Additionally the implementation of LDSB and various globals predicates differs between Chuffed and Gecode, so results also differ between the two solvers.

All experiments were run using a single core on an AMD FX-8350 4.0GHz processor with 16GB of RAM. For each run the number of failures during search, the MiniZinc flattening time and the solver runtime was recorded. For the optimisation runs the value of the solutions was also recorded. A timeout of 15 minutes was used for the solving stage for all runs. The time taken by MiniZinc to flatten models is not considered in the timeout, however it is discussed where relevant as the time taken to flatten depends on the symmetry breaking method used and can be non-negligible.

The following problems are considered, with a focus on problems that contain multiple symmetries.

Latin Squares (n) A Latin Squares problem of size n consists of an $n \times n$ matrix where each row and column contains each of the numbers from 1 to n. It has a value symmetry, variable sequence symmetries (row and column swaps) and at least one permutation symmetry (rotation/reflection).

N Queens (n) An N-Queens problem of size n requires placing n queens on an $n \times n$ chessboard, where the queens cannot attack each other with the normal rules of chess (rows, columns and diagonals can only have one queen). As there must be exactly one queen per column, the model represents the board using an array of the position of the queen in each column. This model has a variable sequence symmetry (column swaps on the chessboard) and a value sequence symmetry (row swaps on the chessboard).

NN Queens (n) An NN-Queens problem requires colouring an $n \times n$ chessboard with n colours, where a pair of queens placed on any two squares of the same colour cannot attack each other. This is represented using an $n \times n$ array of integers from $1..n$ representing the colours of each square on the chessboard. It has a value symmetry, variable sequence symmetries (reflections) and permutation symmetries (rotation).

Balanced incompleted block design (v, k, λ) A BIBD problem with parameters (v, b, r, k, λ) consists of a binary $v \times b$ matrix such that each row sums to r, each column sums to k, and the dot product between any two distinct rows is λ. As b and r depend on the other parameters for all solutions, we consider BIBD-v-k-λ problems. This has variable sequence symmetries (column and row swaps).

Table 1 shows the effectiveness of symmetry breaking methods in reducing the amount of search required for problems with a large number of symmetries. In particular the Latin Squares problem, which has both a large number of solutions and symmetries is drastically faster when redundant parts of the search tree—including symmetric solutions—are pruned either by a static or dynamic method. Across all the runs, the default static symmetry breaking constraints are effective in reducing the search space, as demonstrated by the reduction in the number of failures while searching, as well as the overall run time (flattening and solving). Similarly the decomposition to global predicates passed to the solver for native dynamic symmetry breaking was also effective. Over most runs, the static symmetry breaking performed about as well as the dynamic LDSB method, including on the NN Queens and BIBD problems, which do not have any permutation symmetries (these aren't natively supported by LDSB).

Problem	chuffed-none			chuffed-static			chuffed-ldsb		
	Fails	Flat	Time	Fails	Flat	Time	Fails	Flat	Time
latin-30	208k	0.15	20.51	208k	125.17	93.05	208k	25.92	23.31
latin-40	153k	0.25	8.84	152k	392.80	68.08	153k	81.20	10.96
latin-42	52.0k	0.27	1.34	52.0k	477.48	44.42	52.0k	99.26	2.82
latin-seq-30	208k	0.12	21.25	208k	99.59	88.55	208k	0.16	21.99
latin-seq-40	153k	0.17	9.06	152k	307.69	68.63	153k	0.29	9.74
latin-seq-42	52.0k	0.18	1.37	52.0k	376.47	46.74	52.0k	0.32	1.56
latin-rev-30	207k	0.17	24.38	1.72k	123.32	14.18	207k	0.17	25.56
latin-rev-40	154k	0.27	9.14	102k	389.93	∞	154k	0.31	9.45
nqueens-20	18.1k	0.10	1.09	18.1k	0.14	1.23	18.1k	0.10	1.17
nqueens-22	120k	0.11	43.31	120k	0.16	42.67	120k	0.11	42.36
nqueens-24	30.4k	0.14	3.00	30.4k	0.17	3.26	30.4k	0.13	3.14
nqueens-26	27.2k	0.16	2.66	27.2k	0.19	2.88	27.2k	0.15	2.80
bibd-15-7-3	1.23k	0.54	0.07	277	7.40	0.64	723	0.54	0.07
bibd-16-4-1	536	0.73	0.10	383	12.81	1.07	369	0.71	0.09
bibd-19-3-1	199k	2.71	14.68	11.4k	139.18	34.99	17.0k	2.79	1.97

Table 2: Comparison of symmetry breaking methods: finding the first solution.

Table 2 demonstrates some of the challenges with using static symmetry breaking. Applying symmetry breaking methods is not necessarily beneficial when only the first solution is required, due to overhead and the interaction of symmetry breaking constraints with the search method. This is demonstrated most obviously by the Latin Squares problem, which has a significant number of symmetries that result in a large number of constraints being generated by the flattening process. While the number of failures during the search is the same, there is significant overhead for the flattening time and the overall solver runtime. The *latin* LDSB runs which statically decompose the permutation symmetries suffer from the same overheads, while there are minimal time increases for the *latin-seq* runs which ignore those symmetries altogether. For various problems, completely breaking permutation symmetries statically may not have a desirable trade-off in runtime vs flattening time when these permutations cover a large array of decision variables. This is part of the loss of runtime efficiency and tuning opportunity incurred by using a generic modelling language such as MiniZinc, which we plan to improve through work on better static constraints. Rewriting the constraints directly for a solver, or writing a specific algorithm for each problem will likely give better results at the expensive of the modellers time. However using these symmetry declarations can save a significant amount of modelling time by providing a good enough solution in acceptable time.

The interaction between the additional symmetry breaking constraints and the search strategy is a common problem with static methods. The *latin-rev* runs use the same Latin Squares model, with the search strategy branching on the maximum value from the domain of a given variable instead of the minimum. While no symmetry breaking and LDSB perform similarly with either search order, the static symmetry breaking performs poorly. The default global ordering tends to constrain variables which appear early in the search order so that they are smaller than variables which occur later. While an obvious method for dealing with this example would be to reverse the global ordering for this type of search strategy, generally the interaction between static symmetry breaking constraints and search is not well understood. So while static symmetry breaking can drastically improve search times, it is also possible for a modeller to manually provide a search strategy and global ordering that do not interact well at all, resulting in poor performance for our default constraints.

7. CONCLUSION AND FUTURE WORK

Symmetries appear in many combinatorial problems, and without treatment make finding solutions for these problems much harder. By allowing symmetries to be declared in the model in a uniform way we strengthen models and make it easy to compare different symmetry breaking approaches. We provide a solution to the problem of multiple symmetry declarations by defining compatible lex-least static decompositions that make use of a global order. We believe that having symmetry declarations in a standard modelling language is an important step for the community, and will help advance our understanding and use of symmetries.

Support for MiniZinc symmetry declarations can be added to the solvers discussed in this paper and we hope that having a more consistent base against which to compare symmetry breaking methods will lead to further improvements in the area. There is future work to be done in comparing generic symmetry breaking methods, adding and improving symmetry breaking support in MiniZinc solvers, and exploring the interaction between search and static symmetry breaking.

Acknowledgments

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

[1] Apt, K. R. and Wallace, M. [2006], *Constraint logic programming using ECLiPSe*, Cambridge University Press.

[2] Chu, G. [2011], Improving Combinatorial Optimization, PhD thesis, Department of Computing and Information Systems, University of Melbourne.

[3] Chu, G., Garcia de la Banda, M., Mears, C. and Stuckey, P. [2011], Symmetries and lazy clause generation, *in* 'Proceedings of the 22nd International Joint Conference on Artificial Intelligence', pp. 516–521.

[4] Chu, G. and Stuckey, P. J. [2015], 'Dominance breaking constraints', *Constraints* **20**(2), 155–182.

[5] Cohen, D. A., Jeavons, P., Jefferson, C., Petrie, K. E. and Smith, B. M. [2006], 'Symmetry definitions for constraint satisfaction problems', *Constraints* **11**(2-3), 115–137.

[6] Crawford, J. M., Ginsberg, M. L., Luks, E. M. and Roy, A. [1996], Symmetry-breaking predicates for search problems, *in* 'Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning', Morgan Kaufmann, pp. 148–159.

[7] Fahle, T., Schamberger, S. and Sellmann, M. [2001], Symmetry breaking, *in* T. Walsh, ed., 'Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming', Vol. 2239 of *Lecture Notes in Computer Science*, Springer, pp. 93–107.

[8] Gec [n.d.], 'Gecode - an open, free, efficient constraint solving toolkit', www.gecode.org.

[9] Gent, I. P. and Smith, B. M. [2000], Symmetry breaking in constraint programming, *in* W. Horn, ed., 'Proceedings of the 14th European Conference on Artificial Intelligence', IOS Press, pp. 599–603.

[10] Mears, C., De La Banda, M. G. and Wallace, M. [2009], 'On implementing symmetry detection', *Constraints* **14**(4), 443–477.

[11] Mears, C., Garcia de la Banda, M., Demoen, B. and Wallace, M. [2014], 'Lightweight dynamic symmetry breaking', *Constraints* **19**(3), 195–242.

[12] Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G. and Tack, G. [2007], Minizinc: Towards a standard CP modelling language, *in* C. Bessiere, ed., 'Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming', Vol. 4741 of *LNCS*, Springer-Verlag, pp. 529–543.