Symmetry Declarations for MiniZinc

Geoffrey Chu and Peter J. Stuckey

National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au

Abstract. Underlying symmetries in constraint satisfaction and optimization problems can make the search for solutions or optimal solutions much harder. In contrast, when symmetries are known, they can be used to speed up the search for solutions 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 show how to add symmetry declarations to MiniZinc models. These symmetries can then either be treated as static symmetry breaking constraints using MiniZinc global decomposition, or passed to a dynamic symmetry breaking method if the underlying solver supports it. This will allow a better understanding of the strengths and weaknesses of different symmetry breaking approaches by allowing simpler comparisons of different systems.

1 Introduction

Underlying symmetries in constraint satisfaction and optimization problems can make these problems much harder to solve. In contrast, when symmetries of a problem are known, we can dramatically improve problem solving by breaking the symmetries either statically or dynamically. Unfortunately, while many different approaches to symmetry breaking have been developed [3, 5, 7, 4, 6], there has been little direct comparison of the approaches (although see [9]). The reason is that most symmetry breaking approaches are implemented in only one system.

Declaring symmetries should be a standard part of modelling. If the modeller understands that symmetries exist, then they should be declared so the solver used can take advantage of them. Similarly a standard notation for symmetries gives a target for tools that automatically detect symmetries [8], so these tools can become generic and system independent. A library of models with declared symmetries in a widely supported standard for modelling allows easier comparison of the strengths and weaknesses of different symmetry breaking approaches, and helps to evaluate new symmetry breaking appraoches.

When symmetries are declared in a model we can make use of generic symmetry breaking techniques (e.g., SBDD [4, 6]) to take advantage of the symmetries. Unfortunately, most constraint solvers do not support such symmetry breaking

methods. Another effective way of breaking symmetries is to convert the symmetry declarations into a set of lex-leader symmetry breaking constraints (e.g., [3]). This has the advantage that such constraints will be supported by all solvers which support MiniZinc.

In this paper we describe how to add symmetry declarations to models in MiniZinc. The contributions of this paper are:

- We provide a complete set of symmetry declarations that allow any *solution symmetry* (as defined by Def. 2 in [2]) to be declared as part of a MiniZinc model.
- We provide 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 systems that do treat symmetries better may do so straightforwardly).
- 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 6 basic symmetry declarations for MiniZinc. In Section 4 we discuss how to handle the symmetry declarations, in particular giving default MiniZinc decompositions for each declaration. In Section 5 we give some experiments comparing different systems and symmetry breaking approaches. Finally in Section 6 we conclude.

2 Preliminaries

We assume the reader is familiar with MiniZinc [10].

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 $d \in D(x)$. We denote the set of all pairs of P 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 $\{x/d | (x/d) \in \theta, x \in scope(c)\}$ — 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* from [2].

Definition 1. *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.*

3 Symmetry Declarations

We introduce 6 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 (although perhaps not very concisely), so these declarations are complete in terms of their ability to express symmetries. In the following, we show the integer (`int`) versions only, but analogous versions for Booleans (`bool`) exist.

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

Requires: All variables in x are distinct. *Means*: Every variable in the set x is interchangeable, i.e., for any i, j, if we swap the values of $x[i]$ and $x[j]$, then solutions are preserved.

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

Requires: All variables in x are distinct. All values in s are distinct. *Means*: We can interchange any pair of values in s over all the variables in x, i.e., for any i, j, if we take every $x[k]$ which was set to $s[i]$ and change it to $s[j]$ and take every $x[k]$ which was set to $s[j]$ and change it to $s[i]$, then solutions are preserved.

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

Requires: All variables in x are distinct. *Means*: We can interchange any pair of variable sequences in x, i.e., for any i, j, if for every k, we swap $x[i, k]$ with $x[j, k]$, then solutions are preserved.

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

Requires: All variables in x are distinct. All values in s are distinct. *Means*: We can interchange any pair of value sequences in x, i.e., for any i, j, if for every m, we take every $x[k]$ which was set to $s[i, m]$ and change it to $s[j, m]$ and take every $x[k]$ which was set to $s[j, m]$ and change it to $s[i, m]$, then solutions are preserved.

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

Requires: All variables in x are distinct. Each row of p 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*: We can map any variable sequence represented in p to another variable sequence represented in p, i.e., for any i, j, if for every k, we take $x[p[i, k]]$ and set it to the value of $x[p[j, k]]$, then solutions are preserved.

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

Requires: All variables in x are distinct. Each row of s covers the same set of values. *Means*: We can map any value sequence in s to another value sequence in s, i.e., for any i, j, if we take every $x[k]$ which was set to $s[i, m]$ and change its value to $s[j, m]$, then solutions are preserved.

Note that {`var`,`val`}`_perm_sym` is 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, we can introduce Boolean variables 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.

Notice that the symmetry declarations are no different from normal MiniZinc user-defined predicates. Indeed the default behaviour of symmetry declarations in MiniZinc will be to add static symmetry breaking constraints.

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

```
include "alldifferent.mzn";
int: n; %% size
array[1..n,1..n] of var 1..n: x;
constraint forall(i in 1..n)(alldifferent([x[i,j] | j in 1..n]));
constraint forall(j in 1..n)(alldifferent([x[i,j] | i in 1..n]));
solve satisfy;
```

then this has a number of symmetries. We can declare the value symmetries, row symmetries and column symmetries of the latin squares problem by adding

```
% value symmetries
constraint val_sym([x[i,j] | i in 1..n, j in 1..n], [i | i in 1..n]);
% row symmetries
constraint var_seq_sym(array2d(1..n, 1..n, [x[i,j] | i in 1..n, j in 1..n]));
% column symmetries
constraint var_seq_sym(array2d(1..n, 1..n, [x[j,i] | i in 1..n, j in 1..n]));
```

We can generate new symmetry declarations which are special cases of the above predicates using MiniZinc predicates.

Example 2. A common symmetry for problems with sequences is that the sequence is reversable. We can create a new symmetry predicate `rev_seq_sym` which captures this using `var_perm_sym`, as follows:

```
include "var_perm_sym.mzn";
predicate 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);
```

Example 3. Another common symmetry for problems of square matrices is that of rotation symmetry. We can define a new symmetry predicate `rot_sqr_sym` which captures rotational symmetries as follows:

```
include "var_perm_sym.mzn";
predicate 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,
```

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

```
        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 assert(n == n2, "rot_sqr_sym: rotation symmetry applied to" ++
              " non square matrix",
    var_perm_sym(y,p));
```

We can, for example, declare the rotational symmetries of the latin squares problem of Example 1 by adding the following to our mdoel:

```
% rotational symmetries
constraint rot_sqr_sym(x);
```

4 Handling Symmetry Declarations

A solver supporting MiniZinc can handle symmetry declarations in a model in a number of ways:

- 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 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.

4.1 Single Symmetry Declaration by Default Decomposition

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, we pick a lexicographical order (typically the order in which vars appear in the main argument x), and we add constraints 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. 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.

The lex-leader symmetry breaking constraint for variable symmetries simply requires that the variables are ordered by value:

```
predicate var_sym(array[int] of var int: x) =
   let { int: l = min(index_set(x)), int: u = max(index_set(x)) } in
   forall(i in l..u-1)(x[i] <= x[i+1]);
```

The lex-leader symmetry breaking constraint for value symmetries require that the earliest occurrence of each value is ordered:

```
predicate val_sym(array[int] of var int: x, array[int] of int: s) =
   let { int: l = min(index_set(x)), int: u = max(index_set(x)),
         array[1..length(s)] of var l..u+1:p =
            [ min([ u+1 + bool2int(x[j] = s[i])*(j-u-1) |
                    j in index_set(x) ]) | i in 1..length(s) ] } in
   forall (i in 1..length(s)-1) ( p[i+1] > min(p[i],u) );
```

The lex-leader symmetry breaking constraint for variable sequence symmetries requires that the rows are ordered lexicographically:

```
include "lex_lesseq.mzn";
predicate var_seq_sym(array[int,int] of var int: x) =
   let { int: l1 = min(index_set_1of2(x)),
         int: u1 = max(index_set_1of2(x)),
         int: l2 = min(index_set_2of2(x)),
         int: u2 = max(index_set_2of2(x)) } in
   forall (i in l1..u1-1) ( lex_lesseq([x[i,j] | j in l2..u2],
                                       [x[i+1,j] | j in l2..u2]));
```

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 is easily expressed as a lexicographical constraint:

```
include "lex_lesseq.mzn";
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
   } 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)]);
```

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. We ensure that x is lexicographically less than $\rho_i(\rho_j^{-1}(x))$ for all permutations i, j

in p. This means that no permutation swap will improve the lexicographic value of x, we use the original order x for all constraints to ensure compatibility of the pairwise symmetry breaking constraints.

```
include "lex_lesseq.mzn";
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 in l..u, j in l..u where i != j) (
      var_perm_sym_pairwise(y, %% forces index 1..length(x)
                            [ 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 { array[1..length(x)] of 1..length(x): invp1 =
                    [ j | i,j in 1..length(x) where p1[j] = i ] } in
   lex_lesseq(x, [ x[p2[invp1[i]]] | i in 1..length(x) ]);
```

The following lex-leader symmetry breaking constraint for value permutation symmetries works as follows. We consider each value permutation individually. For a permutation mapping $s[i,k]$ to $s[j,k]$, we look at the most significant variable ($x[1]$) and consider when the value permutation maps it to a better/worse/same lexicographical value. With A, B, C defined as below in val_perm_sym_pairwise, $x[1] \in A$ means it will be mapped to a worse value, $x[1] \in B$ means it will be mapped to the same value and $x[1] \in C$ means it will be mapped to a better value, so we want $x[1] \in A \vee x[1] \in B$. If $x[1] \in B$, then we have consider the next most significant variable $x[2]$, etc. Thus we can enforce $x[1] \in A \vee (x[1] \in B \wedge x[2] \in A) \vee (x[1] \in B \wedge x[2] \in B \wedge x[3] \in A) \ldots$, which can be expressed as a lexicographical constraint.

```
include "lex_lesseq.mzn";
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 Symmetry Declarations by Default Decomposition

A difficulty arises when we wish to convert multiple symmetry declarations into lex-leader symmetry breaking constraints. It is well known that simply taking the con-

junction of multiple lex-leader symmetry breaking constraints is not correct in general. Instead, we need to find a set of symmetry breaking constraints which are compatible. A sufficient condition for a set of lex-leader symmetry breaking constraints to be compatible is that they all follow the same lexicographical ordering.

Suppose we are given a variable ordering "order", which can either be user specified, extracted from the search annotation, or extracted from the symmetry declarations. Then we can convert the symmetry declarations into lex-leader symmetry breaking constraints which are consistent with "order" as follows:

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: MiniZinc builtin function

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

which returns the variables in x sorted as they appear in $order$, so e.g. `sort`($[a, b, c, d, e]$, $[e, c, f, d, a, g, b]$) returns $[e, c, d, a, b]$.

```
include "var_sym.mzn";
predicate 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(x, order) } in
   var_sym(x2);
```

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

```
include "val_sym.mzn";
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(x, order),
         array[1..length(s)] of int: s2 = sort(s)
   } in  val_sym(x2, s2);
```

Variable sequence symmetries can be decomposed into a set of equivalent variable permutation symmetries. We implement the ordered version of variable sequence symmetry breaking by decomposing to the ordered version of variable permutation 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.

```
include "var_perm_sym_ord.mzn";
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)),
   for (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,
               [i | i in 1..n] ++ [n+1-i | i in 1..n])
      } in var_perm_sym_ord(y, p, order);
```

For value sequence symmetries, we simply need to sort the input variable array x, since the values are always treated in increasing order in any case.

```
include "val_seq_sym.mzn";
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(x, order) } in
   val_seq_sym(x2, s);
```

Our decomposition for variable permutation symmetries already make use of an order to make each pairwise symmetry breaking constraint compatible. For multiple symmetries, we simply use the global order rather than the default order of x. This requires relabeling the permutation matrix. We make use of a new MiniZinc builtin function

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

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].

```
include "var_perm_sym.mzn";
predicate 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: r = card(index_set_1of2(p)),
         array[1..n] of var int: y = sort(x, order),
         array[1..n] of 1..n: r = index_sort(x,y),
         array[1..r,1..n] of 1..n: pr = array2d(1..r,1..n,
              [ p[i,r[j]] | i in index_set_1of2(p), j in 1..n ]) } in
    var_perm_sym(y, pr);
```

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

```
include "val_perm_sym.mzn";
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(x, order) } in
   val_perm_sym(x2, s);
```

In order to support multiple symmetry declarations by default decomposition we need to 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.* By annotating the predicate definitions for the base symmetry constraint with a new annotation `symmetry`, we can strightforwardly modify the translator to count the occurences of symmetry predicates. By default the translator should not unroll predicates which are annotated as `symmetry` in the first (usual) flattening stage.

– *Derive a global variable order.* We can extend MiniZinc with an order annotation:

```
annotation global_order(array[int] of var int:g);
```

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 concatentation of the sequences of variables appearing (removing duplicates) are considered the global order. In any case we extend the global order with the all remaining variables that appear in the final model in the order created by the translator, simply 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 translation must be modified. Each symmetry predicate is unrolled in a later phase, where the global order argument is automatically added. If only a single symmetry predicate is used, the simple "unordered" default decomposition can be used.
– *Implement* `index_sort` *and* `sort`. This is a simple matter of programming.

4.3 Solver Specific Static Symmetry Breaking Constraints

Any solver can choose to implement the static symmetry breaking constraints defined herein as using their own decomposition or as a global. There are significant performance advantages to be gained by this, since the decompositions defined above can be extremely large. By defining a specific Minizinc global library for their solver they can specialize the how the symmetry predicate is handled.

Example 4. 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_fz(x, n,
                  [s[i,j] i in index_set_1of2(s), j in index_set_20f2(s)]);
predicate val_perm_sym_fzn(array[int] of var int:x,int:n,array[int] of int:s);
```

which 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 passes in by the MiniZinc translation.

4.4 Solver Specific Dynamic Symmetry Breaking Approaches

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.

5 Experiments

The extension of MiniZinc to automatically handle multiple symmetry declarations is still a work in progress. In particular, the ordered versions of the predicates are not currently supported as `sort` and `index_sort` has not yet been implemented. For the experiments, we get around this by making sure that every symmetry declaration uses the same default variable ordering, so that no sorting is required to enforce the compatibility of the decompositions.

We compare 4 different ways of handling the symmetry declarations: CHUFFED with no symmetry breaking, i.e., just ignore the symmetry declarations (chuffed-none), CHUFFED with an implementation of short-cut SBDS [1] (chuffed-sbds), CHUFFED with the default decomposition into static symmetry breaking constraints described in this paper (chuffed-static), and Gecode with the default decomposition into static symmetry breaking constraints described in this paper (gecode-static).

We try 4 problems with various kinds of symmetries. The *Latin Squares* problem (see Example 1) has value symmetries (val_sym), row and column symmetries (var_seq_sym $\times$ 2), and rotational symmetries (rot_sqr_sym). The well known *N-Queens* problem has a horizontal flip symmetry (var_seq_sym), and a value sequence symmetry (val_seq_sym). The 3-dimensional version of the N-Queens problem, the *NN-Queens* problem, has value symmetries (val_sym), horizontal and vertical flip symmetries (var_seq_sym $\times$ 2), and rotational symmetries (rot_sqr_sym). The *Balanced Incomplete Block Design* (BIBD) problem (see e.g. `http://en.wikipedia.org/wiki/Block_design`) has row and column symmetries (var_seq_sym $\times$ 2). For each problem, we find all solutions of instances with an appropriate size. The models are available from `www.cs.mu.oz.au/~pjs/minisym/`

A time out of 15 minutes was used. For each method, we show the number of solutions found, the number of failures required, and the time spent in Table 1. Note that since none of these methods are complete symmetry breaking methods they will not necessarily find the same number of solutions.

It can be seen from Table 1 that the completeness level of our static decomposition into symmetry breaking constraints (chuffed-static and gecode-static) is greater than that of the shortcut SBDS method implemented in CHUFFED chuffed-sbds on Latin Square and BIBD, and is the same on NN-Queens and N-Queens. chuffed-static is several times slower than chuffed-sbds on NN-Queens for the same level of completeness, but is much faster on BIBD since it is more complete and has a smaller search space on that problem.

Table 1. Comparison of the different ways the symmetry declarations can be handled. We compare CHUFFED with no symmetry breaking (chuffed-none), CHUFFED with shortcut SBDS (chuffed-sbds), CHUFFED with static decomposition (chuffed-static), and Gecode with static decomposition (gecode-static).

Problem	chuffed-none			chuffed-sbds			chuffed-static			gecode-static		
	Sols.	Fails	Time	Sols.	Fails	Time	Sols.	Fails	Time	Sols.	Fails	Time
Latin-5	161280	27	3.13	56	63	0.01	31	11	0.01	31	62	0.02
Latin-6	>14M	>11k	T.O.	9408	7691	0.37	4930	60	0.40	4930	5672	1.42
NN-Queens-7	20160	70235	33.7	4	181	0.01	4	181	0.04	4	650	0.11
NN-Queens-8	0	296246	230	0	1440	0.07	0	1440	0.40	0	92101	13.78
N-Queens-11	2680	18313	1.50	1072	9125	0.47	1072	9125	0.47	1072	13037	0.19
N-Queens-12	14200	74524	34.6	5564	37387	6.22	5564	37387	6.80	5564	63236	0.94
BIBD-7-3-1	151200	6251	14.6	2	25	0.01	1	22	0.01	1	20	0.01
BIBD-8-4-3	>1M	>1M	T.O.	164	943	0.05	92	462	0.03	92	621	0.10
BIBD-16-4-1	>2M	>212k	T.O.	>241k	>42k	T.O.	2436	47473	14.78	2436	774840	192

6 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 compatibly lex-least static decompositions that make use of a global order. All the decompositions are available from `www.cs.mu.oz.au/~pjs/minisym/` We believe that adding symmetry declarations to a standard modelling langauge is an important step for the community, and will help advance our understanding and use of symmetries.

We plan to extending the MiniZinc tool set to handle multiple symmetry declarations as defined in Section 4.2. One can already use MiniZinc for single symmetries, and multiple symmetry declarations where the user takes care to ensure the orderings are compatible. For the MiniZinc Challenge 2013 we hope to use multiple benchmarks that include symemtry declarations which will hopefully give impetus to solver writers to support the symmetry declarations natively.

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. G. Chu, M. Garcia de la Banda, C. Mears, and P.J. Stuckey. Symmetries and lazy clause generation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 516–521, 2011.
2. David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.
3. James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the 5th In-*

ternational Conference on Principles of Knowledge Representation and Reasoning, pages 148–159. Morgan Kaufmann, 1996.

4. Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2001.
5. Pierre Flener, Justin Pearson, Meinolf Sellmann, and Pascal Van Hentenryck. Static and dynamic structural symmetry breaking. In *Proceedings of the 12th International Conference on Principles and Practic of Constraint Programming*, pages 695–699, 2006.
6. Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2001.
7. Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In Werner Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 599–603. IOS Press, 2000.
8. C. Mears, M. Garcia de la Banda, and M. Wallace. On implementing symmetry detection. *Constraints*, 14(4):443–477, 2009.
9. Chris Mears. *Automatic Symmetry Detection and Dynamic Symmetry Breaking for Constraint Programming.* PhD thesis, Clayton School of Information Technology, Monash University, 2010.
10. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer-Verlag, 2007.