Symmetry in the Generation of Constraint Models *

Alan M. Frisch[1], Chris Jefferson[2],
Bernadette Martinez-Hernandez[1], Ian Miguel[3]

[1]Dept. of Computer Science, Univ. of York, UK. {frisch, berna}@cs.york.ac.uk
[2]Computing Laboratory, Univ. of Oxford, UK. chrisj@comlab.ox.ac.uk
[3]School of Computer Science, Univ. of St Andrews, UK. ianm@cs.st-and.ac.uk

Abstract. Symmetry is often introduced in the formulation of constraint models. This paper shows how an automated constraint modelling system can recognise when it takes modelling decisions that introduce symmetry, and build up a description of the symmetry present in the final model of a whole problem class.

1 Introduction

Constraint programming is used to solve a given combinatorial problem in two phases. First, the problem is characterised or *modelled* as a *constraint satisfaction problem* (CSP): a finite set of decision variables, each with a finite set of potential values, and a set of constraints on the allowed assignments of values to the variables. Second, a constraint solver is used to search for *solutions*: assignments to the decision variables that satisfy all constraints.

The modelling process may introduce *symmetries* (solution-preserving transformations) into the model. Consider the following example problem:

Example 1. Find a set of three digits that sum to 16.

A common method of modelling this set is as three decision variables, x, y and z, whose domains are all $\{0, \ldots, 9\}$. In addition, an all-different constraint ensures that x, y and z take distinct values, and a sum constraint ensures that their sum is 16. The solution $x = 1, y = 7, z = 8$ can be transformed into another by permuting the variables, for example: $z = 1, x = 7, y = 8$. This permutation also maps every non-solution to a non-solution. All permutations of the variables preserve solutionhood similarly. Notice that this symmetry is introduced *irrespective* of the constraints on the original set: the original set has no ith element, whereas the model does.

Introducing symmetry into a model in this manner can have a substantial adverse affect on the effort to find a solution by systematic tree search. In the worst case, the search may need to explore all symmetric equivalents of fruitless

* Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship.

sub-trees. There are a variety of ways to eliminate this redundancy from the search space, but to do so requires knowledge of the symmetry in the model. This paper shows how an automated constraint modelling system can recognise when it takes a modelling decision that introduces symmetry, and can therefore build up a description of symmetries present in the final model as the model is generated.

2 Background

A finite-domain constraint satisfaction problem is a triple $\langle \mathcal{X}, D, C \rangle$ where $\mathcal{X}$ is a finite set of variables, D is a function mapping each $x \in \mathcal{X}$ to a finite set of values (its domain) and $\mathcal{C}$ is a finite set of constraints on $\mathcal{X}$. Each constraint $c \in \mathcal{C}$ is defined as a pair $\langle \mathcal{X}', \rho \rangle$ where $\mathcal{X}'$ is a sequence whose elements are drawn from $\mathcal{X}$ (the *scope* of the constraint) and ρ is a subset of the Cartesian product of the domains of the members of $\mathcal{X}'$, which give the set of allowed combinations of values. If $|\mathcal{X}'| = 2$, then c is a binary relation.

A *literal* is a pair $\langle x, v \rangle$, where x is a variable and v is a value from its domain. An *assignment* is a set of literals, no two of which involve the same variable. A *complete assignment* contains exactly one literal involving each variable in the problem. A complete assignment A satisfies a constraint c if and only if the assignments to the variables in the scope of c form one of the tuples allowed by c. A solution to a CSP is a complete assignment that satisfies all of its constraints.

A *symmetry* is a bijection of the set of literals of a CSP that maps every solution to a solution. We usually consider a set of symmetries that is closed under inverse and composition and contains the identity function: a group. This induces a partitioning on the complete assignments into symmetry classes, and in each class every member is a solution or no member is.

There are two common special cases of CSP symmetry. A symmetry s is a *variable* symmetry if there exists a bijection b on variables such that $s(\langle x, v \rangle) = \langle b(x), v \rangle$ for every literal $\langle x, v \rangle$. We usually identify a variable symmetry by b rather than s. Example 1 exhibits variable symmetry. A common type of variable symmetry arises when dealing with matrices of decision variables. Consider the following simple problem:

Example 2. Find a relation on $A \times B$, where $A = \{1, 2, 3\}$ and $B = \{1, 2, 3\}$, such that each element of A is related with two elements of B and each element of B is related to two elements of A.

A common way to model this type of problem is with a two-dimensional matrix R indexed by A and B. The domains of all the variables in R are $\{0, 1\}$, and $R[i, j] = 1$ if and only if tuple $\langle i, j \rangle$ is in the relation. To satisfy the problem constraint, we constrain each row and each column to sum to 2. Consider the following solution:

$$\begin{array}{c|ccc}
 & 1 & 2 & 3 \\
\hline
1 & 1 & 1 & 0 \\
2 & 0 & 1 & 1 \\
3 & 1 & 0 & 1
\end{array}$$

It can be transformed into another solution by permuting the values of the indices of the rows and/or columns. For example, by exchanging the first and third columns and the second and third rows:

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Again, all such permutations preserve solutionhood. This type of variable symmetry is known as *row* symmetry and *column* symmetry. The general term is *index* symmetry.

A symmetry s is a *value* symmetry if there exists a bijection b on values such that $s(\langle x, v\rangle) = \langle x, b(v)\rangle$ for every literal $\langle x, v\rangle$. We usually identify a value symmetry by b rather than s. An example of value symmetry is graph colouring, where the nodes of a graph must be coloured 'red', 'green' and 'blue' and no two adjacent nodes can be the same colour. In any solution, swapping the nodes coloured 'red' and the nodes coloured 'green' generates another solution.

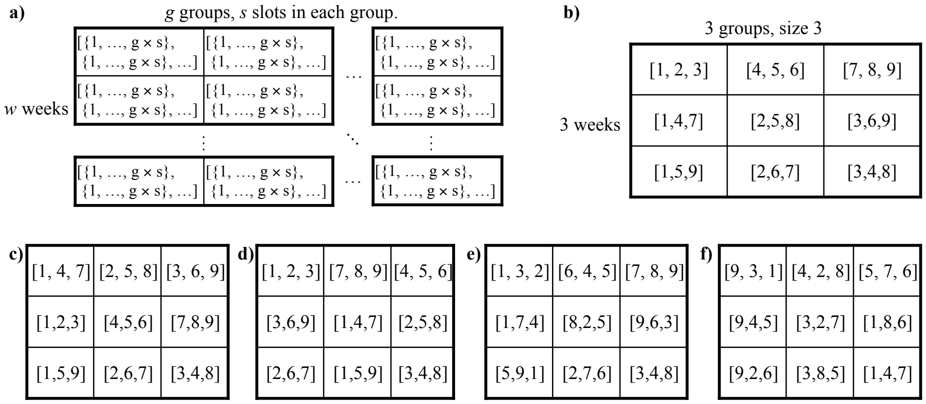
Cohen *et al.* [1] show that automatically detecting all possible symmetries of a CSP is NP-hard. As this is not practical, a common limitation is to only detect those symmetries which map the set of constraints to itself. These can be found by transforming the CSP into a graph, and then detecting the isomorphisms of this graph [7]. The main drawbacks of this approach is that there is no known polynomial algorithm to find the isomorphisms of a graph and the algorithm works on individual CSPs, whereas we are usually interested in problem classes and the symmetry present in all the CSPs that model each element of that class. Furthermore, the difficulty of solving the graph isomorphism problem tends to increase with the size of the instance, for example Mears *et al.* [6] show how the time taken to detect the symmetries of different instances of the same problem class increases rapidly as the instance size increases.

3 Symmetry Introduced by Modelling

To illustrate how symmetry enters a model, consider modelling the Social Golfers problem (problem 10 at www.csplib.org), which is defined as follows:

> In a golf club there are a number of golfers who wish to play together in g groups of size s. Find a schedule of play for w weeks such that no pair of golfers play together more than once.

Hence, we are asked to *partition* the set of golfers into subsets (the groups) for each of the w weeks. Since the order of the weeks does not matter, we must find a *multiset* of partitions (in fact, it is a set, but this requires some inference). Constraint modelling is the reduction of a given problem to a CSP, in which the domains and constraints are supported by the intended constraint solver. Since current solvers do not support multisets or partitions directly, we must create



a)

[{1, ..., g × s}, {1, ..., g × s}, ...]	[{1, ..., g × s}, {1, ..., g × s}, ...]	...	[{1, ..., g × s}, {1, ..., g × s}, ...]
[{1, ..., g × s}, {1, ..., g × s}, ...]	[{1, ..., g × s}, {1, ..., g × s}, ...]		[{1, ..., g × s}, {1, ..., g × s}, ...]
⋮		⋱	⋮
[{1, ..., g × s}, {1, ..., g × s}, ...]	[{1, ..., g × s}, {1, ..., g × s}, ...]	...	[{1, ..., g × s}, {1, ..., g × s}, ...]

b)

[1, 2, 3]	[4, 5, 6]	[7, 8, 9]
[1,4,7]	[2,5,8]	[3,6,9]
[1,5,9]	[2,6,7]	[3,4,8]

c)

[1, 4, 7]	[2, 5, 8]	[3, 6, 9]
[1,2,3]	[4,5,6]	[7,8,9]
[1,5,9]	[2,6,7]	[3,4,8]

d)

[1, 2, 3]	[7, 8, 9]	[4, 5, 6]
[3,6,9]	[1,4,7]	[2,5,8]
[2,6,7]	[1,5,9]	[3,4,8]

e)

[1, 3, 2]	[6, 4, 5]	[7, 8, 9]
[1,7,4]	[8,2,5]	[9,6,3]
[5,9,1]	[2,7,6]	[3,4,8]

f)

[9, 3, 1]	[4, 2, 8]	[5, 7, 6]
[9,4,5]	[3,2,7]	[1,8,6]
[9,2,6]	[3,8,5]	[1,4,7]

Fig. 1. a) 3-dimensional matrix model of the Social Golfers problem presented as a 2-d matrix with a 1-d matrix in each cell. b) Solution to an instance with 3 groups of size 3 in each of 3 weeks. c) Symmetric solution (weeks permuted). d) Symmetric solution (groups within weeks permuted). e) Symmetric solution (slots within group permuted). f) Symmetric solution (golfers permuted).

a model of the problem in terms of a more primitive collection of variables and constraints.

As summarised in Figure 1*a*, a common model of this problem involves a three-dimensional matrix of decision variables. Each row represents one of the w weeks, each column one of the g groups, and the third dimension the slots in each group. The golfers are identified with the integers $\{1, \ldots, g \times s\}$, which is the domain of each variable. For each week, i, there is a constraint that every variable whose row index is i is assigned a distinct value (a golfer plays only once per week). There is also a constraint between each pair of groups in different weeks ensuring that they have at most one common element (otherwise the same pair of golfers play together more than once). Figure 1*b* shows a solution to the instance involving 3 groups of size 3 over 3 weeks.

Each modelling decision taken in formulating this model has introduced symmetry. Recall that we are asked to find a multiset of partitions of golfers. Representing the multiset, which has no indices, as a matrix, and thereby identifying each week with a row of the matrix, introduces symmetry on the w rows (see Figure 1*c*). Consider now modelling a partition as a 1-d matrix. The order of the subsets in a partition is immaterial, so the groups in each row can be permuted (see Figure 1*d*). Choosing to represent each subset in a partition as a one-dimensional matrix introduces symmetry on the 'slots' in each group (see Figure 1*e*). Finally, nowhere in the problem specification are the golfers identified individually. Hence, 'naming' the golfers $1, \ldots, g \times s$ introduces a symmetry on these values.

4 Symmetry Annotation in Automated Modelling

We have developed the abstract constraint specification language Essence [3] and the automated modelling system Conjure [4]. Essence allows a user to write down a combinatorial problem at a level of abstraction above that at which modelling decisions are made. Given an Essence specification, Conjure automatically *refines* it into models at a level of abstraction similar to that commonly supported by existing constraint languages (e.g. Figure 1*a*).

We now show how Conjure and Essence formalises the refinement performed informally in the previous section. Recall that the Social Golfers problem requires us to find a multiset of partitions. In Essence, this abstract decision variable can be written directly:

```
given    w, g, s : int (1...)
letting  golfers be new type of size g * s
find     schedule: mset (size w) of rpartition (size s) of golfers
```

where *schedule* is a decision variable whose (finite) domain is a multiset of regular partitions (in which a set is partitioned into equal-sized subsets) of *golfers.* The type *golfers* is *unnamed* [5] in that it is specified only by its size, which is $g \times s$; the golfers remain indistinguished. Hence, the Essence specification contains none of the symmetry of the model in the previous section.

Conjure refines an Essence specification using *refinement rules.* It has a refinement rule corresponding to each of the modelling decisions described in the context of the Social Golfers problem in the previous section, although they are generally applicable. The refinement rules that introduce symmetry add *annotations* to the model describing the symmetry introduced. Hence, the finished model contains a description of the symmetry introduced by the modelling process, removing the need to detect it. Furthermore, this description holds for all members of a problem class.

To illustrate we now give an overview of the rules required to refine the *schedule* decision variable. We begin with the outer multiset:

Refine	mset (size n) of τ
To	a matrix (indexed by $1..n$) of refine(τ)
Symmetry	1) permutations of the index values of the matrix 2) each element of the matrix independently has every symmetry of refine(τ)

Here, τ stands for any type. In our example τ is "`rpartition` (`size` s) of *golfers*". The rule says that the multiset of size n should be refined into a matrix indexed by 1 to n, each element of which contains the refinement of τ, i.e. we have created the rows of the model in Figure 1*a*. Since a multiset has no indices, but a matrix does, using this rule *always* introduces symmetry, regardless of the constraints on the original multiset. Hence, the first annotation of this rule records that this symmetry has been introduced. All elements of the matrix introduced by this rule share the same type: "refinement of τ". Therefore, the second annotation records that whatever symmetry this refinement introduces is present in each element of the matrix.

Continuing to refine *schedule*, the type of the elements of the matrix introduced by the multiset rule above is regular partition. Consider, therefore, the following rule:

Refine	rpartition (size n) of τ
To	a matrix (indexed by $1..n$, $1..\lvert\tau\rvert/n$) of refine(τ)
Symmetry	1) permutations of the values of the 1st index
	2) within each row, permutations of values of 2nd index

This rule refines a regular partition of size n into a two-dimensional matrix. The first dimension is indexed $1..n$ (one index per subset of τ), and the second $1..\lvert\tau\rvert/n$ (one index per element of each subset). Again, neither the subsets in the partition nor the elements of the subsets have indices. Hence, the application of this rule always introduces symmetry, which is recorded in the annotations. Notice now that, when used to refine the *schedule* decision variable in conjunction with the multiset rule above, *every* week (row of the multiset representation) is automatically annotated with the symmetry recognised here.

Finally, consider the rule for refining an unnamed type:

Refine	type (size n)
To	$1..n$
Symmetry	permutations of the values of $1..n$

This rule simply replaces the unnamed type of size n with the values $1..n$. This is necessary since both variables and values in a CSP must be named. However, it also introduces symmetry since, by definition, these values are not distinguished in the original problem specification.

The result of the application of these rules on the Social Golfers problem is:

```
given    w, g, s : int (1...)
letting  golfers be new type of size g ∗ s
find     schedule′: matrix indexed by [int(1, ..., w)]
                    of matrix indexed by [int(1, ..., g), int(1, ..., s)]
                    of int(1, ..., g ∗ s)
symmetry permutations of values of first index of schedule′
symmetry for each i in 1, ..., w:
             permutations of values of first index of schedule′[i]
             for each j in 1, ..., g:
                 permutations of index values of schedule′[i, j]
symmetry permutations of values 1, ..., g ∗ s
```

In summary, these three rules can both produce a model of the Social Golfers problem (that in Figure 1*a*) *and* detect the symmetry introduced into that model automatically. This avoids the need to detect these symmetries, which is potentially expensive.

5 Conclusions and Future Work

This paper has discussed how the constraint modelling process often introduces symmetry into a model in a systematic way. It showed that rules designed to capture modelling decisions can *annotate* a model for a whole problem *class* with the symmetry introduced. In future work, we will use group-theoretic symmetry annotations to aid in the automation of combining symmetries introduced by different modelling decisions, and in the production of a concise description of the symmetry in the model as a whole.

References

1. D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, B.M. Smith. Symmetry Definitions for Constraint Satisfaction Problems *Constraints* 11: 115-137, 2006.
2. J. Crawford, M. L. Ginsberg, E. Luks, A. Roy. Symmetry-breaking Predicates for Search Problems. *Proc. 5th KRR*, 148-159, 1996.
3. A.M. Frisch, M. Grum, C. Jefferson, B. Martinez-Hernandez, I. Miguel. The Design of ESSENCE: A Constraint Language for Specifying Combinatorial Problems. *Proc. 20th IJCAI*, 2007.
4. A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, I. Miguel. The Rules of Constraint Modelling. *Proc. 19th IJCAI*, 109-116, 2005.
5. A.M. Frisch, I. Miguel. The Concept and Provenance of Unnamed Types. Available at: http://www.cs.york.ac.uk/aig/constraints/Automodel, 2006.
6. C. Means, M. Garcia de la Banda, M. Wallace. On implementing symmetry detection. *Proc. of SymCon '06*, 2006.
7. J.-F. Puget Automatic detection of variable and value symmetries *Proc. CP '05*, 475-489, 2005.